Telektronikk
4/2000

Languages for Telecommunication applications

Telektronikk

Volume 96 No. 4 – 2000
ISSN 0085-7130

Editor:
Ola Espvik
Tel: (+47) 63 84 88 83
email: ola.espvik@telenor.com

Status section editor:
Per Hjalmar Lehne
Tel: (+47) 63 84 88 26
email: per-hjalmar.lehne@telenor.com

Editorial assistant:
Gunhild Luke
Tel: (+47) 63 84 86 52
email: gunhild.luke@telenor.com

Editorial office:
Telenor Communication AS
Telenor R&D
PO Box 83
N-2027 Kjeller
Norway
Tel: (+47) 63 84 84 00
Fax: (+47) 63 81 00 76
email: telektronikk@telenor.com

Graphic design:
Design Consult AS, Oslo

Layout and illustrations:
Gunhild Luke, Britt Kjus (Telenor R&D)

Prepress and printing:
Optimal as, Oslo

Circulation:
4,000

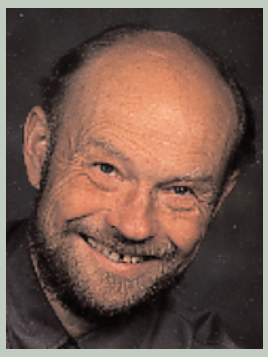
Telenor

Contents

Feature:
Languages for Telecommunications Applications

Special

Guest Editorial



Rolv Bræk

When the CCITT, now ITU-T, initiated work on specification and programming languages back in 1972, it was a bold step. At that time software engineering was in its infancy and the development of communication software very much a pioneering thing. Every system development involved breaking some new technological ground. At the same time it was clear that software offered far more possibilities than mere replacement of electromechanical and electronic solutions. Entirely new functionality was possible and was therefore gradually introduced into the systems. This is a well-known pattern from all areas of computing. But communication systems were not allowed to trade functionality for quality, as has been so common in other strands of computing. Even as the complexity was growing beyond bounds, the systems had to satisfy outstanding requirements to high-performance, reliability and no-break operation. Therefore, it became essential at an early stage to find ways to master the quality in the face of growing complexity.

The combination of high complexity with high reliability forced the communication software industry to take a pro-active approach to software quality from the very beginning. Since communication software always has been embedded real-time software with a high degree of concurrency, distribution and heterogeneity, the solutions that were developed attacked these problems from the very beginning, while they were not yet considered important in mainstream software engineering.

The early techniques developed for software engineering in general, such as SADT and Structured Analysis/Structured Design, focused on activities and data-flow. Quite deliberately they did not deal with sequential behaviour, concurrency and distribution. They emphasised abstraction and human understanding more than formality. They provided no formal semantics, and therefore it was not possible to simulate and analyse the system behaviour before it was implemented. Moreover, the mapping from abstract model to concrete design was unclear, and the value of abstract models was therefore limited to the early phases. They had little documentation value for the final product and were in many cases just thrown away. Apparently, the activity-oriented approach of those techniques did not deliver all the benefits promised, not even outside the communication domain.

Later developments have focused more on data modelling, and these have been considerably more successful, especially for data-intensive applications. In recent years, the trend has been towards object-orientation and more formality. The Unified Modelling Language, UML, is the latest and most notable development in this direction. It combines a set of graphical notations with a partial semantics that makes its meaning more precise. It has notation for sequential and concurrent behaviours based on StateCharts that enable a partial simulation of behaviour before it is implemented, but still it lacks a complete semantics.

The techniques developed for communication systems on the other hand, emphasised formality and dealt explicitly with sequential behaviour and concurrency from the beginning. All the formal description techniques (FDTs) ESTELLE, LOTOS and SDL had state transition based semantics that enabled simulation and analysis to take place before implementation. SDL had the additional benefit of a graphical notation that supported human comprehension combined with an underlying finite state machine semantics that could be implemented effectively. For this reason SDL has been the most successful of the FDTs, with a good track record from numerous industrial development projects.

SDL as a language was object-based already when first recommended in 1976, and since 1992 it has been a full-fledged object-oriented language. It has a semantics that supports formal validation and enables complete simulation to take place before implementation, and also to generate complete and efficient implementation code automatically. These properties enable development organisations to move from an *implementation oriented* development paradigm to a *design oriented* development paradigm. In the latter, a system is documented and maintained primarily using design descriptions and not by implementation code.

Contrary to the popular belief that techniques coming from the communication world are “old fashioned” they are still leading edge in the areas of object and behaviour modelling. When communication systems and information systems now merge into ICT systems, a corresponding merge of techniques from the “I” world and the “C” worlds is bound to take place. As the software industry in general moves towards dis-

tributed heterogeneous solutions we now see a convergence towards a similar merge for the software industry at large. This convergence leads to considerable cross-fertilisation and integration of previously different disciplines such as control systems, user interfaces and databases.

UML – now emerging as a family of languages that is competing with the ITU-T languages – is developing fast and attracting far more attention than the ITU-T languages ever did. From a technical point of view, the ITU-T languages and UML partly overlap and partly complement each other. The overlap area has been greatly extended by introducing into SDL-2000 notation from UML Class Diagrams and by introducing the notion of composite states from UML State Machines/State-Charts. It is now possible to define associations between types and also partially to define types using (parts of) the UML Class diagram notation within SDL. On the other hand, SDL complements UML by providing a complete operational semantics and the possibility to precisely define the component structure of aggregate types. MSC complements UML by providing structuring mechanisms entirely missing in the UML sequence diagrams and collaboration diagrams.

This issue of *Telektronikk* is about the family of languages currently standardised by ITU-T, and related methods, tools and middleware. The ITU-T language family presently consists of:

- The Specification and Description Language, SDL. The new version of SDL, called SDL-2000, is a major revision and is presented for the first time in a popular form in the article by Rick Reed. Rick Reed also presents the history of SDL in an accompanying article.

- Message Sequence Charts, MSC, which are used to describe external behaviour properties by means of interaction cases. MSC provide a useful complement to SDL and is used both as input when making SDL descriptions and as specification when performing verification and testing. Øystein Haugen presents the latest developments of MSC in the article *MSC-2000: interacting with the future*.

- The Abstract Syntax Notation One (ASN.1) is used to describe data structures, especially in connection with protocols. In combination with encoding rules for the physical transfer of data, ASN.1 is much used in protocol development, and may also be combined with SDL. Colin Willcock describes ASN.1 in his article.

- The Tree and Tabular Combined Notation, TTCN, which is used to describe test cases. TTCN may be generated from SDL and MSC. Ina Schieferdecker and Jens Grabowski describe TTCN in their article.

- CHILL – the CCITT HIgh Level (programming) Language. CHILL is an advanced programming language that supports concurrent processes. It has been adopted by many major telecom manufacturers and used successfully to develop a wide range of complex systems. CHILL is described in the article by Jürgen Winkler.

- The Object Definition Language, ODL, which is an extension of the Interface Definition Language, IDL, known from CORBA. ODL is introduced in the paper by Joachim Fischer and Marc Born.

A mapping between SDL and UML has been defined in the ITU-T recommendation Z.109, SDL combined with UML, which is elaborated in the article by Birger Møller-Pedersen. This mapping allows developers and tools to put leverage on the strengths of both languages by facilitating a combined use.

The article by Rolv Bræk and Arve Meisingset presents an overview of the language features of SDL-2000, MSC-2000 and UML. The purpose is to give readers that are unfamiliar with these languages a first introduction, and also a feeling for their main content as a background for the more detailed articles that follow. Readers with a basic knowledge of the languages, who are more interested in the new features, should move directly to the specialist articles.

One important asset of the ITU-T languages, especially SDL, has been its formally defined semantics. Principles for defining formal semantics are presented in the paper by Andreas Prinz, using examples from SDL and MSC as illustration.

It is a common misunderstanding that formal language and formal method is the same thing, but it is not. Methods are concerned with how to use the languages to achieve better results. Several methods have been introduced that are based on the ITU-T languages and UML. The article by Rolv Bræk presents some general methodology issues for using the ITU-T languages and UML. The article by Steve Randall presents specific guidelines for formal use of SDL in development, e.g. of ETSI standards, and Anthony Wiles and Milan Zoric report on their experiences from the application of these guidelines in the development of the Hiperlan standards.

One strongpoint of the ITU-T languages is that abstraction, using concepts suitable for human

comprehension, is combined with semantics suitable both for extensive tool support and efficient implementation. Extensive tool support for validation and testing is one of the benefits that result from this. The article by Dieter Hogrefe, Beat Koch and Helmuth Neukirchen introduces the general principles of validation and testing. The possibility to derive efficient implementations is another benefit elaborated in the article by Richard Sanders.

Two commercial sets of tools that support the ITU-T languages in combination with UML are presented in two separate articles. Anders Olsen and Finn Kristoffersen present the Cinderella tools, while Philippe Leblanc, Thomas Hjelm and Anders Ek present the Telelogic tools.

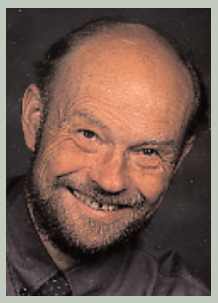
Middleware is an important area where communication and general computing converge. The article by Anastasius Gavras outlines the needs for distributed platforms for telecommunication applications.

Amardeo Sarma presents perspectives on future standardisation in the areas covered by this issue of *Telektronikk* in his article.

The idea behind the feature section was very ambitious. A lot of internationally acknowledged specialists have been involved. I would like to express special thanks to Arve Meisingset for all his co-editing work throughout this process. To my knowledge, this is the first time that all the ITU-T languages with associated topics have been presented in one place. Enjoy!

The ITU-T Languages in a Nutshell

ROLV BRÆK AND ARVE MEISINGSET



Rolv Bræk (56) received his Siv.ing. degree (M.S.E.E.) in 1969 from the Norwegian University of Science and Technology (NTNU) and is currently Professor in the Department of Telematics at NTNU. Rolv Bræk has extensive experience from application development using formal methods as well as from teaching, consulting and introducing systems engineering methodologies to industry. He is co-author of the book "Engineering Real Time Systems – An Object Oriented Methodology using SDL", and "TIMe The Integrated Method" published on CD-ROM by SINTEF. His current research interest is rapid service development.

Rolv.Braek@item.ntnu.no

Arve Meisinset (52) is Senior Research Scientist at Telenor R&D. He is currently working on information systems planning, and has previously been engaged in Case-tool development and formal aspects of human-computer interfaces. He has been involved in several network management projects, and has a particular interest in languages for data definitions and mathematical philosophy. He is ITU-T SG10 Vice Chairman, Working Party Chairman for WP3/10 Distributed Object Technologies, and the Telenor ITU-T technical co-ordinator.

arve.meisingset@telenor.com

This paper provides a condensed overview over MSC, SDL and UML intended both as a quick introduction for novice users and as a quick symbol reference for the more experienced. The last section provides a comparison of the ITU-T language family with UML from the Object Management Group.

1 Introduction to MSC

ITU Message Sequence Charts (MSCs) [1, 2, 3] is a formalised graphical language to define interaction scenarios in terms of asynchronous messages passed between instances.

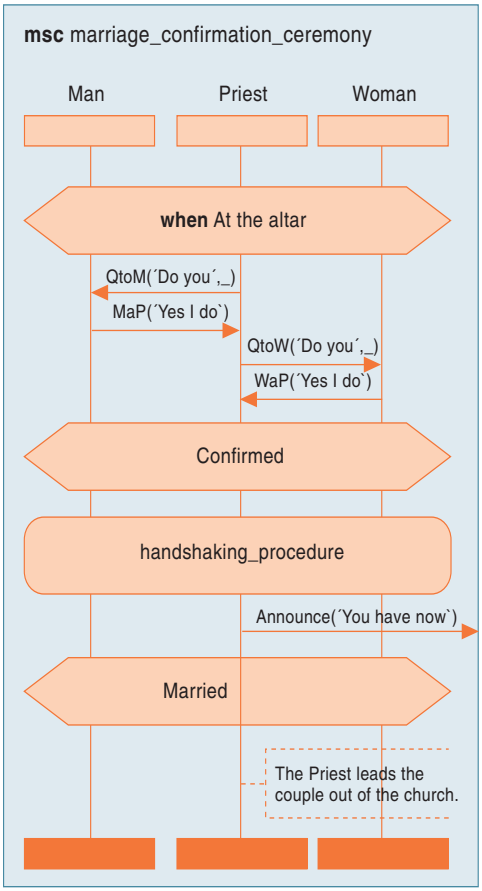
An MSC document comprises a set of graphs of the following kinds:

- simple MSC diagram;
- high-level MSC diagram (HMSC);
- MSC document diagram.

Simple MSC Diagrams

A simple MSC diagram may contain:

- instances;
- messages passed between instances, possibly with references to data;
- events on instances, for each message there is a sending event and a receiving event;
- actions inside instances;
- conditions spanning one or more instances;
- calls to methods and responses;
- references to other simple MSC diagrams;
- inline expressions describing alternatives, loops, exceptions and options;
- comments.



Keyword **msc** followed by diagram name.

Three instances: *Man, Priest, Woman.*

A guard.

Message representing question to the *Man* with a fixed and a wildcard data content.
A response message with a fixed data content.

A similar question to the *Woman* and the response.

A global condition.

MSC reference to ***msc*** *handshaking_procedure*, which is performed here.

Announce message sent to the environment representing a gate definition.

A shared condition that holds for the *Man* and the *Woman*, but not the *Priest*.

Comment.

End of diagram for each instance (but not end of the instance!).

Figure 1 Example Simple MSC

A name and a vertical time line represent each instance. The diagram specifies a total ordering of events along each timeline, but not between instances. The ordering of events between instances follows from the rule that a message must be sent by a sending event before it can be received by a receiving event.

An example Simple MSC is shown in Figure 1. Note that names are unique within the entity class, which means that e.g. a simple MSC diagram and an instance may have the same name.

High-level MSC

A High-level MSC (HMSC) describes how other MSCs may be composed to represent more complex cases. In an HMSC the MSCs are represented by MSC references that may be composed in sequence, in parallel or as alternatives. An HMSC does not depict instances or messages.

A High-level MSC may contain:

- start and end symbols (triangles);
- restrictive conditions;
- MSC references;
- connecting nodes (small round circles);
- connecting lines between the above; if they go downwards, they have no arrow.

An example High-level MSC is shown in Figure 2.

MSC Document

An MSC document diagram defines the context for simple MSC diagrams and may contain:

- instances with inherits (from) and data variables;
- messages;
- wildcards and their types;
- data signatures;
- MSC references;
- utilities, i.e. references to used MSCs to specify the defined MSCs.

An example MSC document diagram is shown in Figure 3.

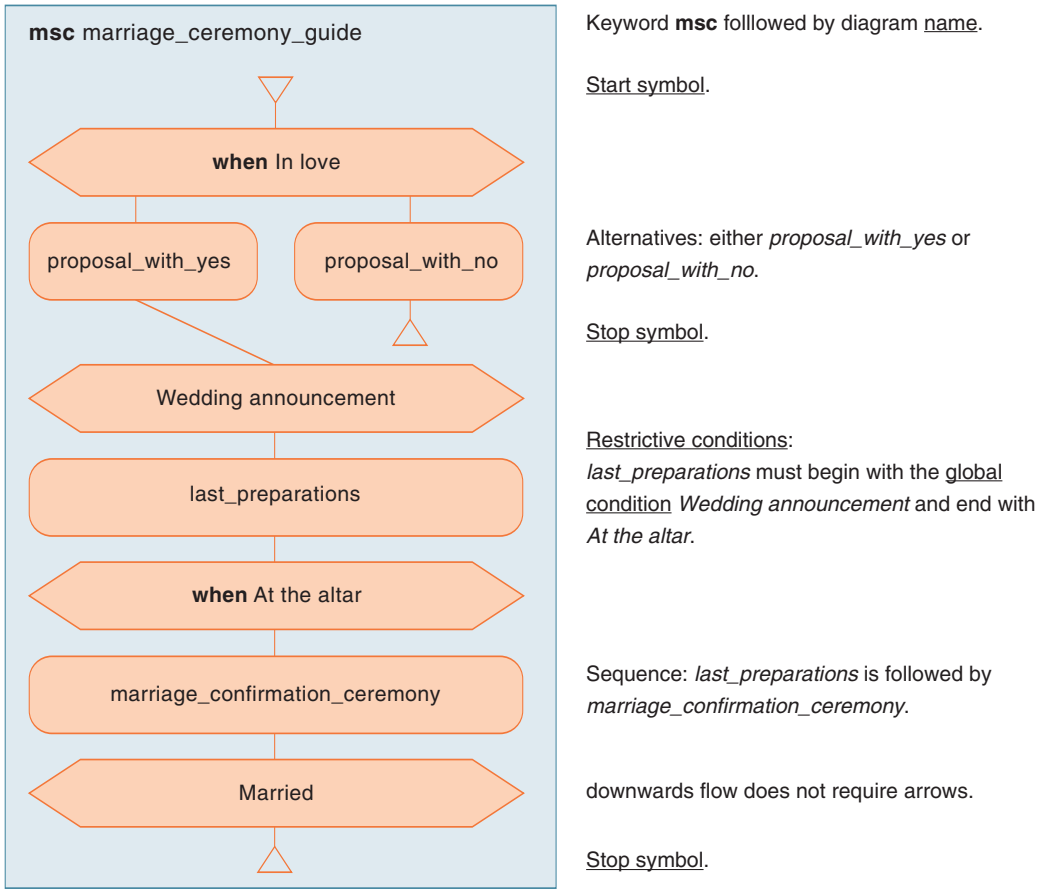
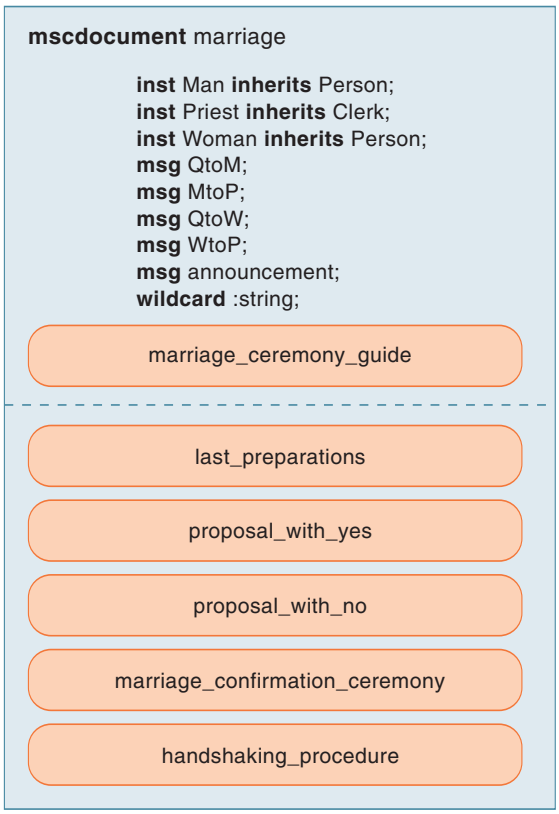


Figure 2 Example High-level MSC

Figure 3 Example MSC document



mscdocument followed by name of document

Declaration of instances with inherits (from) and declaration of variables (not shown).

Declaration of messages (only needed for messages with data parameters)

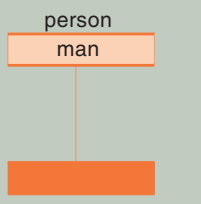
Declaration of a wildcard of type string

Declaration of defining MSC

Utility MSCs are listed under the dashed line.

Box 1 Simple MSC

This box provides an overview of a subset of MSC-2000. Note that not all features of MSC-2000 are listed.

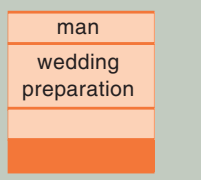


Instances have a name, an optional type, a vertical solid instance axis, which depicts a timeline from a head (blank box) to the end (black box).

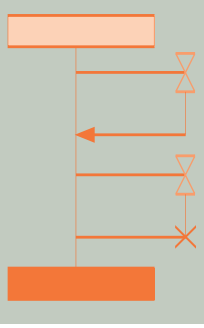
Names are written over or in the head. Names can contain numbers, capital and small letters, and underline, but not blanks.

Instances can be of unspecified or specified kind. The kind is always written above the head. The kind can also be the SDL kinds **system**, **block**, **process** and **service**. The instance kind may be decomposed, and a diagram showing the internal interactions of the decomposed instance can be indicated by the statement “man decomposed as interactions_internal_to_man” where “interactions_internal_to_man” is the name of an MSC.

An informal comment can be associated to any entity.



Instances may be depicted as columns and may contain actions (*wedding preparation*).



Timers have starts and stops, and timers can have min and max elapse time [min, max].

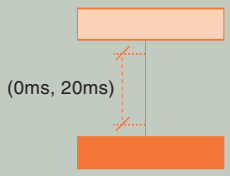
Timer is started

Timer expires

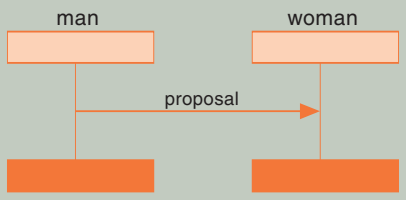
Timer is started

Timer is reset

Box 1 Simple MSC, continued

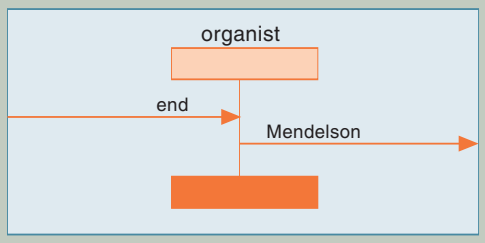


Time constraints can be specified.



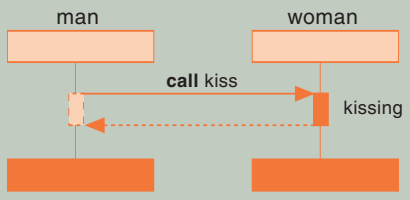
Messages have a sending event and a receiving event and may carry data. Message names are written over the arrows. Data are referred to in parantheses (variable, 'constant', _(i.e. wildcard)).

A message between two instances.

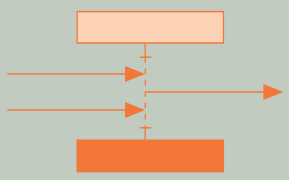


A message from the environment defining a gate.

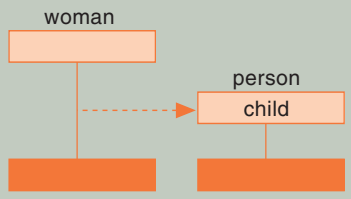
A message to the environment defining a gate.



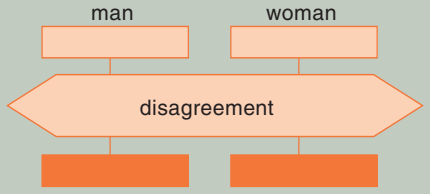
Object instances may invoke a method call (kiss) in another instance which may lock both instances until the end of the operation, indicated by a dotted arrow.



In Co-regions (dashed line) no ordering is imposed on the events.

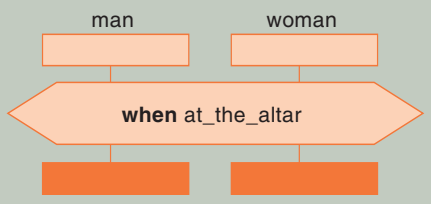


New instances (child of the kind person) can be created from the instance axis of an existing object (woman).

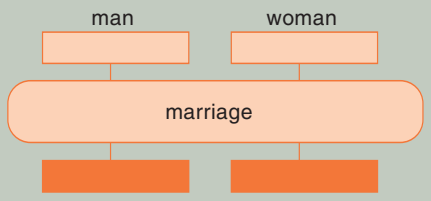


Conditions (disagreement) express states and can span one or more object instances.

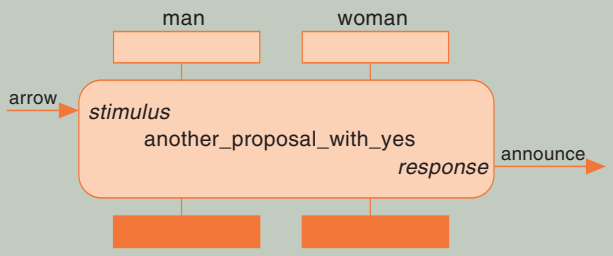
Box 1 Simple MSC, continued



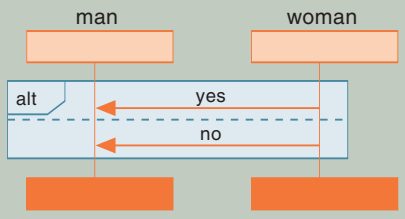
Guard: can only be first in an MSC or operand.



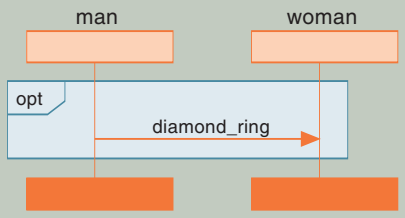
An MSC may contain MSC references (marriage) to other MSCs, and the reference may span several instances.



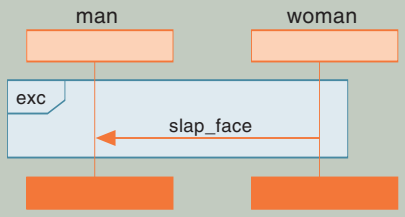
MSCs have explicitly or implicitly named gates, e.g. stimulus and response, for messages.



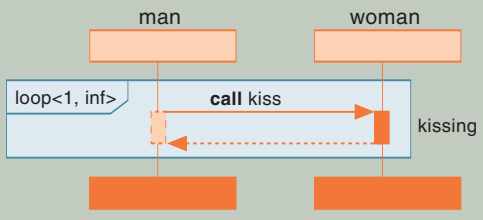
In-line expression giving two alternative sequences: the woman answers either yes or no.



Optional behaviour: giving diamond ring or giving nothing.



Exception: woman slaps man in face and the diagram (wedding) ends, or she does not and the diagram (wedding) goes on as normal.



A loop with one or more runs.

2 Introduction to SDL

SDL (ITU-T Specification and Description Language)) [4, 5, 6, 7, 8, 9] is a language for specifying reactive systems. This presentation provides an introduction to a subset of SDL. Important issues like data types, interfaces and inline expressions are not presented.

SDL *Systems* consist of a structure of communicating *Agents*. Each agent may have variables, procedures, a state machine and a structure of agents. An agent is characterised by the signals it may receive from and send to other agents, and by the procedures that it may perform upon request.

An Agent which contains a structure of concurrently behaving agents is called a Block, while an agent which contains a structure of agents that alternate (only one active at the time) or agents that have no internal agent structure is called a Process.

SDL provides the following kinds of diagrams:

- Agent diagrams that describe the properties of Agents, in terms of variables, procedures, an Agent state machine and contained Agents;
- State diagrams that depict the behaviour of Agents in terms of *States* and state *Transitions*;
- Procedure diagrams that depict the behaviour of Procedures;
- Package diagrams that define types that can be *Used* in other diagrams.

Agent Diagrams

Agent diagrams can be of the following kinds:

- system;
- system type;
- block;
- block type;
- process;
- process type.

Agent diagrams are used to define an agent or agent type and comprise a definition of its:

- locally defined types;
- internal structure.

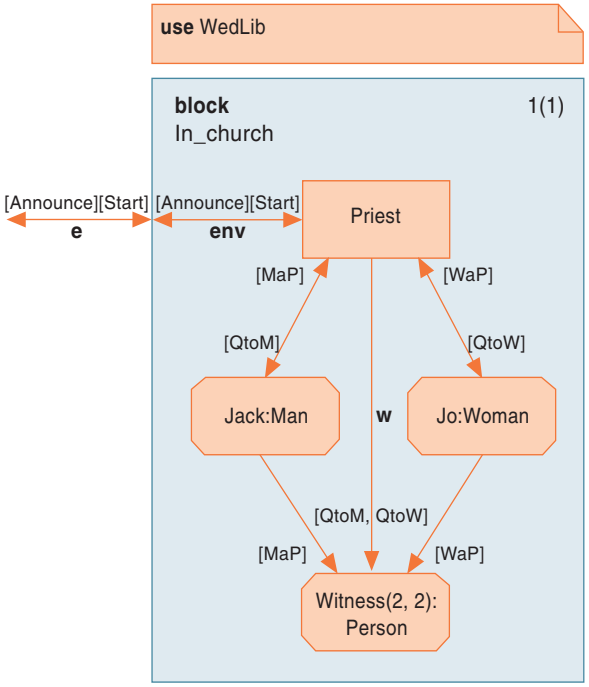
Local types may be defined directly in the agent diagram, but normally only a type reference is placed in the agent diagram, allowing the local type to be defined in a separate diagram. Local types may be block types, process types, data types and signals. Stereotyped UML class symbols may be used as such type references and also to provide partial type definitions as described in Box 3.

The internal structure of an agent may depict:

- sets of agent instances;
- channels;
- signal lists.

Gates may be attached outside the frame symbol of agent types.

The kind of diagram is identified by a corresponding keyword in the upper left corner (inside the frame symbol) of the Agent diagram. An example Agent diagram is depicted in Figure 4.



Reference to Package *WedLib* containing type definitions used by the block *In_church*. This way, the block and process types need not be referenced inside the block diagram itself.

Keyword **block** followed by block name *In_church*
Page 1(of 1)

Block called *Priest*. It is considered as defined here, but actually defined on a separate diagram (called **block** Priest).

Two-way channel called env to the environment
Two-way gate called *e*

Two-way channels between block *Priest* and processes *Jo:Woman* and *Jack:Man*

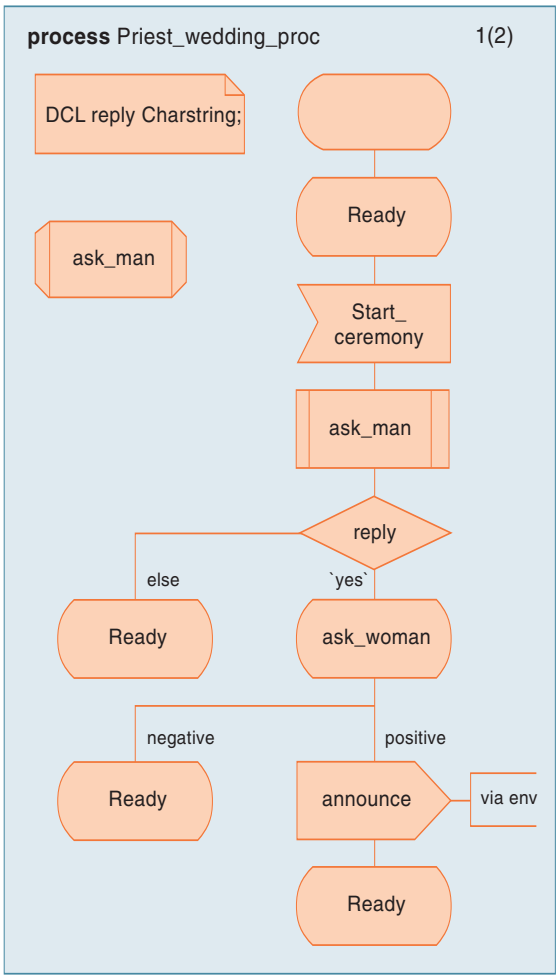
Signals in the direction of the arrows

Process instances called *Jack* of type *Man* and *Jo* of type *Woman*

Process Set with 2 process instances called *Witness* of type *Person*

Figure 4 Example Agent diagram

Figure 5 Example process state diagram



Variable *reply* of type Char string is declared process starts here when created

and enters a state called *Ready*.

Procedure definition refrence for procedure *ask_man*

consuming input signal *Start_ceremony* triggers *transition*,

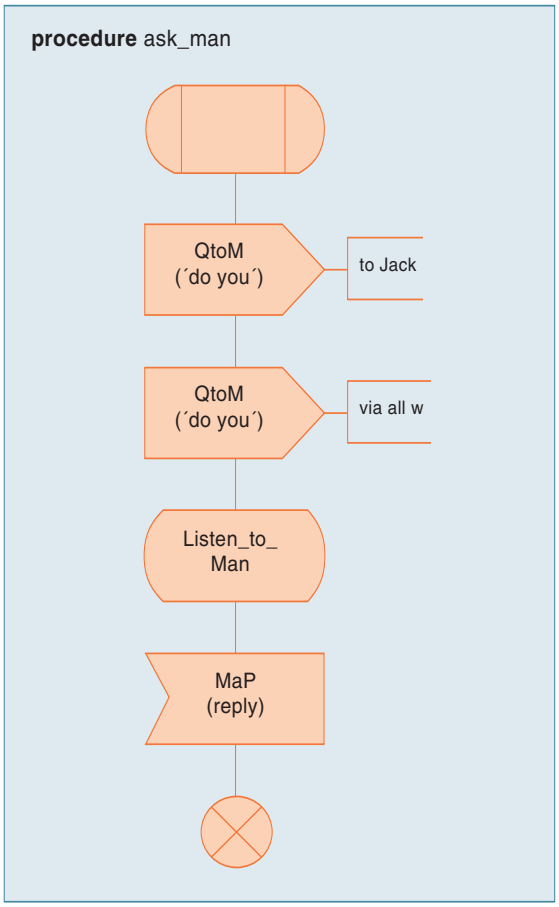
where procedure *ask_man* is invoked.

Then a decision is made depending on the value of *reply* (set by the procedure).

Then a composite state *ask_woman* containing a similar behaviour as the procedure *ask_man* plus the decision. The ask_woman state has two outlet labels *negative* and *postive*.

Signal *Announce* is sent via channel *env*. The text extension symbol is used to give room for more text, here via env.

Next state is state *Ready.*



Procedure start.

Output signal to process instance Jack.

Output signal through channel w to all instances connected to w.

The output signals are provided with the parameter value ‘do you’.

Input signal with parameter value assigned to reply.

Procedure return.

Figure 6 Example procedure diagram

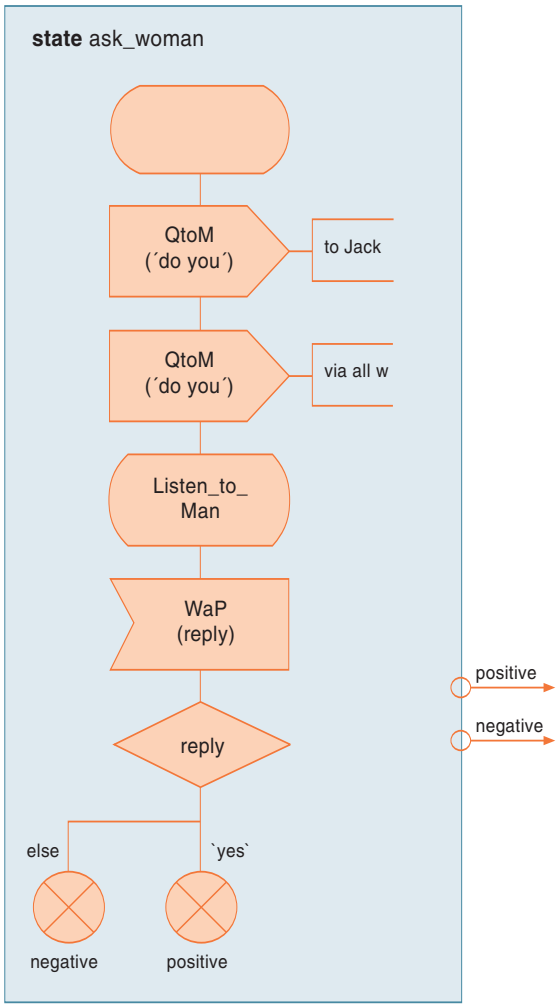
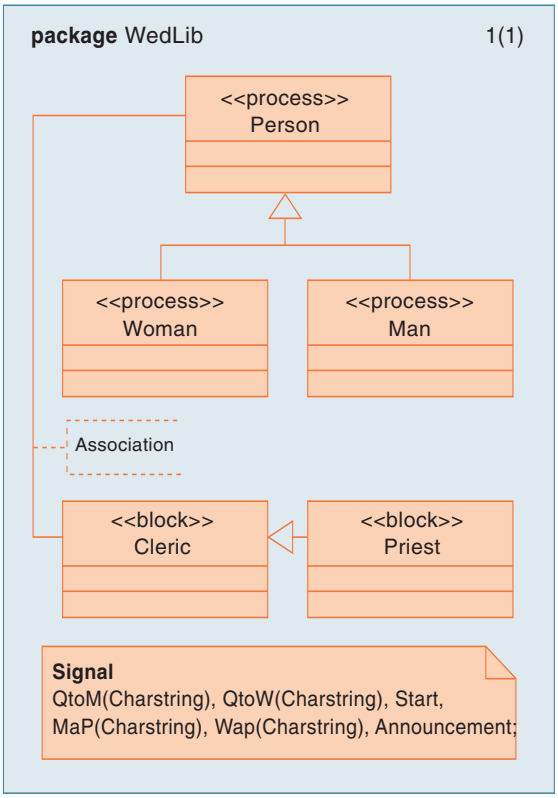


Figure 7 Example composite state diagram

State entry

State exits with labels *negative* or *positive.*



Reference to definition of process type *Person. Person* is stereotyped (by « ») to be of Agent type *Process.*

Process type *Man* is subtype of *Person.*

References to definition of process types *Man and Woman.*

Association between block type *Cleric* and process type *Person.*

Block type *Priest* is subtype of *Cleric.*

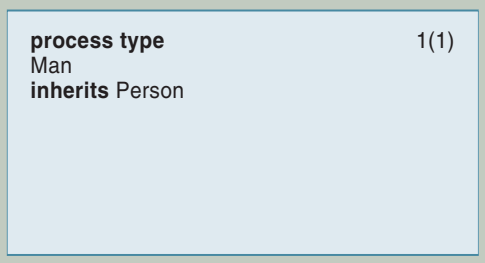
Signal declarations

Figure 8 Example Package diagram

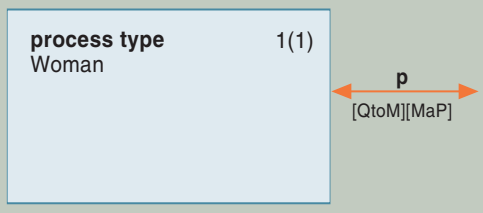
Box 2 Basic SDL

This box provides an overview of a subset of SDL-2000.

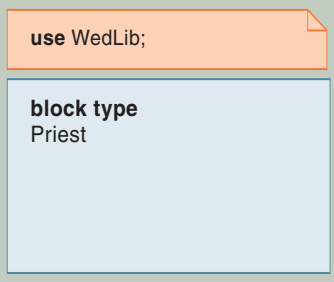
Common Features of SDL Diagrams



Frame symbol. All diagrams have a Frame symbol that represents the boundary of the entity being defined by the diagram. Diagrams have a kind and a name. The following kinds are defined: **system**, **system type**, **block**, **block type**, **process**, **process type**, **procedure**, **state**, **state type**.
Names can contain numbers, capital and small letters, and underline, but not blanks.
Optionally an entity may be defined by means of inheritance from a (super) type. Types defined within a frame symbol of another type may also be defined with a virtuality: **virtual**, **redefined**, **finalized**, e.g. **redefined process type** Man.
Page number and number of pages are indicated in the right hand corner.

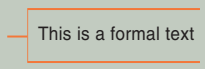


Gate symbols. Gates represent the external connectors of types, and can only be attached to the outside of the Frame symbols for types. Gates may have a name, and specification of signals flowing in each direction. There may be one-way and two-way gates. Attached to the gate may be a constraint on what type of entity may be connected.



Use symbol: Types may be defined in Packages and imported into entity descriptions by means of a use expression inside a text symbol placed on top of the frame symbol where the types are imported.

An informal comment can be associated to any entity.

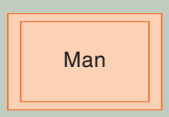


The text extension symbol is used to place formal text outside a symbol.

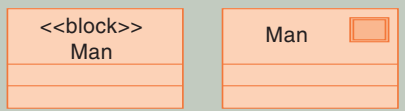
Text symbol. All textual expressions are placed in text symbols: signal definitions; data type definitions; notes; variable declarations; timer declarations. Informal notes are placed between slash asterixes, as follows: /* This is an informal note */.

Types Defined in SDL Agent Diagrams and Package Diagrams

The following symbols are used to represent references to types and partial type definitions in Agent diagrams and Package diagrams. The scope of the type definitions is the diagram where the reference symbol is placed. The actual type definition is provided in a separate diagram. In addition to the symbols shown here, the UML notation described in Box 3 may be usd to provide partial type definitions, and to describe inheritance, associations and dependencies between types.



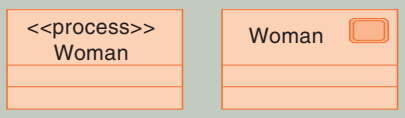
Block type reference symbol.



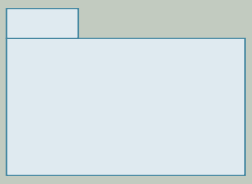
Block type reference symbols with partial type definition. See Box 3.

Box 2 Basic SDL, continued

Process type reference symbol.



Process type reference symbols with partial type definition. See Box 3.



Package symbol. Used to represent packages defined inside packages.

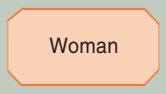
Agent Structures in SDL Agent Diagrams

The following symbols may be used to define the structure of agent (instances) contained in an Agent diagram.
Note that the Agent diagram also may contain type refrences as described in Box 3.

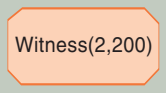
Block set reference symbol, i.e. a Block set containing one initial instance and an unbound upper limit of priests.

Block set symbol with initial and max number of block instances.

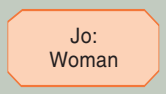
Block set reference of a given block type.



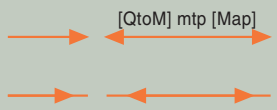
Process set reference symbol, i.e. a Process set containing one initial instance and an unbound upper limit of women.



Process set symbol with initial and max number of process instances.



Process set reference of a given process type.



Channel symbols. Channels represent the connections between objects and are attached to object symbols and/or to the frame. When arrows are at the end, channels are non-delaying. When in the middle, the channels are delaying. Channels may have a name, and specification of signals flowing in each direction. There may be one-way and two-way channels.
Channels have a name and a set of signals specified for each direction.

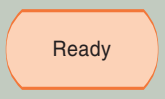
Box 2 Basic SDL, continued

Symbols in SDL State Diagrams

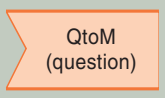
The following symbols may be used to define the behaviour of agents in terms of state machines.

Start symbol.

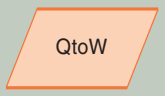
Stop symbol.



State symbol.

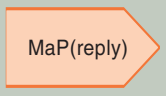


Input symbol. The parameter values (here a Charstring) are assigned to the corresponding variables listed (in the comma separated list) inside the parenthesis (here *question*).

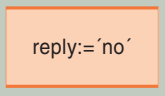


Save symbol. The specified signals will not be consumed, but saved until the next state is reached.

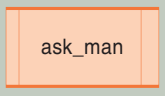
Enabling condition/continuous signal symbol.



Output symbol. The (comma separated list of) values (here the value of *reply*) are assigned to the corresponding signal parameters.



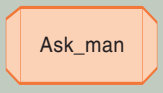
Task symbol. Inside is a comma separated list of assignment or a “program” in the textual action language.



Procedure call symbol. Parameters may be specified as well as a return value.

Decision symbol. The text inside the symbol specifies an expression that has a value, and the following branches are labeled with value ranges. Any number of branches are possible.

Connector symbol. Connector symbols can refer from one page of a state diagram to another. The connector name provides the proper reference.

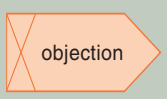


Procedure definition reference symbol. A placeholder for a procedure definition given in a separate diagram.

Create symbol. A new instance is created in the agent set specified inside the symbol.

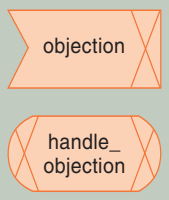
Procedure Start symbol.

Return symbol.



Exception raise symbol.

Box 2 Basic SDL, continued



Exception handle symbol.

Exception handler symbol.

Channels link two agents, or an agent and the diagram frame representing a channel connected to the environment. Channels can be non-delaying (arrows at the ends) or delaying (arrows on the line). Channels can support one-way or two-way asynchronous communication.

Gates terminate channels on instances of Agent types. Arrows outside the frame symbol of the corresponding type indicate gates.

State Diagrams

State diagrams may depict:

- Start symbol;
- State symbols;
- Composite state symbols;
- State types;
- Transitions specifying:
  - Input symbols
  - Save symbols
  - Enabling conditions/continuous signals
  - Output symbols
  - Procedure call symbols
  - Task symbols with
    - Timer operations
    - Expressions on data
  - Decision symbols
  - Create symbols
  - Exception raise symbols
  - Connector symbols
  - Stop symbol
  - Next state.

An example State diagram of a process is depicted in Figure 5.

Procedure and Composite State Diagrams

Procedure diagrams may have the same contents as State diagrams, except that the Start symbol is different, and they have no Stop symbol. Example Procedure diagram and Composite state diagrams are provided in Figures 6 and 7.

Package Diagrams

Package diagrams define types outside the scope of particular Agent diagrams so that the types may be used in any Agent diagram. The types of a package are made available in the definition of an Agent by the Agent diagram having a package use clause. See Figure 4.

In addition, Agent diagrams and Package diagrams may contain a subset of UML for combined use with SDL. See Figure 8 and the next main section.

3 UML Notation in SDL

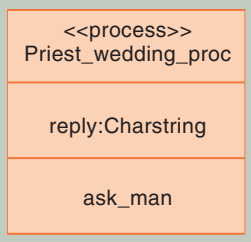
The 'Unified Modeling Language', UML [10, 11, 12] from the Object Management Group, has grown popular to depict many aspects of specifications. However, UML lacks a well-defined semantics (behaviour of its contained constructs). By combining UML with SDL and MSC, this situation can be improved. Previous versions of SDL had no notation to graphically define associations between types. Since UML class diagrams provide a convenient notation for this, part of the UML class diagram notation has been integrated into SDL-2000. (For more on this, see the paper by Birger Møller-Pedersen in this issue.) In this way, SDL-2000 can be used to express facts that were not conveniently expressed in previous versions of SDL. Note, however, that associations are treated as graphical comments in SDL and are not translated into other SDL constructs.

ITU-T Recommendation Z.109 [13, 14] provides a two-way mapping between UML constructs and SDL entities [4, 5, 6, 7, 8, 9]. This allows users to go from pure UML to SDL in a well-defined way. SDL extends UML by providing means to express detailed behaviour with a formal semantics and means to formally define the internal structure of composite entities (using Agent diagrams).

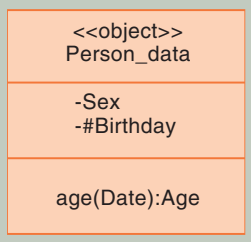
Box 3 UML Notation in SDL

A class (in pure UML) is depicted by a rectangle and has a unique simple name.

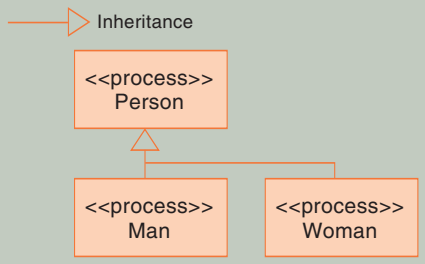
A class (in SDL) may have an attribute compartment and an operation compartment – in that sequence. Attributes in UML are single-valued only. Attribute groups are not supported in UML. The attribute visibility may be +(public to any class), –(private to its class) or #(protected to subclasses only); visibility should not be confused with access control.



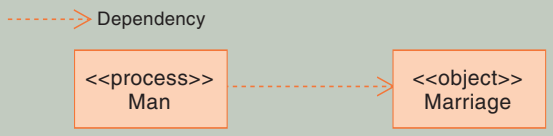
Process type reference symbol with partial type definition.
Local variables may be represented in the attribute compartment and procedures in the operation compartment.



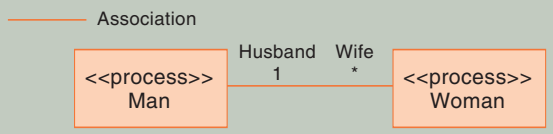
Data type reference with partial type definition.
The internal structure may be shown in the attribute compartment and operations defined for the datatype in the operations compartment. Note that SDL have two kinds of datatypes: <<object>> and <<value>>.



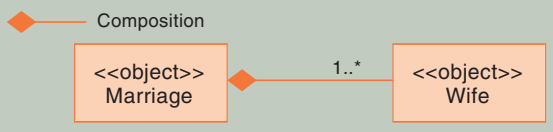
Subclasses may inherit properties of their superclasses by generalisation. Note that the SDL version of UML allows for single inheritance only.



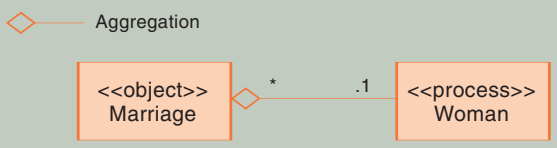
Dependencies between properties of classes can be depicted by a dashed arrow. UML for SDL supports two stereotypes of dependencies, namely <<import>> and <<create>>.



Associations between classes are treated as graphical comments in SDL, and they define that instances of two classes are linked. The association ends can have role names. The cardinality options (in both ends) can be: 1 (exactly one), * (zero or more), (0..1) (zero or one), or (m..n) (numerically specified min and max). In the example a Woman must have one Husband, and a Man can have zero or more Wives.



An object (of a component class) may only be part of one composite object and only exist if the composite object exists. This is in UML called composition.



An object (of a component class) may belong to zero or more instances of aggregate classes. In the example, each Marriage can comprise only one Woman, but a Woman can be engaged in zero or more Marriage-s.

Z.109 does not provide a mapping of every construct in UML to SDL. Also, Z.109 maps only specialised versions of the UML constructs, called stereotypes, to SDL. Hence, the user is safest to express in SDL everything expressible in SDL. UML becomes then a notation for a subset of SDL. This subset is called a profile of UML.

SDL has been extended with UML-like notation elements (notably class symbols and associations) to represent types and associations between types.

UML class symbols and associations can be included in SDL Agent and Package diagrams. They can be both partial type definitions and references to the full type diagrams. The properties defined as part of the class symbols must be consistent with the properties defined in the corresponding SDL types:

- The name compartment of the class symbols contains the type name;
- Class symbols may have attribute compartments; the attributes can have visibility (public, protected or private) and changeability (changeable, frozen or addOnly);
- Class symbols may additionally have an operations compartment: operations represent procedures and their signatures;
- Associations between two classes with min and max cardinality constraints are only interpreted as graphical comments in SDL;
- Roles of classes involved in associations;
- Aggregation as a special kind of association between classes which may be contained in several aggregates are only interpreted as graphical comments in SDL;
- Composition as a restricted form of aggregation;
- Dependency prescribing that (a property of) the dependent class is derived from (a property of) the argument class;
- Generalisation describing inheritance relationships between classes.

Figure 8 provides an example class diagram. Box 3 provides an overview of the SDL UML notation.

4 Comparison of ITU-T Languages with UML

Before embarking on comparing UML with the ITU-T language family, some words are needed on the terminology used in these languages. The text does not contain an evaluation of the languages.

The term ‘modelling’ (in UML) refers to a model of a software system. The term ‘description’ (in SDL) refers to description of a software system. The terms ‘model’ and ‘description’ seems to be synonymous, but must be understood in this context of modelling/describing software only, and should not be confused with other usages of the terms [15]. The use of the term ‘model of’ in model theory of mathematical logic requires that you state a ‘denotation’ mapping between the terms and the sets denoted by the data, e.g. there exists exactly one x, such that William and Bill denote the same x. Also, a similar explicit mapping should be stated between data and “real world” phenomena denoted by the data. The terms ‘model’, ‘description’, ‘denotation’, ‘semantics’ and ‘synonym’ are all synonyms. Note that semantics in programming and specification languages is only concerned with making the behaviour definition unambiguous and not to make the denotation mapping from data to phenomena unique.

In relational mathematics a relationship (set) between two entities (sets) ‘a’ and ‘b’ is denoted by an ordered pair (expression) <a, b> between ‘instances’ (terms) a and b. In relational mathematics a relation is a set of relationships. In UML a relationship associates two classes, and an association is a specialised unordered relationship. Hence, a relationship in UML corresponds to a relation in relational mathematics. In UML a link is an instance of an association – corresponding to a relationship in relational mathematics. Therefore, the terminologies in relational mathematics and UML are not the same. Note also that the relational model (for data bases) is slightly different from relational mathematics [16].

UML could, for example, be used as a graphical notation to define data (structures); however, here the creators of UML themselves give a warning [11, page 111]: “Logical database design is beyond the scope of this book.” UML does not provide means to define the precise structure (e.g. relational database) and formats (e.g. ASN.1) of data.

Aggregation and composition are not well defined in UML. We interpret composition to mean that the name of the component instance

Box 4 The ITU-T Languages Provide a Specialisation of and Parallel to UML

UML	ITU-T languages
Use case diagram; depicts relationships between (users as) actors and their use cases (as tasks or functions). No semantics attached	Not supported. The issue may be addressed in a new Question on User Requirement Notation, URN.
Class diagrams; depict classes and their various relationships, including associations and interfaces. Classes can include attributes, operations and methods **Object diagrams**; depict objects as instances of classes and links as instances of associations. Object diagrams are only illustrations of possible situations and not definition of composite types as in SDL.	Agent diagrams and package diagrams can contain class diagrams that partially define the types. In addition SDL agent diagrams define the composite object structure of agents and agent types. This is not covered by UML. UML classes are specialised (by stereotypes in UML) into **ITU-T SDL**. The stereotypes are <<system>>, <<block>>, <<process>>, <<procedure>>, <<interface>>, <<object>>, <<value>>, <<state>>. UML compositions correspond to SDL decompositions.
Interaction diagrams; can be of the following kinds: **Sequence diagrams**; depict the time-ordering of messages exchanged between objects. **Collaboration diagrams**; depict interactions between objects in an alternative form.	ITU-T MSCs can depict the information in sequence diagrams. MSC support **Simple MSC diagrams**, **High-level MSC diagrams**, and **MSC documents**.
Statechart diagrams; depict optional state machine behaviour associated with classes. **Activity diagrams**; are a special kind of statechart showing the flow between activities only.	**SDL state diagrams** specify (state machine) behaviour associated with agents including creation and deletion of agent instances, data operations and timer operations.
Component diagrams; depict organisation and dependencies between implementation components. Components typically map to classes, interfaces and collaborations. See class diagrams.	**ITU-T ODL** defines objects with multiple interfaces for both operational and stream data.
Deployment diagrams; depict the configuration of processing nodes and the components running in them.	Not supported. The issue may be addressed in a new Question on Deployment and Configuration Language, DCL.
Programming languages are not supported by UML.	**CHILL**; is an object-oriented programming language for real time communicating systems.
Data syntax is not supported by UML.	**ASN.1**; defines data syntax for protocol and other data. ASN.1 is typically used together with SDL.
Testing is not supported by UML.	**TTCN**; defines test cases for protocol testing. TTCN is conveniently used together with MSC.

is functionally dependent on the name of the composed instance, hence, that both names must be given to fully specify the path name to the component instance. This issue is unclear ('bêtes noir' in [10], page 80) in UML, as UML does not address the data formats. In UML a component class (not instance) may belong to more than one composite class, as we interpret as 'alternative name binding' (ref. the GDMO language in Rec. X.722). Note also that from a formal point of view there may be no need to distinguish aggregation from association.

In SDL packages are used for reusable specifications only, while in UML packages may serve as subsystems.

UML is a large language, comprising a large set of diagramming techniques. ITU-T provides a family of languages having parallel, though most often specialised, features compared to UML. Box 4 parallels UML to the ITU-T languages.

Object Constraint Language; OCL defines pre- and post-conditions to state changes by

operations or methods, invariants over state changes and navigation. OCL does not allow any update of the object (instance) model and not state changes. (It is a property language.) ITU-T SDL Abstract Data Types (**ITU-T ADT**) provides an equational logic on data types. SDL-ADT is not comparable to OCL, and OCL is not part of SDL-2000.

From the overview given in Box 4, only class diagrams need to be mapped to SDL. Also, class diagrams provide a(n informal) depiction of data (<<objects>>) in SDL. A profile of class diagrams is therefore included in Recommendation Z.100.

References

1 Haugen, Ø (ed). *Draft revised Recommendation Z.120 – Message Sequence Charts*. Geneva, ITU-T SG10 11/99 TD-115.

2 Haugen, Ø. MSC-2000 Interacting with the future. *Telektronikk*, 96 (4), 54–61, 2000 (this issue).

3 Rudolph, E, Graubmann, P, Grabowski, J. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28 (12), 1629–1641, 1996.

4 Reed, R (ed). *Revised Recommendation Z.100: Languages for telecommunications applications – Specification and Description Language.* Geneva, ITU-T SG10 11/99 TD-79 Rev. 1.

5 Bræk, R. SDL Basics. *Computer Networks and ISDN Systems,* 28 (12), 1585–1602, 1996.

6 Sarma, A. Introduction to SDL-92. *Computer Networks and ISDN Systems*, 28 (12), 1603–1615, 1996.

7 Reed, R. SDL-2000 for new millennium systems. *Telektronikk*, 96 (4), 20–35, 2000 (this issue).

8 Nyeng A. The CCITT Specification and Description Language – SDL. *Telektronikk*, 89 (2/3), 67–70, 1993.

9 Møller-Pedersen, B. SDL-92 as an object oriented notation. *Telektronikk*, 89 (2/3), 71–83, 1993.

10 Booch, G, Rumbaugh, J, Jacobsen, I. *The Unified Modelling Language User Guide*. The Addison-Wesley Object Technology Series, 1999. (ISBN 0-201-57168-4.)

11 Fowler, M, Scott, K. *UML Distilled*. Addison-Wesley, 1997. (ISBN 0-201-32563-2.)

12 Rational Software. *Object Constraint Language Specification*. 2000, November 16 [online] – URL: http://www.software.ibm.com/ad/ocl. ad/97-08-08

13 Møller-Pedersen, B. SDL combined with UML explained. *Telektronikk*, 96 (4), 36–53, 2000 (this issue).

14 ITU-T Q6/10 Rapporteur. *Revised Recommendation Z.100: Languages for Telecommunications Applications – SDL combined with UML.* ITU-T SG10 Geneva 11/99 TD-43 Rev. 2. Geneva, 11–19 November 1999.

15 Meisingset, A. Three Perspectives on Information Systems Architecture. *Telektronikk*, 94 (1), 32–38, 1998.

16 Meisingset, A. *Specification Languages and Environments.* University Studies at Kjeller (UNIK), ver. 3.0, 1991.

SDL-2000 for New Millennium Systems

RICK REED

Rick Reed (53) graduated in electronics at Kent University in 1969. His deep involvement with languages led to him being responsible for the Coral-66 software development facility at GCE for System X. He founded a Software Methods department at GCE which led to the Future Architectures section he was heading when he left to form his own consultancy company, TSE Ltd., in 1991.1988–1993 he technically managed the SPECS project on software methods within the RACE programme. He continued into the ACTS programme as well as consulting on diverse applications of expertise. Recently, all his work has been based on his general experience coupled with his knowledge of SDL. (See also page 173.)

rickreed@tseng.co.uk

SDL is the premier language for specification, design and development of real time systems, and in particular for telecommunication applications. SDL-2000 became the international standard in force in November 1999, replacing the previous version. This paper gives an overview of SDL-2000 and fills the gap between previously published tutorials and the current SDL standard.

1 Introduction to SDL

The success of SDL [1, 2] can be attributed to its graphical presentation form. This makes it easy to understand specifications and designs expressed using SDL. They are good for communication even to anyone that has little knowledge of the language. Another factor is the conceptual suitability of the basis of the language: the notion of an extended finite state machine (EFSM). SDL offers a practical way of specifying systems with several communicating EFSM instances. An SDL system consists of one or more communicating agents. There is one outermost agent: this communicates with the environment. In agents, there is definition of behaviour by EFSM, hierarchical structure with agents containing agents, data variables (owned by agents) of value or reference data types, and communication based on asynchronous message exchange.

When systems are specified or designed (in the rest of this article the verb "specify" should be taken to include design), the usual starting point is some kind of top level picture showing the connection between components of the system and the environment. Such pictures usually take the form of labelled boxes joined by labelled lines. SDL can be used, even at this level, to start turning sketches into a formal system description: the names on boxes become the names of SDL components and the names on lines can become the names of SDL channels or associations. Such descriptions are abstract models of real systems. Of course, as an object oriented language, SDL can also be used bottom-up, based on a set of components, or "middle-out".

The SDL specification for a system is a set of diagrams. Each diagram has one or more presentation "pages", and each page has:

- a frame (often with some information attached on the outside);
- the diagram heading giving the kind and identity of the item described by the diagram in the top left corner;
- the page name and number of pages in the top right corner.

1.1 Simple Structure

A very simple example is shown in Figure 1. This system agent diagram contains two process agents. A channel (→) conveys signals between two agents or between an agent and the environment of a diagram. The signal names are listed in the ▭ symbol near the arrowhead, which gives the direction. Channels can have names, but these are omitted here, as they are

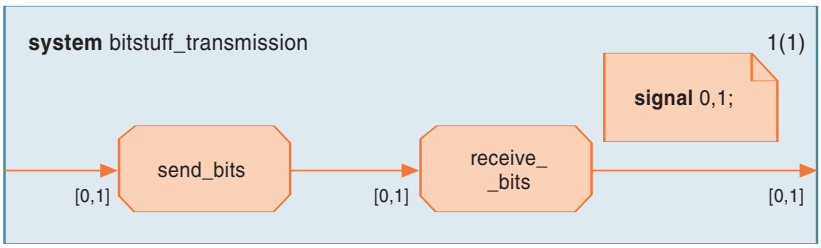


Figure 1 Example simple system model – Bit-stuffing one-way transmission. This system consists of a send-bits transmitter and a receive-bits receiver. The transmitter inserts ("stuffs in") bits so that there are never n bits the same. The receiver removes the inserted bits. This technique is used in real systems to protect against "stuck at zero or one" or (for example in Signalling System 7) to allow flags that consist of n ones or zeros to be inserted without the risk that they are imitated by signals

Names and Underlines

Names consist of letters, digits and underlines; names that only contain digits are allowed. However, an underline character at the end of a line is a continuation and not part of a name.

not needed. A system diagram can contain process agents (symbol), or block agents (symbol). The system itself is the special case of the outermost block agent.

The essential difference between a block (or system) agent and a process agent is that the instances of agents within a block agent behave concurrently and asynchronously with each other, whereas instances within a process are scheduled one at a time. A block agent can contain process agents or block agents. A process agent can only contain other process agents.

As well as containing other agents, agents can contain a state machine, data variables and procedures. An agent SDL diagram is the definition of a set of agent instances. Each agent instance of such a set is created either when the instance containing the set is created or by a create-action in another agent instance. The system agent is created when the system is initialized.

Agent diagrams act as scope units hiding internally defined items. These include the items mentioned above, signals for communication and locally defined types of data. Items defined in enclosing agent diagrams are visible in inner agents. Thus, the signals *0* and *1* are visible inside *send_bits* and *receive_bits*. On the other hand, items defined inside these process agents are not visible at the system level.

The symbols containing the names (and similarly symbols containing names) are links (called "references" in Z.100) to other diagrams considered to be defined where the symbol occurs (**process** *send_bits* is defined in the **system** *bitstuff_transmission*). The defining context and kind of the entity (such as **block**, **process**, and **signal**) is part of entity's identity. Complete identities must be unique, but names need not be unique.

1.2 Simple Behaviour

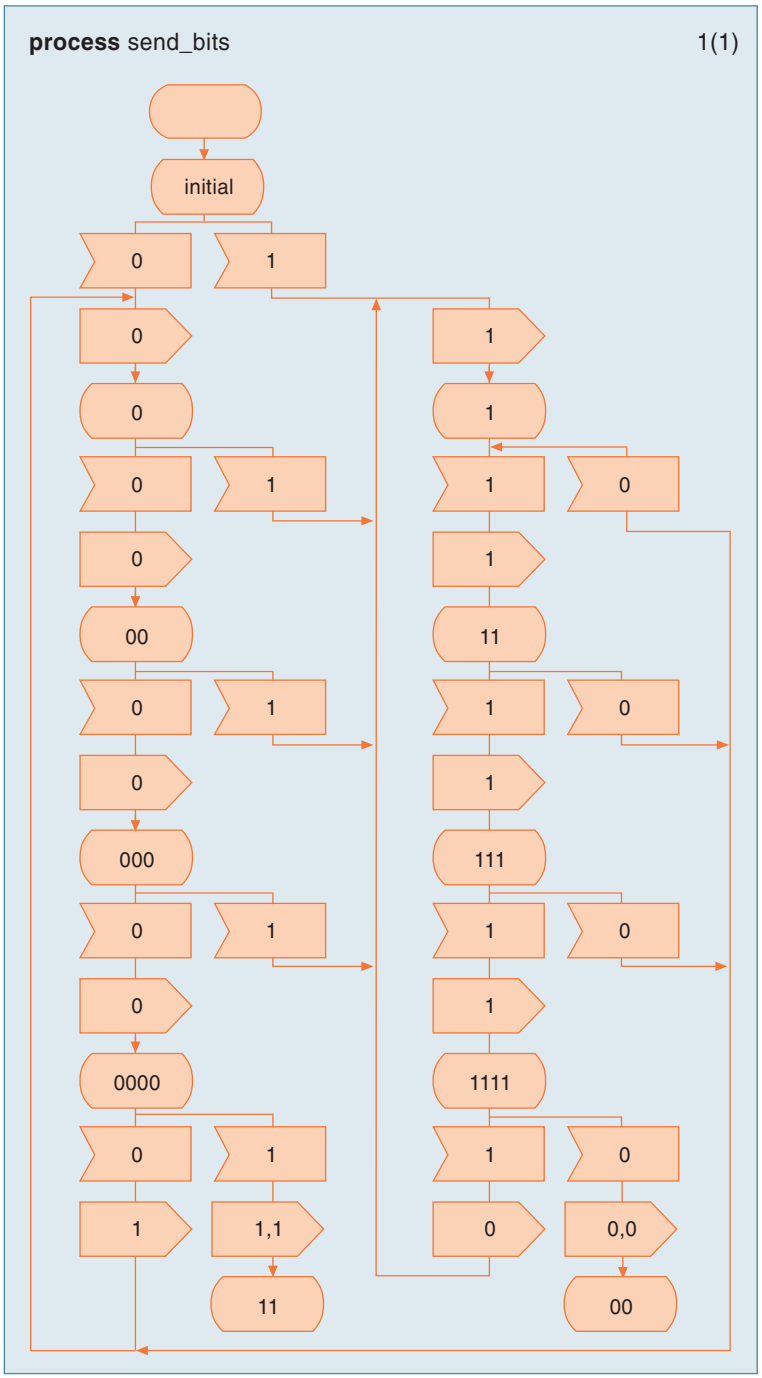
An agent diagram, such as Figure 1, has the possibility to show the interaction between the contained agents, and is called an interaction diagram. An agent that only contains one state machine (typically a **process**) can have the behaviour graph in the agent diagram (otherwise, it has to be linked to a **state** diagram for the state machine graph such as Figure 4.

In Figure 2, the *send_bits* process contains a finite state machine that has:

- a start (symbol) where it starts;
- states (symbols) containing state names: *initial, 0, 00, 000, 0000, 1, 11, 111* and *1111*) with associated inputs (symbols) for the stimulus signals *0* and *1*;
- transitions to the next state with outputs (symbols) for the response signals *0* and *1*.

The response of the state machine is determined by following the flow from state to state in the diagram. The start leads to a state, possibly via

Figure 2 The send bits process as a finite state machine



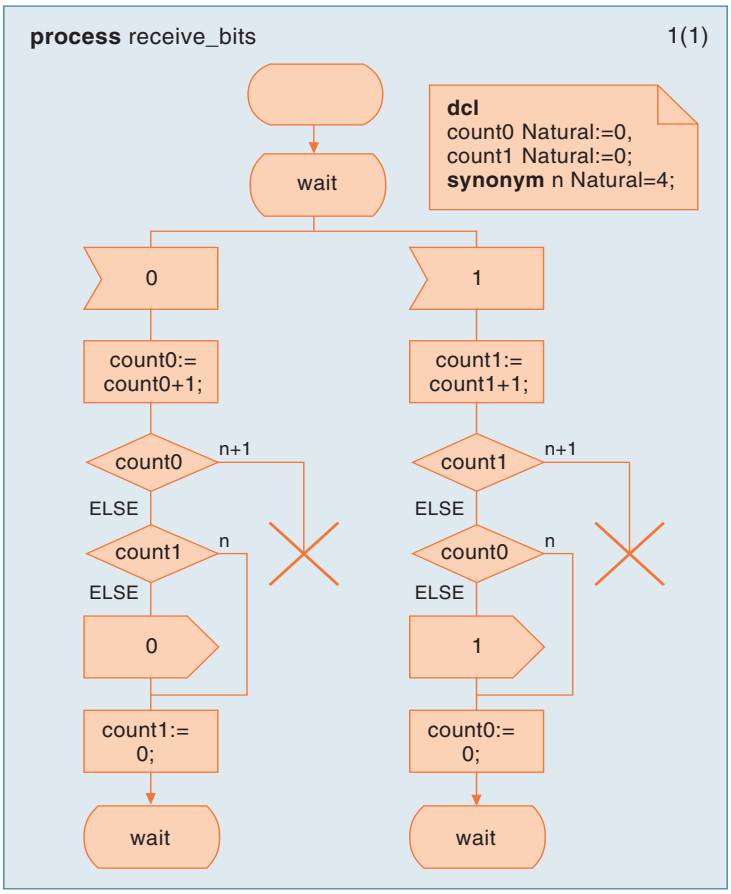


Figure 3 The receive bits process as an extended finite state machine

other symbols. Once at a state, the machine waits until one of the signals that can be consumed in the state is available. This is immediately if the first signal queued in the agent's input port can be consumed, otherwise the machine will wait. Each input leads to other states via other symbols (such as outputs) to the next state.

An output symbol may contain more than one signal (in the example *1*, *1 meaning that two 1 signals are sent*). The next state can be indicated by a ⬭ symbol with the state name (in the example 11 after the output of *1, 1*), which in this case acts as a connector.

SDL extends the finite state machine paradigm in two important ways:

1. Each agent has an input port that queues received signals on a first-in-first-out basis, so that the signals are (normally) processed in the order they are received;

2. Data can be received in signals, stored in variables, manipulated, used in expressions, used to decide how the agent will behave, and passed in output signals.

The *receive bits* process in Figure 3 uses data, and therefore the number of explicit states is reduced to one and the specification allows *n* to be easily changed to any value. The data declaration (**dcl**) introduces two variables, *count0* and *count1*, and a **synonym** relation is defined between *n* and a constant value 4. The *receive bits* process also has:

- Decisions (◇) that can have two or more alternatives, one of which can be **else** – the path taken after a decision is the one labelled with a value that matches the expression in the symbol;

- Tasks (▭) that contain one or more statements – typically assignment statements, but can include textual loops, textual procedure or method calls, textual if, and textual decision statements;

- Text (▱) symbols that are used to contain data definitions, signal definitions and other textual definitions;

- Stops (✕) for terminating the state machine and in this case the process agent.

Note that the stops are unreachable in the example if the *send_bits* process works correctly.

2 Basic Communication and Timers

As seen in the example in the previous section, signals are the primary means of communication between state machines (see 7 for other means). Timers provide a real time element to SDL, and generate associated timer signals.

2.1 Signal Communication

Signals can be defined with or without parameters, and the paths used are shown by the lists attached to channels and gates as shown throughout the figures in this article. An output using the signal name generates an instance of the signal. When a signal instance arrives at the destination agent, it remains in the input port until it is consumed, at which time the instance ceases to exist. On output, parameters of a signal can be given the values of expressions listed in parentheses after the signal name. On input, the

Uniqueness and Qualifiers

A name is usually sufficient to identify an entity, but the full identifier includes a qualifier that gives the context where the entity is defined.

The qualified signal <<**system** bitstuff_transmission>>0 is distinct from the Integer data item called *0* or a signal <<send_bits>>0 (that is, a signal *0* defined in *send_bits*). In practice, these qualified names (<<*context path*>> is a "qualifier") need only be used when necessary, which occurs rarely.

parameters of a signal can be assigned to variables listed in parentheses after the signal name.

When there is more than one path, communication can be directed in the output to specific destinations by a processing identity (Pid) value, an agent name or **via** path. If there is more than one path, an arbitrary one is used.

These values can be stored in variables for use later. In Figure 5, *X* can only take path *c1*, but *Y* can take *g1* or *c1*. *Y* **via** *c1* ensures the signal goes to *p2*. *Y* **to sender** or *Y* **to** kid directs the signal to a specific destination but on either path.

Four Pid expressions are available to each agent for communications:

self an agent's own identity;

parent the agent that created the agent – Null for initial agents;

offspring the most recent agent created by the agent – Null initially or if creation fails because the maximum number of instances already exists;

sender the agent that sent the last signal input – Null before any signal received.

2.2 Timers

An agent can have timers defined. A timer is created by a definition, such as

timer t4 := 10.5;

A timer can be started with a set and cancelled with a reset. When the timer is set it becomes active and will expire when the time specified in the set has been past.

The expression
active (t4) tests if the timer *t4* is active.

set(now+3.2, t4) – sets the timer to 3.2 from the current time.

set(t4) – sets the timer t4 to the duration (optionally) given in the timer definition from the current time, which for *t4* is now+10.5, see Figure 6.

If the timer expires then a signal of the same name (in this case *t4*) is put in the input port of the agent. It is quite usual to have a **reset** (*t4*) before the timer expires in which case it is cancelled, or if the signal is already in the input port, it is removed.

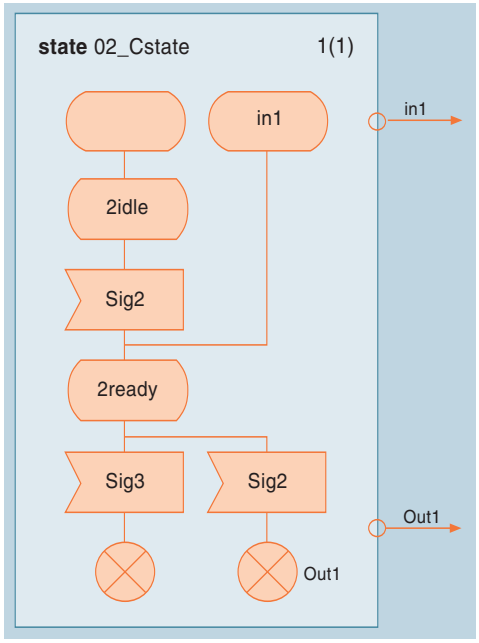


Figure 4 The state 02_Cstate

A typical use of a timer is shown in Figure 6.

Timer definitions are NOT allowed in state diagrams or procedure diagrams (outlined in Sections 6.3 and 6.4 respectively).

3 System Engineering

Although the state machines are essential to specify behaviour (that is, what responses are given to particular stimulus sequences), complex systems often involve several levels of decomposition before state machines are reached. After producing a top-level diagram, the next step is often to determine the various attributes and structures of each component rather than designing state machines. Some would argue that

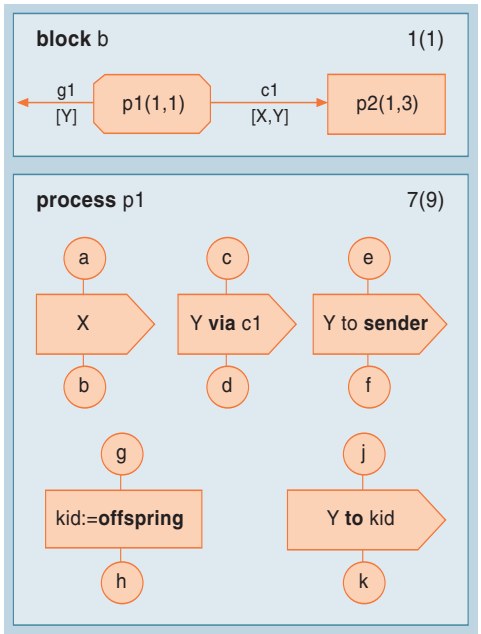
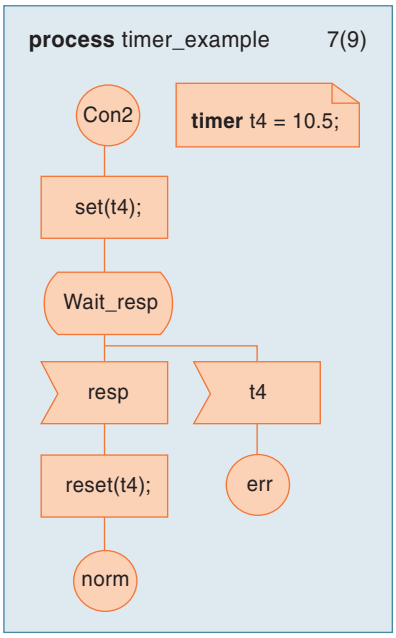


Figure 5 Number of instances; signal directions in output

Figure 6 Typical timer use



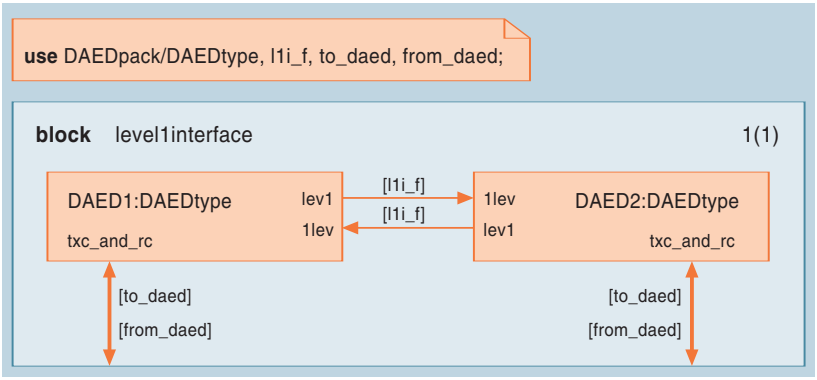


Figure 7 The level 1 interface for Q.703, re-using the same BLOCK for both ends. The communication carried by the channels is defined by attached interface names: l1I-f, to_daed and from_daed

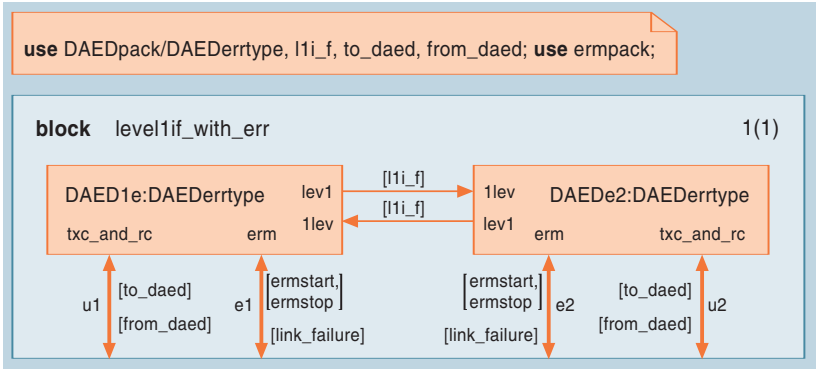


Figure 8 The level 1 interface for Q.703, with error handling

recognising the "objects" in the system, their attributes and the relationships between objects should be the first step.

Rarely are engineers given such a simple case as in Figure 1. More likely the case would be more complex as indicated by the following informal statement: *"The message transfer part of our system has some control transfer functions that interface with link control functions. Link control uses data signalling links defined by the following standards ... Design the Link Control Function (LCF) to support the Message Transfer Part (MTP) with the following characteristics ... LCF is expected to ..."*.

In most cases, engineering involves domain and requirements analysis as well as application specification, design and implementation. For analysis, knowledge and experience are important factors, but natural languages have proved inadequate to complete the task effectively and efficiently [3]. Well-defined notations are needed to provide common understanding of the object and property models and to enable the models to be checked (before too much money is spent).

The essential models for analysis are use scenarios with use sequences (these can be captured in MSC-2000 [4, 5]) and the object model. SDL-2000 uses the same object model notation as UML [6] for this purpose. A feature of engineering is that the diagrams change and evolve and there may be many different versions, even if only one is retained at the end. The final object model can be a traceable evolution of the initial analysis model.

In the rest of this article, an example has been taken from ITU Recommendation Q.703: Signalling System No. 7 – Message Transfer Part – Signalling Link, otherwise known as level 2. Of the several functions of level 2, the signal unit delimitation, alignment and error detection are considered, which interfaces with level 1, the signalling data link. For delimitation, an eight-bit flag 01111110 is inserted into messages after "bit-stuffing" to ensure six ones cannot otherwise occur. On reception, the flags are removed, and the messages "unstuffed".

The initial model of a system would normally be considered a "context model" showing the main objects and interfaces. This is usually the initial version of the final top level specification, which for the example is the SDL diagram in Figure 7, the details of which will be described subsequently.

Analysis of the small part of Q.703 results in the diagrams in Figures 7, 8, 9 and 10 containing:

- an interface *I1i_f* for transmission and reception of *Bits* from level 1;
- two interfaces with the rest of level 2, *to_daed* and *from_daed*;
- two agents *DAED1* and *DAED2* of type *DAEDtype*, each containing agents for:
  1. "delimitation, alignment and error detection (transmission)" *DAEDT*;
  2. "delimitation, alignment and error detection (receiving)" *DAEDR*;
  3. if error handling is included a "signal unit error rate monitor" *SUERM*, see Figure 10.

3.1 Structure and Types

The block *level1interface*, Figure 7, uses the **block type** *DAEDtype* from **package** *DAEDpack* (for packages and their use see Section 3.5). End-to-end signal unit transport has two DAED units connected by level 1. In Figure 7, the type *DAEDtype* is used twice as the basis for *DAED1* and *DAED2*. A diagram that contains types is often called an "object model". For the example, such a diagram corresponding to the analysis for DAEDtype is shown in Figure 14. Note that for illustration in this article, it is assumed that two systems for the level 1 interface are defined: one without and one with error rate monitoring. Therefore, two versions of the *DAEDR* agent are provided in Figure 14. These two different specifications could (for example) be used as the basis for different conformance tests.

DAED1 and *DEAD2* in Figure 7 are linked by the name *DAEDtype* to the diagram in Figure 9, which is linked by the *daedrtype* and *daedttype* in **block type** (▭) or **process type** (⬭) symbols to the diagrams that define these agent types.

The labelled arrows outside the frame in Figure 9 are gates. Channels are connected to these inside the frame, and when the type is used, channels are connected to the gates from outside the relevant symbol. For example, *daedttype* has a gate *txc* that consumes *signal_unit* signals and generates *transmission_request* signals. Interface names could have been used instead of signals.

For a system that consists of a single block or process, enclosing diagrams are not essential, therefore in Figure 7 there are no connections to the channels outside the frame. Normally a gate or a channel name would have to be shown. In general channel and gate names can be omitted from diagrams if there is no need to refer to the channel. No name is needed on the channels inside the frame, as the communication is clear from the interface names given for each direction.

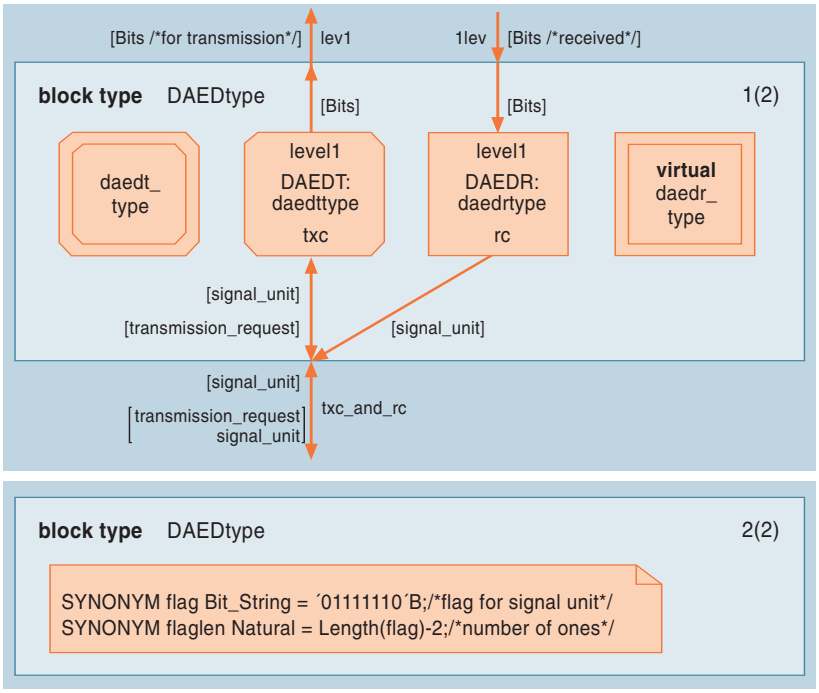


Figure 9 The diagram for DAEDtype consisting of two "pages"

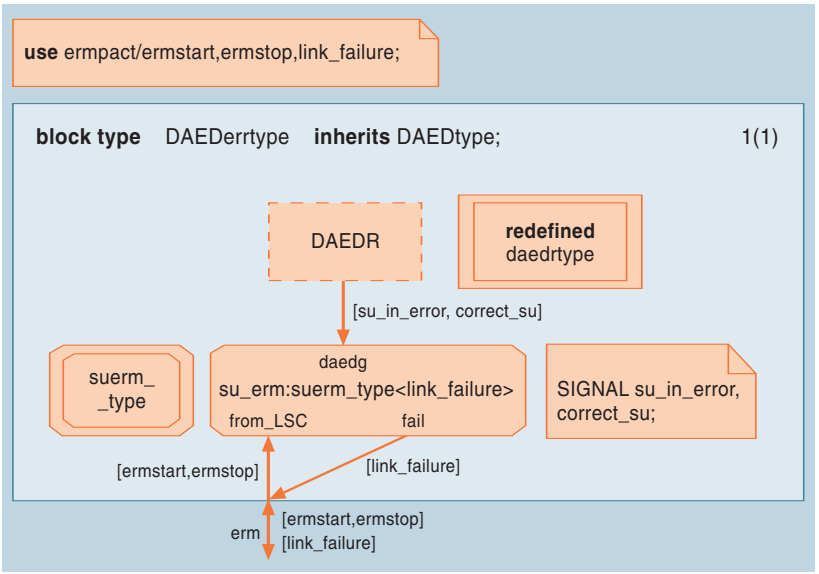


Figure 10 The error handling version inherits the basic version

Even when names are not needed by SDL, it is sometimes useful to put them in. In the alternative version of the system (Figure 8), the channels have been named (*u1,e1,e2,u2*) so that it is possible to distinguish between the two sides.

In simple systems such as Figure 1, the object instances are shown as SDL definitions (such as a **block** or a **process**) that have an implied type definition. If several objects have the same properties, using explicit types makes the SDL simpler.

Figure 11 The virtual block type daedrtype

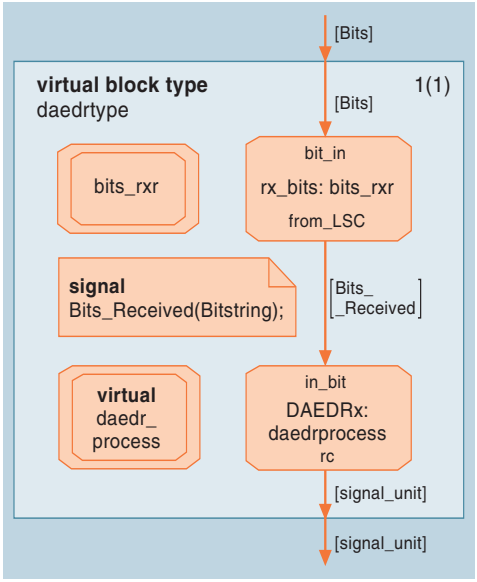
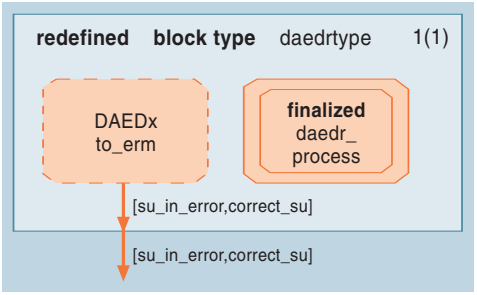


Figure 12 The redefined block type



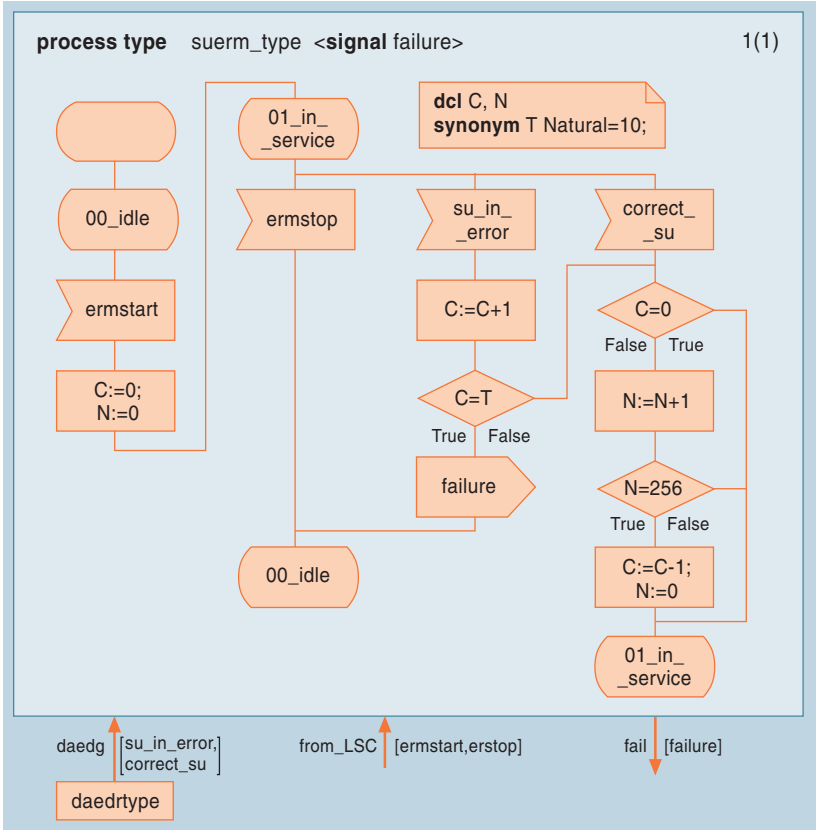


Figure 13 The Q.703 signal unit error rate monitor, adapted with a context parameter

A type definition can be re-used in several places in the SDL specification, and its properties can be inherited to make specialisations of the type. For example in two-way systems, it is quite usual for the transceiver description to be re-used at both ends. In a system with several kinds of termination, a general type of termination can be specialised for each case.

Types have to have fewer context dependencies, so that they can be used in different contexts, and context independence means that types can be used as components in different systems.

3.2 Inheritance and Virtuality

When a type simply inherits from another type, it has the same set of properties as the original type, but a distinct identity. More typically, additional properties are also specified at the same time. For example, *DAEDtype* in Figure 9 is inherited by the *DAEDerrtype*, which handles errors in Figure 10. The extra process agent *su_erm* is added, based on an extra type *suerm_type*.

Inheritance is a general mechanism that applies to interaction diagrams, to behaviour diagrams and to data types. In behaviour diagrams new transitions can be added leading to new states. In data types, new operations can be added.

However, it is not always sufficient to add new properties to a type: it may be necessary to redefine some existing properties. For example, in Figure 10 the additional signals needed are generated by *DAEDR* based on the **redefined** *daedrtype*.

SDL clearly distinguishes those parts that are **virtual** and can be **redefined**. All other parts are inherited unchanged and cannot be changed: these are “finalized”. The fact that the properties defined by the unchanged parts can be relied upon in sub-classes, is a major advantage over languages where any property of a super-class can be changed in a sub-class. A **redefined** item is virtual, and can be **redefined** again if the sub-class (here *DAEDerrtype*) is inherited again. On the other hand, a virtual or redefined item does not have to be redefined in a sub-class, in which case the definition from the super-class is used. When redefinition is given, an item can also be made **finalized**, so that it then cannot be changed in sub-classes.

The agent type *daedrprocess*, defined in Figure 11 is redefined to generate the extra signals (see Figure 12), but otherwise the structure and behaviour of the rest is the same as in the original *daedrtype* in *DAEDtype*. Symbols with dashed lines indicate the use of existing items defined in a super type. The examples here are

the existing process ( ) in Figure 12 and existing block ( ) in Figure 10.

3.3 Context Parameters

Specialisation of types can also be done using context parameters, for which actual parameters must be given before a type is used. As an example, *suerm_type* has been defined (Figure 13) to have a signal parameter for the *failure* signal, so that the actual signal output can be changed. The actual parameter, *link_failure*, is given after the use of *suerm_type* in Figure 10.

Formal context parameters are given in a type definition after the name of the type and enclosed in < and >. The actual parameters are given after the use of the type name enclosed in < and >.

As well as being a **signal**, a context parameter can be a **block**, a **process**, a data variable, a **synonym,** a gate, an interface, a procedure, an exception or timer; or a **type** for a **block** or **process** or data.

3.4 Constraints

Context parameters, virtual types and gates can have constraints. A constraint limits the actual parameters, type redefinition and gate connections (respectively). By default, a virtual type is constrained to be a sub-class of the base type (the one with **virtual**). For example, any redefinition of *daedrtype* in Figure 11 must by default be a sub-class of *daedrtype*. However, it is permitted to specify that the constraint is **at least** some other type, in which case a redefinition can use **inherits** to explicitly inherit another type.

There are no defaults for context parameters, but these can also be constrained by an **at least**. Similarly, gates can normally be connected to any channel that conveys the appropriate signals, but a constraint restricts connections. In Figure 13, gate *daedg* must be connected to a block based on *daedrtype*.

3.5 Packages

A package groups several type definitions together and allows them to be used in several systems. Packages can also be used within other packages, and it is quite usual to have a hierarchy of dependencies between packages, which can be shown diagrammatically (not illustrated here for reasons of space).

Figure 14 supports both the systems defined in Figure 7 and Figure 8. Each **interface** contains the definition of the relevant signals, or links to signal definitions by **use** (see *from_daed*). Interfaces can also include definitions or uses of two other ways of communicating between processes: remote procedures and remote variables (see 7.1 and 7.2).

The three compartment class ( ) symbols are linked to type definitions. The top compartment contains the kind, here **block type** ( ) or **process type** ( ), and identity of the type, such as *DAEDtype*. Where the types are actually defined elsewhere, a qualifier is put before the name: *deadt_type* is defined in <<*DAEDtype*>>. The specialization relation (symbol) indicates that *DAEDerrtype* **inherits** *DAEDtype* so that it includes the properties of *daedt_type*. The **block type** *daedr_type* is also inherited, but is **redefined**, which is possible because the original is **virtual**. The **process type** suerm is added.

The lower two compartments of a class symbol optionally give a definition of some properties of the linked type, so that a reader does not have to refer to another diagram for them. The middle compartment can contain attribute properties, such as the variables if the linked type is an agent type. The lower compartment can contain behaviour properties such as a procedure name and its parameter sorts or a signal used in the inputs of the linked object. In Figure 14, this use of the class symbol is illustrated only for *daedt_type*, which has a variable attribute property, *su_bits* and a procedure behaviour property, *insert_zeros*. During engineering, it would be

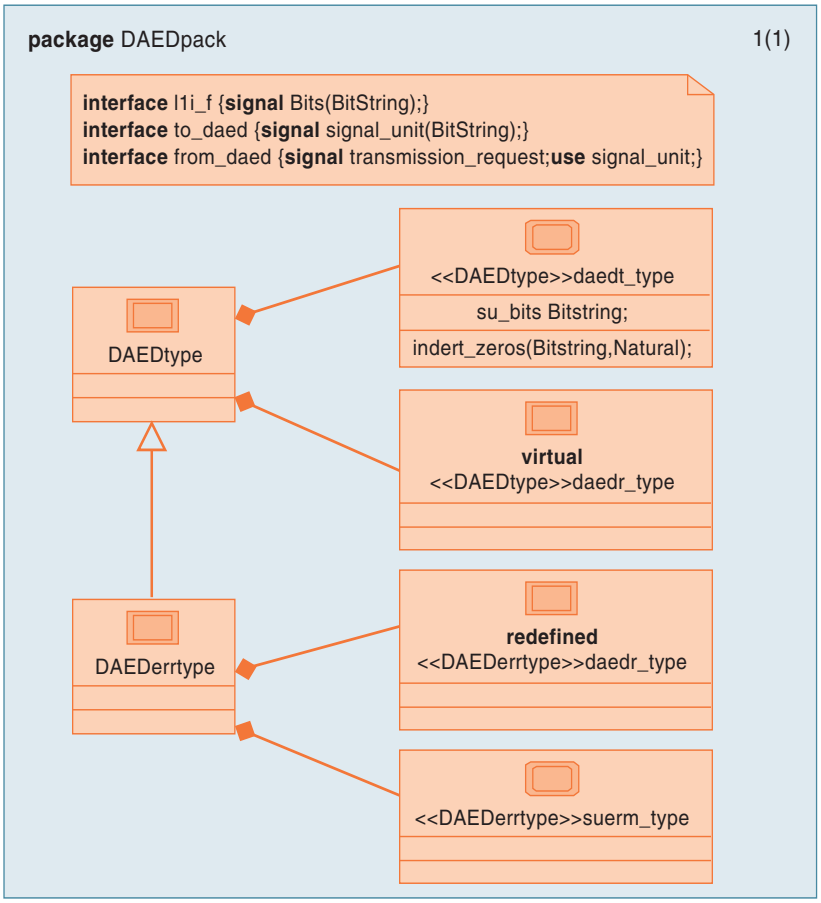


Figure 14 An object model of part of ITU Recommendation Q.703, as an SDL package

quite normal to fill in some properties in the compartments first, and elaborate the linked type later. The real property definition is in the linked type, but tools can assist in copying or checking consistency.

An association (♦—) is a form of annotation – it makes no difference to the SDL meaning if it is removed. However, associations are intended to have meaning in UML, and it is expected that tools will do some checks between associations and the SDL. Associations can be given meaningful names and have attributes at each end (role name, multiplicity range, **ordered**, private or restricted or public visibility). The line can be plain, or with ◆ or ◇ at one end indicating "composition" or "aggregation". A → at either end shows that the end is "bound". The terms "composition", "aggregation" and "bound" are not further defined by SDL.

If no properties are included, the class symbols can be "iconized": that is replaced by the identity inside the type symbol, (such as ▭ for a **block type**). For examples see *daedttype* and *daedrtype*, used in Figure 9 for the process and block *DAEDT* and *DAEDR* respectively.

The text box at the top of Figure 10 makes **use** of the signals *ermstart*, *ermstop* and *link_failure*, all defined in a package, *ermpack*. Also the whole object model is enclosed in a package called *DAEDpack* in Figure 14.

One package named *Predefined*, is an integral part of the language. It defines the data types: Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Vector, Powerset, Duration, Time, Bag, Bit, Bitstring, Octet and Octetstring. Some of these have context parameters that need to have actual parameters to create new data types before they can be used to declare variables (some examples follow).

4 Data

SDL data is strongly typed. A data type can be a **value type** that represents a set of values, or can be an **object type** that represents object references. Each sort of data is distinct. An element of one **value type** cannot be assigned where another **value type** is required. An element of one **object type** cannot be assigned where another **object** type is required that is not a subtype of the first. A **value type** element can be assigned to an **object type** if they are based on the same sort of data: an Integer value can be assigned to an **object** Integer.

value type astring **inherits** String <Natural>;

defines *astring* as a data type that is a string of Natural elements.

value type chlookup **inherits** Array <Character, Integer>;

defines a data type that is mapping for Character values to Integer values.

value type c_array10 **inherits** vector <mystruct, 10>;

defines *c_array10* as a data type indexed with an Integer in the range 1:10 that gives *mystruct* values (where *mystruct* is a defined data type).

package *Predefined* is implicitly part of every SDL model. The example uses Bitstring, which is a string of Bits. Note that Bitstring is indexed from zero to be compatible with ASN.1: all other strings in SDL (including Octetstring) are indexed from 1. Bit values are 0 and 1. Bitstring values can be '0'B, '1'B, '00'B, '01'B etc. or (for example) 'B3'H meaning the same as '10010011'B. The bit '...'B and hexadecimal '...'H notations are also valid for Integer.

The Pid (processing identity) and Any data types are considered as defined in **package** *Predefined*. Pid has a special role in the language for referencing agents or interfaces to agents, and therefore has a Null to indicate no reference. An Any variable can be assigned a value or reference of any other data type, and is therefore fully polymorphic.

Each of the data types defined in *Predefined* has a set of operations. Some of these provide the normal infix notations for Boolean, Integer and Real (such as **and**, **or**, +, –, *, /). Other *Predefined* operations (such as *mkstring*) are operators that use functional prefix notation.

String, Vector and Array based types can be indexed to give an element.

```
dcl a1,a2 c_array10, i Integer;
        /*allows assignments*/
    a2:=a1;              /*the whole array*/
    a1[3]:=a2[i+1];      /*an element*/
```

4.1 User Data Types

For data types beyond simple types such as Integer, user-named types are defined either using Predefined types with parameters (see *astring*, *chlookup*, and *c_array10* above), or by constructing new data types. Constructed data types are enumerated with a list of literals, or a **struct**ure, or **choice** type.

An example of an enumerated list is:

value type rbg {**literals** blue, red=0, green}

and has operators <, <=, >, >=, first, last, succ, pred and num. Each literal must have a unique

number. Literals without numbers are given (left to right) the lowest available Natural number, so blue=1 and green=2.

A **struct**ure has any number of fields, each of which can be any named type including other structures, strings, vectors or arrays.

```
value type S { struct
    a   Integer;
    b   Charstring optional;
    c   Character default 'd';}
dcl s1 S, I Integer, X Character;
    s1:=(.3,'21','e'.);          /*structure value*/
    s1.b:=mkstring(s1.c);        /*field access*/
```

The presence of the **optional** field *b* can be tested by *s1.bPresent*, which gives a Boolean value. A field that has not been assigned a value is undefined unless it has a **default** value.

A choice is similar to a structure, but can only contain one field at any one time, and assigning one of the choices makes all other choices undefined.

```
value type C { choice
      hue       rgb;
      bs        Bitstring;}
```

A named data type can be defined that **inherits** the properties of another data type, and properties can be added including operations. An operation can be either an **operator** or a **method**. An operator has a list of parameters and produces a result. A method acts on a variable of the data type (and may change it), and optionally takes a list of arguments, and may produce a result. An operator uses functional prefix notation: *f(a,b)*, whereas a method uses dot notation: *var.methodname(c,d)*. The body of an operation can be defined using a textual algorithm (for example see Figure 15) or by a linked diagram.

A synonym type (**syntype**) can be defined for any data type that is assignment compatible with the parent type. Though this could be used just to give the type another name, this is usually combined with some limitation of the values of the parent type. Values of a **syntype** can be assigned to the parent type, but only those values defined by the **syntype** can be assigned to a **syntype** variable or parameter. A common use is to limit the range of Integer.

```
syntype Int16 = Integer constants 0:65535;
```

For types that have a Length operator (such as strings), a **syntype** can include a **size** constraint. For example **size**(0,10) means the length must be zero or 10.

4.2 Support for ASN.1

Several of the predefined data types have a direct equivalence in ASN.1 [7]. SDL adds operators to ASN.1 data types, so that the values can be manipulated in expressions. Bit, Bitstring, Octet, and Octetstring were added to specifically support ASN.1.

Other mappings from ASN.1 to SDL are defined in Z.105 [8]. This allows an ASN.1 module to be used with SDL, so that the data types defined in ASN.1 are equivalent to data types defined in SDL.

A value assignment in ASN.1 is mapped to a **synonym**.

```
myvalue INTEGER ::= 100;
            is mapped to
synonym myvalue Integer = 100;
```

A constrained type in ASN.1 is mapped to a **syntype**, so that

```
T ::= INTEGER(1..10)
            is mapped to
syntype T = Integer constants 1:10;
```

An ASN.1 SEQUENCE (or SET – these are treated the same) is mapped to a structure type in SDL, which allows a variable to have a number of fields. For example, the ASN.1

```
S ::= SEQUENCE {
a INTEGER,
b CHARSTRING OPTIONAL,
c CHARACTER DEFAULT 'd' }
```

is mapped to *S* as defined in 4.1 above, and similarly CHOICE is mapped to the SDL **choice**. The corresponding mapping for SEQUENCE values is to omit the field names and convert the value to a structure value.

```
object type Slist {struct
            elem      S;
operators marke(S)->Slist;
methods add(S)
operator make(s S)
            {return (. s, Null.);}
method add(s S) {
            dcl last S;
            for (     last:=this,
                      last.next/=Null,
                      last.next);
            last.next:=make(s);
            }         }
```

Figure 15 An object type for a linked list of S elements

Figure 16 Description of a state machine in a block that creates instances

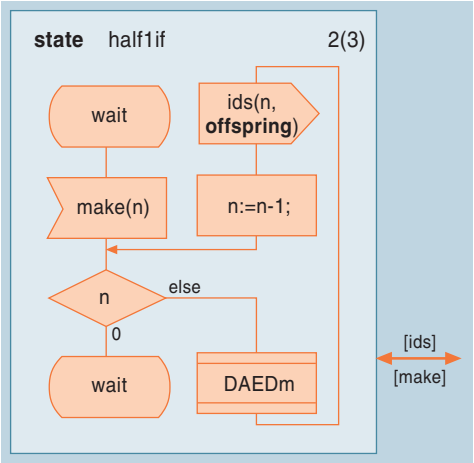
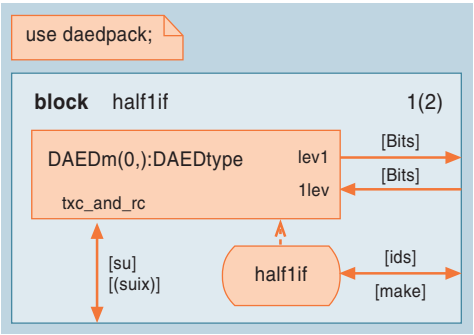


Figure 17 State machine in block that creates instances of an inner block



seqval S ::= { a 22, b 'pqr', c 'x' }
is mapped to
synonym seqval S = (. 22, 'pqr', 'x' .);

SEQUENCE OF, and SET OF, are mapped to the String and Bag data types respectively. ENUMERATED types are mapped to types with **literals**. In addition, Z.105 gives mappings for ASN.1 parameterized types, object classes, objects and object sets.

5 Agent Creation

In most systems, there are multiple instances of various agents, and some agents are created dynamically, particularly when these are realised as software rather than hardware or firmware. The definition of an agent therefore includes how many initial instances of the agent there will be, and the maximum number of instances. The default is one initial instance and no limit on the maximum, and applies if explicit numbers are not given by parentheses after the name. In Figure 5 (also used in 2), **process** p1 is defined to have 1 initial instance and a maximum of 1 instance, and **block** b2 is defined to have a maximum of 3 instances.

Agents can be created by other agents in a create request (symbol) as part of a transition. One instance of the agent definition identified in the request is created each time the request is interpreted. Values can be passed to the agent in parameter variables. A create request can also be used with an agent type, in which case an instance is created as a member of an agent set (implicitly created if one does not exist) of the agent type in the scope surrounding the creator. Creation can be indicated by create line (symbol) originating from the creator and with its arrowhead at the created agent.

In Figure 17 the state machine of **block** half1if creates (multiple) instances of *DAEDm* and Figure 16 shows the create request in the state machine. Note that the state machine of the block is represented by a single state symbol, linked to the state diagram in Figure 16.

6 State Machine Diagrams

The state machine diagrams determine the behaviour of Agents. They define what happens in each state and the transitions between states. States are defined by both the symbol and the attached symbols such as describing the handling of stimuli in the state. Transitions are defined by the symbols between the symbols for states, and the next state symbol. Input is part of a state, not part of a transition, though the signal mentioned triggers one.

6.1 Stimulus Handling

Other symbols that can be attached to a state symbol to describe stimulus handling are:

- save symbol that contains the names of signals that are not consumed in that state;
- 〈 〉 continuous signal symbol that contains a Boolean expression – if there is no signal that can be consumed and the expression is true, the attached transition is triggered;
- immediately followed by 〈 〉 containing a Boolean expression making a signal with an enabling condition – the signal named in is consumed and the attached transition entered, only if the expression is true (the expression cannot depend on the signal parameters);
- containing the keyword **none** indicating a spontaneous event – the attached transition can be entered at any time while waiting in the state.

The save is particularly important, because the channels leading to the state machine determine signals that are valid for all states in the machine. If a signal is not mentioned in any of

the ⟩ for the state, it is implied that it can be consumed and there is an empty transition back to the state. Defining a signal as saved in that state by using ▱ prevents this from happening.

Whether a signal is saved or consumed is defined for each state independently. If the machine enters a state and a signal is saved in that state, all instances of that signal remain in the input port and are not consumed as long as the machine remains in that state. The next transition is triggered by the first signal instance in the input port that can be consumed (that is, not saved and not inhibited by an enabling condition being false). If the triggered transition goes to a new next state, this next state defines the sets of consumed and saved signals.

6.2 Transitions

The components of a transition, such as output (⬭), task (▭) and decision (◇) seen in 1.2, are called actions. Other actions and symbols that can occur within a transition are:

- ▭ procedure call – see 6.4;
- ⊗ return symbol – see 6.4;
- ▭ create request – see 5;
- ⬭ raise exception – see 6.6;
- ○ connector – this contains a label.

When a transition ends in a next state ⬭ that does have any stimulus handling attached, this symbol acts as a connector to the ⬭ symbol that defines the state. Although this means that it is possible to avoid connectors, they are sometimes necessary. Out connectors have arrows pointing to them at the end of the flow lines leading to the connectors. An in connector can only have one flow line leading from it. Logically, connectors are a continuation of flow. Figure 18 has an example of a connector and procedure calls.

Two other symbols are also introduced in Figure 18, though they can be used generally:

- ⊣ text extension symbol – this can be attached to any symbol and allows continuation of the text inside the symbol. So the task containing the comment /**generate flags*/ also logically contains *su_bits:=flag//su_bits//flag;*
- ⋯ comment symbol – contains comment text and can be attached to any symbol, such as the initial *transmission_request* output with the comment *DAEDT -> TXC For first su.*

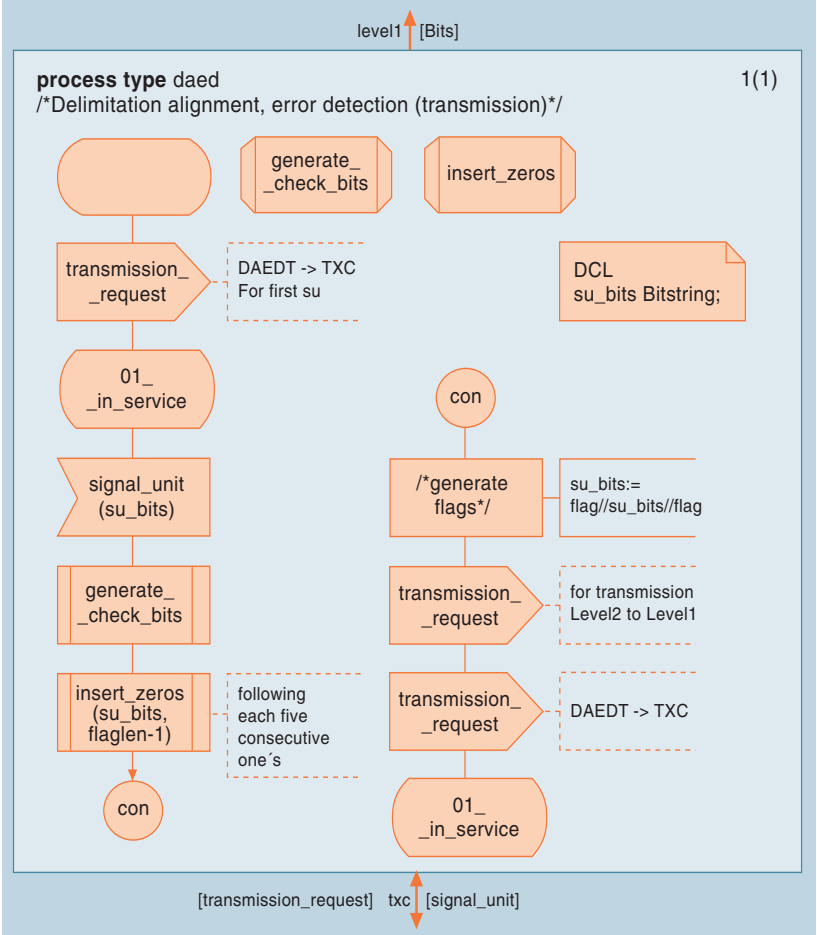


Figure 18 daedttype with connectors and procedure call

6.3 Composite States

In all the above examples, there has been either no explicit state machine or just one. If no explicit state machine is given for the agent, an implicit one exists. If the agent contains other agents, the explicit state machine must be given as in Figure 17 and Figure 16. There can be channels connecting the ⬭ symbol with the other agents and the environment. The ⬭ links to a composite state description that can be either a state aggregation diagram, or a state machine diagram.

The state aggregation (see Figure 19) is similar to an agent diagram containing a number of agents, except that it contains ⬭ links to composite states instead of agents and no channels are allowed. The linked composite state can be considered as a partitioning of the state machine of the agent into state machines that are interpreted in an interleaving manner: only one machine can be in a transition at any one time. When that transition reaches a state node, one of the state machines that can enter a transition is scheduled. If no machine is ready, the agent waits for a stimulus. Each of the aggregated state machines must handle a different set of inputs. An aggregate state only terminates when all the contained states terminate.

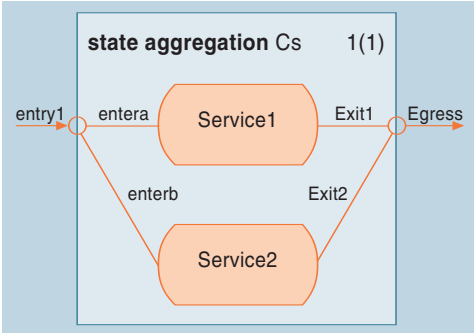


Figure 19 The specification of a machine that is partitioned into two interleaved machines. Cs can be entered without giving an entry point or giving entry1. If no entry point is given both Service1 and Service2 are entered via the start transitions without names. If entered via entry1, Service1 is entered via entera, Service2 via enterb. If Service1 terminates at Exit1 and Service2 terminates at Exit2, Cs will exit via Egress. There can be more than one exit, and if the terminations are inconsistent, an arbitrary one is used. Named entry and exit point are only meaningful if Cs is a composite state in a state machine diagram with named entries and exits. →° entry and °→ exit connection points are optional

A state machine diagram, which is linked from an agent diagram containing other agents, has the same form as an agent state machine diagram. This is the case in Figure 16, and another example (**state** ATM) is given in [9].

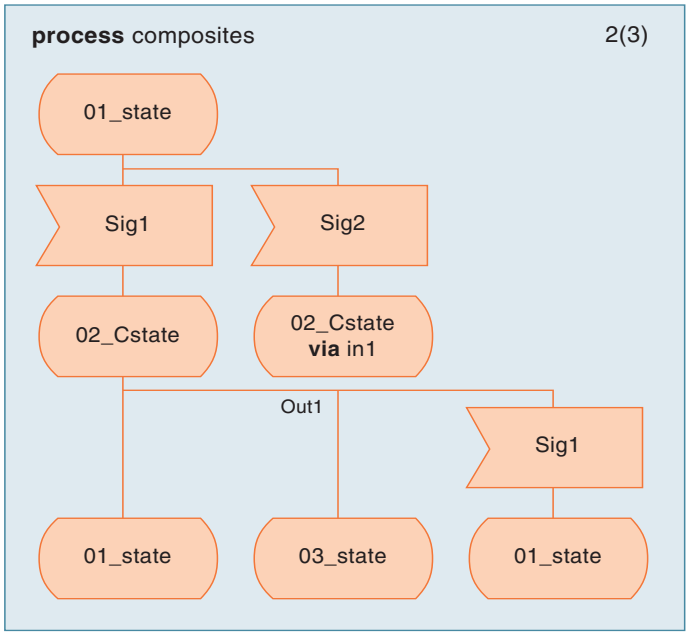
A state machine diagram can also specify composite sub-states of a state in another diagram. Figure 20 and Figure 4 show a simple example. When *02_Cstate* is entered, the process remains in this composite state until either one of the returns is reached, or a *Sig1* is received, which forces the sub-state to terminate. Return via the unlabelled return (⊗) takes the unlabelled transition from the *02_Cstate* in *composites*. Return via the labelled return (⊗) *Out1* leads to the transition to *03_state*. If *02_Cstate* is entered **via** *in1* the start symbol containing *in1* is used.

As well as **state** diagrams, **state types** can also be defined, which allows composite states to be reused many places, like procedures.

6.4 Procedures

Figure 21 shows a procedure diagram. It is similar to a state machine diagram except that it starts with a procedure start (⬭) and ends with a return (⊗). The procedure link (⬭) containing the procedure name shows where it is defined.

Figure 20 The state 02_Cstate is a composite state that can be entered via in1 and has an exit Out1



A procedure is part of a state machine diagram that is separated out and encapsulated, providing a level of abstraction and a component for reuse. Procedures can have dynamic parameters. A procedure can return a result, and such a procedure can be used in an expression. Procedures can contain states and can be recursive.

A procedure is a type. The calls of the procedure are instances of the type. As well as dynamic parameters for variables, procedures can also have context parameters, for example for signals. A procedure definition can **inherit** from another definition or can be **virtual** and **redefined** in sub-types of the enclosing type.

6.5 Textual Algorithms

A task (▭) contains one or more statements separated by semicolons. These statements are not limited to assignments, and can include:

- compound statement;
- **if** or **decision** statement;
- **for** statement;
- **break** and labelled statements;
- procedure **call** (see 6.4);
- **set** or **reset** action (see 2.2);
- **raise** statement (see 6.6);
- **export** action (see 7.2).

There are some occasions when graphical description of an algorithm is not the most appropriate form, though this is clearly a matter of opinion. A long and complex procedure without any states might be better written textually. A statement list can be used in a task symbol, as the body of a compound statement, or as the body of a textual procedure or operation in a text symbol. The last three cases all have the list enclosed in curly brackets { }. The *make* and *add* in Figure 15 are defined in this way.

A compound statement (and textual body of a procedure or operation) can have local variables only used in the statement.

An **if** statement takes the form
if (<Boolean expression>)
    <consequence statement>
    else <alternative statement>;
where the **else** part is optional.

A decision has the form
decision (<expression>){
(<range>): <statement>
(<range>): <statement>
...
}

where <range> specifies the constants for the statement to be interpreted. It is exactly equivalent to a graphical decision and there can be one **else**.

The **for** statement does not have one equivalent graphical construct, and therefore is one benefit of using textual algorithms. The general form is
for (<loop variable assignment>,
    <loop test>,
    <loop variable step>)
<controlled statement>
though, for simplicity, options and some alternatives have been omitted here.

If a statement is preceded by a label, the statement can contain a
    break label
statement, which goes to the label. Note that it is not possible to jump into a statement.

6.6 Exceptions

Some checks can only be made dynamically on SDL models. This causes language defined exceptions to occur:

- OutOfRange, when a value is out of the range for a **syntype**;
- InvalidReference, when there is an attempt to use Null to reference an object or a Pid is used in an output to identify a destination process that cannot receive the signal;
- NoMatchingAnswer, when no answer matches a decision value;
- UndefinedVariable, when trying to get the contents of a variable before it has been assigned a value;
- UndefinedField, when trying to get the contents of a field that is undefined;
- InvalidIndex, when an index is out of range;
- DivisionByZero – division by zero;
- Empty – trying to *take* an element from a set (created using a type derived from Powerset) that has no elements.

Handlers can be provided for these exceptions and for user defined exceptions. If the exception occurs and is not handled locally in a procedure, operation or compound statement that item is terminated, and the exception can be passed to the point of invocation in the caller.

An exception handler can be defined in an agent, agent type, procedure or operation. The exception handler (⌦) symbol contains the name,

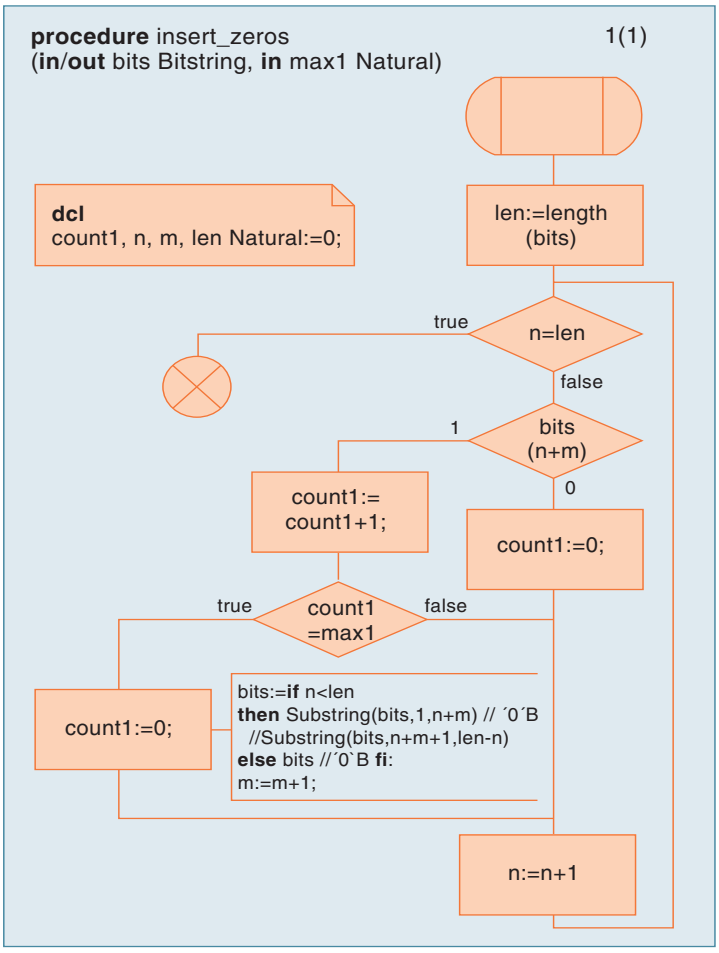


Figure 21 The procedure definition for zero_bits

and handles one or more exceptions whose names are in handle (⋈) symbols attached to an exception handler ⌦ symbol by lines.

An on exception (→) has its arrowhead connected to a ⌦ symbol containing a name. Like a next state symbol, this ⌦ symbol may be a connector to the actual definition of the handler, or may be the head of the handler. The other end of the on exception (→) is either not connected, in which case the handler applies to the whole diagram, or is connected to a specific symbol (such as a start, or state or input) in which case it applies until the end of the transition. An exception attached to an action applies just to that action.

exception e1, e2;

in a text symbol defines user exceptions *e1* and *e2*.

An exception can be explicitly raised by a **raise** (⌧) containing the exception name. This terminates a transition, and no symbol can follow it.

Figure 22 gives an example. *check_bits_correct* has the heading

procedure check_bits_correct -> Boolean;**raise** su_error;

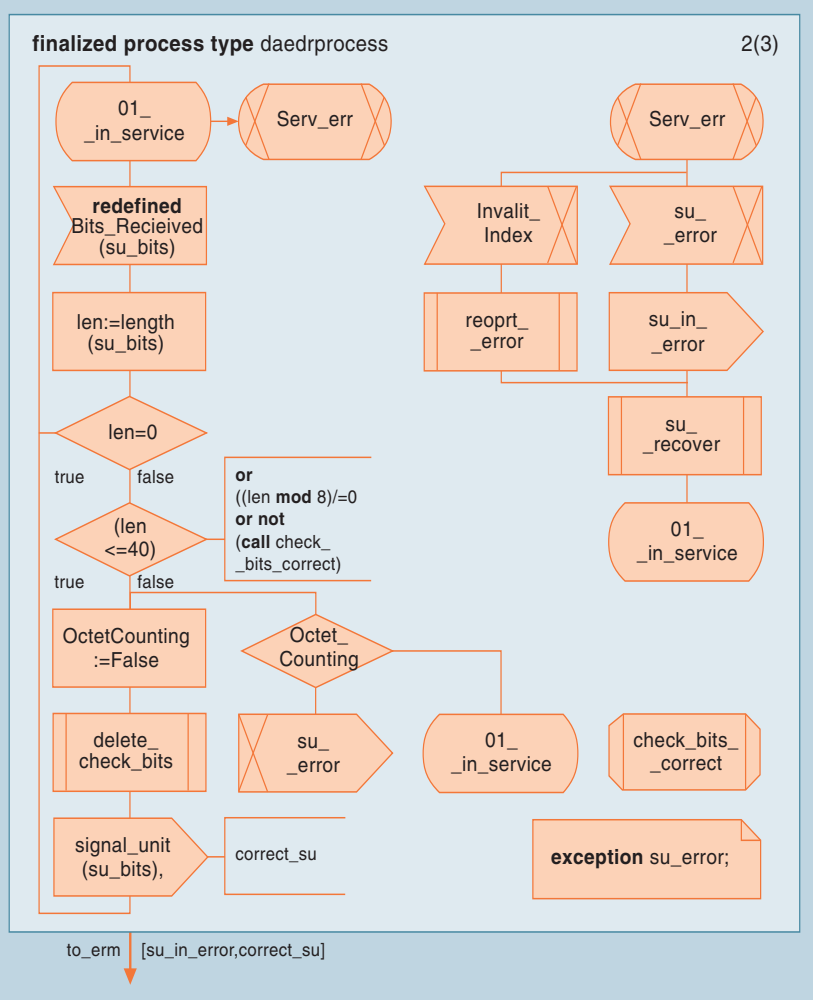


Figure 22 Finalized daedrprocess with exception handling

7 Communications

Communication between agents takes place by signals (see 2.1), remote procedures and variables.

7.1 Communication Using Remote Procedures

One agent can communicate with another agent by a remote procedure mechanism, so that the calling agent waits for a response from the called agent. The agent that offers the communication defines the procedure in the normal way, but with **exported** in the heading, such as

exported procedure rp (**in** x xsort)->rsort;

There are some restrictions on the parameters and return values of remote procedures. In a scope or interface common to both the called and calling agents, a definition is given

remote procedure rp (**in** xsort)->rsort;

The procedure call can be written in the caller in a similar way as any other procedure, but with the added possibility to specify the destination of the call and a timer on the response.

myx:= rp (myx) **to parent timer** trp;

7.2 Communication Using Variables

One agent in a block cannot access the variables of another agent in the same block or any other block. However, there is a notation for exporting the value of a variable from one process to another. If the owning process defines the variable as exported (by **dcl exported** x xsort;), it can have an **export** action that copies the variable value. In a scope or interface common to both the exporter and importer, a definition is given (**remote** x xsort;). The importer can invoke **import** expression to the exporter (myx:= **import** (x);) and obtain the copied value. In this way, the value is safely under the control of the exporter.

A variable of an enclosing agent that has its **dcl** definition directly visible to an enclosed agent, can be read or written by either agent without the need to define a remote variable.

Where the enclosing agent is a process, no special mechanisms are needed to access the variable. This is because the scheduling of state machines within the process is alternating at the transition level, no two state machines can be accessing the variable at the same time.

Where the enclosing agent is a block, the scheduling of state machines within the block can be interleaved at the action level. To ensure safe access to the shared variables of the block, these are accessed by implicit remote procedures of the state machine of the block.

8 Learning More

The SDL-2000 Recommendation is 200 pages of concise information and is probably only suitable as a reference document. Obviously, a short article such as this one cannot be comprehensive. At the time of writing no tools have been released and no books have been published, and as far as the author knows this is the first tutorial style article to be published.

However, beyond the year 2000 the author expects the situation to change, and the best way of getting up-to-date information on SDL is to access http://www.sdl-forum.org.

References

1 ITU-T. *Specification and Description Language (SDL)*. Geneva, 2000. (Z.100 (11/99).)

2 ITU-T. *CCITT Specification and Description Language (SDL).* Geneva, 1994. (Z.100 (03/93).)

3 Bræk, R et al. *TIMe – The Integrated Method version 4.0 – TIMe at a glance.* Trondheim, SINTEF, 1997.

4 ITU-T. *Message Sequence Chart (MSC)*. Geneva, 1999. (Z.120 (11/99).)

5 Haugen, Ø. MSC-2000 : interacting with the future. *Telektronikk*, 96 (4), 54–61 (this issue.)

6 ITU-T. *SDL combined with UML*. Geneva, 2000. (Z.109 (11/99).)

7 Willcock, C. A Tutorial Introduction to ASN.1 97. *Telektronikk*, 96 (4), 62–69 (this issue.)

8 ITU-T. *SDL Combined with ASN.1 (SDL/ASN.1).* Geneva, 1995. (Z.105 (03/95).)

9 Møller-Pedersen, B. SDL Combined with UML. *Telektronikk*, 96 (4), 36–53 (this issue.)

SDL Combined with UML

BIRGER MØLLER-PEDERSEN

Birger Møller-Pedersen (51) joined Ericsson in 1997. He has been working with the standardisation of SDL-2000 within ITU, and is now responsible for the Ericsson engagement in UML 2.0 within OMG. Before joining Ericsson he worked at Telenor R&D with Java and network management. While working at the Norwegian Computing Center, Birger Møller-Pedersen was one of the four designers of the BETA programming language. He was one of the key persons in extending the ITU formal specification language SDL to support object orientation in 1992. His background in object orientation dates back to implementations of Simula, espcially via an intermediate code (S-code) in the spirit of Java byte-code, but long before internet was a reality. Møller-Pedersen is co-author of two books.

Birger.Moller-Pedersen @eto.ercisson.no

The ITU-T Recommendation Z.109, “SDL combined with UML”, defines a one-to-one mapping between a subset of SDL and a specialised subset of UML. With this mapping it is possible to use UML for what UML is good at (multiple views of the same system, informal object models, and property model views) and SDL for what SDL is good at (detailed and formalised object models, especially with respect to execution semantics).

1 Introduction

This is a presentation of the ITU-T Recommendation Z.109, “SDL combined with UML” [1], in a form that is assumed to be somewhat more readable than the Recommendation itself.

Z.109 defines a specialisation of a subset of UML [2] that has a one-to-one mapping to a subset of SDL. The semantics of this specialisation is given by the semantics of the corresponding SDL. This is also the case with the CORBA/IDL profile and any other language specific UML profile. The intention with Z.109 is however not to use the specialised UML *instead* of SDL, but to use SDL *combined* with UML. This means that even though Z.109 is based upon SDL for the semantics, not every concept in SDL has a mapping to UML. As they will be used in combination, there is no reason to make an artificial and complex mapping for concepts that are better supported by SDL and for which UML is not the right notation. For example, SDL has support for detailed specification of the object structure of systems and for detailed specification of behaviour, while UML is not meant to have this kind of support.

This presentation introduces a UML model of the most important concepts from the SDL subset. This model is not used to define the mapping in Z.109, but it may form the basis for the definition of a profile solely based upon the *concepts* of SDL, i.e. without requiring the language SDL as such. This UML model of SDL is intended for readers familiar with UML and plays the same role as the meta (UML) model that forms the basis for the definition of UML. Readers familiar with SDL may read this model as an alternative to the abstract syntax of SDL, but knowledge about the abstract syntax of SDL is not needed in order to read the rest of the document.

With the understanding of SDL in terms of the UML model, it should be possible for designers to make UML models without knowing the detailed mapping and the detailed semantics of SDL, and still use the UML in a way that lends itself to a mapping to SDL for detailed design.

2 Overview

UML/SDL Coverage

The main difference between SDL and UML is *not* that UML is especially well suited for analysis and SDL especially well suited for design. With support for associations, SDL may be used for making analysis object models in terms of classes and associations, as well as for design.

The main differences between UML and SDL are that

- UML is a collection of concepts and notations for *several views* of the same system: e.g. Object-, State Machine-, Use Case-, Collaboration and Interaction views;
- SDL is a language (with concepts, abstract grammar and graphical/textual grammars) focussing on the Object- and State Machine views of a system. For these views, SDL is however a complete language with static and dynamic semantics and with concrete syntax (graphical/textual) for the specification of actions. Users of SDL rely on other languages like MSC for specification of interactions between instances;
- UML has a weak semantics with many variation points, while SDL has a complete semantics, including execution semantics for state machines.

These differences are the reasons for Z.109. They are illustrated in Figure 1, which also introduces the following terms:

- UML_{SDL}: the specialised subset of UML with a mapping to SDL according to Z.109;
- SDL_{UML}: the corresponding subset of SDL.

With the mapping defined in Z.109 it is possible for SDL users to use not only MSC for interaction modelling, but also to use UML for Use Case and Collaboration modelling. SDL users may also use the Object and Statemachine mod-

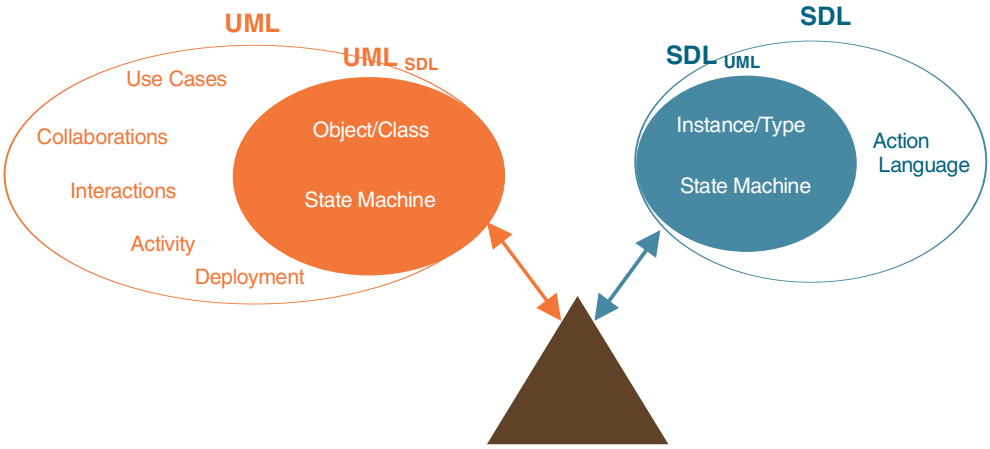


Figure 1 Integrated tool provides both views

elling of UML at stages where the detailed semantics is not determined, and then turn to SDL when detailed specification is needed. With Z.109 it is also possible for UML users to use SDL for more precise models, including the specification of actions.

Z.109 implies no sequence in the use of UML and SDL. As indicated in Figure 1 a tool supporting Z.109 should be able to provide both the SDL- and the UML view of the subsets covered by Z.109.

Z.109 provides a mapping between the UML meta-model and the (abstract) grammar of SDL. For the notation in UML_{SDL}, the notation defined in SDL (Z.100) can be used, where this is appropriate. Otherwise the UML Notation Guide applies. One example of a difference between the notation defined by SDL and the UML Notation Guide is the notation for tagged values. While UML, and thereby UML_{SDL}, have tagged values enclosed by { }, SDL uses keywords preceding the type names.

Presentation Structure

The rest of this presentation is a description of the main concepts of SDL and their representation in UML according to Z.109. It is based upon an example, an Automatic Teller Machine (ATM), in order to illustrate the use of Z.109 for the combined use of SDL and UML. The example is only intended to give an idea on how the mapping between UML and SDL may work and does not claim to cover all details of the mapping.

The presentation requires a detailed knowledge of UML – on the other hand there is no reason to use Z.109 without a fairly good knowledge of UML. The whole idea of Z.109 is to enable users of SDL or users of UML to take advantage of the combined use of UML and SDL.

For each of the main SDL concepts the following is described:

- A short textual description that describes the *SDL concept*;

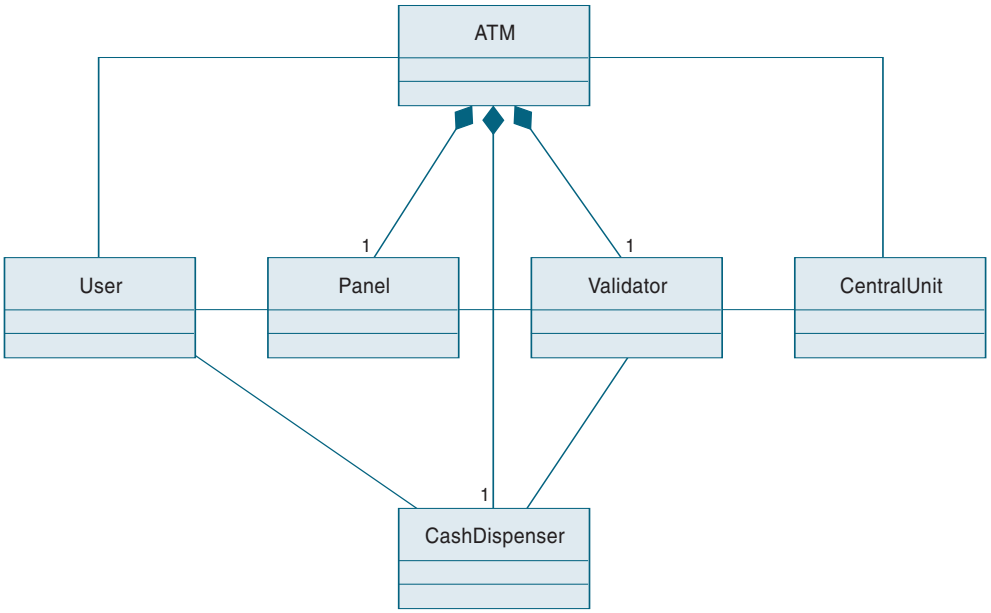


Figure 2 Analysis object model in UML – Class Diagram

UML Profile: The Mechanisms for Defining a Specialisation of UML

The notion of a UML profile is not yet well defined within OMG, but the following is taken from the RFP (Request For Proposal) requiring a definition of a profile:

"Definition of a profile:

- In general, a UML profile is a mutually consistent set of predefined specifications that collectively customize UML for a specific domain or purpose (e.g. a "unified process" profile). The specifications in a profile typically consist of UML stereotypes, tagged values, constraints, notational elements (icons, diagrams, etc.), and other possible specifications. By definition, a profile does not extend UML by adding any new basic concepts and fully conforms to the semantics of the general UML standard.

- More precisely, a UML profile is defined as a specification that does one or more of the following:

  - Identifies a subset of the UML meta-model (which may be the entire UML meta-model);

  - Specifies "well-formedness rules" beyond those specified by the identified subset of the UML meta-model. "Well-formedness rule" is a term used in the normative UML meta-model specification (ad/97-08-04) to describe a set of constraints written in UML's Object Constraint Language (OCL) that contributes to the definition of a meta-model element;

  - Specifies "standard elements" beyond those specified by the identified subset of the UML meta-model. "Standard element" is a term used in the UML meta-model specification to describe a standard instance of a UML stereotype, tagged value or constraint;

  - Specifies semantics, expressed in natural language or in any appropriate language, beyond those specified by the identified subset of the UML meta-model;

  - Specifies common model elements (i.e. instances of UML constructs), expressed in terms of the profile.

- The above definition is taken verbatim from the UML 1.3 specification with one important exception: the ability to extend the UML meta-model with new meta-types. The latter capability may be included in a more comprehensive future definition of a profile, but is out of the scope of the current profile."

Z.109 follows this definition, with the exception of the definition of the semantics. This is not described as part of the text, but obtained by the mapping to the SDL subset.

- *Conceptual UML model:* the SDL concept in terms of UML, like a meta-model for SDL;

- *UML Mapping:* a specification of how the various semantic elements of the SDL concept are mapped to the UML meta-model. UML meta-model elements are written in Italics, while the SDL terms are capitalized.

  - *Base class:* the name of the class from the UML meta-model to which the SDL concept is mapped, with stereotype or without stereotype;

  - *Tags:* the set of tagged values that every model element bearing the stereotype must have.

- *Constraints:* the additional constraints (relative to the base class) that apply.

The example does not cover the whole mapping of Z.109, nor does it cover a typical full-fledged use of Z.109: the use of the parts of UML that are not mapped. A typical use would be to make Use Case models and overall Collaboration models in UML, and turn to SDL when it comes to (detailed) Object and Statemachine models. It would also be natural to use UML Deployment models during implementation design. The use of Sequence Diagrams or MSC is not covered either.

3 Agents and Agent Types

An (analysis) object model of an ATM will typically include a class diagram with classes and associations and one or more collaboration diagrams showing the interaction between the involved objects for selected scenarios. In Figure 2 we have identified that an ATM consists of objects that model the panel, the validator and the cash dispenser, and we have specified the associations between the classes. The object model includes associations between ATM and User and between ATM and CentralUnit because this is the only way to specify that they do not only have associations to parts of the ATM, but also to the ATM as such.

For the purpose of this presentation, we have only included one collaboration diagram and just used the structural part of the collaboration to specify which instances will be linked (Figure 3). A full collaboration diagram may include the specification of the interactions between the objects. Note that the ATM instance is not present in the collaboration diagram. It could be, but it would just be an object similar to the other objects; it is not possible to specify as part of the collaboration diagram that the ATM object is composed of the Panel-, Validator-, and Cash-Dispenser objects.

Given the UML Object model in Figures 2 and 3, there is no unique SDL specification. As soon as the UML is elaborated to conform to UML_{SDL}, the UML model is a partial specification of a potentially more detailed SDL specification. In order to elaborate the UML model, we have to know what the concepts of UML_{SDL} are and how they are expressed in the specialised UML.

SDL: Agents

An SDL System consists of Agents that are connected by means of Channels. Agents may communicate by sending Signals or by requesting other Agents to perform Procedures.

An Agent may have both a StateMachine and an internal structure of Agents (a composite Agent). The internal Agents and the StateMachine are connected by Channels. The connection points for Channels are Gates.

Conceptual UML Model. Figure 4 gives a UML conceptual model of this part of SDL. An SDL specification may specify both singular Agents and types of Agents. Types correspond to classes in UML. Because UML only prescribes classes of objects (and not singular objects), Figure 4 only gives the UML model of Agent types. UML supports the notion of object diagram, but that is only for describing snapshots of UML run time objects and not for the prescriptions of object structures in the body of types.

Agents come in different *kinds* with different execution semantics: Block Agents are *concurrent* Agents with possibly interleaved execution of the transitions of the state machines, while Process Agents are *alternating* Agents with run-to-completion execution of transitions. The overall system is a special System Block Agent.

UML Mapping. An Agent type maps to a *Class* of active *Objects*, with constraints as described below. System, Block and Process agent types are mapped to classes with stereotypes “system”, “block” and “process”, respectively.

Part of the structural content of a composite Agent (in terms of Agents connected by means of Channels) is mapped to a combination of *Composition* and the structural part of a *Collaboration*. The *representedClassifier* of the *Collaboration* is the *Class* representing the composite Agent type. Channels between sub-agents are represented by *AssociationRoles* in the *Collaboration*. The *ClassifierRoles* of the *Collaboration* represent the types of the sub-agents. An eventual stateMachine of the composite Agent is not (or rather cannot be) represented as part of the *Collaboration*.

Types defined locally to an Agent type are mapped to *Classes* in the *Namespace* of the *Class* representing the enclosing Agent type.

Base Class for stereotypes “system”, “block” and “process”

- Class for Agent type without internal Agents
- Class with associated Collaboration for a composite Agent type.

(Stereotype) Constraints

- an agent is an active object;
- an agent can have at most one state machine;
- an agent can have operations that are public, protected or private, i.e. no constraints;
- an agent may have attributes that are ordinary attributes in addition to gates, channels, or sub-agents;
- attributes may be public, protected or private, sub-agents are private, and gates are public.

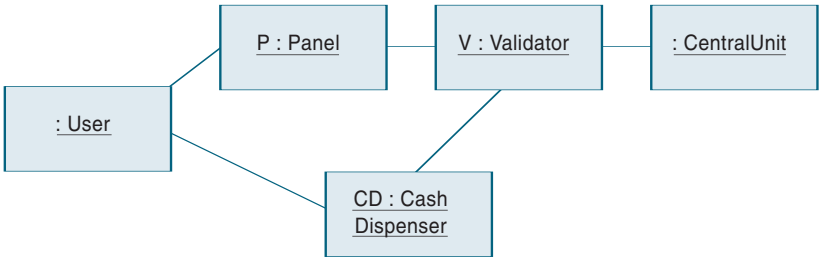


Figure 3 Structural part of a Collaboration Diagram

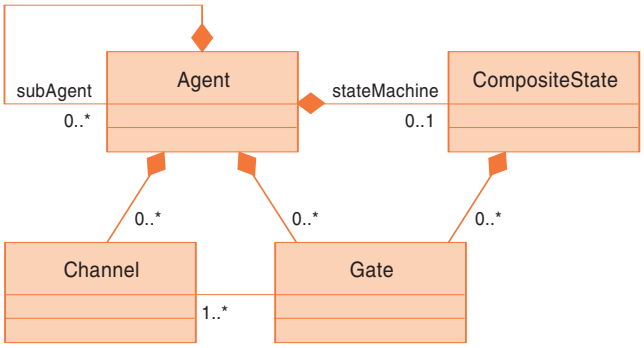


Figure 4 A UML conceptual model of the basic SDL modelling concepts and relationships

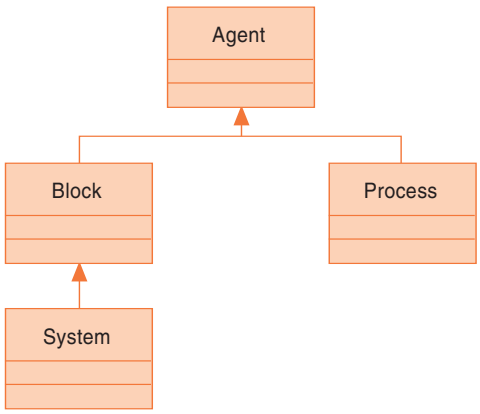


Figure 5 Kinds of Agents

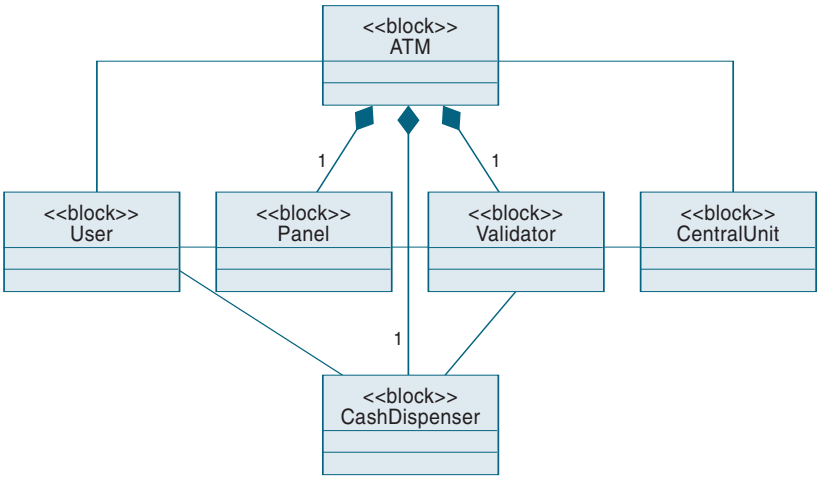


Figure 6 Analysis model in UML_{SDL}

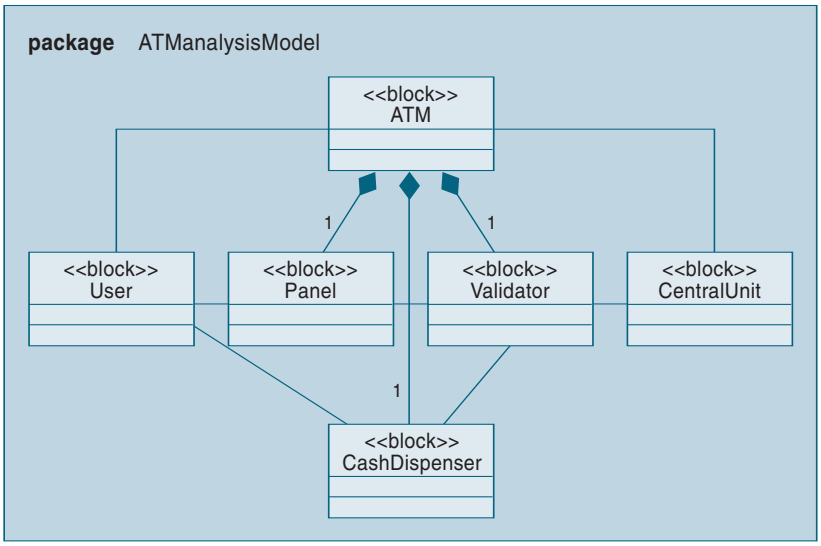


Figure 7 Type references in an SDL package diagram

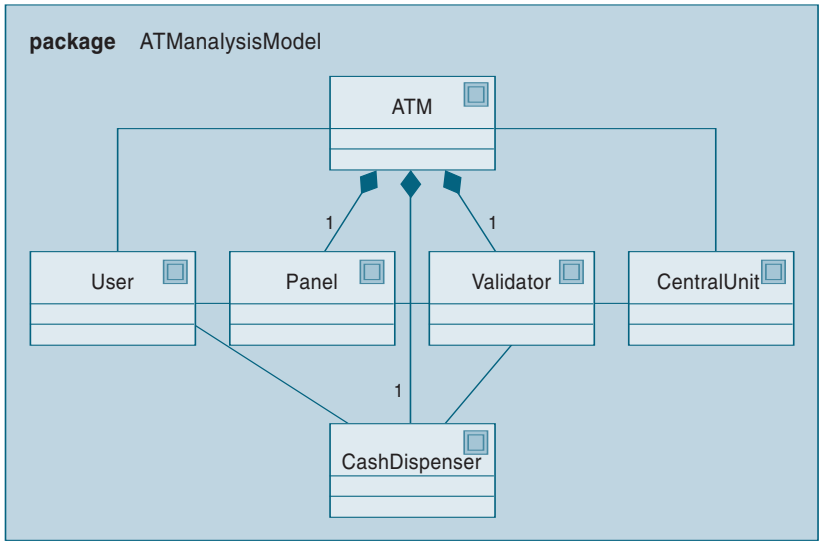


Figure 8 Alternative type references with icons

With this introduction to the notion of agents in SDL and how they are mapped to UML_{SDL}, we can now elaborate the general UML model of the ATM. We decide that all involved classes shall be classes of block agents, because they execute concurrently with each other. This is reflected in Figure 6, where the classes have been stereotyped with “block”.

A class diagram as in Figure 6 implicitly defines a set of classes in a package (or in the name-space of another class). This is according to the definition of UML, and a UML tool will normally provide this information in a browser. The corresponding SDL specification, see Figure 7, includes the information of the enclosing package graphically. SDL has a package diagram that contains class symbols for the types that are defined in that package. As associations are supported by SDL and as class symbols represent type definitions, there is a one-to-one mapping between the two specifications.

Icons can be used as an alternative to stereotypes, see Figure 8. The icons for the different kinds of agents are defined in Z.100.

In addition to the package diagram with type references and associations, the SDL specification includes a type diagram for each of the types, see Figure 9. The package diagram with the class symbols only tells that there is a number of types defined in the scope of the package, while the detailed specifications of the types are given in the separate type diagrams.

As demonstrated so far, UML can be used in situations where the object kinds have not been decided yet. Figure 2 only specifies that some kind of ATM objects will be composed of objects of other classes, but not whether these are concurrent objects or not.

As illustrated in Figure 7 and Figure 8, a class symbol represents a type. If the name of the class has no qualifier, then the class symbol specifies that a type is defined in the scope defined by the enclosing diagram, and that the complete type definition is given in a (referenced) separate, complete type diagram, as those found in Figure 9. If the name contains a qualifier (a path expression denoting some other scope), then the class symbol just represents an application of a type. The type is then defined in the scope unit denoted by the qualifier.

A class symbol is, however, not just an indication that there is a type being defined (in a separate diagram), but also a *partial type specification*. Specification of an attribute in a class symbol, see Figure 10, implies a corresponding spec-

ification of a variable in the corresponding SDL type diagram as in Figure 11.

4 Associations

The example includes both ordinary associations and compositions. This presentation does not make any effort to convey the details on the use of the many properties that associations can have.

SDL – Associations

Types can be associated. Associations are defined at the level of the types being involved. Associations are properties of the enclosing entity. Associations have no implied semantics for the types involved.

UML Mapping. The notion of associations in SDL is a strict subset of associations in UML, so the mapping between SDL and UML is straightforward. As we shall see below, associations are also used to represent two special concepts of SDL:

- Internal structure of Agents, represented by a combination of *Composition* and *Collaboration* in UML_{SDL};
- Gate with endpoint constraint, i.e. a gate that can only be connected with instances of a certain type, represented by a stereotyped *Association*.

5 Internal Structure of Agents

Assume that the next step is to model the details of block type ATM. In Figure 2, Figure 6 and Figure 7 it is only specified that instances of classes may be linked (as the classes are associated), and it is specified that ATM instances will contain instances of Panel, CashDispenser and Validator (by composition). Figure 3 says that the instances of Panel, CashDispenser and Validator collaborate, and it also tells which of these collaborate with the user and with the central unit.

The composition in Figure 6 can be mapped to the corresponding composition in SDL (Figure 7 and Figure 8), but it can also be mapped to a partial specification of the more detailed internal structure of agents. This is done in the SDL diagram for block agent type ATM in Figure 12. It specifies that each ATM instance will contain a number of sub-agents that are connected by means of channels.

This detailed SDL specification introduces two new concepts: Interfaces and Gates. The following defines these and provides the mapping to UML.

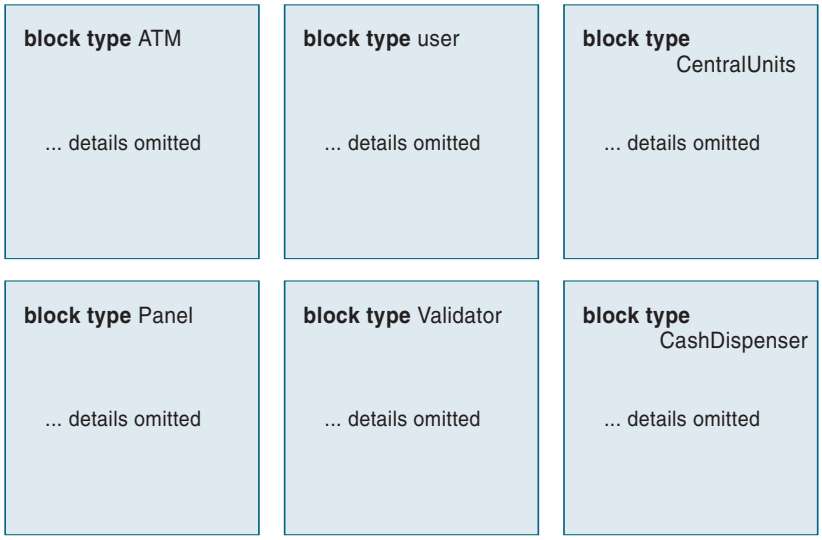


Figure 9 ... and the corresponding block type diagrams, here only partially specified

SDL – Interfaces and Gates

An Agent may have a number of required and implemented Interfaces. An Interface defines Signals, Variables, RemoteProcedures and Exceptions. A Signal defines the types of the data (parameters) that will be sent with each signal instance. A RemoteProcedure defines the signature of procedures that may be exported by Agents and thereby be requested by other Agents.

Interfaces are associated with Gates. Gates are connection points for Channels connecting Agents. Communication between Agents takes place via Channels.

An *implemented* Interface defines which Signals and which RemoteProcedure call requests that may be sent to an Agent. A required Interface defines which Signals the Agent may send and which RemoteProcedures it may request from other Agents.

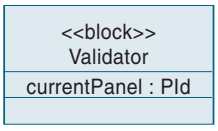


Figure 10 Partial type specification ...

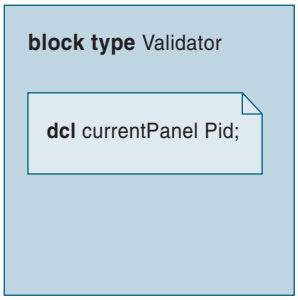


Figure 11 ... corresponding to a textual specification in the diagram

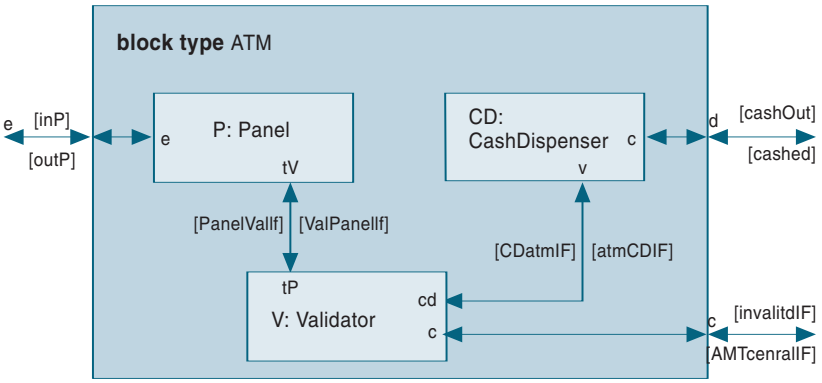
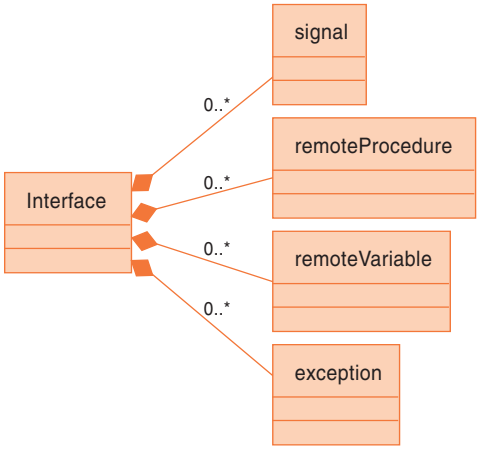


Figure 12 SDL specification of the ATM block type

Figure 13 UML conceptual model of SDL interfaces

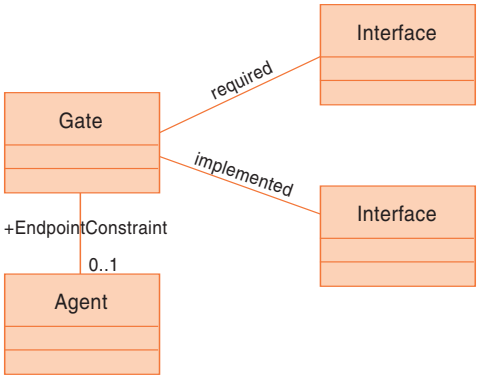


A Gate with an EndpointConstraint can only be connected to Agents of the same type as or to a subtype of the Agent type of the constraint.

UML Mapping. An Interface is mapped to an *Interface*. UML *Interfaces* can only have *Operations* (corresponding to procedures), so Signals, Variables and Exceptions are mapped to stereotyped *Operations*.

It is not possible to represent the Gates of an Agent in UML directly according to the conceptual model, unless representing them as attributes or objects. A Gate is, however, not an instance but rather a connection point and as such similar to an Association of UML Objects. Therefore, Gates are mapped by means of different kinds of Associations:

Figure 14 UML conceptual model of SDL gates



- For gates as connection points between instances of two agent types the mapping is to an association between the two types.

- For a gate of a single agent type the mapping is to a stereotyped association. The name of the association maps to the name of the gate. The type at the other end of the association is either the type of an endpoint constraint, or no type in case there is no endpoint constraint.

Base Class for gate

- AssociationEnd for the definition of a Gate as a possible connection point based upon associations between classes representing Agent types;

- Association stereotyped with gate for the definition of a gate (possibly with endpoint constraint) as part of the definition of an Agent type.

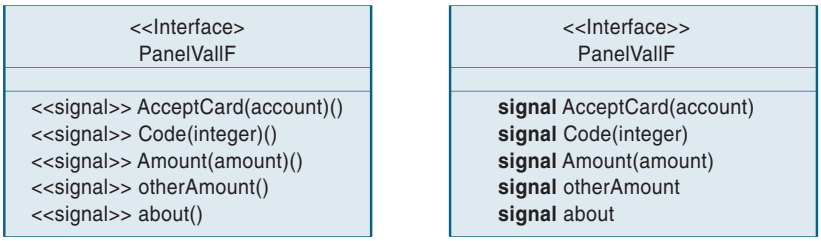
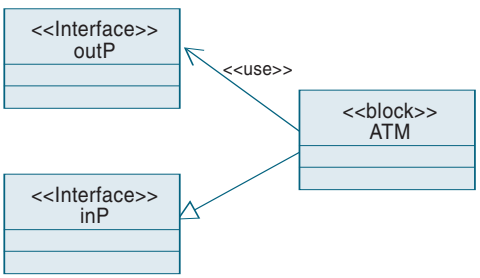


Figure 15 Interface in UML (left) and SDL (right)

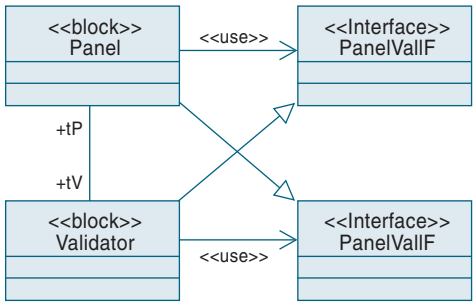
The example in Figure 15 shows the use of stereotype “signal” to specify that the interface PanelValIF defines a set of signals. The corresponding SDL graphical interface definition is also given.

If we were to map Figure 12 back to UML, the gates of the ATM block agent type are not specified as objects, even though the conceptual model in Figure 4 defines gates as objects. The gates of the ATM are defined by means of a combination of interfaces and associations to other classes. In Figure 16 one of the gates of the ATM block type is represented by two interfaces, one for incoming signals and one for outgoing signals.

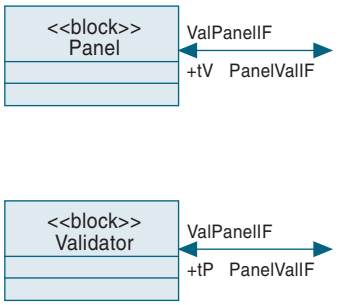
Figure 16 Block Agent Type with Gate in UML$_{SDL}$



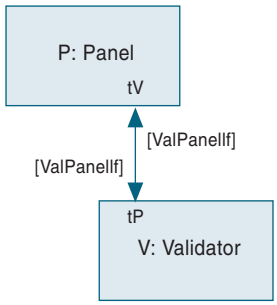
In Figure 17 the gates of two of the agent types are defined by the role names of an association between the two types. The implemented and/or required interfaces are specified either on the



association between types in UML



represents gates on each type in SDL



forming the basis for connecting instances in SDL

Figure 17 Gates by means of Associations

types (as in Figure 17) by means of use and realises dependencies, or as part of the role names (not shown here). If more than one gate at each agent type should be specified, then the role name alternative must be used in order to specify which interfaces belong to which gates.

Figure 17 illustrates that gates with two-way interfaces are better specified in SDL. This is not surprising, since this is one of the distinguished features of SDL. Z.109 defines the mapping, but in actual use it is recommended to specify the gates as part of the SDL specification.

From the SDL specification in Figure 12, it is possible to apply the mapping from SDL_{UML} to UML_{SDL}. This is illustrated in Figure 18 and Figure 19. The mapping is to a combination of a Composition with representation of the gates of the composite class and a Collaboration with roles representing the parts of the composite that interacts.

The internal connections are mapped to associations between the classes (Figure 18) and to association roles between the roles of a Collaboration (Figure 19). Note that compared with the initial collaboration (Figure 3), the collaboration in Figure 19 does not include the User and the CentralUnit roles.

The association roles linking P with V and V with CD correspond to the channels of the SDL model. For the purpose of specifying the contents (in terms of objects) of a type of objects, (here the type ATM), the SDL diagram is superior. The UML specification provides the specification at the class composition level and a separate collaboration diagram, while the SDL specification combines these two.

6 Signals

As described above, possible interactions between Agents by means of signals is defined by interfaces and gates. The signals themselves are defined by signal definitions, either as part of interface definitions or as part of packages or agents.

SDL: Signals and Signal Types

A Signal may carry a set of values called Parameters.

A Signal definition defines a type of Signal instances. An Agent sending a Signal does this by generating a Signal instance according to a Signal type and providing the actual Parameters. The receiving Agent may assign the values of the Parameters to local Variables.

Signals can be defined either in the Agent enclosing the Agents that use the signals for com-

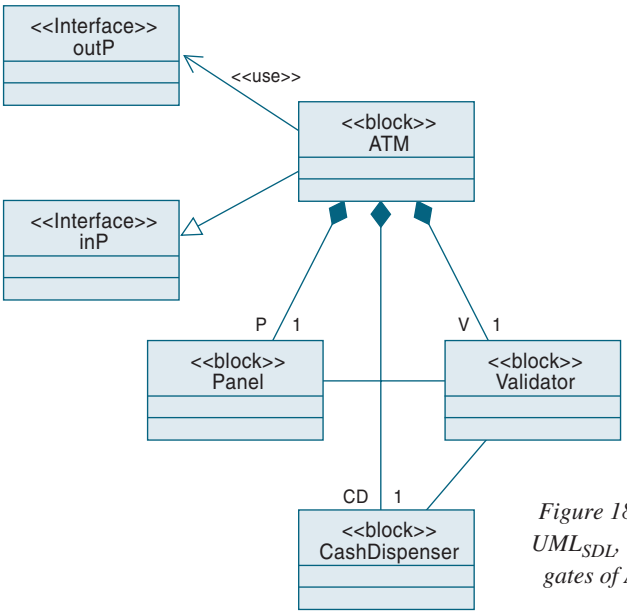


Figure 18 Analysis model in UML_{SDL}, with only one of the gates of ATM represented in UML

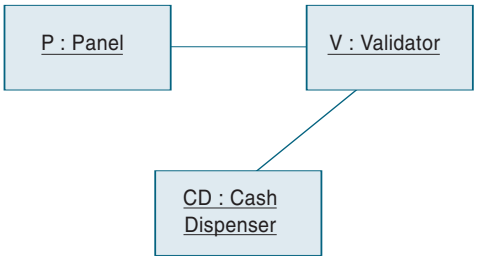
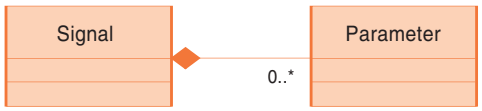


Figure 19 Collaboration representing ATM

Figure 20 Signal



munication, as part of an Interface, or in a separate Package.

A Signal can be defined as a subtype of another Signal. The subtype inherits the Parameters of the supertype and may add Parameters. Inherited Parameters cannot be changed.

A Signal type defined in an Agent type can be defined to be a virtual Signal type. A virtual Signal type can be redefined/finalized by extension in subtypes of the Agent type with the virtual Signal type definition.

UML Mapping. Signal is in UML represented by the standard UML subclass of *Classifier* stereotyped with "signal". The Parameters are represented by *Attributes*.

Tags

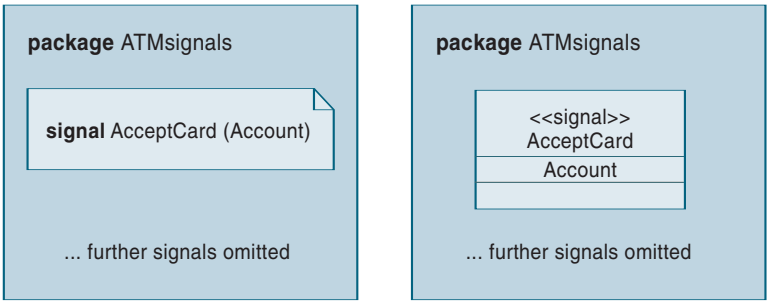
- Virtual - the Signal type is a virtual type
- Redefined - the Signal type is a redefined type, but still virtual
- Finalized - the Signal type is a finalized type, and no longer virtual

(Stereotype) Constraints

- A Signal type *Classifier* can have at most one super (Signal) type *Classifier* and it must be stereotyped with "signal";
- A virtual, redefined and finalized Signal type Classifier must be defined in the namespace of an Agent type *Classifier*;
- *Visibility* of attributes representing Parameters does not apply.

A signal can in SDL be defined in a textual form, and in a combination of graphical and textual forms. As an example, the signal AcceptCard introduced in Figure 15 is defined in a Package called ATMsignals. Figure 21 provides the SDL and the corresponding UML specification.

Figure 21 Signal definition in SDL (textual), in SDL (graphical) and in UML_{SDL}



7 Specialisation of (Agent) Types, and Virtual Types

In order to illustrate the use of tagged values, suppose that the panel of the ATM needs to be redefined for different kinds of ATM. The different kinds of ATM are defined by specialisations of the general ATM class. Both SDL and UML cover specialisation, although in a slightly different form.

SDL: Agent Types and Subtypes

An Agent type can be the specialisation of another Agent type. A specialisation may add properties to those specified for the supertype, including subAgents and Channels, and it may redefine/finalize virtual types and/or Procedures being defined in the supertype.

A virtual type/Procedure can be either redefined (in which case it is still virtual) or finalized (in which case it is no longer virtual). Redefinitions/finalizations must obey the constraint of the virtual. A virtuality constraint is in terms of a general type/procedure, and the redefinitions/finalizations must be subtypes/subprocedures of the constraint.

UML Mapping. Specialisation maps to Generalisation, with the constraint that there is only one superclass and that the kind of the superclass is the same as the kind of the subclass.

Tags

A virtual type is mapped to a class with one of the following tagged values:

- Virtual - the agent type is a virtual type;
- Redefined - the agent type is a redefined type, but still virtual;
- Finalized - the agent type is a finalized type, and no longer virtual.

The virtuality constraint is mapped to the *constraint* association of the *Class*.

(Stereotype) Constraints

- An agent type *Class* can have at most one super (agent) type *Class* and it must be of the same kind as the agent type *Class*;
- A virtual, redefined and finalized agent type *Class* must be defined in the namespace of another agent type *Class*;

- A *Collaboration* representing the structural content of a super agent type *Class* is inherited by subtype *Classes*.

In SDL a general ATM type is specified by having the type of the panel (ATMpanel) defined locally to the type ATM and by having it defined as a virtual type, see Figure 22.

Note that the locally defined type ATMpanel is indicated by a type symbol within the diagram of the enclosing type. The full specification of the type ATMpanel will be in a separate diagram, see Figure 23. The heading of this diagram will give additional specification of the type, e.g. that it is constrained by the general type Panel, so that it can only be redefined to a subtype of Panel.

The constraint type Panel, see Figure 24, defines the common properties of panels, including the interfaces in terms of gates. By specifying the Panel as the constraint of the virtual type ATM-panel, it is enforced that all redefinitions have this interface.

In UML_{SDL}, the fact that ATMpanel is defined in the namespace of ATM is specified by the plus encircled line, and the virtuality is specified by the tagged value "virtual", see Figure 25. So far, we have only used the filled diamond type of UML graphics for Composition. Figure 25 includes the symbol enclosing style, in order to illustrate that a class symbol within another class symbol is not the same as name space containment. In both variants in Figure 25 the plus encircled line must be there in order to specify that the class is defined locally to the ATM.

The constraint of the virtual type is not shown in the class symbol for the virtual type.

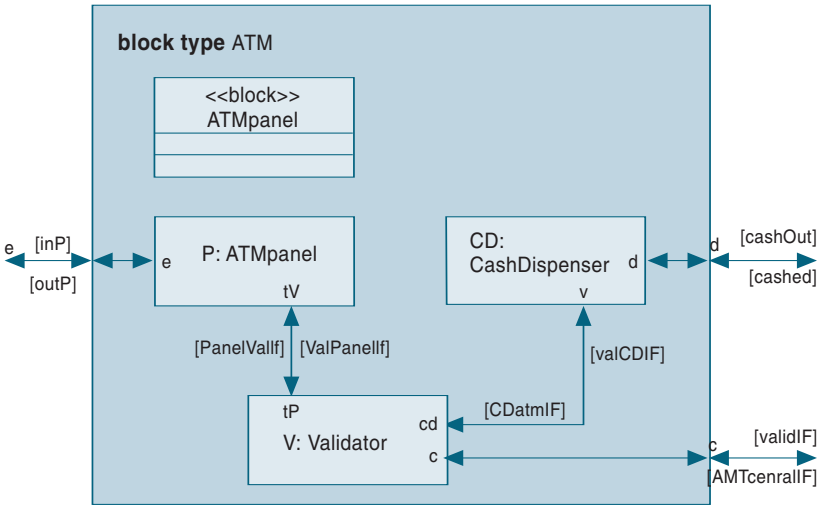


Figure 22 Virtual type in SDL

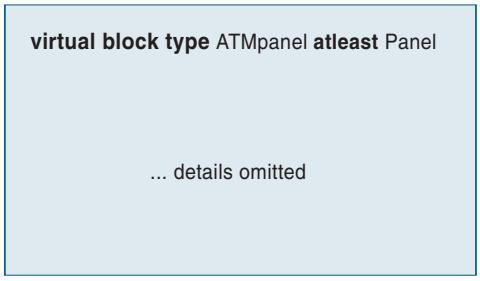


Figure 23 Virtual type diagram – only heading

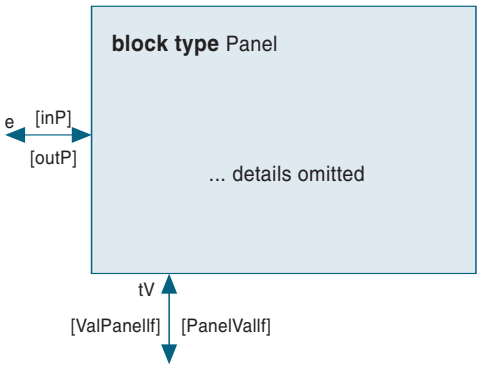


Figure 24 The Panel constraint type – only heading and gates

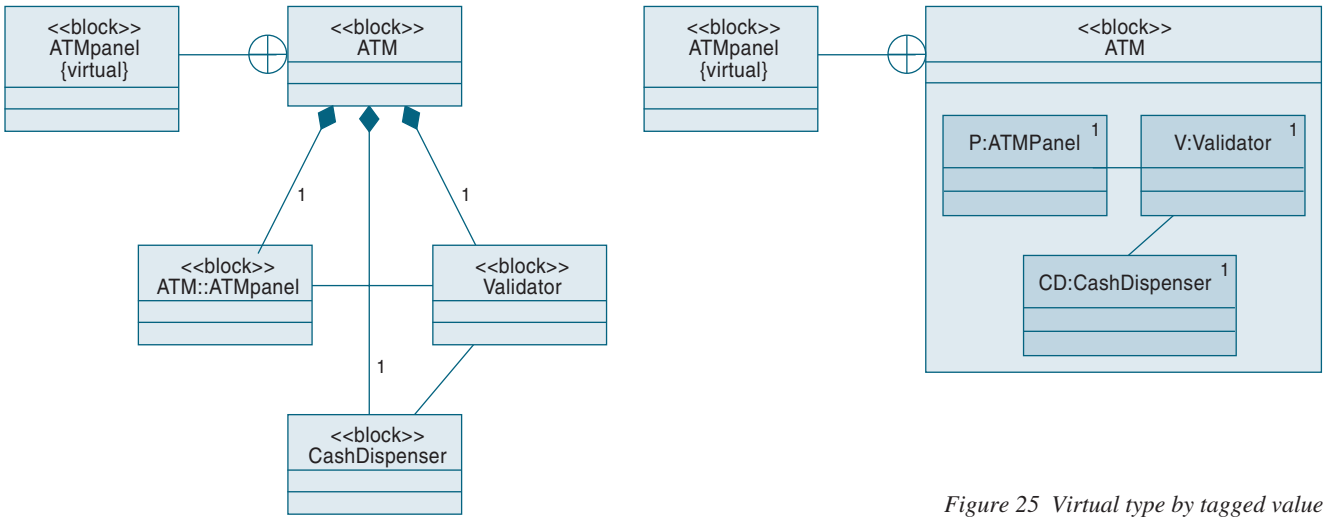


Figure 25 Virtual type by tagged value

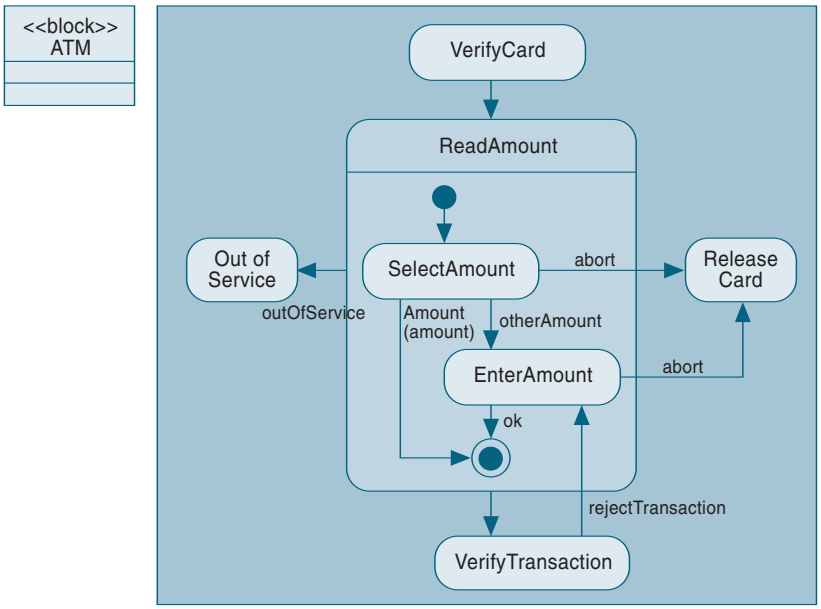


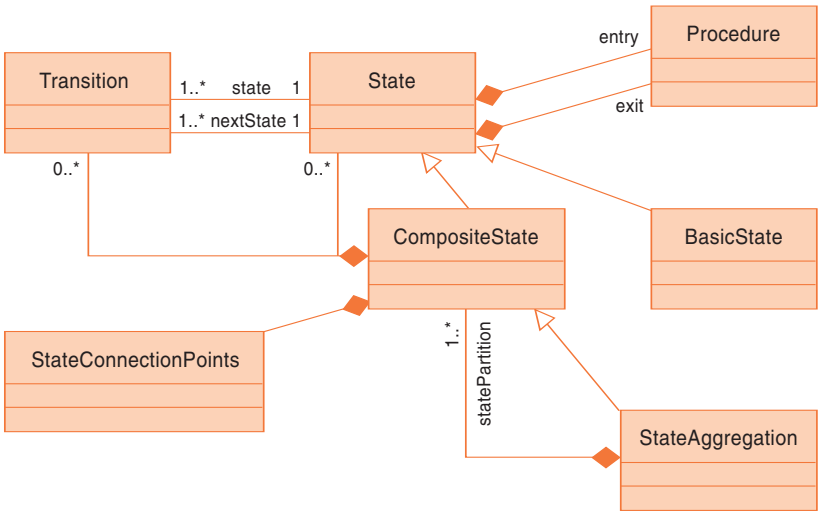
Figure 26 UML Statechart for ATM

8 State Machines

In order to illustrate state machines, the block type ATM is changed by substituting the Validator block with a state machine. The effect would be the same if the Validator block had its own state machine: the ATM state machine will execute concurrently with the state machines of Panel and CashDispenser.

The state machine of the ATM illustrates the notion of composite state. Figure 26 is a UML statechart diagram associated with the class ATM. In UML, the association between the class and statechart is part of the UML meta-model, but there is no graphical notation for it. In a Class Diagram there will be a class symbol for ATM, while a separate Statechart Diagram defines the state machine. There is nothing in the class symbol saying that this class has an associated state machine.

Figure 27 UML conceptual model of SDL state machine



SDL: State Machines

States may be either BasicStates or CompositeStates. The StateMachine of an Agent is a CompositeState. A CompositeState has a number of States and Transitions. Transitions are triggered by events like input of a Signal or a remote Procedure call. Composite states may be StateAggregations, i.e. states with a number of StatePartitions. A StateAggregation is in one of the states in each StatePartition, while an ordinary CompositeState is in just one of its states.

UML Mapping. In the UML meta-model there is an aggregation between *ModelElement* and *Statemachine*, with role names *context* (the *ModelElement*) and *behaviour* (the *Statemachine*). The *stateMachine* of an Agent is represented by a *Statemachine* with the *context* associating it to the class of the Agent type.

A StateAggregation is mapped to a *CompositeState* with *isConcurrent=true*. A StatePartition is mapped to a State with *isRegion=true*, but with a different semantics than UML (see above). *isRegion* is an attribute of the UML meta-model element *CompositeState*.

The SDL specification corresponding to Figure 26 has a block type diagram for ATM. This diagram shows the contents of ATM objects, including the fact that there is a state machine. The state machine is defined in a separate diagram.

The state symbol (with the name ATM) in the block type diagram in Figure 28 specifies that the ATM has a state machine. The definition of the state machine is given in a separate state diagram, see Figure 29.

Note that the state machine of ATM is defined by two state diagrams. The reason is that the state machine of ATM contains a composite state ReadAmount. In the state diagram ATM this state is specified by a state symbol, and its internal specification (in terms of states and transitions) is given in a separate ReadAmount state diagram. Even though there is a mapping between UML and SDL state machines at the meta-model/grammar level, the graphical syntax for state machines is very different. While SDL-2000 has (independently of Z.109) introduced UML-like class symbols for types and associations in the style of UML, the state machine notation of SDL is so different from the Statechart notation that composite states was introduced in SDL-2000 as part of the existing SDL notation. Z.109 can be used to switch between the SDL notation and the Statechart notation.

As demonstrated by the example above, the main rationale for Z.109 to cover state machines is that the view offered by UML Statecharts is

good for the state overview, while the SDL view is good for specifying the details of transitions. In case of large state machines, the SDL way of dividing them into separate state diagrams is superior to the graphically nested form of state-charts.

9 Variables and Procedures of Agents

Somewhere behind the scene presented so far, there will be Account objects. Transactions on the ATM will imply transactions on the account of the user. A simple model of an account includes attributes like account number, balance and credit limit and operations like deposit, withdraw, open, and close.

The standard way of modelling this is to define an Account class. SDL and UML are similar here, with the exception that SDL makes a distinction between value- and object classes.

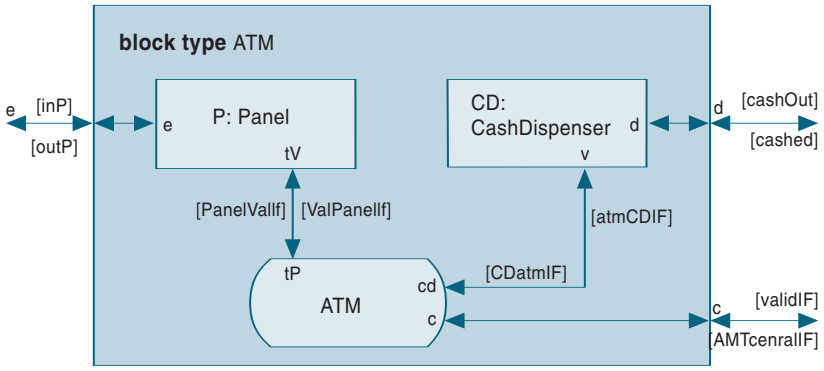


Figure 28 Block type diagram with internal agents and a state machine

SDL: Object and Value Typed Variables

In addition to a StateMachine and sub-Agents, an Agent may have both attributes in terms of Object and Value type Variables, and operations in terms of Procedures. A Variable may be an exported variable, in which case other Agents may observe its value.

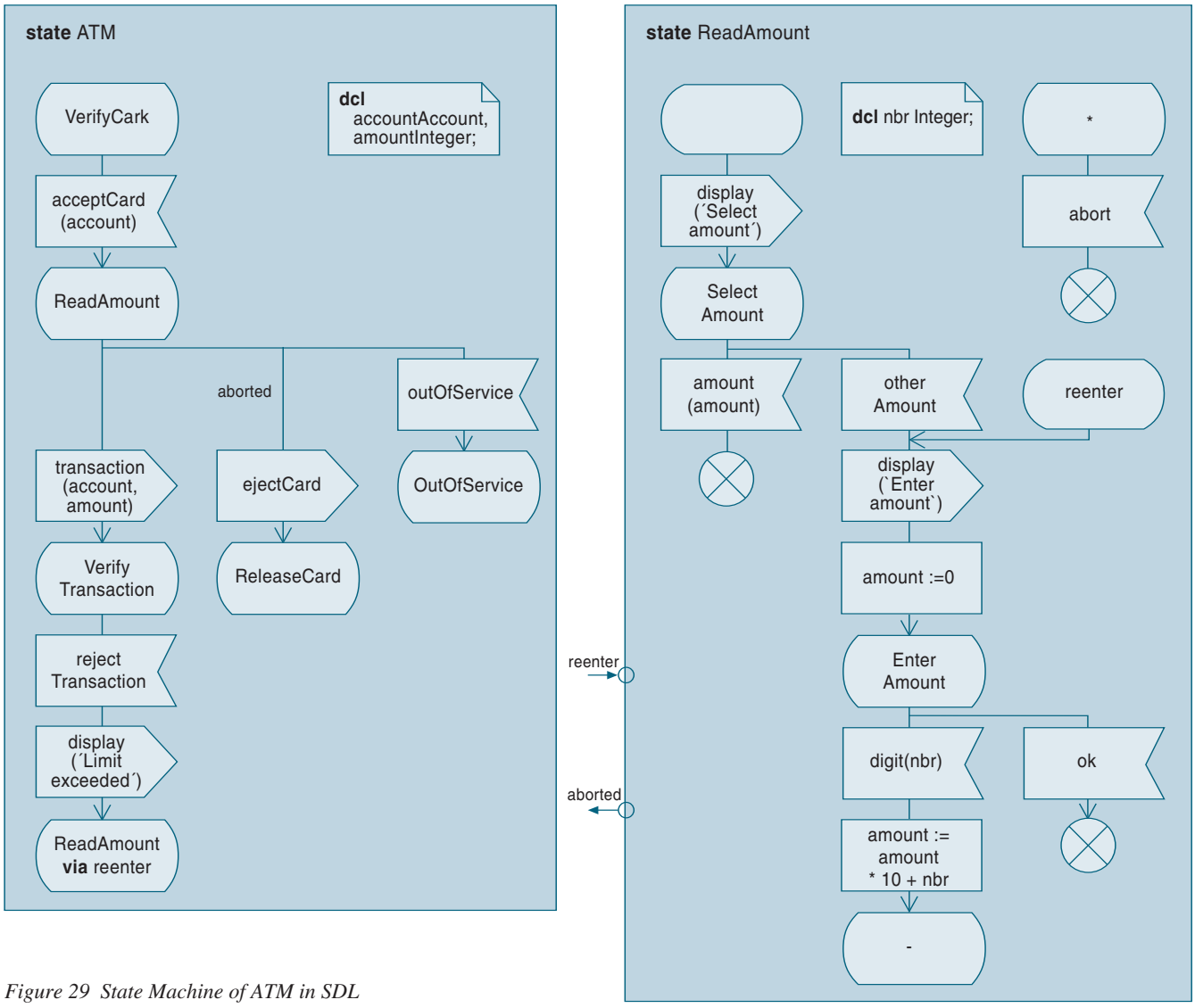


Figure 29 State Machine of ATM in SDL

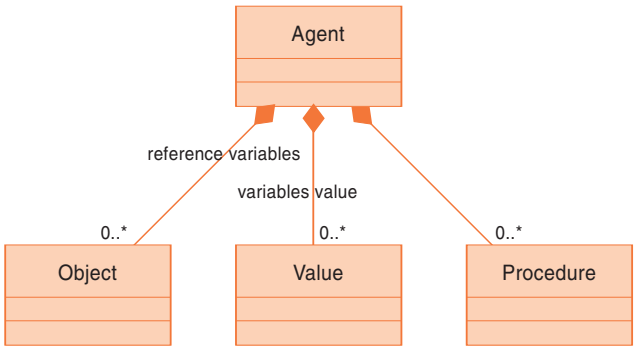


Figure 30 Variables and Procedures of Agents

Figure 31 UML Class with Attributes and Operations

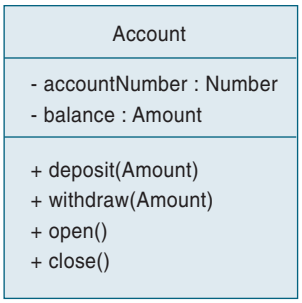
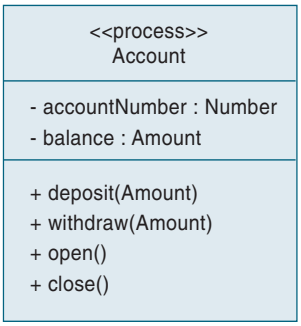


Figure 32 UML$_{SDL}$ Process Type



UML Mapping. Variables are mapped to *Attributes*, and Procedures are mapped to a combination of *Operations* and *Methods*.

Base Classes

- *Attribute* for reference- and value variables;
- *Operation/Method* for Procedure;
- *Class* for Procedures that are specialisations of general Procedures.

Constraints

- *Visibility* = private is not applicable.

A local Variable is mapped to an *Attribute* with *visibility* = protected. An exported Variable is mapped to an *Attribute* with *visibility* = public.

Figure 31 illustrates the UML way of specifying a class with attributes and operations.

From this general UML model it is possible to elaborate into SDL in two different ways: either Account objects are active objects which execute their operations themselves, or Account objects are just data objects with associated operations.

In Figure 32 it has been decided to model the account by an active object, and therefore, the stereotype “process” has been applied.

In Figure 33 the corresponding SDL is given, in both graphical and textual form. The process type diagram is only sketched. In addition to the declaration of the variables and of the procedures, a process type diagram will typically contain the specification of the state machine, specifying in which states the different procedures will be accepted and executed.

The procedure symbols in the process type diagram in Figure 33 play the same role as class symbols for agent type. They specify that the enclosing type has a number of procedures and that the detailed specification of the procedures is to be found in separate procedure diagrams, just as for type diagrams, see Figure 34.

10 Procedures

Procedures have been introduced above as properties of Agents. The example will not contain any detailed definitions of procedures. Procedures are meant for specifying patterns of behaviour that an Agent can perform as part of transitions. Procedures can be specified in the same way as Agents: by means of a state

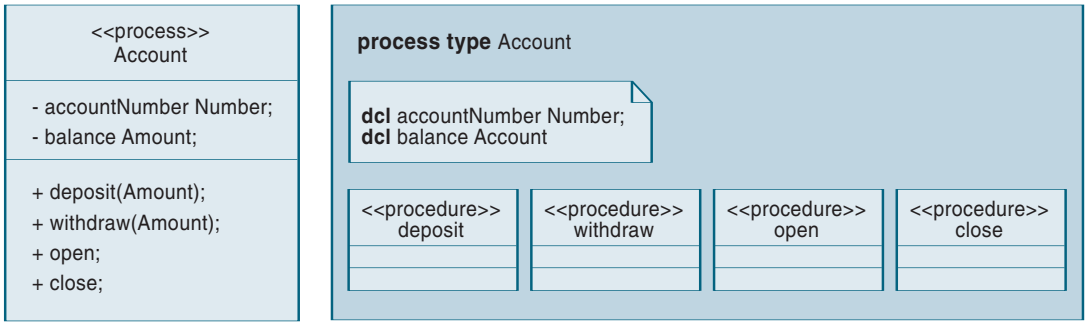


Figure 33 Process Type Symbol and the corresponding Process Type Diagram

machine. Simple Procedures will just have one transition and no states.

SDL: Procedure

A Procedure is a pattern of behaviour specification that can be used by Agents performing the Procedure by calling it as part of transitions. Procedures may either be LocalProcedures or ExportedProcedures. A LocalProcedure is a procedure that is used locally in an Agent, while other Agents (according to the signature defined by a RemoteProcedure) may request the execution of an ExportedProcedure.

A Procedure can be a specialisation of another (general) Procedure, thereby inheriting parameters, eventual local variables and behaviour specification.

A virtual Procedure is a Procedure that can be redefined in subtypes of the enclosing Agent type. A virtual Procedure can be either redefined (in which it is still virtual) or finalized (in which case it is no longer virtual). Redefinitions/finalizations must obey the constraint of the virtual. A virtuality constraint is in terms of a general Procedure, and the redefinitions/finalizations must be subtypes/subprocedures of the constraint.

UML Mapping. Procedures are mapped to a combination of *Operations*, *Methods* and *Classes*. The signature of the Procedure is represented by an *Operation*, while the body is represented by a *Method*. A Procedure that is a specialisation of another Procedure is in addition represented by a *Class* (in the namespace of the *Class* representing the enclosing type) with stereotype “procedure” and with a *Generalisation* relationship to the *Class* (also stereotyped with “procedure”) representing the superprocedure.

Base Classes

- Operation/Method for Procedure;
- Class for Procedures that are specialisations of general Procedures.

Tags

- For Procedure:
  - Virtual - the Procedure is a virtual Procedure;
  - Redefined - the Procedure is a redefined Procedure;
  - Finalized - the Procedure type is a finalized Procedure.

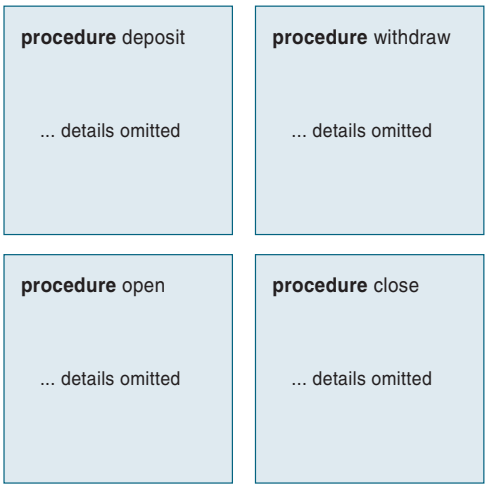


Figure 34 Procedure diagrams corresponding to the procedure symbols in Figure 33

Constraints

- LocalProcedures have private visibility only, while ExportedProcedures have public visibility only;
- *Visibility* = private is not applicable.

A LocalProcedure is mapped to an *Operation* with *visibility* = protected. An ExportedProcedure is mapped to an *Operation* with *visibility* = public.

11 Data Types

Data types are used to define the properties of attributes and parameters to signals and operations. The access code that follows each card, with a card identification number and a personal identification number, is an example of an attribute defined by a data type. The data type AccessCode will define a structure consisting of two fields: cardId and pin.

SDL: Data Types

SDL data types come in two different forms. A value data type<data type definition>:value:use in text defines a set of values. An object data type <data type definition>:object:use in text defines a set of objects.

Variables of Object types are references with an associated reference semantics. Assignment

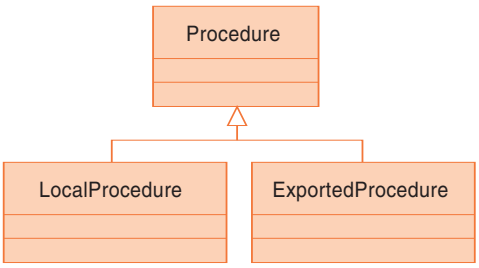


Figure 35 Procedures

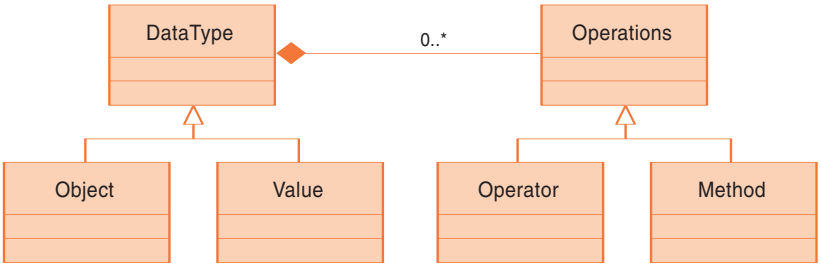


Figure 36 Data Type

between these variables is reference assignment, and two reference variables may denote the same Object instance. Variables of Value types exhibit value semantics: assignment means e.g. copying the value from one variable to another.

Data types can be defined in various ways: either by enumerating the elements of the type (literal) or by constructing a tuple from elements of given sorts (structure of fields).

Operations are Operators or Methods. Operators are functions that produce new objects or values, while Methods are applied to instances and can modify properties of the actual instances. Operations in general can be defined as virtual, redefined and finalized Operations.

In addition to Operations a DataType may also define local DataTypes. DataTypes can be defined as part of Packages and as part of types. DataTypes defined locally to a type can be defined as virtual DataTypes and thereby redefined/finalized in a subtype of the enclosing type.

UML Mapping. Object and Value data types are mapped to stereotyped Classes, i.e. the predefined UML DataType is not used. Variables of Object and Value types are therefore mapped to attributes with user-defined classes as attribute types. Fields of structured types are mapped to Attributes. Operators and Methods are mapped to a combination of Operations and Methods.

Base Classes

- Object type is a stereotyped Class (“Object”);
- Value type is a stereotyped Class (“Value”).

Tags

For both “Object” and “Value” the following Tags apply:

- Virtual - the object/value type is a virtual type;
- Redefined - the object/value type is a redefined type;
- Finalized - the object/value object/value type is a finalized type.

Operators and Methods have the same visibility options (public, protected, private) as Operations in UML, so there is a direct mapping.

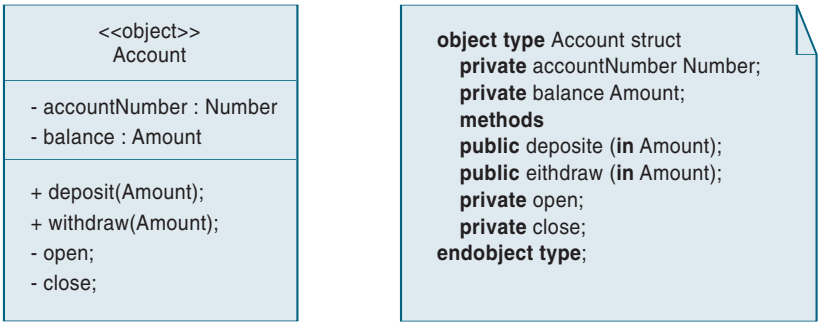


Figure 37 Object Type in UML_{SDL} and in SDL_{UML} (textual form)

In Figure 37 it has been decided to model Accounts by means of objects with fields and methods. The corresponding textual SDL partial definition (not including the bodies of the methods) is also included in the figure.

The SDL version of the data type is given in the textual form, contained in a text symbol, which will be part of a diagram. SDL also allows a class symbol as a partial specification of the data type, but in order to have it completely defined, the textual form is necessary.

Figure 38 gives an example of the definition of a simple value data type AccessCode in both UML_{SDL} and in SDL_{UML}.

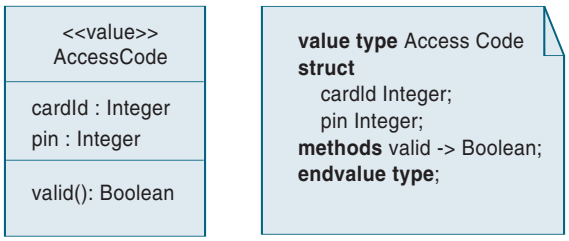


Figure 38 Value data type in UML_{SDL} and in SDL_{UML} (textual form)

12 Packages and Overall System Specification

Both SDL and UML have the notion of package for the grouping of elements and of a topmost unit of specification. Figure 39 is a package diagram in SDL, using the signal types defined in the package ATMsignals and defining three block types.

SDL: Packages and Systems

A Package is a grouping of definitions, including type definitions but not instance definitions. A Package may also contain other Packages. A Package may use another Package, including all its definitions or just a subset of these. The PackageInterface of a Package defines the elements of a Package that are visible outside the Package.

The complete SDL SystemSpecification consists of a number of Packages and possibly a System. If no System is included, the SystemSpecification is simply just a means to define a set of Packages.

A System may also use Packages, with the implication that the definitions in the Packages become visible as if they were defined in namespace enclosing the System namespace. Systems do not contain Packages. Decomposition of a System into parts is obtained by the general mechanism for structuring of Agents. The System is the special outermost Block Agent (see Figure 5), and as Agents may contain Agents (connected by Channels), a System may be decomposed into a number of Block Agents or a number of Process Agents, depending on the concurrency involved between the parts of the System.

UML Mapping. A Package is mapped to a *Package* constrained as specified below. The use of a Package is mapped to an import *Dependency*. The PackageInterface is mapped to public visibility for the model elements to which the elements in the PackageInterface are mapped.

A SystemSpecification is mapped to a *Model Package* with a system-stereotyped class for the System Agent.

While a UML Model may have a number of Models, each representing a view on the modelled system, the corresponding SDL systemSpecification may only contain one view, represented by the System.

Note also that the UML notion of *Subsystem* is not used in the mapping. The reason for this is that parts of an SDL system are Block Agents. Blocks can be created dynamically, they have unique identifiers and they can have attributes and operations – and that is not possible with UML *Subsystems*.

Constraints
A SystemSpecification can only have Packages and a System, i.e. no other Models.

The UML model corresponding to Figure 39 is given in Figure 41.

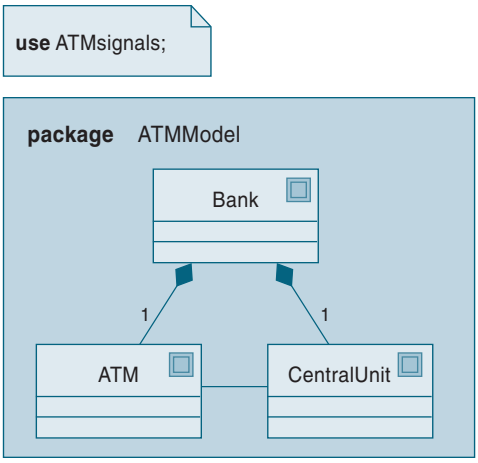


Figure 39 Package diagram in SDL

13 Context Parameters

An SDL type can be parameterised, so that it can be used in different contexts. Parameterised types are often defined in packages, as they are supposed to be usable in different systems. *Context parameters* are typically types, but they may also be variables and instance sets.

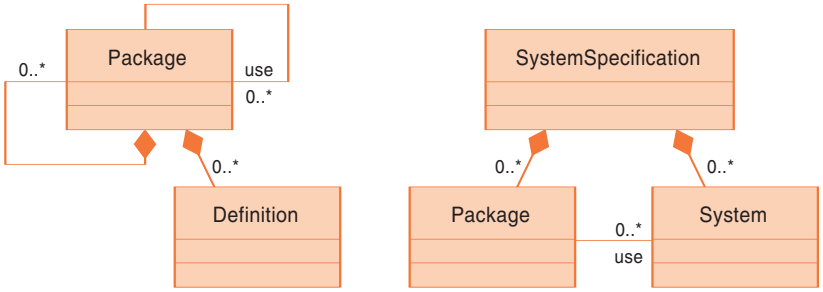


Figure 40 Package and SystemSpecification

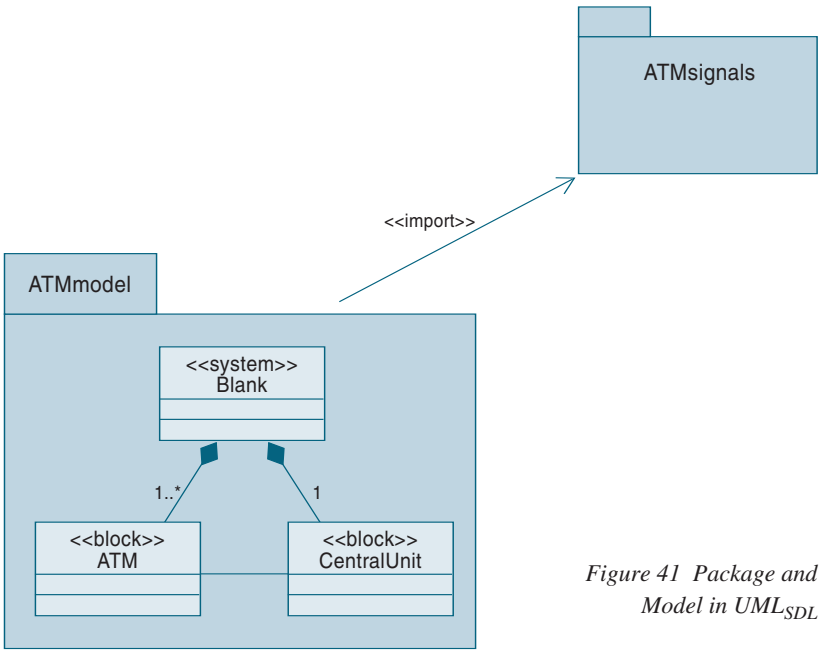
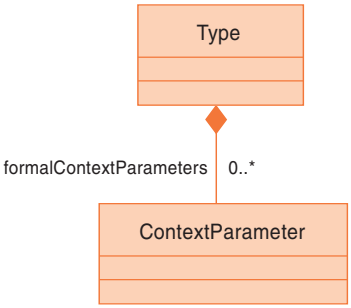


Figure 41 Package and Model in UML_{SDL}

Figure 42 Parameterised Type



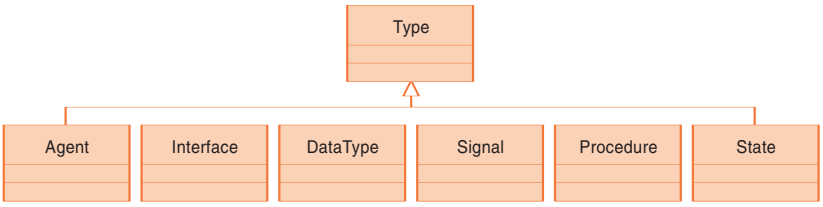


Figure 43 Types that can be parameterised

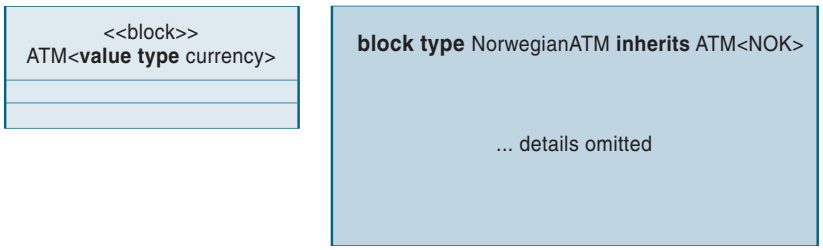


Figure 44 Parameterised ATM type and a type with actual parameters, in SDL

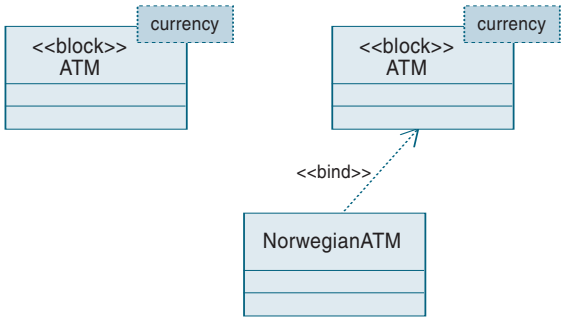


Figure 45 Parameterised ATM type and a binding, in UML

SDL: Context Parameters

A type in general can have formal context parameters. A formal context parameter represents a corresponding entity in the context (scope) of the type.

The types in Figure 43 can be parameterised by context parameters.

Depending on the kind of type, context parameters can be agent types, procedures, data types, signals and even variables. The simple rule is that entities that can be defined in the context of a given type also can be context parameter of the type.

When defining a new type based upon a parameterised type, the actual context parameters are types, procedures, data types, signals or variables in the scope where the new type is defined.

Context parameters are constrained, so that the specification of the parameterised type can be analysed independently of where it is used for defining new types.

UML Mapping. Context parameters are represented by *TemplateParameters* of the *Class* that represents the SDL type.

The provision of actual context parameters is represented by a *binding*. The binding as an explicit relationship does not exist in SDL. It is part of the definition of the type as a subtype of the parameterised type.

In SDL the context parameters are specified as part of the name of the type, see Figure 44.

In Figure 45 the corresponding ATM class in UML is a class with template parameters. The actual parameters are provided as part of the binding.

14 References

1 ITU-T. *SDL combined with UML*. Geneva, ITU-T, 2000. (Z.109.)

2 UML documentation on: www.omg.org/uml.

Summary of Z.109 SDL Combined with UML

This paper provides an introduction to the ITU Recommendation Z.109: SDL Combined with UML. The style is different from the one of Z.109, in the sense that SDL (and a conceptual UML of SDL) is the entry point. From this viewpoint it should be clear which subset of SDL is covered.

Z.109 provides a specialisation and restrictions of the following UML model elements, with an indication of the mapping to SDL:

- Packages represent packages in SDL;
- Models represent SDL specifications, each consisting of a set of packages and a system. Subsystems are not used; the structuring of systems into subsystems is in SDL covered by a special kind of objects (block agents) and thereby mapped to Composition in UML;
- Classes represent SDL types, with the following stereotypes representing the different kinds of entity types in SDL. The features of classes represent different SDL type properties, depending on the stereotype:
  - «system»;
  - «block»;
  - «process»;
  - «procedure»;
  - «interface»;
  - «object»;
  - «value»;
  - «state»;
  - «signal».
- State machines represent state machines of agents;
- A subset of Associations represents the corresponding concept in SDL. Two special kinds of association represent partially other SDL concepts:
  - composition as a partial representation of containment between agents;
  - association stereotyped with gate representing a gate with endpoint constraint;
- Generalization represents the corresponding specialisation in SDL;
- The following UML Dependencies are used to represent different dependencies in SDL:
  - «import»: a package using another package;
  - «create»: an agent being created by another agent;

Virtual types and procedures/operations are represented by the following Tags:

- virtual;
- redefined;
- finalized.

MSC-2000: Interacting with the Future

ØYSTEIN HAUGEN

Øystein Haugen (45) works for Ericsson Research NorARC, Norway and has been heading the standardisation effort within ITU for MSC leading to MSC-2000. Haugen holds a PhD in computer science from the University of Oslo. His interests have been in the area of making precise software methods more applicable to common engineers. He also holds a position as associate professor at the Institute for Informatics at the University of Oslo. In 1993 he co-authored (with prof. Rolv Bræk) the textbook "Engineering Real Time Systems" (Prentice Hall) in connection with the SISU project.

oystein.haugen@ericsson.no

MSC-2000 [1] is the latest recommendation from the International Telecommunication Union (ITU) defining Message Sequence Charts. Earlier recommendations of MSC have been issued in 1992 [2] and in 1996 [3].

This tutorial goes through MSC-2000 by means of examples from an Access Control system.

1 Introduction

The system[1)] is a simple system to provide access to users to certain access zones through access points. In the most abstract specification, nothing is said about what means are used to achieve this access other than the fact that some individual code is the base for authentication. In a more concrete description, it is uncovered that we have a system based on a card bearing the necessary information for access, and that there is a door that needs to be opened and closed by the user.

The language MSC is a formal language. MSC-96 has a formal semantics defined in Annex B of the recommendation Z.120 [5]. The language is well suited to define interaction sequences. We shall use this language to specify our example system of Access Control.

The reader may make good use of earlier descriptions of MSC. Tutorials on MSC-92 and MSC-96 can be found in TIMe [6]. There are also MSC methodology papers presented at SDL fora in 1995 and 1997 [7, 8]. For the more formally inclined we recommend to have a look at Michel Renier's doctoral thesis [9] or a shorter presentation of formal semantics of MSC [10], or the paper by Andreas Prinz on Formal Semantics of Specification Languages in this issue of *Telektronikk*.

MSC is being used on its own as a precise way to describe interaction behavior or it is used together with other languages. Requirements expressed using MSCs can be used in model-checking of an SDL description [11] and as a base for producing TTCN test cases [12, 13].

Thus MSC is being used extensively in all phases of system development.

2 Basic MSC

We start by specifying the service of controlling the access of users to some Access Zones. A very simple description of a scenario where the User is accepted by the system is given in Figure 1. The simple MSCs merely describe a set of event traces. An event is something that happens in one moment in time. An event in MSC is typically an output of a message or an input of a message, or the setting of a timer, or the timeout of a timer. A trace is a sequence of events.

In the diagram UserAccepted we have two *instances* called User and ACSystem. The User sends a *message* Code to the ACSystem and later receives an OK message indicating that the ACSystem will Unlock the door. The MSC is bounded by a *frame* representing the border between the environment and the inner instances. The points on the frame that represent relations between the environment and the MSC such as the Unlock message to environment in Figure 1 are called gates. The MSC has a *name* in the upper left corner preceded by the keyword **msc**. The six cornered shapes at the start and at the end of the MSC are *conditions*. They represent a description of the situation present at that point. We shall turn to conditions later.

Most people, regardless of their background in software engineering, will normally understand such message sequence charts. One should be aware, however, that to specify the complete set of event traces is not always so easy. In MSC UserAccepted there are two possible traces as the diagram does not give explicit ordering between the output of the Unlock message from the ACSystem and the reception of the OK message by the User. More informally one may say that whether the door is unlocked before or after the User recognizing the OK, is not defined.

---

1) *Throughout this tutorial we shall use one example, the* Access Control System *pioneered by Bræk & Haugen in Engineering Real Time Systems [4].*

Both solutions are possible. This is implied formally by the two major invariants of event ordering in MSC:

1 The events on an instance line are ordered from the instance head to the instance end.

2 The output of a message comes before the input of the same message.

The problem with the basic MSCs was that when they became popular due to their simplicity, a normal system would have a large number of basic MSCs such that maintaining the base of MSCs would be very difficult. There were no language mechanisms to help structure the total description. MSC references were introduced to improve this as they made it possible to include references to separately defined MSCs in the body of other MSCs, as shown in Figure 2. This is very similar to the introduction of subroutines in FORTRAN. We see that the Unlock gate of User_Accepted in Figure 1 naturally appears as an actual gate from the MSC reference in Figure 2. Gates can be compared with parameters, or with interface points.

The introduction of MSC references was practical also for the new overview diagram that showed how MSCs were to be composed, as illustrated in Figure 3. The idea is of course that the High Level MSCs (in short called HMSCs) show the combination of MSCs in diagrams where the instances and the messages are abstracted away. This has proved to be very helpful to get an overview of MSC descriptions, and to prescribe the intended composition.

3 Time in MSC

The basic MSC approach to time is characterized by the use of timers, based on the same principle as SDL [14]. Typically the user will describe the timers that will be present in an implementation. The description will specify the starting of the timers and the consumption of timer messages. Stopping the timer is of course also a possibility if other messages are received that make the timing irrelevant.

3.1 Imperative Description of Time

In Figure 4 we see a diagram with an *inline expression*. The initial condition is that the door is unlocked. Then the timer door is started. The first operand of the alternative description describes the normal situation where the door is opened by the User and the timer is stopped and the door is closed. The second operand describes that the User neglects the open door and the timer expires. Finally, in both cases the door locks again. An inline expression is bounded by an expression frame, the operands are separated by a dashed separator line, and the operator is

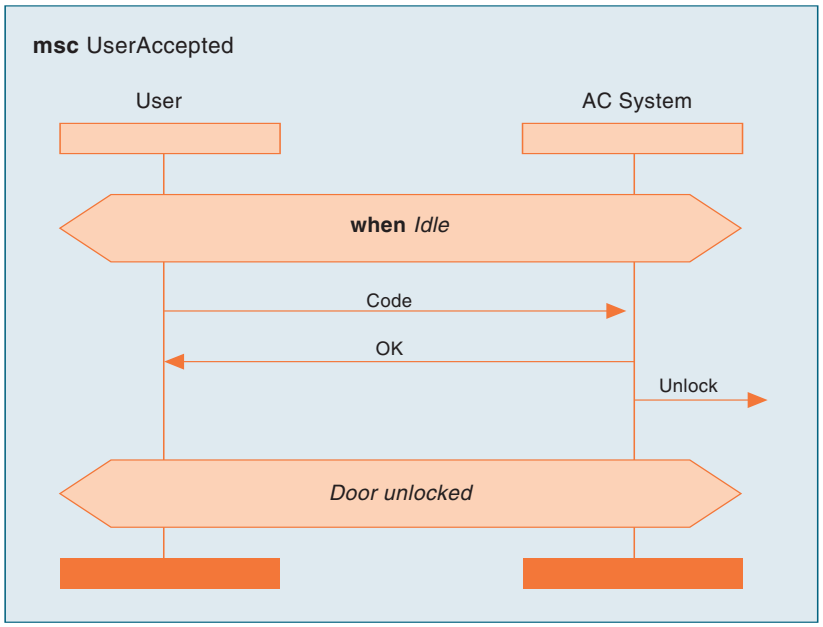


Figure 1 Simple MSC msc User_accepted

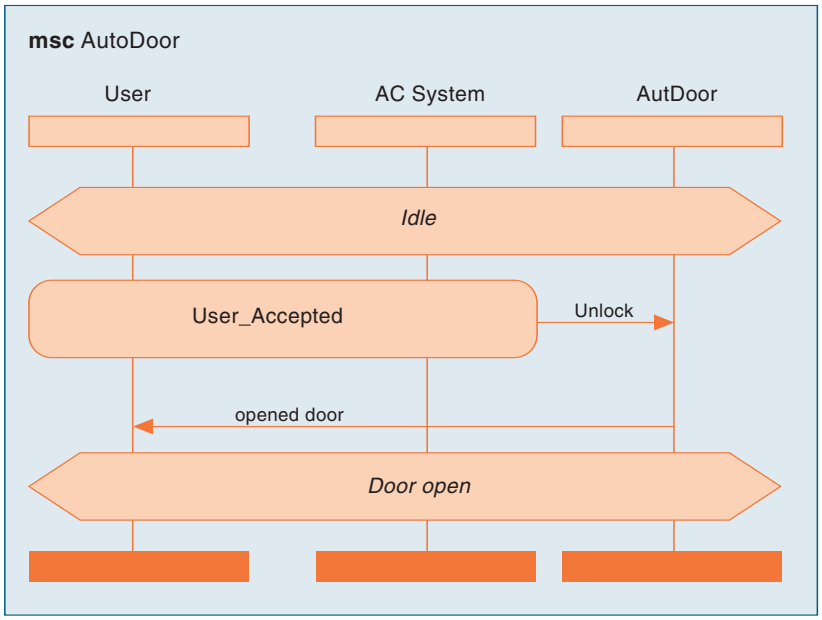


Figure 2 MSC with MSC reference and actual gates

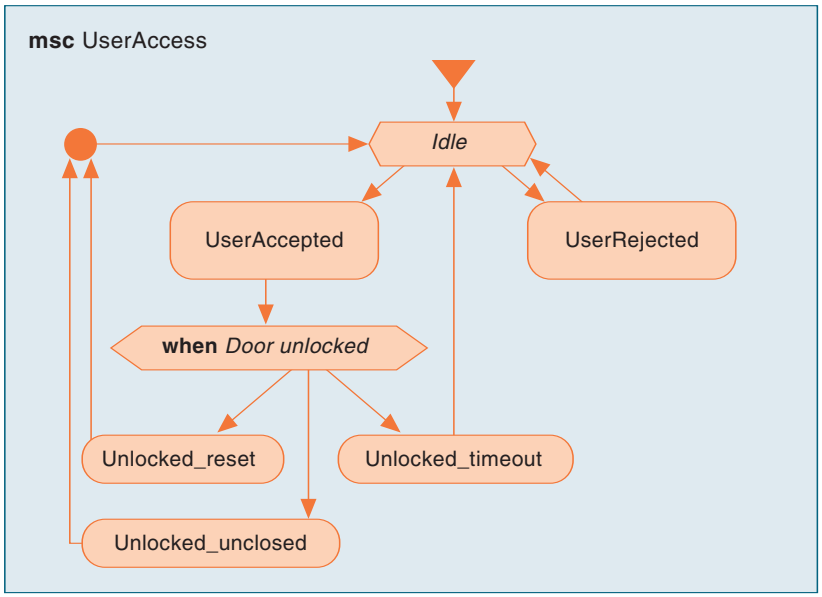


Figure 3 High-level MSC for improved overview

Figure 4 Using timers

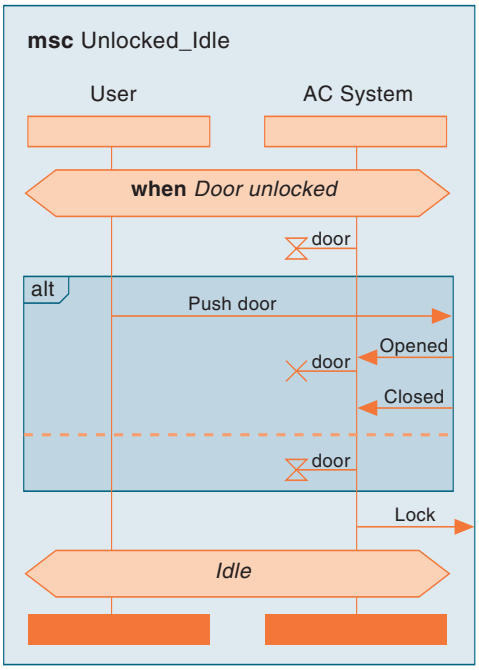


Figure 5 Using time constraints

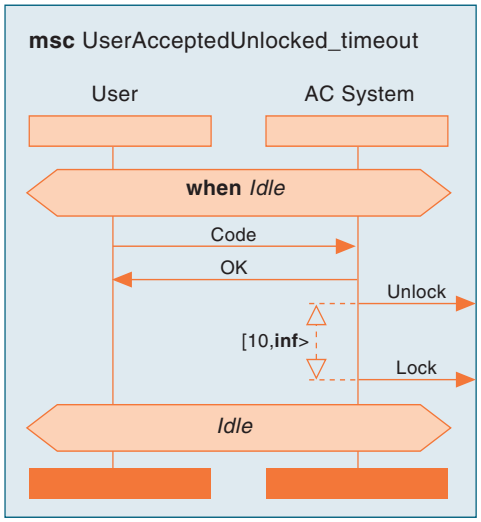
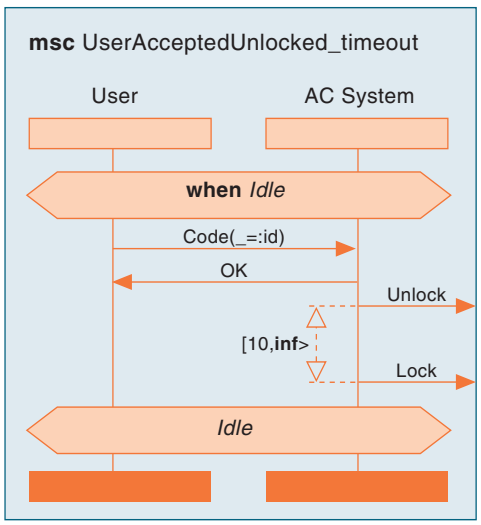


Figure 6 Data in messages



shown in the upper left corner of the expression frame.

We note that the timer events may appear individually. The events associated with the same timer occurrence may also be connected by a vertical line (not shown).

The initial conditions containing "**when** Door unlocked" in Figure 4 fit well with the final condition of Figure 1. The initial conditions check that the appropriate label is the current label for the covered set of instances while the final conditions set the value of the label. Thus conditions are used to restrict how the MSCs can be legally composed in sequence.

3.2 Constraint-Oriented Approach to Time

Another approach to describing time was introduced in MSC-2000. Now it is possible to describe time-stamps as well as time constraints. In Figure 5 we describe the same situation again using time constraints. While timers are conceptually more according to the view of the technical system, time constraints are more according to the (human) User. The User knows that when the system is neglected the unlocking of the door will be followed by a Lock after more than 10 seconds. The keyword **inf** describes infinity. Time constraints describe time relations between events that must be fulfilled for the whole scenario to be valid. What makes the time constraints hold is not described.

In MSC-2000 we can describe time constraints on intervals or on time points. We may also measure time intervals or points and use those measurements in time.

4 Data in Messages

Data has been made more precise in MSC-2000 and this implied that there was a need for more formal declaration of data. In Figure 6 we see that different aspects of data are defined in a diagram that defines a context for the individual MSC diagrams.

For each instance, the variables are declared with their name and associated data type. The data type is of course to be used for typechecking of expressions. The message types that have data parameters are also declared such that parameter type checking can be performed on message passing.

For MSC-2000 it was decided that instead of defining a data language entirely on its own, and instead of adopting one particular data language, the data language should be parameterized. This means that the designer can use his favorite data language as long as it is properly declared.

The language-clause indicates the name of the chosen data language such that the MSC analyzer can get access to the proper interfaces between MSC and the chosen data language. The data-clause gives the necessary data declarations in that given language. The wildcard-clause indicates an identifier which designates a wildcard value of a data type.

Finally, the parenthesis-clauses give declarations of different kinds of parentheses in the data language such that a pure MSC-parser can parse the data expressions even when the analyzer has no interface to the chosen data language. The escape-clause has the same purpose when parenthesizing is not sufficient.

In Figure 6 we see that the Code message is supplemented by the binding "_ =: id". The underline is defined (by the developer as shown in Figure 7) as a *wildcard* value meaning that it designates just any value. "id" is a name of a variable of the ACSystem. Thus the Code message with the binding denotes that the Code message transmits a value which may just be any integer from the User to ACSystem where the transmitted value is bound to the variable id.

5 Decompose MSC Instance Kind

An MSC document diagram defines the context in which a set of MSC diagrams appear. This set of MSC diagrams share the same set of interacting instances. This set of instances are components of the context and the MSC document can be understood as defining an enclosing instance. Actually a diagram normally defines a type and in case of MSC documents they define *instance kinds*. This corresponds well with decomposition as we shall see in greater detail below.

5.1 ACContext

In Figure 7 we have in addition to the instance and data declarations also MSC references in two groups. Above the dashed separator there are the *defining* MSCs and under the separator there are the *utilities*. The distinction can be compared with public and private attributes of classes in Java or C++. The semantics of an MSC document is the set of all traces given by the defining MSCs. Some of the MSCs referenced from the MSC document in Figure 7 have been presented as single MSCs earlier in this paper.

5.2 ACSystem

We have up to now only dealt with the ACContext, but now we want to go into the ACSystem instance (kind).

The ACSystem, defined by the MSC document in Figure 8, is an instance in ACContext. The

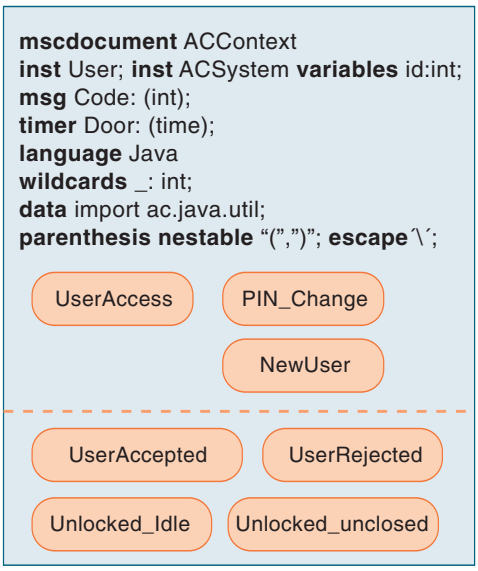


Figure 7 Mscdocument of the Access Control context

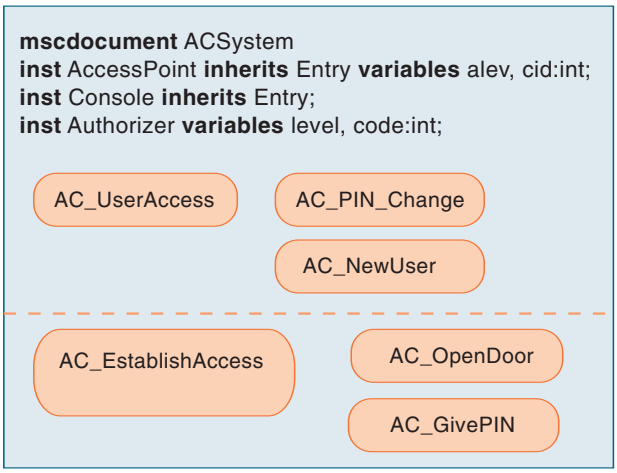


Figure 8 The Access Control System structure

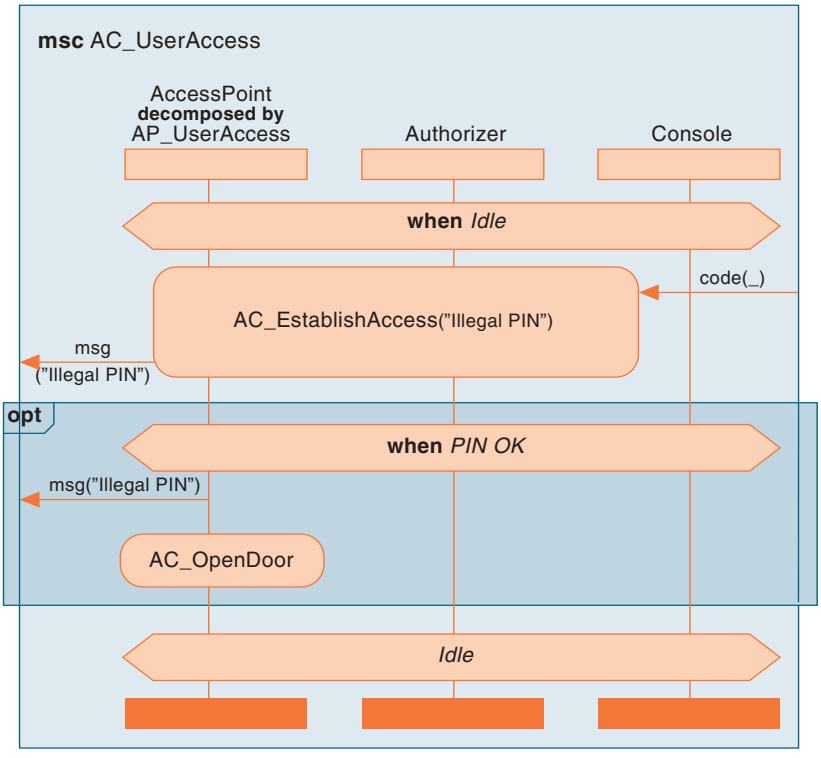


Figure 9 AC_UserAccess

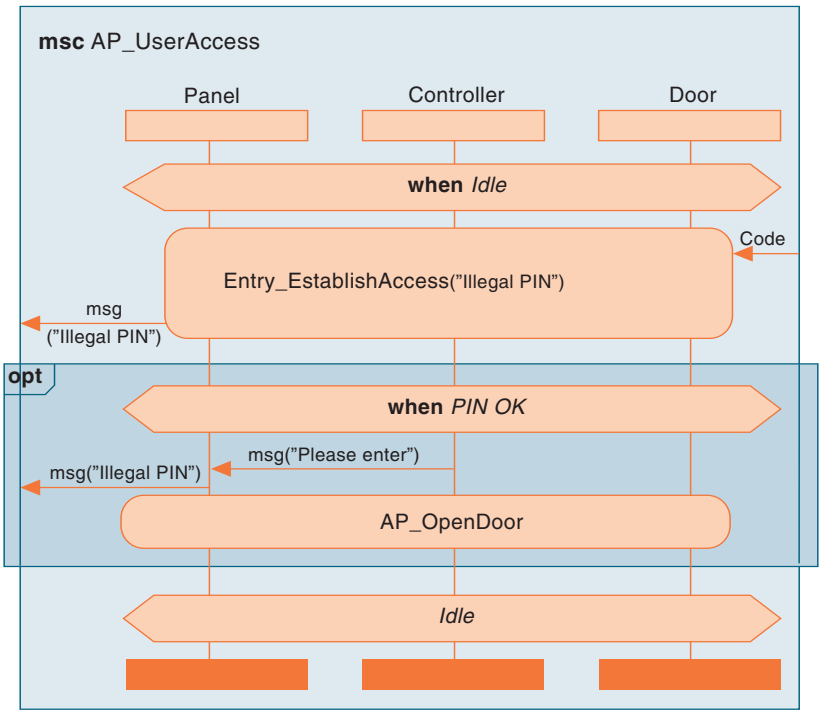


Figure 10 AP_UserAccess

MSC document of ACSystem can be defined implicitly by decomposing the ACSystem in every MSC diagram of ACContext. In Figure 9 we see that such decomposition is indicated syntactically by a decoration of the instance name. Associated with that decoration there is a reference to another MSC that represents the decomposition of that instance with respect to the MSC within which the decomposition appears. Thus the total decomposition of the instance is defined by the MSC document defining that instance.

For reasons that will become clearer later, we shall not make the diagrams of ACSystem (Figure 8) structurally equivalent to those of ACContext. In Figure 9 we show the user access scenario redefined independently within the ACSystem. The user access applies an MSC reference to an MSC that establishes the access rights of the user followed by an option specifying that if the user is eligible for the access zone he will be asked to enter.[2)] The option is a special alternative inline expression (see also Figure 4) where the second operand is empty.

In order to explain decomposition, we shall now go into the inner interactions of the AccessPoint with respect to the UserAccess service. According to the diagram in Figure 9 we will find this in the MSC AP_UserAccess described in Figure 10. From the decomposition in Figure 10 we see that an AccessPoint must contain at least the instances Panel, Controller and Door.

The decomposed instance can be understood as a sequence of language constructs of which some should be interpreted as gates relative to the decomposition diagram. Such gates are not only peer gates as we know from basic MSC (see Figure 2), but also decomposition specific gates such as MSC references, MSC reference expressions and inline expressions.

One major principle must be respected in decomposition. When an MSC reference covers the decomposed instance, the decomposition shall contain a corresponding global MSC reference. In Figure 9 AC_EstablishAccess covers AccessPoint and in Figure 10 Entry_EstablishAccess is the corresponding global MSC reference.

The same principle applies to inline expressions, a corresponding "extra-global" inline expression with the same operation and operand structure shall appear in the decomposition. An extra-global inline expression is an expression where the frame goes beyond the diagram frame indicating that the scope of the inline expression is beyond the scope of the diagram, typically from being decomposed.

Comparing the two **opt**-expressions of Figure 9 and Figure 10 we see that the structures of the inner operand correspond.

Regarding the decomposition of MSC references there is also a requirement of *commutativity* which in this context means that understanding the system through the reference first and then doing decomposition should result in the same interpretation as understanding the decomposition first and then the corresponding reference. In Figure 11 we show how commutativity works.

From AC_UserAccess (Figure 9) follow the MSC reference AC_EstablishAccess (Figure 12). We see that the one instance Entry is decomposed as Entry_EstablishAccess which is exactly what we would expect from the other route of decomposition and referencing given by decomposition of AccessPoint in AC_UserAccess to AP_UserAccess (Figure 10).

In AC_EstablishAccess we have not directly used AccessPoint, but rather Entry. This is acceptable as AccessPoint is a specialization of Entry as shown also in the MSC document ACSystem in Figure 8. The conceptual reason for this is that an AccessPoint and a Console will both have a means to control the entry of the user. The AccessPoint will control a physical

[2)] *Note that the **opt**-expression is beyond the MSC frame to indicate that the option has a larger scope than this diagram.*

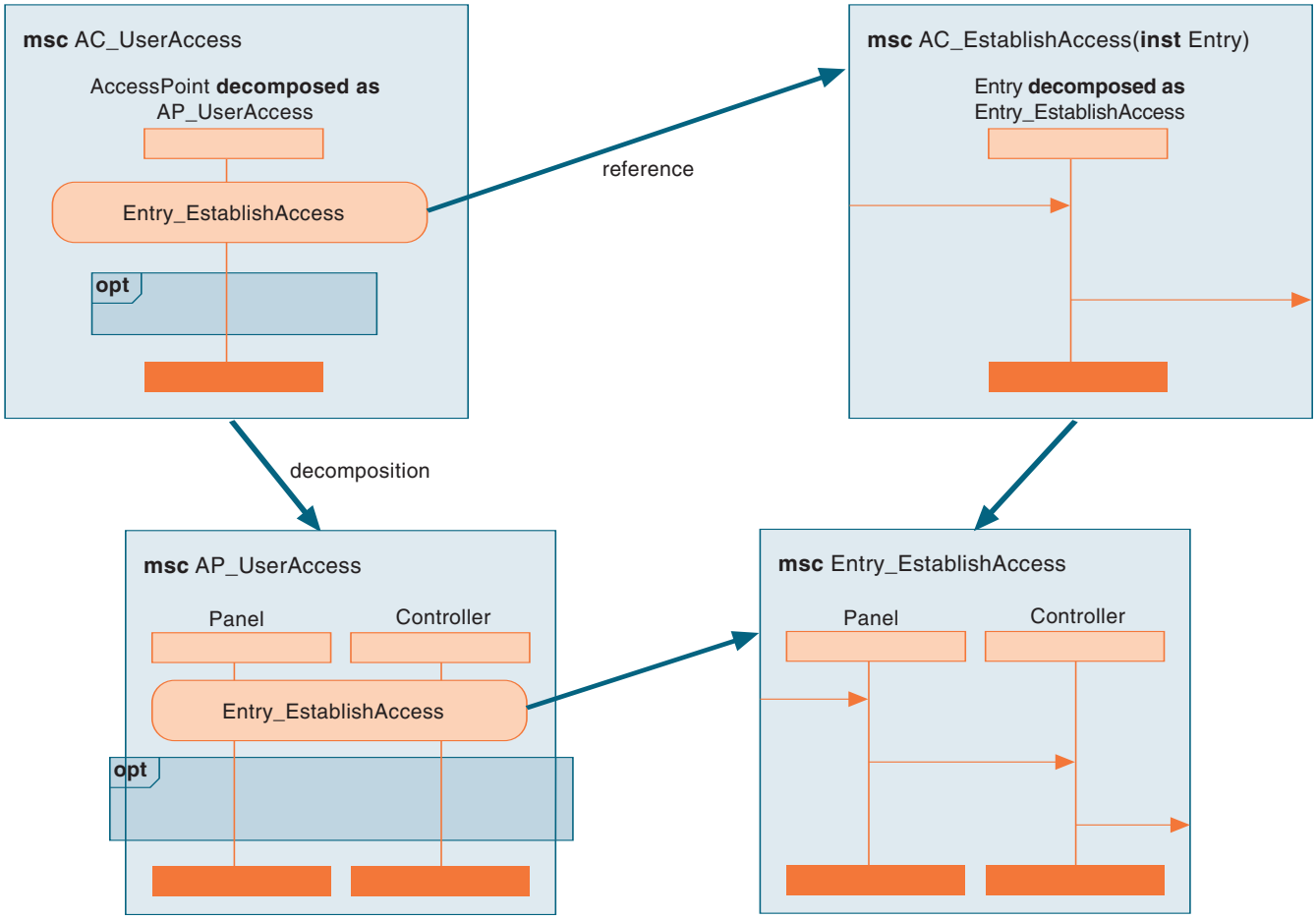


Figure 11 Commutative decomposition

door while the Console will control the entry to the database. Both share a CardReader and a Keypad.

Entry is also an *instance parameter* of AC_EstablishAccess (Figure 12). In AC_UserAccess (Figure 9) the actual instance parameter AccessPoint is given implicitly by the MSC reference covering the AccessPoint instance line.

Since there is no need to include instances with no events in a diagram, we need not have Door inside the diagram Entry_EstablishAccess.

5.3 Data Again

AC_EstablishAccess in Figure 12 can also be used to illustrate the similarities and differences between static and dynamic variables. The static variables are given in the parameter list of the diagram, in this case txt:String. This variable is not changed within the diagram and can therefore be applied anywhere within the diagram.

The dynamic variables are associated with an instance and defined in the MSC document. In Figure 8 we see that AccessPoint has variables cid and alev which are both integers. Authorizer has the integers level and cde. The message passing also indicates which variables are affected by and affecting the message through the parameter

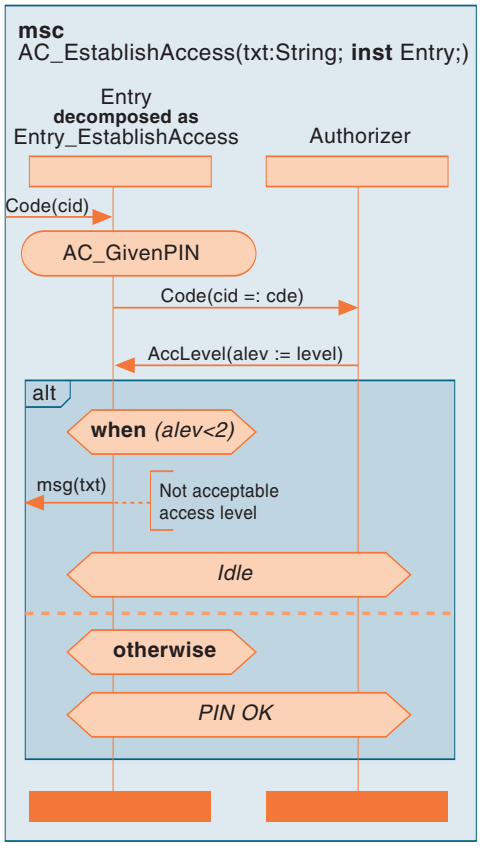


Figure 12 AC_EstablishAccess

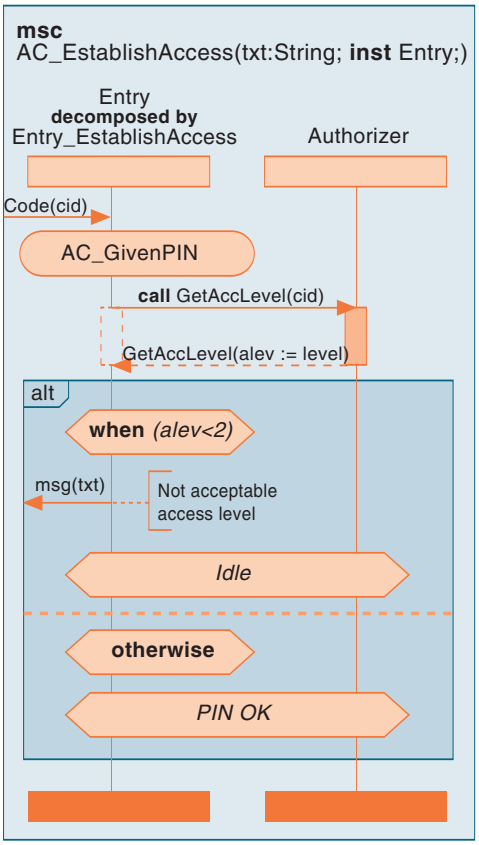


Figure 13 AC_Establish-Access with method call

binding given with the message. Complete messages are shown in Figure 12.

Messages that go to and from the environment will have only one end of the binding depending on the direction of the message. In Figure 12 the message msg(txt) is such a message that shows only the parameter value and not its pattern binding.

Initial conditions of MSC diagrams or of inline expression operands are called *guards*. Their text is preceded with the keyword **when**. When the dynamic variables are used in a guard, only dynamic variables of one instance should appear in the guard. This only instance must be the only instance that is ready to execute an event within that operand. In the example alev is the variable of Entry and in the first operand only Entry is ready to execute. The second operand is guarded by the **otherwise** guard which is equivalent to the situation where none of the other guards of the alternative hold.

6 Method Calls

In our case we may have realized that the authorization of the Code is very much a synchronized activity and that the AccessPoint will definitely wait for the result of the authentication before continuing. In MSC-2000 the use of method calls and suspension region are used to exactly specify this as shown in Figure 13.

7 Decomposing HMSCs?

In MSC-2000 there are still no mechanisms or requirements on how to decompose instances relative to an HMSC. We have seen above that decomposition is well defined for simple MSCs where the instances and messages etc. are all present. With HMSCs there is a difference. In an HMSC there are no explicit instances and messages. It is possible to specify the contained instances, but their relation to messages, references etc. is not explicitly defined in the HMSC.

In our example we have an overview HMSC description in Figure 3. There we are focusing on how the situation will be perceived by the User or an observer external to the MSC. User-Access will either result in the user being accepted or rejected. If the User is accepted there are three situations relating to how he goes through the door. If we had chosen to try and decompose the ACSystem from this overview diagram, we could have transformed the diagram to a simple MSC with inline expressions. This can always be done. Then we would have constructed the decomposition according to the structural static requirements mentioned above. This would not have resulted in the decomposition given in Figure 9.

The AC_UserAccess was created partly to illustrate some other points in MSC-2000. Furthermore it was designed from the perspective of the ACSystem. The ACSystem must specify why the distinction between the user being accepted and the user being rejected appears. This is why the AC_EstablishAccess is performed. Another reason for this sub-MSC is that it can be reused also in other services such as NewUser and ChangePIN.

We now have the following methodological choices:

1 Start from the overview diagram and transform to simple MSC and then construct the decomposition from that; or

2 Develop the more detailed view somewhat independently and then afterwards make sure that the results are compatible.

We have here chosen the latter approach. From an MSC language perspective the MSC documents ACContext and ACSystem will remain independent, but from a logical perspective (also formally) they are interrelated.

8 Summary

We have gone through some parts of an example using MSC. We have focused on those mechanisms that are new to MSC-2000 namely improved structuring of MSC documents, time and data, and method calls.

9 References

1 ITU. *Message Sequence Charts (MSC)*. Ø Haugen (ed.). Geneva, 1999. (Recommendation Z.120, p. 126.)

2 ITU. *Message Sequence Charts (MSC)*. E Rudolph (ed.). Geneva, 1993. (Recommendation Z.120, p. 36.)

3 ITU. *Message Sequence Charts (MSC)*. E Rudolph (ed.). Geneva, 1996. (Recommendation Z.120, p. 78.)

4 Bræk, R, Haugen, Ø. *Engineering Real Time Systems*. New York, Prentice Hall, 1993.

5 ITU. *Formal Semantics of Message Sequence Charts*. S Mauw et al. (eds.). Geneva, 1998. (Recommendation Z.120, p. 76.)

6 Bræk, R et al. *The Integrated Method – TIMe.* Trondheim, Sintef, 1997.

7 Haugen, Ø. Using MSC-92 Effectively. In: *SDL'95 with MSC in CASE. Proceedings of the Seventh SDL Forum, Oslo.* North-Holland, Elsevier, 1995.

8 Haugen, Ø. The MSC-96 Distillery. In: *SDL '97 Time for Testing. SDL, MSC and Trends. Proceedings of the Eighth SDL Forum.* Evry, Paris. North-Holland, Elsevier, 1997, 167–182.

9 Reniers, M A. *Message Sequence Chart : Syntax and Semantics*. Eindhoven, Eindhoven University of Technology, 1998.

10 Mauw, S, Reniers, M A. Operational Semantics for MSC'96. In: *Tutorials of the Eighth SDL Forum SDl'97: Time for Testing – SDL, MSC and Trends*. Cavalli, A and Vincent, D (eds). Evry, Institut national des télécommunications, 1997, 135–152.

11 Ek, A. Verifying Message Sequence Charts with the SDT Validator. In: *SDL '93 Using Objects. Proceedings of the Sixth SDL Forum*. Darmstadt, Germany. North Holland, Elsevier, 1993, 237–249.

12 Grabowski, J. *Test Case Generation and Test Case Specification with Message Sequence Charts.* Institut für Informatik und angewandte Mathematik, Universität Bern, 1994.

13 Nahm, R E M. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts.* Universität Bern, 1994.

14 ITU. *Specification and Description Language*. R Reed (ed.). Geneva, 1999. (Recommendation Z.100.)

A Tutorial Introduction to ASN.1 97

COLIN WILLCOCK

Colin Willcock (36) is Research Manager at Nokia Research Center. He received his BSc from Sheffield University in 1986, his MSc from Edinburgh University in 1987 and his Doctorate from the University of Kent at Canterbury in 1992. He is currently working on testing methodology, tool development and standardization of specification languages. He is a member of ETSI STF16 developing TTCN-3 and a member of the joint ISO/ITU-T ASN.1 Editors group. He is the ex-rapporteur for ITU-T Z.105 and current rapporteur for ASN.1 Encoding Control within ETSI MTS.

colin.willcock@nokia.com

ASN.1 is an established formal notation for defining data structures and message formats. The new versions of this language standardised in 1994 and 1997 includes a number of powerful new constructs, which improve and extend the existing notation. This paper provides an overview of basic ASN.1 notation, and provides a tutorial introduction to the new features.

1 Introduction

ASN.1 [1, 2, 3, 4] is already a widely used language with established supporting tools. It provides a means to formally specify data structures independent of machine or transfer syntax and is directly integrated into the SDL [5] and TTCN [6] languages. The integration with SDL enables the specification of associated semantics for the ASN.1 data structures, and the integration with TTCN enables direct testing of protocols defined using ASN.1. The typical application area for ASN.1 is in the structure definition of protocol messages. ASN.1 definitions can be combined with one of several standardised encoding rules which specify the physical representation (i.e. bit-pattern) of the abstract ASN.1 types. The combination of ASN.1 definitions and selected encoding rules can be used to automatically produce the required encoders and decoders for a protocol.

The purpose of this paper is to increase the use of the ASN.1 language. This paper is divided into two main parts. The first part – 'language basics' – is designed for those new to ASN.1. It attempts in a very concise manner to give an overview of the most important elements of the notation by providing simple examples and brief explanatory text. See Box 1.

The second part of this document – 'new features' – is designed for those who already have a grounding in the basics of ASN.1, but would like to understand and use the new features of the language introduced in the 1994 and 1997 versions of the standard.

To complete the ASN.1 picture, the last clause of the document covers some of the on-going work within the ASN.1 standardisation community to further extend the language into new areas.

2 New Features

Having considered the basics of the ASN.1 language in the other section, we now consider the new features of the language introduced in the latest versions of the standard.

2.1 Automatic Tagging

Each ASN.1 type has a tag associated with it. A tag gives a unique identity to the type, as explained below.

Within a message or syntax definition it is possible to specify structures which from the decoding point of view are ambiguous, for example consider the type below:

```
MyModule DEFINITIONS EXPLICIT TAGS ::=
BEGIN

   ProblemType ::= SEQUENCE {
      sometimes      INTEGER OPTIONAL,
                     -- Problem here
      always         INTEGER
   }

END
```

When receiving such a structure without some extra information, the decoder will not be able to tell whether the first integer value it decodes inside the sequence is the value of the field **sometimes** or that of field **always**.

In older versions of ASN.1, before 1994, the specifier would have solved such a problem by explicitly adding unique tags before the ambiguous fields. These tags would be visible in the encoding of the structure and therefore the decoder could use these tag values to distinguish between the fields.

```
ProblemType ::= SEQUENCE {
   sometimes      [0] INTEGER OPTIONAL,
                  -- [0] is a tag
   always         [1] INTEGER
                  -- [1] is another tag
}
```

Box 1 Language Basics

ASN.1 provides the ability to specify both type and value definitions, these are referred to as type notation and value notation respectively. These types and values can be either simple, based directly on the existing ASN.1 built-in types, or be structured, using the ASN.1 constructors.

• Examples of simple type definitions using ASN.1 built-in types are:

```
NumberOfElephants ::= INTEGER                                  -- use INTEGER to define positive and negative discrete values
RainInBochum ::= BOOLEAN                                       -- use BOOLEAN to define binary values
StudentState ::= ENUMERATED { sober, tipsy, drunk, inebriated } -- use ENUMERATED to define values with finite no. of states
ExactHeight ::= REAL                                           -- use REAL to define continuous values
AdultArtImage ::= BIT STRING                                   -- use BIT STRING to define raw data or bit maps
Data ::= OCTET STRING                                          -- use OCTET STRING to define binary data with length a
                                                                  multiple of 8
RudeWord ::= VisibleString                                     -- use the appropriate string type for character string values
```

• Example of simple value definitions using either built-in types or the defined types above are:

```
elephantsInTheRoom INTEGER ::= 0                               -- using builtin type INTEGER
elephantsOutside NumberOfElephants ::= 10                      -- using simple user defined type NumberOfElephants
rainingNow RainInBochum ::= TRUE                               -- unfortunately almost always true
stateOfJens StudentState ::= tipsy                             -- enumerated value
heightOfIna ExactHeight ::= {mantissa 16, base 10, exponent 1} -- REAL value definition
bitmap1 BIT STRING ::= '10011'B                                -- the B after the string means it's a bit value string (i.e. 1 or 0s)
messageData Data::= '08985550'H                                -- the H after the string means it's a hex value string (i.e. 0-9 A-F)
finnish RudeWord ::= "perkele"                                 -- a character string value uses " " as delimiters
```

Note that all type identifiers must start with a capital letter and all value identifiers must start with a lower case letter. The assignment symbol is '::=' and the notation requires no statement termination symbol.

ASN.1 also enables the definition of sub-types by the addition of constraints to type definition as shown below.

• Examples of sub-type definitions are:

```
HolidayDays ::= INTEGER(0..10)                                 -- The subtype may only take the values 0 to 10
ModelStudent ::= StudentState (sober | tipsy)                  -- The subtype may only take the values sober or tipsy
SmallImage ::= BIT STRING (SIZE (1..100))                      -- The length of the BIT STRING must be between 1 and 100
```

ASN.1 constructors are used to define structured types and values.

• Examples of structured types and values are:

```
-- use SEQUENCE type to define a record structure, i.e. a collection of variables of known number and definite order

FinnishStereotype ::= SEQUENCE {                               -- type definition
    numberOfRainDeer        INTEGER,
    hasMobilePhone          BOOLEAN OPTIONAL,                  -- field is optional
    location                ENUMERATED {helsinki, oulu, other}
}

marko FinnishStereotype ::= {                                  -- value definition
    numberOfRainDeer        1,
    hasMobilePhone          TRUE,
    location                other
}

-- use SEQUENCE OF to define a list or array, i.e. a collection of variables that have the same type, a definite order
-- and a large or unknown number
GirlFriends ::= SEQUENCE OF VisibleString                      -- type definition
firstLoves GirlFriends ::= {"Janice", "Lindsea", "Ally"}       -- value definition
```

Box 1 Language Basics, continued

```
-- ASN.1 also provides SET and SET OF constructors. These are analogous to SEQUENCE and SEQUENCE OF

-- respectively except the collection of variables has no implicit ordering

-- use CHOICE to define a variable selected from a known collection of possible variables

MobilePhone ::= CHOICE {                                    -- type definition
      nokia                     INTEGER,
      ericsson                  INTEGER,
      none                      NULL                        -- NULL means no associated component
}

myHandy MobilePhone ::=         nokia : 8210                -- value definition
markosHandy MobilePhone ::=     nokia : 9110
jensHandy MobilePhone ::=       none
```

The presented ASN.1 simple and structured types can be arbitrarily nested to build up the required message or data structure. The required collection of ASN.1 type and value definitions are grouped together using ASN.1 modules; these provide a mechanism for structuring and referencing.

• Example of an ASN.1 module:

```
MyModule DEFINITIONS ::=
BEGIN
                                                            -- ASN.1 definitions go here
END
```

Automatic tagging is a feature which, when used, frees the specifier from ever having to explicitly add or even consider tagging within a message or syntax specification.

The process of manually adding tags is no longer necessary. The specifier must simply select AUTOMATIC TAGS as the tag default in the module header and the tags will be automatically generated as and when they are needed by a set of language transformations.

```
MyModule DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

  ProblemType ::= SEQUENCE {
    sometimes   INTEGER OPTIONAL,
                -- [0] magically associated with sometimes
    always      INTEGER
                -- [1] magically associated with always
  }

END
```

2.2 Extensibility

Extensibility is a mechanism to provide forward and backward compatibility between different versions of an ASN.1 definition. Extensibility can be specified locally by use of the extension marker or globally by using the optional module header field EXTENSIBILITY IMPLIED. The extension marker '...' is normally used in the definitions of ENUMERATED, SEQUENCE and CHOICE types.

If we consider the example shown in Figure 1, the use of the ASN.1 extensibility mechanism allows the specification of the CallMsg such that the phones running different versions of the protocol can still communicate with each other. The first version of the protocol uses the extension marker to indicate that this type is expected to be extended in some later version. In version 1.1 this does indeed happen, and the new extension additions (any number are allowed) are enclosed in the '[[' and ']]' version brackets. In the protocol version 2.0 we can see that there is a further extension addition.

The extension mechanism works by allowing the decoder to skip over unknown additions. If we consider the case of Figure 1, the entity in the middle, running protocol version 1.1 will receive messages from the entity to the left which has fewer components in the sequence than defined in its version of the protocol. Because of the presence of the extension marker the decoder will know that the last expected field (cost) is an extension addition and therefore treat this in a similar way to an optional field, accepting the message and passing it to the application.

In the case where the entity in the middle receives a CallMsg from the entity on the right running version 2.0, there will be more components in the message than specified in its protocol specification. Because of the presence of the extension marker, the decoder knows that there is a possibility it might receive such a message from a later protocol version and although it cannot decode that later part, it can skip over it and continue decoding the parts of the protocol it does know.

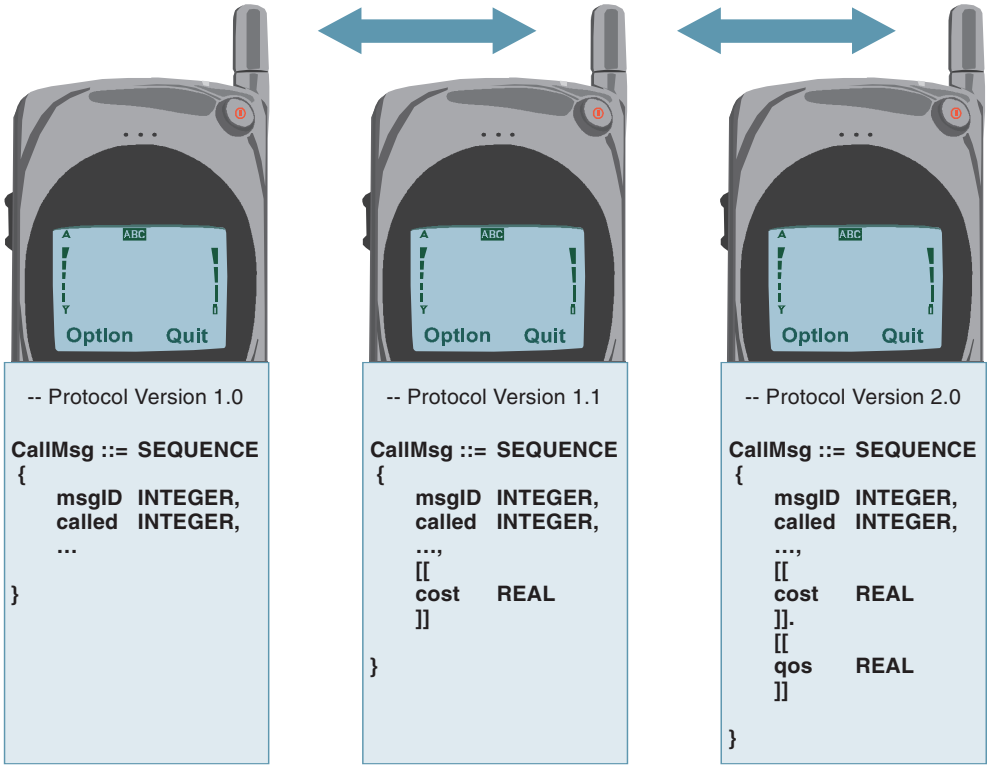


Figure 1 ASN.1 Extensibility mechanism

Although the ASN.1 extension mechanism enables forward and backward compatibility from a decoding point of view, the actual application which calls the ASN.1 decoder must also be written in such a way that it can handle the presence or absence of extra elements after the extension marker.

2.3 Exception Identifier

The exception identifier provides the ability to indicate to an application above the ASN.1 decoder where within the abstract syntax a decoding error occurred. The exception identifier consists of the symbol '!' followed by either a type and a value of that type or in the absence of a type, an integer value. The exception identifier can be associated with any constraint or extension marker, for example:

```
dayError INTEGER ::= 1

DateType ::= SEQUENCE {
   year        INTEGER (1970..2001),
   month       INTEGER (1..12),
   day         INTEGER (1..31 ! dayError)
               -- if decoding fails on this field send dayError value
}              -- to the application
```

When an exception identifier is present in a specification, the associated standard or application documentation should state the required functionality of the application on reception of this value.

2.4 New String Types

The new string types BMPString, UniversalString and UTF8String allow the specification of character strings based on the ISO/IEC 10646 standard [7]. This allows the use of a huge number of symbols from many languages and nations. Because the character set is so large, four octets are normally needed to identify a particular character, as opposed to a single octet in normal ASCII. To allow for this, a new value notation has been added to ASN.1 for these string types, this allows the definition of named values from the character set. For example:

```
LatinCapitalLetterA BMPString ::= { 0, 0, 0, 65}
```

UniversalString includes the entire ISO/IEC 10646 character set with each character encoding to four octets. BMPString is a subset of UniversalString which only includes the characters from the Basic Multilingual Plane. BMPString characters, because they are a subset of the full ISO/IEC 10646 character set, can be encoded into two octets. UTF8String includes the entire character set, but allows variable encoding lengths for characters, depending on which part of the character set the character is from. The encoding is such that the most often used characters are encoded in the smallest space and the least used characters are encoded in the largest space [8].

2.5 Information Objects

Information objects can be considered a generic table mechanism which allows the association of specific sets of field values or types. Information objects replace the macro notation and the ANY DEFINED BY construct present in older versions of ASN.1 and have the advantage that they are directly machine processable.

Typically information objects are used to model protocols. They enable in a specification to define that the value of a particular field will determine the structure of the following fields. For example given a simple protocol message which consists of a message type, followed by indication of what to do if the message type is unknown by a receiver, followed by the message data,

where the type of the message data is dependent on the message type, we could define the following information object class:

```
-- Definition of Information Object Class

MESSAGE ::= CLASS {
   &msgCode          INTEGER UNIQUE,
                     -- INTEGER type value field
   &msgCriticality   ENUMERATED,
                     { ignore, report, reject }
                     -- ENUMERATED type value field
   &MsgData          OPTIONAL
                     -- Open type field
}
WITH SYNTAX {
   CODE              &msgCode
   CRITICALITY       &msgCriticality
   [DATA TYPE        &MsgData]
}
```

This defines the information object class MESSAGE, which is like a template for all possible messages within our simple protocol. This information object class has three fields msgCode, msgCriticality and MsgData, these represent the three parts of the protocol messages. The msgCode field is a value field of type integer in which the message type is stored. The unique keyword signifies that all instances of this information class will have a unique value for this field (i.e. message type). The msgCriticality field is a value field of type enumerated and the last field MsgData is a type field – i.e. a field which can contain a type. The WITH SYNTAX clause at the end of the definition merely provides an user friendly syntax for defining instances of this information class.

Information objects are often best visualised as a table. The table associated with the MESSAGE information Object class is shown in Table 1.

Once the overall information object class has been defined, information object instances can be specified to represent the constituent messages in the protocol. For instance the information object definitions below specify the four messages of our simple example protocol:

```
-- Information Object Definition

setup      MESSAGE ::= {
           CODE              1
           CRITICALITY       reject
           DATA TYPE         OCTET STRING
}

setupAck           MESSAGE ::= {
           CODE              2
           CRITICALITY       report
           DATA TYPE         INTEGER
}

release    MESSAGE ::= {
           CODE              3
           CRITICALITY       reject
}

relAck     MESSAGE ::= {
           CODE              4
           CRITICALITY       ignore
}
```

The information objects can be organised into information object sets which can represent all the messages in a protocol or subsystem, for instance:

```
-- Information Object Set Definition

ConnectPhaseMsgs        MESSAGE ::= {
   setup                |
   setupAck             |
   release              |
   relAck               ,
   ...
                           -- Other messages can be added
}
```

The associated table for the ConnectPhaseMsgs information object set is shown in Table 2.

So far we have defined the information object class which provides a protocol template and then used this class to define information objects representing the individual messages in the protocol. These information objects have then been brought together into an information object set which represents the entire set of protocol messages.

Information objects are just collections of information. Information objects cannot be transmitted, for that purpose we need separate type definitions. Therefore, we need to relate the information object definitions to an actual overall ASN.1 type which can be directly used in message transfer. The associated ASN.1 type definition for the simple example protocol is:

```
-- Associated PDU type definition for MESSAGE Information Object
   Class

ConnectPhasePDU ::= SEQUENCE{
   id                MESSAGE.&msgCode,
                     -- INTEGER field
   criticality       MESSAGE.&msgCriticality
                     -- ENUMERATED field
   data              MESSAGE.&MsgData OPTIONAL
                     -- open type field
}
```

This definition defines the type ConnectPhasePDU which can contain any message which conforms to the MESSAGE informa-

Table 1 Table template for MESSAGE *object class*

Field	msgCode	msgCriticality	MsgData
Type	INTEGER	ENUMERATED	OPEN TYPE

MsgCode	msgCriticality	MsgData
1	reject	OCTET STRING
2	report	INTEGER
3	reject	-
4	ignore	-

Table 2 Associated table for ConnectPhaseMsgs *object set*

tion class (i.e. could be defined as an information object of this class).

The full power of information objects and information object sets requires the use of tabular and relational constraints which are described in the following section.

2.6 Constraints

The newer versions of the ASN.1 language provide three new forms of constraint specification: *user defined constraints, tabular constraints and component relational constraints*.

User defined constraints are used to informally specify constraints which are too complex to define using any of the other existing ASN.1 constructs; they provide a syntax to specify an extended comment:

```
– Example of user defined constraint

SMIMStatus ::= OCTET STRING ( CONSTRAINED BY {-- each
octet must have pattern 1010 for bit 2-5 --})
```

The user defined constraint is specified using the keywords CONSTRAINED BY; the associated curly brackets contain the specification for the constraint. The contents of the curly brackets is considered to be just a special form of comment.

Tabular constraints provide a way of limiting the contents of a field associated to an information object class to those contained within a specified object set. For example if we consider the previously defined type ConnectPhasePDU we can constrain the allowed values of this type by applying a tabular constraint to its constituent fields.

```
-- Associated PDU type definition for MESSAGE Information Object
-- Class with tabular constraint

ConnectPhasePDU ::= SEQUENCE{

   id                MESSAGE.&msgCode
                     ({ConnectPhaseMsgs}),
                     -- constrained to 1,2,3 or 4

   criticality       MESSAGE.&msgCriticality
                     ({ConnectPhaseMsgs}),

   data              MESSAGE.&MsgData
                     ({ConnectPhaseMsgs}) OPTIONAL
                     -- OCTET

                     -- STRING/INTEGER

}
```

In this example each field may only take the values present in the specified information object set for that field. For example the field id which is associated with the MESSAGE class field &msgCode may only have the values in the information object set ConnectPhaseMsgs for that field, i.e. 1, 2, 3 or 4.

Constraints using information object sets can be taken one step further by using component relational constraints. Relational constraints provide a way of constraining the contents of one field depending on the value of a second field. For example to constrain the values of criticality and data to be consistent with the associated value of the id field defined within the information object set ConnectPhaseMsgs, the connectPhasePDU type definition can be extended as follows:

```
-- Associated PDU type definition for MESSAGE Information Object
   Class with Relational constraint

ConnectPhasePDU ::= SEQUENCE{

   id                MESSAGE.&msgCode
                     ({ConnectPhaseMsgs}),

   criticality       MESSAGE.&msgCriticality
                     ({ConnectPhaseMsgs}{@id}),

   data              MESSAGE.&MsgData
                     ({ConnectPhaseMsgs}{@id}) OPTIONAL

}
```

The @id is the relational constraint in this example. It links the value of the stated field (id) to the contents of the field containing the relational constraint, ensuring that the two are consistent with the information object set definition, i.e. if the id field has the value 1 then the criticality field is constrained to the value reject and the data field must be of type OCTET STRING. More generally relational constraints can be best visualised with reference to the table associated with the information object set. The relational constraint is equivalent to selecting only the values of one (or more) rows within the table.

2.7 Parameterisation

The ASN.1 language now provides a general parameterisation mechanism which allows for example value and type parameters for both type and value notation. The formal parameter list is specified after the identifier in the definition, and the actual parameter list is specified when the definition is referenced.

Value parameterisation allows the passing of a value of a defined type, this can be used to complete a value definition or provide constraint values for type definitions, for example:

```
-- Value parameterisation in value definitions

-- Value definition with value parameter
genericGreeting{ IA5String : name}  IA5String ::= {"Hello", name}

-- Use of parameterised value in value assignment

firstString IA5String ::= genericGreeting{ " World" }
            -- assign value "Hello World"

-- Value parameterisation in type definitions

-- Type definition with value parameters
MyMessage{ INTEGER : maxSize, INTEGER : minSize} ::=
SEQUENCE
{
   asp      INTEGER,
   pdu      OCTET STRING( SIZE( minSize.. maxSize))
            -- limit size to be within bounds
}

-- Use of Parameterised type definition. MyMessage is instantiated
   with different actual parameters.

MyLargeMessage ::= MyMessage{10000, 1}
MySmallMessage ::= MyMessage{10, 1}
```

In addition to value parameters, ASN.1 supports type parameters. Type parameterisation allows the passing of a type into a definition, for example:

```
-- Type parameterisation in type definitions

-- Type definition with type parameter
GenericMessage{ MsgDataType} ::= SEQUENCE
{
   asp      INTEGER,
   pdu      MsgDataType
            -- this field will be of the type passed in as a parameter

}

-- Use of Parameterised type definition
SetupMessage ::= GenericMessage{ OCTET STRING}
```

3 Future Developments

This paper has described the new features introduced into the ASN.1 language in the 1994 and 1997 versions of the standard. To complete the ASN.1 overview this section briefly considers some of the areas which are currently under development within the ASN.1 standardisation community and will likely become additions to the ASN.1 language in the future.

One area where extensions are being worked on is constraints for string types. The current proposal is to introduce a constraint scheme which is similar to regular expressions. This would allow string constraints like:

```
– Possible future string constraints
MyString ::= IA5String(PATTERN "train..to.*" )
```

The allowed values for MyString would be limited to strings that match the regular expression defined within the constraint. In this case a matching string would start with the five characters "train", followed by any two characters, followed by the two characters "to", followed by any string of any length.

These new string constraints will be very useful in the specification of new text based protocols, which form a major part of many web based technologies.

Another area which at present is under development is encoding control notation [9]. Currently there is a gap in the formal specification techniques relative to the definition of encoding rules. This means that although the abstract structure of a protocol message can be formally defined, the actual format of the bits transmitted on the line or through the air cannot be formally specified. There is a small number of standardised encoding rules (e.g. Basic Encoding Rules – BER [10], and Packed Encoding Rules – PER [11]) but in many application domains such as radio interfaces, these do not satisfy today's requirements for very efficient transmission of information. The development of advanced encoding control will extend the use of ASN.1 in protocol specifications which presently use other less formal notations due to the implied encoding restrictions of the current language.

References

1 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.* Geneva, 1998. (ITU-T Recommendation X.680 (1997). ISO/IEC 8824-1:1998.)

2 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.* Geneva, 1998. (ITU-T Recommendation X.681 (1997). ISO/IEC 8824-2:1998.)

3 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.* Geneva, 1998. (ITU-T Recommendation X.682 (1997). ISO/IEC 8824-3:1998.)

4 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Parameterisation of ASN.1 specifications.* Geneva, 1998. (ITU-T Recommendation X.683 (1997). ISO/IEC 8824-4:1998.)

5 ITU. *Programming Languages – CCITT Specification And Description Language (SDL)*. Geneva, 1993. (ITU-T Recommendation Z.100.)

6 ISO. *Information technology – Open System Interconnection Conformance testing methodology and framework Part 3: The Tree and Tabular Combined Notation (TTCN)*. Geneva, 1994. (ISO/IEC 9646-3:1994.)

7 ISO. *Information technology – Universal Multiple-Octet Coded Character Set (UCS): – Architecture and Basic Multilingual Plane.* Geneva, 1993. (ISO/IEC 10646-1:1993.)

8 ISO. *Information technology – Universal Multiple-Octet Coded Character Set (UCS): – Architecture and Basic Multilingual Plane – Amendment 2: UCS Transformation Format 8 (UTF-8)*. (ISO/IEC 10646-1:1993/ Amd.2:1996.)

9 Willcock, C. New Directions in ASN.1: Towards a Formal Notation for Transfer Syntax. In: G Csopaki, S Dibuz, K Tarnay (eds.). *Testing of Communicating Systems Methods and applications, Volume 12*. Kluwer, 1999, 31–40.

10 ITU-T. *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* Geneva, 1998. (ITU-T Recommendation X.690 (1997). ISO/IEC 8825-1:1998.)

11 ITU-T. *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).* Geneva, 1998. (ITU-T Recommendation X.691 (1997). ISO/IEC 8825-2:1998.)

Further Reading

Larmouth, J. *ASN.1 Complete*. Morgan Kaufmann, 1999. (ISBN 0-12233-435-3) (http://www.oss.com/asn1/larmouth.html)

Dubuisson, O. *ASN.1 – Communication between heterogeneous systems.* Morgan Kaufmann, 2000. (ISBN 0-12-6333361-0.) (http://www.asn1.elibel.tm.fr)

CHILL 2000

JÜRGEN F.H. WINKLER

Jürgen F.H. Winkler (57) has since 1993 been a full professor of Computer Science at the Friedrich Schiller University in Jena, Germany. His main interests are program correctness, object-orientation, programming languages and their implementation. Before joining the university he was with the corporate research of Siemens AG in Munich. Among other projects he has been involved in the definition and implementation of Object-CHILL, a forerunner of CHILL 2000, and with the Siemens Ada compiler. He also founded the "International Workshop on Software Configuration Management". Dr. Winkler received his PhD and his Diploma, both in Computer Science, from the University of Karlsruhe, Germany.

jwinkler@acm.org

CHILL is a programming language mainly used in the area of telecom systems. This paper gives an overview of the language elements of CHILL and reports in more detail on new language elements which have been added recently, especially object-orientation and genericity.

1 Introduction

This paper gives a tutorial overview of CHILL, the ITU-T Programming Language [1]. CHILL is an acronym with the original long form "CCITT High Level Language", which reflects the fact that ITU-T was formerly called CCITT.

CHILL has been originally developed in CCITT during the period 1975 – 1983. After this, it has been continuously updated and used for the development of many telecom systems around the world [2]. This paper also contains more details about the history and application of CHILL.

Today CHILL is a modern object-oriented language, which also supports concurrency in an object-oriented manner. In the last Study Period (1997–2000) the following language elements have been included:

- Interfaces;
- Support of Unicode;
- Friend-procedures;
- Overloading of procedures;
- Final (unmodifiable) components in objects.

In the body of the paper we give an overview of the language elements of CHILL and describe in more detail those elements of CHILL that were added more recently.

In this paper we use the typical terminology of the field of programming languages, especially for basic terms. CHILL, as many other languages, has a number of specific terms. Especially for the following terms we use the traditional terminology:

type	"mode" in CHILL
variable	"location" in CHILL
statement	"action" in CHILL

2 Language Overview

CHILL is a procedural and object-oriented language, which contains a number of elements that support the development of large programs, as they are typical for the telecom field. The following tree shows the language elements of CHILL 2000.

Data Structures
  Scalar: integer, float, characters, Boolean, enumerations, pointer, procedure type, process type, event, time;
  Composite: string, record, array, set, buffer, signal.

Sequential Programming
  Variable, constant, expression, function call;
  Assignment;
  Procedure call;
  EXIT, RESULT, RETURN, GOTO;
  Statement sequence;
  Selection statements: IF, CASE (multidimensional);
  Repetition statements: DO, WHILE, FOR.

Object-oriented Programming
  Sequential, unsynchronized object;
  Sequential, synchronized object;
  Concurrent, synchronized object;
  Interface;
  Friend.

Concurrent Programming
  Process;
  Start process;
  Communication via buffer;
  Communication via signal;
  Critical region and co-ordination with events;
  Concurrent, synchronized object.

Program Structure
  Block;
  Procedure / Function / Process;
  Object-Type / Class;
  Module / Region.

Genericity
  Generic Procedure / Process;
  Generic Module / Region;
  Generic Object Type / Class;
  Generic Interface.

Program Verification
  Precondition and postcondition for methods;
  Invariant for object type / class;
  ASSERT statement.

Box 1 contains a number of small examples for most of the elements listed above, in order to give the reader some impression of CHILL 2000 as a programming language.

Box 2 contains a comparison of CHILL2000 and Java based on the tree structure of the overview on CHILL given above. If one of the languages does not contain a certain element the corresponding entry is empty (e.g. “Genericity” in Java).

3 New Elements in CHILL 2000

During the last two study periods (1993–1996, 1997–2000) new and important language elements have been added to the language. The most important of them are:

- Object-Orientation;
- Genericity.

3.1 Object-Orientation

Object types, which are typically called classes in the area of object-orientation [3, 4], come in CHILL 2000 in four different flavors

- *Module type*
  An object (or instance) of such a type has the typical properties of a module. It has components, which can be public or internal, and it does not do any co-ordination in case of concurrent accesses to its components. With respect to concurrency module objects are passive, i.e. they do not have an own thread of control.

- *Region type*
  An object (or instance) of such a type has the typical properties of a region. It has components, which can be public or internal, and it co-ordinates concurrent accesses to its components. With respect to concurrency, region objects are passive, i.e. they do not have an own thread of control.

- *Task type*
  An object (or instance) of such a type has a similar structure as module and region objects. It has components, which can be public or internal. With respect to concurrency it has its own thread of control and it co-ordinates concurrent accesses to its components. It is therefore similar to task objects in Ada [5] and this is the reason for its name.

- *Interface type*
  An interface type defines an interface, which consists of the specification of public components. There are no objects of interface types. Interface types are typically used as base types of other object types.

Together the new object types are called moreta types, where moreta has been formed from the first letters of module, region, and task.

A common characteristic is that the definition of a non-interface moreta type (= class) consists of a specification part and a body. This separation is very useful from a software engineering point of view. The interface describes what a user (client) of the given type must know in order to use the type or its objects. The body contains the internal implementation of the components specified in the interface.

As an example we look at the definition of a stack type.

```
SYNMODE IntStackType1 = MODULE SPEC
   GRANT Push, Pop;
   Push: PROC(Elem INT IN)
         EXCEPTIONS(Overflow)  END Push;
   Pop:  PROC( ) RETURNS(INT)
         EXCEPTIONS(Underflow)  END Pop;
   SYN Length = 10_000;
   DCL StackData ARRAY (1:Length) INT,
         TopOfStack RANGE(0:Length) INIT := 0;
END IntStackType1;
```

The type IntStackType1 is defined like other types in CHILL. The keyword MODULE indicates that it is a module type and the keyword SPEC indicates that it is the specification part of this type. The procedures (methods) Push and Pop are exported and are therefore public components of IntStackType1. Length, StackData, and TopOfStack are internal components. This is an example of encapsulation and is necessary to guarantee the stack protocol.

The corresponding body contains in this case the bodies of the two procedures.

```
SYNMODE IntStackType1 = MODULE BODY
   Push: PROC(Elem INT IN) EXCEPTIONS(Overflow)
      IF TopOfStack = Length
      THEN CAUSE Overflow;
      ELSE  TopOfStack +:= 1;
            StackData(TopOfStack) := Elem;
      FI;
   END Push;
   Pop: PROC( ) RETURNS(INT) EXCEPTIONS(Underflow)
      IF TopOfStack = 0
      THEN CAUSE Underflow;
      ELSE  RESULT StackData(TopOfStack);
            TopOfStack -:= 1;
      FI;
   END Pop;
END IntStackType1;
```

Objects of the type IntStackType1 are declared in the same way as variables for the traditional types. The manipulation of these variables is done in the typical style of object-orientation.

```
DCL Stack1, Stack2 IntStackType1;
Stack1.Push(10);
Stack2.Push(100);
. . . Stack1.Pop(). . .
```

Stack1 and Stack2 are adequate for sequential programming. It is now quite easy to define a stack type CIntStackType1 whose objects co-ordinate concurrent calls of their methods. In CHILL there are two ways to accomplish this:

a) change the keyword MODULE into REGION

```
SYNMODE CIntStackType1 = REGION SPEC
     /* same as before */
END CIntStackType1;
```

And analogously for the body.

b) derive the type CIntStackType1 from the existing type IntStackType1.

```
SYNMODE CIntStackType1 = REGION SPEC
    BASED_ON IntStackType1
END CIntStackType1;
```

Since there are different kinds of object types there exist several possibilities for the derivation of types from base types.

- A class can be directly derived from one base class (single inheritance between classes);
- A class can be directly derived by combining an arbitrary number of base interface types (multiple inheritance between interfaces and classes);
- An interface type can be derived from an arbitrary number of base interface types (multiple inheritance between interfaces).

These conditions can be summed up to the rule that CHILL uses single inheritance for classes and multiple inheritance for interfaces.

Since module, region and task differ in their properties, the following derivation constraints have to be observed:

Base type:	Permissible derived type:
module	module, region, task
region	region
task	task

The derivation mechanism of object-orientation is a mechanism for the realization of structural polymorphism. A derived type DT and its objects contain the components inherited and possibly additional components defined in DT. As an example, we define a stack type IntStackType2, which is derived from IntStackType1 but contains the additional function Top() (INT) which returns the value of the topmost element, but does not change the contents of the stack.

```
SYNMODE IntStackType2 = MODULE SPEC
                        BASED_ON IntStackType1
   GRANT Top;
   Top: PROC( ) RETURNS(INT) EXCEPTIONS(Underflow)
   END Top;
END IntStackType2;

SYNMODE IntStackType2 = MODULE BODY
                        BASED_ON IntStackType1
   Top: PROC( ) RETURNS(INT) EXCEPTIONS(Underflow)
      IF TopOfStack = 0
      THEN CAUSE Underflow;
      ELSE RETURN StackData(TopOfStack);
      FI;
   END Top;
END IntStackType2;
```

3.2 Genericity

The stack is a good example to demonstrate the concept of genericity. In section 3.1 the element type of the stack is INT. If we need stacks with other element types, we have to duplicate or in general replicate the code for each new element type. From a software engineering point of view, this code replication is very unwelcome. There are two ways to try to avoid this problem.

a) Use “REF UltimateBaseType” as the element type of the stack type. If the language does not have an ultimate base type, an appropriate base type has to be used.

```
SYNMODE IntStackType3 = MODULE SPEC
    GRANT Push, Pop, ElemType;
    SYNMODE ElemType = REF UltimateBaseType;
    Push: PROC(Elem ElemType IN)
          EXCEPTIONS(Overflow) END Push;
    Pop:  PROC( ) RETURNS(ElemType)
          EXCEPTIONS(Underflow) END Pop;
    SYN Length = 10_000;
    DCL StackData ARRAY (1:Length) ElemType,
          TopOfStack RANGE(0:Length) INIT := 0;
END IntStackType3;
```

The body of IntStackType3 is essentially the same as that of IntStacktype1. The difference is in the identifiers IntStackType3 and ElemType.

The objects of IntStackType3 are now heterogeneous stacks, i.e. due to polymorphism, they may contain objects of different types.

```
DCL Stack3 IntStackType3;
Stack3.Push(new IntStackType1);
Stack3.Push(new IntStackType2);
Stack3.Push(new IntStackType3);
```

b) If we want to have homogeneous stack objects, as those of the types IntStackType1, IntStackType2, or CIntStackType1 are, genericity (or parametric polymorphism) is the right mechanism to use. A generic entity is an entity which is parameterized in a more general way than traditional procedures. In CHILL the following entities may be used as parameters of a generic entity:

values of arbitrary types;
types;
procedures and functions.

It is especially the possibility to use types as parameters which provides new possibilities for the formulation of programs.

The use of genericity is typically done in two steps:

i) define a generic entity, i.e. an entity which has formal generic parameters. Such a generic entity is a template for more specific entities.

ii) define an instantiation of the generic template by providing actual generic parameters for the formal ones.

A generic stack type may now look as follows:

```
GenericStackTemplate1:
    GENERIC MODE ElemType = ANY_ASSIGN;
MODULE SPEC
    GRANT Push, Pop;
    Push: PROC(Elem ElemType IN)
          EXCEPTIONS(Overflow) END Push;
    Pop:  PROC( ) RETURNS(ElemType)
          EXCEPTIONS(Underflow) END Pop;
    SYN Length = 10_000;
    DCL StackData ARRAY (1:Length) INT,
    TopOfStack RANGE(0:Length) INIT := 0;
END GenericStackTemplate1;
```

As for IntStackType3, the body of GenericStackTemplate1 is essentially the same as that for IntStackType1.

GenericStackTemplate1 has one formal generic parameter, ElemType, which is of the kind ANY_ASSIGN. This means that variables of type ElemType can be assigned inside the definition of GenericStackTemplate1. This property is needed in the bodies of Push and Pop. On the other hand, any type which is used as a corresponding actual generic parameter must at least support the operation of assignment. This guarantees that any legal instantiation will produce a legal type.

Using GenericStackTemplate1, we obtain non-generic stack types by instantiating the template with an actual generic parameter. If we use INT as actual generic parameter, we obtain an object type which is essentially equivalent to IntStackType1.

```
SYNMODE IntStackType4 = NEW GenericStackTemplate1
    SYNMODE ElemType = INT;
END IntStackType4;
```

If we use FLOAT as actual generic parameter we obtain a type FloatStackType whose objects can only take float values as elements.

```
SYNMODE FloatStackType = NEW GenericStackTemplate1
    SYNMODE ElemType = FLOAT;
END FloatStackType;
```

After having created two generic instantiations of the template GenericStackTemplate1 we see that with genericity the code duplication is avoided.

We see that both structural polymorphism (through inheritance) and parametric polymorphism (through genericity) are very useful mechanisms for the formulation of programs.

4 Use of CHILL in Telecom Systems

Since its birth, CHILL has been used quite widely in the world of telecommunications. Rekdal mentions about 13 companies [2], and if we account for the fact that several companies in Korea have built systems using CHILL, we can say that about 15 significant companies in the telecom field have built systems using CHILL. Since large companies as e.g. Alcatel and Siemens sell their systems all over he world, CHILL is passively used by hundreds of millions of people. In Germany for example, the conventional telephone network is essentially based on systems written in CHILL. There are mainly two systems used: EWSD from Siemens and System12 from Alcatel.

A lot more details about these aspects of CHILL are given in [2].

References

1 ITU-T. *CHILL – The ITU-T programming language).* ITU, Geneva, 1999. (Recommendation Z.200 (11/99.) (http://www.itu.int/itudoc/itu-t/approved/z/z200.html)

See also: ISO/IEC 9496:1998 CCITT high level language (CHILL). http://www.iso.ch/cate/d30537.html

2 Rekdal, K. CHILL – the international standard language for telecommunications programming. *Telektronikk*, 89 (2/3), 5–10, 1993.

3 Dahl, O, Myhrhaug, B, Nygaard, K. *Common Base Language.* Oslo, Norwegian Computing Center, 1970.

4 Goldberg, A, Robson, D. *Smalltalk-80 – The Language.* Reading, Mass., Addison Wesley, 1989. (ISBN 0-201-13688-0)

5 ISO/IEC. *Information Technology – Programming Languages – Ada.* Geneva, ISO/IEC, 1995. (ISO/IEC 8652:1995(E).)

Box 1 CHILL in Examples

This box gives a tutorial overview on the language elements of CHILL in three pieces:

- Sequential programming (types and statements)
- Object-oriented programming and Genericity
- Concurrent programming

Sequential Programming: Types

From a structural point of view we may distinguish between scalar types and composite types. In this overview we follow roughly this pattern.

Scalar Types

The values of scalar types are indivisible entities. Important scalar types are numbers, enumerations and references.

As is usual in computing, we distinguish integer numbers and types, and floating point numbers and types.

Integer numbers are written as usual:

```
1, 123, -450
```

Large numbers may be structured for better readability using the underscore character: `1_721_119`

We may write numbers using different bases:

Binary numbers:	`b'1010`
Octal numbers:	`o'12367`
Hexadecimal numbers:	`-h'12ABC`

There are predefined integer types (e.g. INT) and the user may also define his own types, especially types with specific value ranges:

```
NEWMODE line = RANGE(1:8);
       /* e.g. the lines of a chess board */
```

A variable of a given type is defined in a declaration statement:

```
DCL CurrentLine line INIT := 1;
```

Such a variable can be initialized with a specific value.

For rational numbers CHILL uses floating point types. FLOAT is a predefined type, but it is also possible to define problem specific floating point types, e.g. a type for temperature in a given range.

```
NEWMODE Temp = FLOAT(-273.15:1000.0);
```

For numbers the usual arithmetic operations are defined:

```
DCL I INT INIT := 25*25 + 17;
DCL J INT INIT := I/2;
```

The type BOOL contains the two truth values FALSE and TRUE and can be used for conditions and computations in propositional logic:

```
DCL CallFinished BOOL INIT := FALSE;
. . .
IF NOT CallFinished THEN . . .
```

Very useful are also the enumeration types, e.g.

```
NEWMODE ActionType = SET(A1, A2, A3);
NEWMODE ColorTy = SET(red, green, blue);
SYNMODE month = SET
       (jan,feb,mar,apr,may,jun,
       jul,aug,sep,oct,nov,dec);
```

Composite Types

The values of composite types consist of several components which may themselves be scalar or composite values. The composite types in CHILL are structures (records), arrays and strings, buffer and signal, sets, and objects.

Structures are heterogeneous tuples:

```
NEWMODE DateType =
       STRUCT ( day INT(1:31),
                mo       month,
                year     INT(1:3000) );

NEWMODE TimedActionType =
       STRUCT ( action   ActionType,
                date     DateType );
```

The values of structures can be denoted by unlabelled or by labelled tuples:

```
DCL Today DateType INIT := [24,aug,2000];
DCL Today DateType INIT :=
      [day: 24, mo: aug, year: 2000];
```

If we want to implement a linked list of timed actions, we can use a reference type (pointer type). The values of reference types point to other values.

```
NEWMODE RefToTimedActionListType =
       REF TimedActionListType;

NEWMODE TimedActionListType =
       STRUCT(action TimedActionType,
              next   RefToTimedActionListType );

DCL TimedActionList TimedActionListType;
```

The following two assignment statements now create a linked list containing two timed actions.

Box 1 CHILL in Examples, continued

```
TimedActionList :=
       ALLOCATE(TimedActionListType,
                [[A1, [16,sep,2000]],NULL]);
TimedActionList :=
       ALLOCATE(TimedActionListType,
                [[A3, [28,aug,2000]],
                TimedActionList]);
```

For homogeneous tuples, as e.g. vectors or matrices, array types can be used. They can have an arbitrary number of dimensions.

```
NEWMODE VectorType = ARRAY(1:3)FLOAT;
NEWMODE SqMatrixType =
       ARRAY(1:3, 1:3) FLOAT;
DCL Vect1 VectorType
          INIT := [1.0, 2.5, 5.0];
DCL Matrix1 SqMatrixType
       INIT := [ [1.0, 2.0, 3.0],
                 [4.0, 5.0, 6.0],
                 [7.0, 8.0, 9.0] ] ;
```

String types are similar to one-dimensional arrays with a special element type, which is either CHARS (= Latin-1), WCHARS (= Unicode) or BOOL.

```
NEWMODE NameType = CHARS(20) VARYING;
DCL MyName NameType INIT := "Winkler";
DCL FirstLetter CHAR INIT := 'W';
```

Sequential Programming: Statements

The section on types already contains several assignment statements. It is therefore not necessary to give further examples.

There are two kinds of selection statements: IF and CASE.

```
IF a>b THEN max := a; ELSE max := b; FI
```

The CASE-statement selects among more alternatives. The CASE-statement of CHILL can also select an alternative using a tuple of n selection values.

```
CASE A, B OF Bool, Bool;
       (false),(false) : Res := false;
       (false),(true)  : Res := false;
       (true), (false) : Res := false;
       (true), (true)  : Res := true;
ESAC
```

There are FOR-loops and WHILE-loops to express repetitive computations.

```
DO WHILE sieve/=empty;
       primes OR:= [MIN(sieve)];
       DO FOR j := MIN(sieve)
                   BY MIN(sieve) TO max;
          sieve -:= [j];
       OD;
OD;
```

Object-Oriented Programming and Genericity

CHILL supports object-oriented programming in a very versatile way in that it combines object-orientation, concurrency and genericity. We show the popular example of the stack data type.

First the specification / interface:

```
SYNMODE IntStackType1 = MODULE SPEC
       GRANT Push, Pop;
       Push: PROC(Elem INT IN)
             EXCEPTIONS(Overflow) END Push;
       Pop:  PROC( ) RETURNS(INT)
             EXCEPTIONS(Underflow) END Pop;
       SYN Length = 10_000;
       DCL StackData ARRAY (1:Length) INT,
             TopOfStack RANGE(0:Length)
                        INIT := 0;
END IntStackType1;
```

The corresponding implementation/body looks like this:

```
SYNMODE IntStackType1 = MODULE BODY
       Push: PROC(Elem INT IN)
             EXCEPTIONS(Overflow)
          IF TopOfStack = Length THEN
             CAUSE Overflow;
          ELSE
             TopOfStack +:= 1;
             StackData(TopOfStack) := Elem;
          FI;
       END Push;
       /* body of Pop */
END IntStackType1;
```

Stack objects are declared in the same manner as variables of other types.

```
DCL Stack1, Stack2  IntStackType1;
Stack1.Push(10);
Stack1.Push(20);
IF Stack1.Pop() > 10 ...
```

Since Stack1 and Stack2 have a finite capacity, it would be better to check whether the operations have been executed normally, i.e. check whether an exception has occurred.

```
Stack1.Push(30)
       ON(Overflow): TempValStack1 := 30;
                     PushStack1 := True;
       END;
```

IntStackType1 is a sequential stack without coordination of concurrent calls, i.e. Stack1 behaves very much like a module. It is easy to define a stack type whose objects behave like regions:

Box 1 CHILL in Examples, continued

```
SYNMODE IntStackType2 = REGION SPEC
       /* as in IntStackType1 */
END IntStackType2;
SYNMODE IntStackType2 = REGION BODY
       /* bodies of Push and Pop */
END IntStackType2;
```

If we use inheritance such a stack type with coordination can be obtained even simpler:

```
SYNMODE IntStackType2 = REGION SPEC
       BASED_ON IntStackType1
END IntStackType2;
```

Both IntStackType1 and IntStackType2 have a fixed element type. If we need stack types for other element types, we have to duplicate the code.

It is simpler first to define a generic stack template StackTemplate1 and then define IntStackType1 and DateStackType1 as generic instantiations of StackTemplate1.

```
GenericStackTemplate1: GENERIC
       MODE ElemType = ANY_ASSIGN;
MODULE SPEC
       GRANT Push, Pop;
       Push: PROC(Elem ElemType IN)
             EXCEPTIONS(Overflow) END Push;
       Pop:  PROC( ) RETURNS(ElemType)
             EXCEPTIONS(Underflow) END Pop;
       SYN Length = 10_000;
       DCL StackData ARRAY (1:Length) INT,
           TopOfStack RANGE(0:Length)
             INIT := 0;
END GenericStackTemplate1;
```

The corresponding implementation/body looks like this:

```
GenericStackTemplate1:
       GENERIC MODE ElemType = ANY_ASSIGN;
MODULE BODY
       /* bodies of Push and Pop */
END GenericStackTemplate1;
```

This template can be used to define object types as instantiations of the template. We do not have to duplicate the code, but only have to provide an actual generic parameter.

```
SYNMODE IntStackType4 =
       NEW GenericStackTemplate1
           SYNMODE ElemType = INT;
END IntStackType4;
SYNMODE DateStackType1 =
       NEW GenericStackTemplate1
           SYNMODE ElemType = DateType;
END DateStackType1;
```

IntStackType4 is essentially equivalent to IntStackType1.

Concurrent Programming

One essential difference between sequential and concurrent programming is the presence of active entities, i.e. entities which have their own thread of control. Such entities are called active entities in contrast to passive entities, as e.g. procedures.

CHILL contains two kinds of active entities: the process and the task object.

Processes typically communicate via buffers, signals or regions. A traditional example is the producer-consumer problem, where a number of processes produce data items and a number of processes consume these data items.

```
ProducerConsumer: MODULE
       DCL PCBuffer BUFFER(100)ItemType;
       ProducerType: PROCESS()
          DCL Item ItemType;
          DO WHILE NotFinished
             /* produce new data item */
             Item := NewValue;
             SEND PCBuffer(Item);
          OD;
       END ProducerType;
       ConsumerType: PROCESS()
          DCL Item ItemType;
          DO WHILE NotFinished
             RECEIVE (PCBuffer IN Item);
             /* consume the data item */
             OD;
       END ConsumerType;
       /* Two producers and one consumer */
       START ProducerType();
       START ProducerType();
       START ConsumerType();
END ProducerConsumer;
```

If there are several kinds of consuming or processing the items produced by the producers, we can define a task type with corresponding methods.

```
ProducerConsumer2: MODULE
   ProducerType: PROCESS()
      DCL Item ItemType;
      DO WHILE NotFinished
         /* produce new data item */
         Item := NewValue;
         CASE KindOfProcessing OF
            (Kind1): Consumer.Consume1(Item);
            (Kind2): Consumer.Consume2(Item);
         ESAC;
   END ProducerType;
   SYNMODE ConsumerType = TASK SPEC
      GRANT Consume1, Consume2;
      Consume1: PROC(Item ItemType IN);
      Consume2: PROC(Item ItemType IN);
   END ConsumerType;
```

CHILL in Examples, continued

```
SYNMODE ConsumerType = TASK BODY
   Consume1: PROC(Item ItemType IN)
      /* consume the data item */
   END Consume1;
   Consume2: PROC(Item ItemType IN)
      /* consume the data item */
   END Consume2;
END ConsumerType;
```

```
   /* Two producers and one consumer */
   DCL Consumer ConsumerType;
      /* automatic start */
   START ProducerType();
   START ProducerType();
END ProducerConsumer2;
```

Box 2 CHILL vs. Java

CHILL	Java
Data Structures	Data Structures
Scalar: integer, float, characters, boolean, enumerations, pointer, procedure type, process type, event, time range types	Scalar: integer, float, characters, boolean no range types
Composite: string, record, array, set, buffer, signal	Composite: string, array, set, and many others (in the predefined APIs)
Sequential Programming	Sequential Programming
Variable, constant, expression, function call	Variable, constant, expression, function call
Assignment	Assignment
Procedure call	Procedure call
EXIT, RESULT, RETURN, GOTO	BREAK, RETURN
Statement sequence	Statement sequence
Selection statements: IF, CASE (multidimensional)	Selection statements: IF, SWITCH (onedimensional)
Repetition statements: DO, WHILE, FOR	Repetition statements: WHILE-DO, DO-WHILE, FOR
Object-oriented Programming	Object-oriented Programming
Sequential, unsynchronized object	Sequential, unsynchronized object
Sequential, synchronized object	
Concurrent, synchronized object	Concurrent object
Interface	Interface
Friend	
Concurrent Programming	Concurrent Programming
Process	
Start process	
Communication via buffer	
Communication via signal	
Critical region and coordination with events	Synchronized method and synchronized statement
Concurrent, synchronized object	Concurrent object
Program Structure	Program Structure
Block	Block
Procedure / Function / Process	Procedure / Function
Object-Type / Class	Object-Type / Class
Module / Region	Package
Genericity	Genericity
Generic Procedure / Process	
Generic Module / Region	
Generic Object Type / Class	
Generic Interface	
Program Verification	Program Verification
Precondition and postcondition for methods	
Invariant for object type / class	
ASSERT statement	
	Additional Elements

Applet	java.applet
Reflection	java.lang.reflect
GUI definition	javax.swing, java.awt
SW components	java.beans, org.omg.CORBA
Remote Procedure Call	java.rmi
Internet access	java.net
Data security	java.security
Data base access	java.sql
Data compression	java.util.zip
Painting	java.awt
Music	java.sound.midi

Object Definition Language

MARC BORN AND JOACHIM FISCHER

The ITU Object Definition Language (ODL) is an extension of CORBA IDL version 2.0 supporting multiple interfaces, both operational and stream interfaces, and supports groups of objects and user-defined data types.

Marc Born (29) received his MSc in Computer Science from Humboldt Universität zu Berlin in 1996, whereupon he started work at GMD FOKUS as a scientist, working mainly in the area of object-oriented development and specification of telecommunication systems in national and international projects. Since 1999 he has been active in EURESCOM and other international projects. He is currently involved in the standardisation activities of the ITU-T SG 10 regarding ITU-ODL, SDL and DCL. He has started on his PhD thesis on a methodology for distributed telecommunication system design, in particular the definition of a suitable notation for relevant modelling concepts.

born@fokus.gmd.de

Joachim Fischer (48) has been University Professor for Modelling and Computer Simulation at the Humboldt-Universität Berlin since 1994. He received his MSc in Mathematics in 1977 and his PhD in Computer Science in 1982. In 1988 he published his habilitation thesis at the HU on rapid prototyping of distributed systems using executable formal description techniques. His current working area is in the development of tool-supported description and design techniques for complex distributed systems and their application in the telecommunications domain. He is involved in many European projects and a member of SDL-Forum, ITU and ASIM (German Simulation Society).

fischer@informatik.hu-berlin.de

1 Fundamentals

The Reference Model for Open Distributed Processing (RM ODP) [1] defines an architecture for the design of distributed services where the basic idea is to split the design concerns into several viewpoints: enterprise, information, computational, engineering and technology. This is in order to overcome the immense complexity of today's distributed systems by structuring the design process.

To describe models of distributed services from the computational viewpoint a combination of Object Definition Language (ODL) [2] and Specification and Description Language (SDL) [3] is proposed by ITU. An ODL computational model defines the objects (or components) and groups of them together with the interface signatures. All definitions are given in terms of types (called templates in ODL), so ODL does not support the definition of instances, their configurations and the precise formal behavior description. Additional information especially on the behavior and on the connection between the component instances is needed in order to allow validation, automated code generation and automated testing. A structural equivalent mapping of ODL to SDL is one possible starting point for a more precise behavior description and object configurations. However, it should be noted that the intention is not to require a detailed SDL model for each individual component. SDL is needed for components where validation, automated code generation and/or automated testing should be performed. There are two different ways to come to component implementations. One is based on a mapping of ODL to C++ (which follows the corresponding mapping of CORBA IDL[4] as a subset of ODL), the other is based on an automated code generation from SDL.

ODL is based on the work done by TINA-C on TINA ODL [5] from 1996. The motivation for this work and the following ITU activities was to provide specification support for stream interfaces and of objects with multiple interfaces, which is not possible in CORBA IDL. These features have been added using CORBA IDL as the base notation. The ITU activities lead to the ITU ODL standard Z.130, which is a strict superset of CORBA IDL.

2 Computational Object

A computational object (CO) is an autonomous interacting data processing unit in the computational model of a distributed system. The COs interact through their well defined computational interfaces. The modelling process focuses on how a particular functionality can be provided without taking into account what kind of computing or network infrastructure is used to implement the object.

Hence, the task of a computational model is to define the object (or component) structure together with the interface signatures and to describe the behavior provided at the interfaces on a high level of abstraction. ODL focuses on the first aspect while behavior definitions are provided informally only.

Object class O supports two kinds of interfaces I1 and I2, nothing is said about the instantiation of these interfaces. I1 is a name of an operational interface, which here supports only one operation Op1, where O is the server of that operation. I2 is a stream interface, which is able to handle two information (media) flows F1 and F2, where O is the producer of F1 and the consumer of F2.

```
CO O{
    behavior this is an example;
    supports
        ModA:I1, // Interface I1 is defined
        // in Module ModA
        ModB:I2; // Interface I2 is defined
        // in Module ModB
}; // end of O
```

3 Interface

Interfaces are access points to object implementations. A computational interface template comprises a textual (informal) behavior specification and, as appropriate, either an operational interface signature, or a stream interface signature.

Operational Interface

The following information is specified:

- The signature of each operation which is supported, so that it can be invoked by clients (operation name, parameter types, return types, exceptions);
- The definition of one-way operations with neither return types nor exceptions;
- The semantics of each operation, including sequencing, and concurrency constraints applicable to the operations;
- The type and the name of each attribute (specifying attributes implies to have get and set operations later in the implementation code).

Stream Interface

The following information is specified:

- The signature of each stream flow (flow name, sink or source indication, flow type/format);
- The semantics and QoS aspects of each stream flow.

An interface template can be shared between several object templates. Interface template specifications can be included in an object template declaration as supported interfaces or as required interfaces.

Supported interfaces

Supported interfaces are offered interfaces of a computational object. They are the only interface templates for which instances may exist on the objects instance.

Required interfaces

The required interfaces of an object prescribe interfaces on other objects that this object invokes operations on.

Initial Interface

One and only one of the supported operational interfaces of an object template may be declared as being the initial interface of that object. That interface will be instantiated when the object template itself is instantiated. This interface instance may be used for initialization or configuration purposes.

Interfaces which are supported by an object may only exist as long as the object itself exists.

4 Object Group

A group template allows the specifications of ensembles of computational objects. An informal predicate defines the interpretation of such a group. One important application of an ensemble is to define a composite object. Another application is to specify a loosely coupled logical unit of objects, for instance a collection of objects

Figure 1 Computational Object (CO) with supported interfaces

which are managed by one manager. The later corresponds to the idea of groups in TINA ODL.

Interface template specifications can be referred to in an object group template as supported or required interfaces (contracts). These contracts are the interface templates whose instances can be used by COs external to the object group (supported) or needed by the instances of the group members from the environment (required).

```
group G1 {
   predicate
     "This group manages a subnetwork. The
     NetworkCoordinator manages this group."
   members
     CMC, NetworkCoordinator, NetworkCP,
     ElementCP; //computational objects
   supported
     Configurator, Trail, TC;
};
```

5 Data Description

Data types and constants can be declared in almost any scope within an ODL specification. Later, these types or constants can be used for declaration of operation, exception, flow, and other template constructs. As for any template declaration, it is required that a type or constant is declared prior (i.e. earlier in the file) to its use.

The syntax supported by ODL for type and constant declaration is identical to OMG-IDL.

```
const string str="xyz";
typedef float Bps;
enum Guarantee {
        Deterministic, Statistical, BestEffort
};
struct AudioQoS {
   union Throughput switch (Guarantee){
      case Statistical:       Bps mean;
      case Deterministic:     Bps peak;
      case BestEffort:
         struct Interval {
            Bps min;
            Bps maxd;
      } range ;
   };
   union Jitter switch (Guarantee) {
      case Statistical:       Bps mean;
      case Deterministic:     Bps peak;
   };
};
```

6 Template Inheritance for Interfaces, Objects and Object Groups

Interface templates, object templates and object group templates are considered units of specification modularity. A template is derived from another template of the same kind. Rules for inheritance will allow new interface templates, object templates and object group templates to be declared as extensions of previously defined ones. For all template kinds, multiple inheritance is supported.

Only the essential elements of interface template inheritance are presented here. It is possible to declare interface template I4 inheriting operation11 from I1, and adding operation operation41. Similarly, S2 may inherit from S1 the source flow voiceDownStream and the sink flow voiceUpStream, and add the source flow videoFlow21. I1 and S1 are defined as follows:

```
interface I1{
    ...
    // data types
    typedef ... DataType11;
    typedef ... DataType12;

    void operation11 (in DataType11 ...,
                      out DataType12 ...);
}; // end of I1

interface S1{
    ...
    // flow types
    typedef ... VoiceFlowType;

    source VoiceFlowType voiceDownStream;
    sink VoiceFlowType voiceUpStream ;
}; // end of S1
```

The inheriting interface templates can then be defined as follows:

```
interface I4: I1{
    ...
    typedef ... DataType41;
    void operation41 (in DataType41 ...);
}; // end of I4

interface S2: S1{
    ...
    typedef ... FlowTypeS21;
    source FlowTypeS21 videoFlow21 ;
}; // end of S2
```

Object template O, using the inherited specialised interface templates, can be defined as follows:

```
CO O {
    behavior ...
    supports I4, S2;
    ...
}; // end of O
```

7 Module

A module introduces a scope for contained definitions. It may contain definitions of templates for objects, interfaces, object groups and user-defined data types. Modules can be nested. The usage of modules enables a structured specification development.

8 Naming and Scoping

The following kinds of definitions form nested scopes within an ODL specification: module, object group template, object (CO) template, interface template, data types (struct, union), operation and exception. For example, the following ODL definitions are contained in one specification.

```
module M1 {
    ...
    group G1 {
        ...
        CO O1 {
            ...
            interface I1 {
                ...
                typedef ... DataType1;
                ...
                void operation1
                (in DataType1 parameter1...);
                ...
            }; // end of I1
        }; // end of O1
    }; // end of G1
}; // end of M1
```

An identifier can only be defined once in a scope, but can be redefined in nested scopes. Based on the ODL example above, the qualified name of G1 is M1::G1. Similarly the qualified name of DataType1 is M1::G1::O1::I1::DataType1.

9 Design Methodology

Since ODL describes structures and signatures of components only, there are two ways of covering the semantics of them. The traditional one is to step directly into the implementation. For supporting this, there is a need for a language mapping from ODL into the used implementation language to make use of the specified structure and signature information. Such a language mapping to C++ has been developed and is part of Z.130. By applying this mapping, the time needed for the implementation can be reduced compared to using the IDL to C++ mapping pro-

Box 1 Mapping from ODL over SDL to C++

The most important steps of the methodology are:

Step (1): Take the information and the enterprise (viewpoint) specification, partially developed using UML[6], and define a computational (viewpoint) specification expressed in ODL, using the mapping rules of Z.130.

Step (2): The ODL specification is mapped into a structurally equivalent SDL skeleton specification.

Step (3): The SDL inheritance feature is applied to enrich the SDL specification generated by step 2 with behavior descriptions for both the interface and object templates. The behavior description is based on the specification of states and transitions. Step 3 can be repeated to achieve different levels of abstraction in the design of object composition and behavior. The result is an executable SDL model in the computational viewpoint. It is not yet an engineering viewpoint solution.

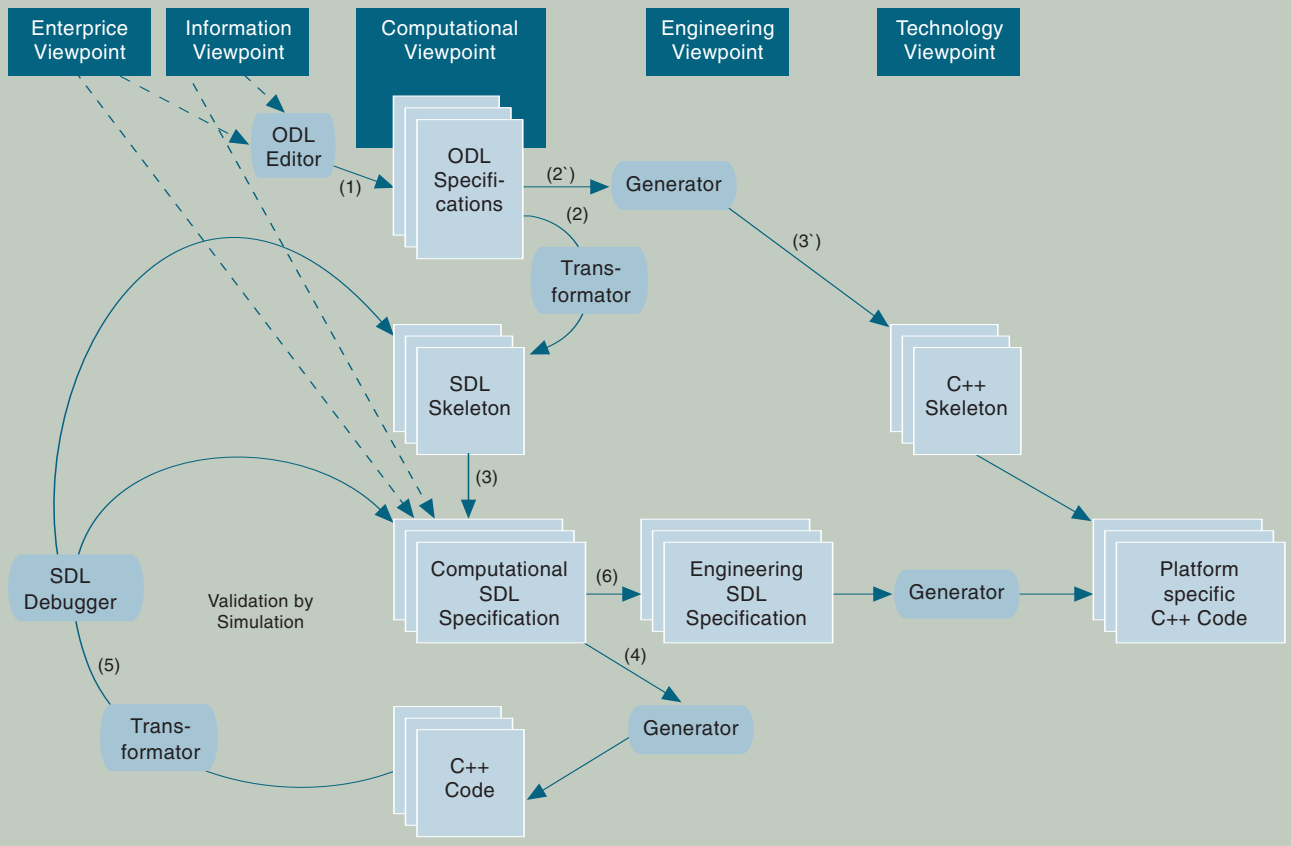


Figure 2 Steps in system design using ODL and SDL

Step (4): With help of tool packages, the SDL specification from step 3 can be checked for correctness of syntax and static semantics. Additionally, it is possible to generate C++ code that can be linked with a simulation library. This leads to a simulator for the SDL system, which represents the ODL specification and includes the computational behavior aspect and hence allows to check the dynamics of the system.

Step (5): A simulation of the computational model is used to detect design errors prior to implementation. An SDL debugger is a component that supports this kind of validation. Another way of error detection is to explore the state space of the SDL model to find lifelocks or deadlocks. If design errors are detected, a repetition of the steps 1 to 4 could be necessary.

Step (6): Platform specific C++ code can be generated out of the SDL model. If necessary, the SDL model can be refined before; this specification is called Engineering SDL specification.

The result of applying this method is a computational model of either a complete telecommunication service or a single component or a set of components.

Box 2 Direct mapping from ODL to C++

Since ODL does not provide means for behavior description, there must be a solution for describing the behavior of a distributed application outside the ODL specification. As already indicated, this could be done via a mapping to SDL. However, it is not realistic to require a detailed SDL model for each individual component. It depends on the application for which component an SDL specification makes sense. This is due to the fact that not all problems can be adequately described by state machines. As a consequence, not all component implementations can be derived from their SDL specification via automatic code generation. If performance aspects have to be considered, or if SDL is not suitable to specify the component behavior in every detail, the components have to be implemented by hand. In this case, there is a need to have language mapping not only from ODL to SDL, but also to an implementation language like C++ in order to use the structural information contained in the ODL specification for the implementation (Step (2')).

Since ODL is a strict superset of CORBA IDL, all aspects concerning the communication between the components of a system are described using CORBA IDL interface descriptions. In order to be able to use existing ORB implementations as a platform to implement an ODL specification, the mapping for the CORBA IDL part of ODL is adopted. The intention is that the existing ORB specific CORBA IDL compilers can be used to map the IDL part of an ODL specification to C++.

Additionally, the structural information of computational objects or components should be reflected in the implementation language, as well. This information reflects design decisions made during system design. It is not a requirement to map the structural information into the implementation language, since the application will work even if only the IDL part is mapped (Only the interfaces are of importance for communication). But ODL should be understood as a computational design language, and the structural information contained in it makes the programming of distributed applications easier. Therefore, a language mapping of ODL to C++ is part of Z.130. This language mapping allows flexible structuring of implementations and ensures that the developer can make use of the computational design information in the implementation stage (Step (3')). The mapping rules are aligned with those from ODL to SDL and from SDL to TTCN. This facilitates the application of automated testing facilities and increases the benefits resulting from the application of ODL and SDL in the design methodology. See Figure 2.

vided by the Object Request Brokers (ORBs) only in terms of IDL compilers.

Though the way of direct mapping from ODL to the implementation language is the most common method, another approach is needed in order to allow validation of the component behavior before its implementation and to perform automated code generation and testing. This approach is to provide a computational viewpoint behavior description for the components. This behavior description should be an abstract one since in most cases only the external visible behavior should be specified without prescribing any implementation details. SDL is a convenient language to state the behaviour description. A mapping to SDL is part of Z.130 too.

The mapping is supported by tools, and has been applied in different EU projects dealing with the development of telecommunication services.

10 ODL Limitations

ODL was defined before the OMG standardization activities reached the final stage where components with multiple interfaces [7] were defined. An interesting question to ask now is how the current version of ODL compares to the computational modelling concepts of ODP, SDL and Component IDL.

Limitations of the Computational Concepts of ODL

ODL does not support behavior specification; especially there is no description concept for binding functionality. It does not provide mechanisms for template instantiations and the configuration of systems based on components. Furthermore, signal communications cannot be described directly. Meanwhile, this is supported by OMG with events. Practical experiences have shown that the support of stream flow types in ODL is not adequate. The concept of flow types should be substituted by the usage of standardized formats (MP3, MPEGII, ...).

Language Mapping Problems

If a solution were to be found for the problems of missing ODP concepts, the resulting ODL language would have more expressive power than OMG Component IDL. However, there is a problem of mapping stream interfaces. The only possible way of mapping is to map these interfaces to implicit operational interfaces for managing the stream flows, which are declared by the stream interfaces. A generation of concrete operations (and their signatures) can only be realized for a concrete platform which itself has to be standardized first.

Regarding the SDL mapping, there are also some open issues:

Box 3 Example: Interactive TV

Problem Statement

A design task, which should be solved using ODL, can be summarized as follows:

An Interactive TV Station Server should be developed, which should provide a number of channels for clients (audio & video data). The server should be able to receive inputs from its clients in the form of joystick and mouse events. Furthermore, the server has to provide a mechanism to set configuration data. There have to be two configuration variants, where the second variant should provide a more sophisticated behaviour.

The client for this station should also be designed, and it should be able to receive the channels and to provide the mouse and joystick events which trigger some changes in the sent channel. The client should also be configured.

Relevant Constructs

A system designer who has to produce an ODL specification for the above task would design the computational viewpoint in the following way:

- Specification of the mouse and joystick events as ODL structs with two members for the x and y pos of the mouse or joystick.
- Specification of an operational interface, which has two one-way push operations for the mouse and joystick events.
- Specification of the flow types for the audio and video flows that have to be exchanged. This could be octet types.
- Specification of a stream interface which has two sources, one of the audio flow type and one of the video flow type.
- Specification of operational interfaces with the basic configuration operation for both the client and the server and a subtype of this operational interface with the extended configuration operation for the server.
- Specification of two computational objects representing the servers. The base CO supports the stream interface and the interface with the push operations. It has the base configuration interface as initial interface. The second CO is a subtype of the base CO, it has the interface with the extended configuration operation as initial interface.
- Specification of a CO, which requires the stream interface and the interface with the two push operations. It declares the base configuration interface as initial interface.

ODL Specification

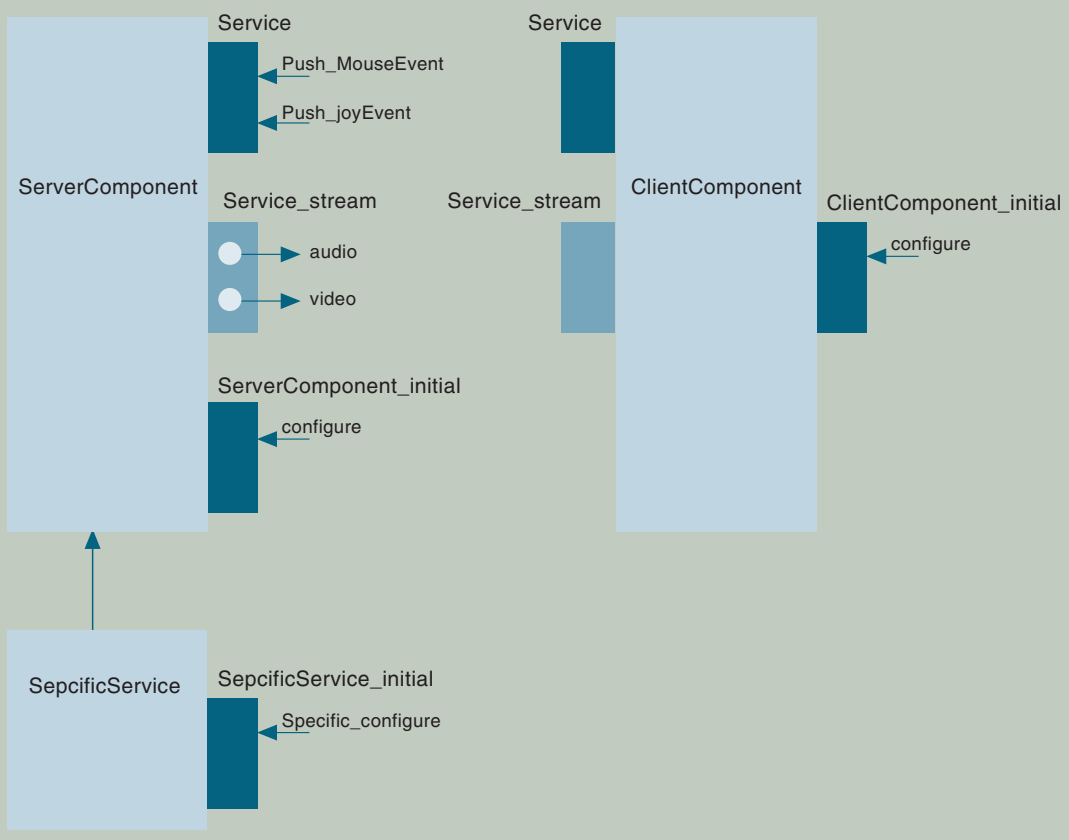


Figure 3 Graphical ODL specification for Interactive TV example

Box 3 Example: Interactive TV, continued

```
module InteractiveGame {
    typedef octet Audio;
    typedef octet Video;

    struct MouseEvent {
        long pos_x;
        long pos_y;
     };

    struct JoyEvent {
        long pos_x;
        long pos_y;
    };

    interface Service {
        oneway void push_mouseEvent( in MouseEvent mouse
);
        oneway void push_joyEvent( in JoyEvent joy);
    };

    interface Service_stream {
        source Audio audio;
        source Video video;
    };
}; // end module InteractiveGame

module ServiceComponents {
    interface ClientComponent_initial_;

    CO ClientComponent {
        requires ::InteractiveGame::Service,
                 ::InteractiveGame::Service_stream;
        initial ClientComponent_initial_;
    }; // end object ClientComponent

    interface ClientComponent_initial_ {
        void configure ( );
    };

    interface ServiceComponent_initial_;

    CO ServiceComponent {
        supports ::InteractiveGame::Service,
        ::InteractiveGame::Service_stream;
        initial ServiceComponent_initial_;
    }; // end object ServiceComponent

    interface ServiceComponent_initial_ {
        void configure ( );
    };

    module SpecificServiceComponents {
        interface SpecificService_initial_;

        CO SpecificService: ::ServiceComponents::
            ServiceComponent {
            initial SpecificService_initial_;
        }; // end object SpecificService

        interface SpecificService_initial_ :
            ::ServiceComponents::ServiceComponent_initial_ {
            void specific_configure ( );
        };
    }; // end module SpecificServiceComponents
}; // end module ServiceComponents
```

- A more harmonized language mapping would be achieved if SDL would offer the use of ODL data types as an alternative data type concept.

- Since SDL does not support multiple inheritance a flattening process is necessary for the mapping.

Intended ODL Improvements

ODL does not yet have a well-defined graphical syntax and some graphical symbols are missing, for required interface and initial interface, for inheritance, and for text boxes for data types. Therefore, there are some degrees of freedom here for the development of tools. It is expected and intended that tool vendors come up with their own graphical notation and use textual ODL as an interchange format.

References

1 ITU-T. *Open Distributed Processing – Reference Model Part 3/4.* Geneva, ITU, 1995. (Rec. X.903/X.904 | ISO/IEC 10746-3/-4.)

2 ITU-T. *Object Definition Language.* Geneva, 1999. (Rec. Z.130.)

3 ITU-T. *SDL – Specification Description Language*. Geneva, 2000. (Rec. Z.100.)

4 OMG. *The Common Object Request Broker Architecture and Specification, Version 2.3*. Needham, MA, 1999.

5 TINA-C. *Object Definition Language Manual, Version 2.3.* Trinton Falls, NJ, 1996.

6 OMG. *Unified Modeling Language Specification, Version 1.3.* Needham, MA, 1999.

7 OMG. *Corba Components – Vol. I.* Needham, MA, 1999. (ORBOS 99-07-01.)

Conformance Testing with TTCN

INA SCHIEFERDECKER AND JENS GRABOWSKI

Ina Schieferdecker (33) received her PhD from the Technical University Berlin in 1994. Since 1993 she has been a researcher at GMD Fokus, and a lecturer at Technical University Berlin since 1995. Her interests cover testing methods for distributed systems and formal methods for the design, validation and prototyping of distributed systems. She has been head of the Competence Center for Testing, Interoperability and Performance (TIP) since 1997 and is actively involved in several testing projects. She has published several papers on testing telecommunications systems and developing test systems, and is involved in the definition of MSC in ITU-T SG10 and of TTCN-3 in ETSI.

Schieferdecker@fokus.gmd.de

Jens Grabowski (38) graduated from the University of Hamburg with a diploma degree in Computer Science and Chemistry. He spent two years at SIEMENS AG in Munich focusing on protocol specification and protocol validation based on Petri Nets, SDL and MSC. 1990–1995 he was research scientist at the University of Berne, where he received his PhD in 1994. Since 1995 Grabowski has been researcher and lecturer at the Institute for Telematics at the Medical University in Lübeck; since 1996 he has also worked as expert in several ETSI standardization projects. He is a member of the ETSI experts team which develops the third edition of TTCN.

jens@itm.mu-luebeck.de

The Tree and Tabular Combined Notation (TTCN) is a semi-formal notation which supports the specification of abstract test suites for protocol conformance testing. An abstract test suite is a collection of abstract test cases[1)]. As indicated by the name TTCN, test cases are described in the form of behavior trees and different kinds of tables are used for the graphical representation of test suites.

"Product testing is still seen as the only reliable way to assure that outsourced products meet the required specification and are suitable for inclusion in the live network."
Cited from Counting on IT, Issue 7 by National Physical Laboratory, UK, Summer 1998.

1 Introduction

TTCN [3] is the means of the Conformance Testing Methodology and Framework (CTMF) for the description of test suites for conformance testing. See terminology and explanations in Box 1. TTCN has two syntactical forms (Figure 1), called TTCN/gr (TTCN GRaphical form) and TTCN/mp (TTCN Machine Processable form). TTCN/gr is intended to be used by humans and TTCN/mp is developed for the exchange of documents between different computers and for further processing of TTCN test suites. A TTCN/gr description can be translated into an equivalent TTCN/mp representation and vice versa. In this paper only TTCN/gr examples are presented.

In the following, the different TTCN constructs are described by developing an example test suite[2)]. The system to be tested is a parcel service. A test case should check whether the parcel

Test Step Dynamic Behaviour

Test Step Name	:	Preamble
Group	:	
Objective	:	To bring the SUT into the initial state
Default	:	
Comments	:	
Description	:	

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		(PS_Init:= Reset_ParcelService())			
2		[PS_Init]		(P)	
3		[NOT PS_Init]		I	

Detailed Comments:

```
...
$Begin_TestStep
$TestStepId Preamble
$TestStepRef Example_ATS/
$Objective /* To bring the SUT into the initial state */
$DefaultsRef
$BehaviourDescription
$BehaviourLine
$LabelId
$Line [0] (PS_Init:= Reset_ParcelService())
$Cref
$VerdictId
$End_BehaviourLine
$BehaviourLine
$LabelId
$Line [1] [PS_Init]
$Cref
$VerdictId (P)
$End_BehaviourLine
$BehaviourLine
$LabelId
$Line [1] [NOT PS_Init]
$Cref
$VerdictId I
$End_BehaviourLine
$End_BehaviourDescription
$End_TestStep
```

Figure 1 The TTCN forms: TTCN/gr and corresponding TTCN/mp code below the table

1) CTMF and TTCN use the terms abstract and executable to distinguish between implementation-independent and implementation-dependent concepts, e.g. abstract test suite and executable test suite, abstract test case and executable test case or abstract service primitive. This paper introduces mainly implementation-independent CTMF and TTCN concepts. Qualifiers like abstract or executable will only be used in case of ambiguities.

Conformance Testing Framework

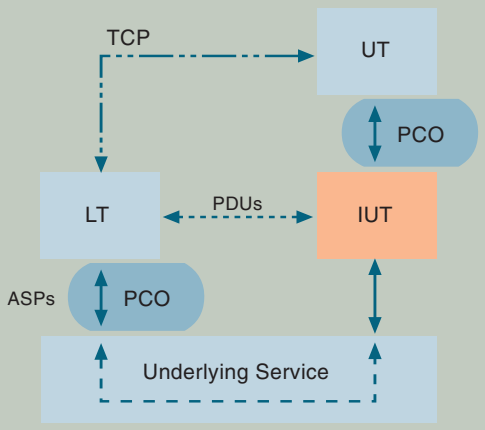
Testing a system is performed in order to assess its quality and to find errors. An error is considered to be a discrepancy between observed or measured values provided by the system under test and the specified or theoretically correct values. Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements. It approves a quality level of a tested system.

Conformance testing in particular is the process of testing the extent to which implementations of OSI protocol entities adhere to the requirements stated in the relevant standard or specification. Conformance testing is functional black-box testing. The term functional refers to the correct functional behavior of an Implementation Under Test (IUT), i.e. the correct input/output behavior in each state. Black-box testing means that the internal structure of the IUT remains hidden, i.e. it is a black box for the test developer.

The OSI conformance testing procedure is defined in the international ISO/IEC standard 9646 Conformance Testing Methodology and Framework (CTMF) [1]. CTMF consists of seven parts and covers the following aspects: concepts (part 1), test suite specification and test system architectures (part 2), test notation (part 3), test realization (part 4), means of testing and organizational aspects (part 5, 6, and 7). The Tree and Tabular Combined Notation (TTCN) is defined in part 3 of CTMF. By definition, the target systems to be tested according to the CTMF principles are implementations of OSI protocol entities. However, CTMF and TTCN are applicable in a much wider scope than OSI-based systems. The CTMF principles and TTCN have also been applied successfully for conformance testing of ODP-, TINA-, CORBA- and IP-based systems, APIs and reactive systems in general[3)].

In conformance testing, the IUT is an implementation of an OSI protocol entity. The IUT is part of an open system called System Under Test (SUT). The conceptual conformance test architecture is shown in the figure below.

The IUT has an upper and a lower interface through which it is tested. Conformance testing is done at standardized interfaces called Points of Control and Observation (PCOs)[4)]. Typically, the lower interface of an IUT is accessible only from remote. Therefore, the underlying service of the IUT is used to define an appropriate PCO on a remote site, i.e. the lower interface of the IUT is moved to the remote site. Communication is always meant to be asynchronous and therefore, a PCO is modeled by two FIFO queues, i.e. one queue for each direction.



Conceptual Test Architecture

CTMF distinguishes between an Upper Tester function (UT) and a Lower Tester function (LT). As indicated by the names, the upper interface of the IUT is controlled by the UT and the lower interface is controlled by the LT. During the test, the UT plays the role of a user that makes use of the service provided by the IUT and the LT plays the role of a peer entity of the IUT, i.e. the LT and the IUT communicate in order to provide the service to the UT.

IUT and UT communicate by means of Abstract Service Primitives (ASPs). Conceptually, IUT and LT provide their service by exchanging Protocol Data Units (PDUs). In practice, the PDUs are encoded in ASPs of the underlying service, i.e. PDUs will not be exchanged directly. However, CTMF allows to abstract from the encoding of PDUs, i.e. allows to specify the exchange of PDUs in abstract test cases. Therefore, it is not necessary to distinguish between ASP and PDU explicitly, and hence, only the term PDU is used.

As shown in the figure, Test Coordination Procedures (TCP) can be used to coordinate the actions of LT and UT. This might be necessary if LT and UT are realized in separate tester processes. The figure presents the conceptual test architecture only. In practice, several variations of the conceptual test architecture are used. The test methods defined in CTMF are local, distributed, coordinated and remote test method. They differ in the possibilities to coordinate LT and UT and the ability to control and observe

---

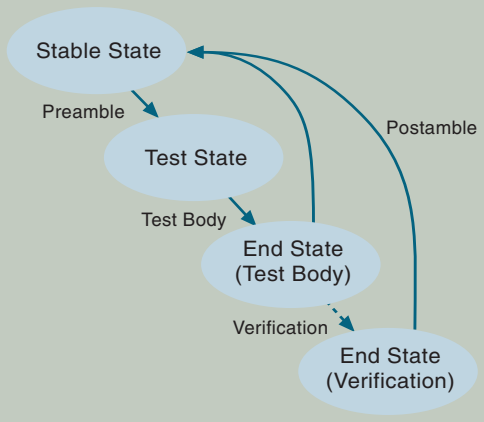
2) Only a few TTCN tables can be presented in this paper, but the complete example test suite is available from the authors.
3) An overview of the use of conformance testing and TTCN is given in [9].
4) In most cases, a PCO maps to a Service Access Point (SAP) in the OSI basic reference model.

Conformance Testing Framework, continued

the IUT. In addition, CTMF defines a multi-party context which allows to combine the different test architectures in order to specify tests with several UT and LT processes.

The test case development starts with the identification of test purposes. A test purpose is a prose description of a single requirement or a set of related requirements which should be tested. Test purposes are identified based on the requirements in the specification of the IUT.

A test case is the implementation of a test purpose for a particular test architecture, i.e. a complete specification of the actions required to achieve a specific test purpose. The definition of a test case follows the schema shown below.



Test Case Scheme

A test case starts and ends in stable testing states, which need not to be identical. It consists of a preamble, a test body, an optional verification step and a postamble. With the preamble, the IUT is driven from a stable testing state to the test state from which the test body is performed in order to check the test purpose. If the end state of the test body is not unique, it has to be checked by a verification step and then a postamble is used to drive the IUT into a stable testing state again. Otherwise, the IUT is put into a stable testing state immediately with the postamble.

Test cases developed according to the principles of CTMF are abstract. Executable test cases are derived from abstract ones by compilation and adaptation to the Means of Testing (MoT). The MoT is the combination of equipment and procedures that can perform the derivation, selection, parameterization and execution of test cases. It consists typically of dedicated test devices and facilities for the coordination of test devices and the observation of the IUT. These facilities may be installed inside the SUT.

service behaves as shown in Figure 2. A producer asks for a service offer and the parcel service indicates within a certain time frame that a 24h delivery service is available. Then, the goods are sent to the parcel service, which delivers them to the consumer. The consumer accepts the goods by sending an acknowledgement to the parcel service. The acknowledgement is forwarded as a confirmation to the producer. The confirmation is expected within 24h in accordance with the service assured by the parcel service. Not shown in Figure 2 is the possibility of the parcel service promoting new services to the producer by sending advertisements at any time.

The test architecture for testing the parcel service is shown in Figure 3. The IUT is the parcel service which is connected to the LT functions *Consumer* and *Producer* through the PCOs *LT_Cons* and *LT_Prod*. This test architecture can be seen as a combination of two remote test methods in a multi-party context, i.e. a special variant of the conceptual test architecture described in Box 1. In this section, a non-concurrent test case will be developed, i.e. the behaviour of both LT functions will be implemented in one test component which controls and observes both PCOs.

2 Basics of TTCN

A TTCN test suite is composed of four parts: an overview part (Section 2.1), a declarations part (Section 2.2), a constraints part (Section 2.3) and a dynamic part (Section 2.4).

2.1 Overview Part and Test Suite Structure

The overview part of a TTCN test suite can be seen as a table of contents and provides all information needed for the general presentation and understanding of the test suite. It defines the test suite name and test architecture, describes the test suite structure, provides references to additional documents related to the test procedure and includes indexes for the test cases, test steps and default behaviour descriptions[5)].

[5)] *The meaning of test steps and default behaviour descriptions will be explained in Section 2.4.*

Figure 2 Behaviour of a Parcel Service

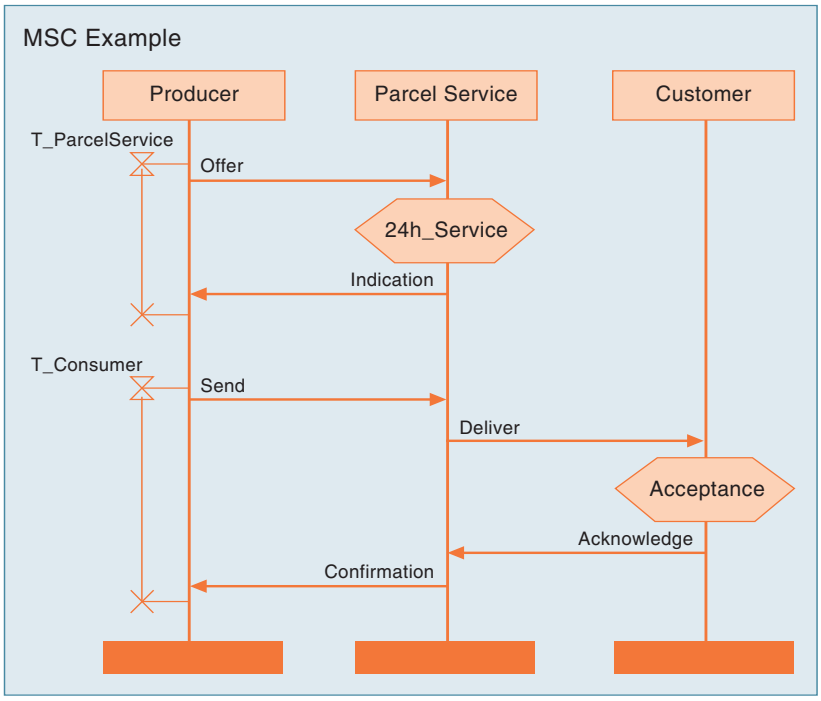


Figure 3 Test Architecture for the Parcel Service

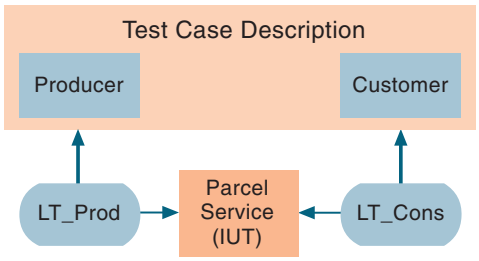
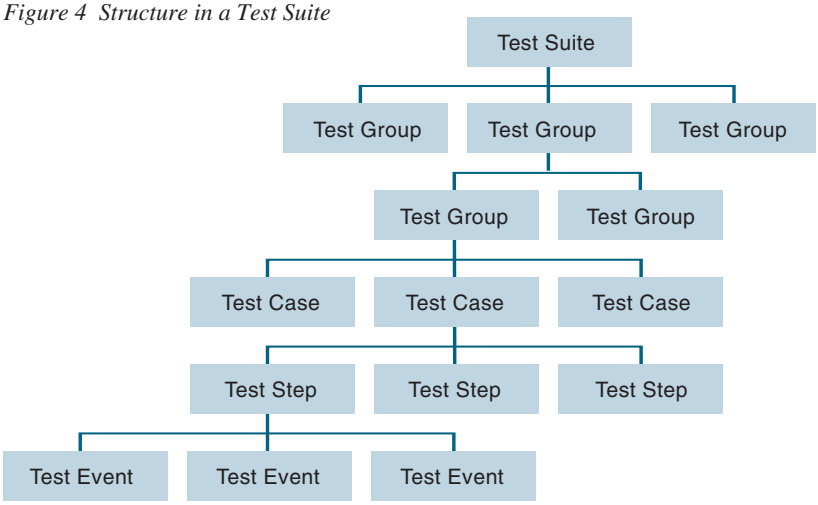


Figure 4 Structure in a Test Suite



The documents related to the test procedure are the specification on which the test suite is based, a PICS (Protocol Implementation Conformance Statement) document and a PIXIT (Protocol Implementation eXtra Information for Testing) document. In most cases, the referenced specification is a protocol standard. The PICS document is a questionnaire on mandatory and optional features of the IUT and the PIXIT document is a questionnaire on additional information required for the test execution such as address and timer information.

The different elements of a TTCN test suite appear in a predefined strict order. Only in the dynamic part it is possible to define a logical structure for test cases, test steps and default behaviour descriptions by putting them into groups and subgroups (Figure 4). Test events are the smallest elements and are explained in Section 2.4.2. The test suite for the parcel service example contains only one test group, which is specified in Figure 5.

2.2 The Declarations Part

The declarations part provides definitions and declarations used and referenced in the subsequent parts of the test suite. Specifically, the declarations part defines and declares types, operations, selection expressions, test components, PCOs, timers, variables, constants and the encoding of ASPs and PDUs. In the following, the data types are explained and examples of operation definitions, test suite parameter declarations, variable declarations, timer declarations and PDU type definitions are given.

2.2.1 Data Types

TTCN has its own data type system and allows the usage and definition of ASN.1 data types.[6)] The TTCN type system includes the predefined data types INTEGER, BOOLEAN, BITSTRING, HEXSTRING, OCTETSTRING, and various character strings such as IA5String, NumericString and PrintableString. In addition, TTCN allows to define structured types which are comparable to C structures or ASN.1 sequences. For the usage of ASN.1, TTCN provides special tables, which include pure ASN.1 code.

2.2.2 Test Suite Operations

Test suite operations are comparable to functions in common programming languages like C or Pascal. They can be used to encapsulate any functionality relevant for the test execution such as setting up basic connections, resetting the IUT or just calculating a specific value. An example for the definition of a test suite operation is shown in Figure 6. The table header contains the name of the operation, a description of the input parameters and the type of the result. The table body includes the behaviour specification of the operation which may either be given in the form of a pseudo-code like procedural definition language or in the form of an informal textual description. For simplicity, the behaviour specification of the operation in Figure 6 has been omitted.

2.2.3 Test Suite Parameters

Test suite parameters are global parameters of the test suite. Typically, they are derived from the PICS and PIXIT documents and are constant during test execution. Test suite parameters serve as a basis for test case selection and for the parameterization of test cases. The declaration of the test suite parameter *Duration_T_ParcelService* of the parcel service example is shown in Figure 7. The parameter is used to set the duration of timer *T_ParcelService* in Figure 9.

2.2.4 Variables

TTCN supports two types of variables: test suite variables and test case variables. Test suite variables are defined globally and retain their values throughout the whole test campaign. They are used to pass information from one test case to another. Test case variables are also declared globally, but their scope is local to a test case. Each test case receives a fresh copy of all test case variables when it is started. The declaration of the test case variable *PS_Init* with the initial value *TRUE* is shown in Figure 8. In Figure 1, *PS_Init* is used to store the result of the test suite operation *Reset_ParcelService* given in Figure 6.

Test Suite Structure

Suite Name : Example_ATS
Standards Ref : None
PICS Ref : None
PIXIT Ref : None
Test Method(s) : Remote Test Method
Comments : This is an ATS for explaining selected TTCN constructs.

Test Group Reference	Selection Ref	Test Group Objective	Page Nr
Valid/		Test the valid behaviour of the Parcel Service.	18

Detailed Comments:

Figure 5 Test Suite Structure

Test Suite Operation Definition

Operation Name : Reset_ParcelService
Result type : BOOLEAN
Comments : This is used to initialize the Parcel Service, e.g. the storage capacity should be reset.

Description
...

Detailed Comments:

Figure 6 Definition of a Test Suite Operation

Test Suite Parameter Declarations

Parameter name	Type	PICS/PIXIT Ref	Comments
Duration_T_ParcelService	INTEGER		This is the timeout period for the timer to watchdog the indication process of the Parcel Service.

Detailed Comments:

Figure 7 Declaration of a Test Suite Parameter

Test Case Variable Declaration

Variable Name	Type	Value	Comments
PS_Init	BOOLEAN	TRUE	This is to store the result of the Reset_ParcelService TSO.

Detailed Comments:

Figure 8 Declaration of a Test Case Variable

6) *For further information on ASN.1, please refer to [7] and to the paper on ASN.1 in this issue of the journal.*

Timer Declaration			
Timer Name	**Duration**	**Unit**	**Comments**
T_ParcelService	Duration_T_ParcelService	min	Timer between Offer and Indication
T_Consumer	24*60	min	Timer between Send and Confirmation
Detailed Comments:			

Figure 9 Timer Declaration

ASN.1 PDU Type Definition		
PDU Name	:	Offer
PCO Type	:	LT_PCO
Encoding Rule Name	:	
Encoding Variation	:	
Comments	:	Ask for an offer to deliver certain goods by the Parcel Service
Type Definition		
SEQUENCE {type_of_good OBJECT IDENTIFIER, amount_of_good INTEGER}		
Detailed Comments:		

Figure 10 A PDU Type Declaration

ASN.1 PDU Constraint Declaration		
Constraint Name	:	Offer_Large
PDU Type	:	Offer
Derivation Path	:	
Encoding Rule Name	:	
Encoding Variation	:	
Comments	:	Sending a question for an offer to the Parcel Service
Constraint Value		
{type_of_good {1 2 3 4 5}, amount_of_good 100}		
Detailed Comments:		

Figure 11 Sending Constraint for an Offer PDU

2.2.5 Timer Declarations

As shown in Figure 9, timers are declared with their name, an optional default duration and a timeout period in the range from pico second (ps) up to minute (min). The two timers *T_ParcelService* and *T_Consumer* of the parcel service example (Figure 4) are declared in Figure 9. The default duration of *T_ParcelService* is set to the test suite parameter *Duration_T_ParcelService* (Figure 7). The default duration of *T_Consumer* is given by an expression of type INTEGER reflecting the *24h=24*60min* duration for the 24h service.

2.2.6 PDU Definitions

Instances of PDU types are (either directly or embedded in ASPs) sent to or received from the IUT at PCOs. As presented in Figure 10, the definition of a PDU type consists of its name, the PCO type associated with the PDU type, and a list of PDU fields. Each PDU field is defined by its name and its type. The encoding of PDU fields follows the relevant protocol specification unless encoding information is included in the test suite. Figure 10 defines the *Offer* PDU of the parcel service example (Figure 2). The definition is given in the form of an ASN.1 PDU type definition and shows the usage of ASN.1 in TTCN. The table body includes a pure ASN.1 type definition. The Offer PDU type uses a unique object identifier referring to the type of goods and an INTEGER value referring to the amount of goods that should be delivered.

2.3 The Constraints Part

The constraints part of a TTCN test suite provides the values of the PDUs (and ASPs) to be sent to or received from the IUT. This is done by means of PDU (and ASP) constraints. A PDU constraint is related to a PDU type and describes a concrete value or value ranges of the PDU type. Constraints are referenced in the dynamic part of a test suite (Section 2.4) in order to describe the PDU exchange in the different test cases. A constraint specification will follow the structure of the corresponding PDU type and can be specified either in tabular form or in the form of the ASN.1 value notation.

A constraint for a PDU which should be sent to the IUT, has to provide concrete values for all PDU fields. An example constraint for the *Offer* PDU type (Figure 10) is presented in Figure 11. It defines the value of the *type_of_good* field to be *{1 2 3 4 5}* and the value of the *amount_of_good* field to be *100*.

A constraint for a PDU that is received from the IUT may define value ranges for the PDU fields. TTCN provides powerful matching mechanisms to specify specific values (if concrete values are expected) and to specify value ranges (if several values are expected). Value ranges can be specified by referring to any value of a given type, by listing specific values, by complementing specific values or by providing value patterns. Value patterns are described by using wildcards such as '*', '-' or '?'.

An example for a constraint of the *Indication* PDU which should be received from the IUT of the parcel service example is shown in Figure 12. The *Indication* PDU which matches the constraint must have information about the delivered goods, i.e. the kind and amount of goods must be identical to the information contained in the preceding *Offer* PDU (Figure 11). In addition, an order number and an indication of the 24h service has to be received, but these can have any values (indicated by '?'). The comment field can be omitted or have any value (indicated by '*').

ASN.1 PDU Constraint Declaration		
Constraint Name	:	Indication_24h
PDU Type	:	Indication
Derivation Path	:	
Encoding Rule Name	:	
Encoding Variation	:	
Comments	:	Receive an offer from the Parcel Service.

Constraint Value
{goods {type_of_good {1 2 3 4 5}, amount_of_good 100}, order ?, delivery_time 24, comments*}
Detailed Comments:

Figure 12 Receiving Constraint for an Indication PDU

2.4 The Dynamic Part

The dynamic part describes the dynamic behaviour of the tester processes by test cases, test steps and default behaviour descriptions.

2.4.1 Test Cases, Test Steps and Default Behaviour Descriptions

A test case is a complete program, which has to be executed in order to judge whether a test purpose (cf. Box 1) is fulfilled or not. Test cases can be structured into test steps and default behaviour descriptions.

A test step can be seen as a procedure definition which can be called in test cases by means of an ATTACH operation. Figure 13 presents a TTCN test case description. In lines 1 and 4 of the table body, the test steps *Preamble* and *Postamble* are attached to the test case behaviour. The corresponding TTCN specifications can be found in Figure 1 and Figure 14.

A default behaviour description is a special test step and copes with exceptional test situations where the IUT does not behave in an expected manner. In contrast to a test step, a default behaviour description is not used inside a test case or test step behaviour description. Instead, it is referenced in the table header. Figure 15 presents the default behaviour description *OtherwiseFail*. This is the default for the parcel service example and referenced by the test case shown in Figure 13.

The specification of the test behaviour is identical for test cases, test steps and default behaviour descriptions and can be found in the body of the corresponding tables (Figures 1, 13, 14, 15). The body consists of columns and rows. The Nr. column includes row numbers. The *Label* column allows to specify labels for the TTCN statements defined in the *Behaviour Description* column. The *Constraints Ref.* column provides references to constraints (Section 2.3). The *Verdict* column includes verdict assignments to indicate the success or failure of a test run with respect to the sequence of statements that have been performed. In the follow-

Test Case Dynamic Behaviour		
Test Case Name	:	Indication_1
Group	:	Valid/
Purpose	:	
Configuration	:	
Default	:	OtherwiseFail
Comments	:	
Selection Ref	:	
Description	:	Test the offer and indication sequence of behaviour.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+Preamble			
2		LT_Prod !Offer START T_ParcelService	Offer_Large		
3	L1	LT_Prod ?Indication CANCEL T_ParcelService	Indication_24h	(P)	
4		+Postamble			
5		LT_Prod ?Advertisement	Advertisement_Any		Ignore advertisement
6		GOTO L1			

Detailed Comments:

Figure 13 Test Case Description

Test Step Dynamic Behaviour		
Test Step Name	:	Postamble
Group	:	
Objective	:	To reset the test system and to assign the final verdict.
Default	:	
Comments	:	
Description	:	

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CANCEL			
2		[TRUE]		R	

Detailed Comments:

Figure 14 Test Step Description

Dynamic Dynamic Behaviour

Default Name : OtherwiseFail
Group :
Objective : Cover all unexpected reactions from the IUT.
Comments :
Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		LT_Prod?OTHERWISE		F	
2		LT_Cons?OTHERWISE		F	
3		?TIMEOUT		F	

Detailed Comments:

Figure 15 Default Behaviour Description

ing sections, all TTCN statements are introduced, the execution of behaviour descriptions is explained and the assignment of test verdicts is described.

2.4.2 TTCN Statements

The *Behaviour Description* column includes TTCN statements. TTCN statements can be grouped into test events, control constructs and pseudo events.

Test events are SEND, IMPLICIT SEND, RECEIVE, OTHERWISE and TIMEOUT. SEND and IMPLICIT SEND specify the sending of PDUs. RECEIVE and OTHERWISE denote the processing of received PDUs. OTHERWISE is the mechanism for dealing with unforeseen test events and denotes that the test system shall accept any incoming PDU. TIMEOUT events check for the expiration of timers. Test events may be qualified and/or followed by assignments and timer operations. Instead of keywords, TTCN uses '!' to describe send events and '?' to denote receive events. For example, the statement *LT_Prod ! Offer* (Figure 13, line 2) describes the sending of PDU *Offer* via PCO *LT_Prod* to the IUT and the statement *LT_Prod ? Advertisement* (Figure 13, line 5) denotes the reception of PDU *Advertisement* at PCO *LT_Prod* from the IUT.

Control constructs are ATTACH, GOTO and REPEAT. The ATTACH construct allows to attach test steps. GOTO transfers control to a statement identified by a label in the *Label* column and REPEAT is used for the specification of loops.

Pseudo events are qualifiers (i.e. boolean expressions), timer operations (i.e. SET, READTIMER and RESET) and assignments.

The TTCN statements in a behaviour description can be grouped into statement sequences and sets of alternatives. Statement sequences are represented one after the other on separate lines, being indented from left to right. The statements on lines 1 to 4 in Figure 13 constitute a statement sequence. Statements on the same level of indentation and identical predecessor form a set of alternatives. In Figure 13, the statements on lines 3 and 5 form a set of alternatives. They are on the same level of indentation and their common predecessor is the statement on line 2.

2.4.3 Behaviour Execution

The execution of a behaviour description will be explained by means of Figure 13. Execution starts with the first level of indentation (line 1) and proceeds towards the last level of indentation (lines 4 and 6). Only one alternative out of a set of alternatives at the current level of indentation is executed, and execution proceeds with the next level of indentation relative to the executed alternative. For example, the statements on lines 3 and 5 are alternatives. If the statement on line 3 is executed, processing continues with the statement on line 4. Execution of a behaviour description stops if the last level of indentation has been visited, a final test verdict has been assigned (see below), or a test case error has occurred.

Before a set of alternatives is evaluated, a snapshot is taken. This means that the state of the test component, the state of all PCOs and all expired timer lists related to the test case are updated and frozen until the set of alternatives has been evaluated. This guarantees that the evaluation of a set of alternatives is an atomic and deterministic action.

Alternatives are evaluated in the order of their specification. The first alternative with successful evaluation is executed, i.e. all conditions of that alternative are fulfilled. Execution then proceeds with the set of alternatives on the next level of indentation. If no alternative can be evaluated successfully, a new snapshot is taken and the evaluation of the set of alternatives is started again.

2.4.4 Verdict Assignment

Test verdicts are assigned in the *Verdict* column of test cases, test steps and default behaviour descriptions. TTCN supports three different verdicts: PASS to indicate that the test behaviour gives evidence for conformance, FAIL to describe that the specification has been violated, and INCONCLUSIVE for cases where neither a PASS nor a FAIL can be given.

TTCN distinguishes between preliminary and final test verdicts. Preliminary verdicts are given in parentheses, e.g. the preliminary PASS in line 3 of Figure 13. Final verdict assignments are specified without parentheses, e.g. the three final FAIL verdicts in Figure 15. The difference

between a preliminary and a final test verdict is that the assignment of a final test verdict terminates the test case execution, i.e. it can be considered to be a combination of a verdict assignment and a subsequent stop operation.

For the handling of test verdicts, each test case has a predefined variable *R*. Variable *R* stores the current preliminary verdict of a test case and its value becomes the final verdict if the test case ends without the assignment of a final verdict. In other words, the assignment of a test verdict in the *Verdict* column of a behaviour description is an assignment to variable *R*. As shown in Figure 14, variable *R* can also be used to calculate the final verdict of a test case. The entry *R* in the *Verdict* column indicates that the test case ends and that the actual value of *R* will be the final verdict.

There are special rules for the assignment of verdicts during the execution of a test case. They are shown in Figure 16 and can be summarized as: *"A verdict can only become worse"*. For example, if the value of *R* is (FAIL), then the assignment of (PASS) or (INCONCLUSIVE) will have no effect on *R*. Please note that the value *none* in Figure 16 describes the situation where *R* has not been initialized, i.e. no preliminary verdict has been assigned to *R*.

2.4.5 The Example Test Case

The test case *Indication_1* in Figure 13 should be read as follows: the test case starts with the execution of test step *Preamble* in order to initialise the parcel service. Afterwards, an *Offer* is sent to the parcel service at PCO *LT_Prod* and the timer *T_ParcelService* is started. Then, two alternative events are expected:. Either, an *Indication* or an *Advertisement* is received. If an *Indication* is received (line 3), the timer T_ParcelService is cancelled, a preliminary *PASS* verdict is assigned and *Postamble* is executed in order to reset the test system. If an *Advertisement* is received (line 5), a GOTO statement is used (line 6) to put the test case control back to the set of alternatives at label *L1* in order to await the expected *Indication* PDU.

The *Preamble* (Figure 1) executes the test suite operation *Reset_ParcelService* and stores the result in the test case variable *PS_Init*. In case of a successful initialisation, i.e. the value of *PS_Init* is TRUE, a preliminary *PASS* verdict is assigned (line 2) and the test case proceeds with the execution. If the initialisation is not successful, i.e. *PS_Init* has the value FALSE, a final *INCONCLUSIVE* verdict is assigned (line 3) which terminates the test execution.

The *Postamble* (Figure 14) resets the test system by cancelling all running timers (line 1). Finally,

Current value of R	Entry in verdict column (PASS)	(INCONC)	(FAIL)
none	pass	inconc	fail
pass	pass	inconc	fail
inconc	inconc	inconc	fail
fail	fail	fail	fail

Figure 16 Handling of Test Verdicts

it assigns the final verdict by referring to the value of the special verdict variable *R* (line 2).

The default behaviour *OtherwiseFail* (Figure 15) defines that the reception of any other PDU at *LT_Prod* (line 1) or *LT_Cons* (line 2) will lead to the assignment of a FAIL verdict and the termination of the test case. In addition, the occurrence of a timeout (line 3) will also terminate the test case with the final test verdict FAIL.

3 Concurrency in TTCN

The term concurrent TTCN refers to TTCN language constructs and concepts for the description of concurrent test cases. In concurrent TTCN, each test case consists of several test components that execute independently and in parallel. A Main Test Component (MTC) controls the test case execution and creates Parallel Test Components (PTCs). The MTC cannot stop PTCs but has the possibility to check their termination by means of a DONE statement. A test case always ends when the MTC ends. Each test component controls its own local verdict. The final verdict of a test case is calculated according to the rules described in Figure 16 by the MoT.

Test components can coordinate themselves by exchanging Coordination Messages (CMs) at Coordination Points (CPs). CPs connect test components and are similar to PCOs. CMs are similar to PDUs, but they are used for the information exchange among test components only.

A concurrent test configuration for the parcel service example is given in Figure 17. It uses *PTC_Main* as MTC, which creates the PTCs *PTC_Prod* and *PTC_Cons*. The PTCs control

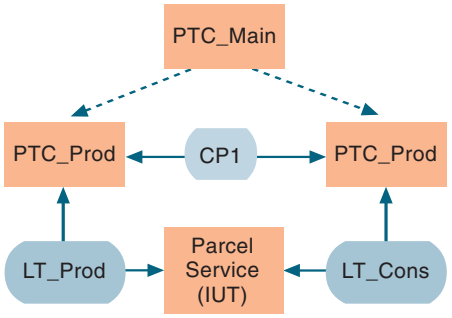


Figure 17 Example of a Concurrent Test Architecture

Test Case Dynamic Behaviour

Test Case Name : Deliver_1
Group : Valid/
Purpose :
Configuration : Example_Conc_Conf
Default :
Comments :
Selection Ref :
Description : Test the offer and indication sequence of behaviour.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+Preamble			
2		CREATE (PTC_Prod: Deliver_1_Prod, PTC_Cons: Deliver_1_Cons)			
3		?DONE(PTC_Prod, PTC_Cons)		(P)	
4		+Postamble			

Detailed Comments:

Figure 18 Description of PTC_Main

Test Step Dynamic Behaviour

Test Step Name : Deliver_1_Cons
Group : ConcurrentVersion/
Objective :
Default : OtherwiseFail_LT_Cons
Comments :
Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CP1 ?Continue(Order_No:= Continue.order)	Continue_Order_Recv		
2		LT_Cons ?Deliver	Deliver_Large (Order_No)	(P)	
3		LT_Cons !Acknowledge	Acknowledge_Large (Order_No)		

Detailed Comments:

Figure 19 Description of PTC_Cons

```
testcase Indication_1() runs on MTCType {
// Test the offer and indication sequence of behaviour

        activate(OtherwiseFail)
        Preamble();
        LT_Prod.send(Offer_Large);
        T_ParcelService.set;
        alt {
                []      LT_Prod.receive(Indication_24h) {
                                T_ParcelService.stop;
                                verdict.set(pass);
                                Postamble()
                        }
                []      LT_Prod.receive(Advertisement_Any); {
                                goto alt;
                        }
        }
}
```

Figure 20 TTCN-3 Description of Figure 13

and observe the IUT via the PCOs *LT_Prod* and *LT_Cons*. They coordinate themselves through the coordination point *CP1*.

The definition of the MTC *PTC_Main* for an example test case is shown in Figure 18. After the *Preamble* (line 1), *PTC_Main* creates *PTC_Prod* and *PTC_Cons* (line 2). The behaviour of the PTCs is given by the test step descriptions *Deliver_1_Prod* and *Deliver_1_Cons*. The test behaviour can be related to Figure 2: *Deliver_1_ Prod* covers the testing of the sequence from *Offer* to *Confirmation* at PCO *LT_Prod*, while *Deliver_1_Cons* covers the *Delivery* and *Acknowledgment* events at PCO *LT_Cons*. *PTC_Main* waits for the termination of the two PTCs through a DONE statement. The MTC assumes their successful termination by assigning a preliminary pass verdict[7)] (line 3). Finally, the *Postamble* finishes the test case.

The use of a CM is shown in Figure 19. The CM Continue on line 1 enables the PTC to proceed with the test execution by expecting to receive a *Delivery* PDU at PCO *LT_Cons* (line 2). The *Delivery* should be parameterized with the correct order number which was sent to *PTC_Cons* as a parameter of the CM. This number is used as a parameter to the constraint for *Delivery*. At the end, *PTC_Cons* initiates a proper *Acknowledgment* and terminates.

4 Outlook: Next Version of TTCN

Currently, the third edition of TTCN (TTCN-3) is in work at ETSI [4, 10]. TTCN-3 is a text-based language for the specification of tests for reactive systems. TTCN-3 is on a syntactical (and methodological) level a drastic change compared to the previous TTCN versions. However, the main concepts of TTCN have been retained and improved and new concepts have been included, so that TTCN-3 will be applicable for a broader class of systems. New concepts are e.g. a test execution control program to describe relations between test cases such as sequences, repetitions and dependencies on test outcomes, dynamic concurrent test configurations, and test behaviour in asynchronous and synchronous communication environments. Further improved concepts are, e.g. the integration of ASN.1, the module and grouping concepts to improve the test suite structure, and the test component concepts to describe concurrent test setups.

---

7) Please note that any worse verdict returned by one of the test components will overrule this assignment according to the table given in Figure 16.

The top-level unit of a TTCN-3 test suite is the module which can import definitions from other modules. A module consists of a definitions part and a control part. The definitions part of a module covers definitions for test components, their communication interfaces, type definitions, test data templates (previously known as constraints), functions, and test cases. The control part of a module calls the test cases and describes the test campaign. For this, control statements similar to statements in other programming languages (e.g. if-then-else and while loops) are supported. They can be used to specify the selection and execution order of individual test cases. TTCN-3 provides a variety of constructs to describe test behaviour within a test case such as the alternative reception of communication events and their interleaving. Moreover, default behaviour can be covered, e.g. unexpected reactions from the system under test. In addition to the automatic test verdict assignment, more powerful logging mechanisms are provided, e.g. for detailed tracing. An example of a TTCN-3 test case definition is shown in Figure 20. It is the TTCN-3 representation of the TTCN test case in Figure 13.

In addition to the pure textual format, TTCN-3 will define at least two presentation formats: A tabular conformance testing presentation format [5] that resembles the tabular form of TTCN and a graphical presentation format [6, 8] that supports the presentation and also the development of TTCN-3 test cases, as Message Sequence Charts (MSC).

References

1 ISO. Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework. – Seven Parts Standard. Geneva, 1991–1999 (includes [2] and [3]). (ISO/IEC 9646.)

2 ISO. *Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 2: Abstract test suite specification.* Geneva, 1991. (ISO/IEC 9646-2.)

3 ISO. *Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN).* 2nd ed. Geneva, 1998. (ISO/IEC 9646-3.)

4 ETSI. *TTCN-3 – Core Language. European Norm (EN) 00063-1 (provisional).* Sophia-Antipolis, 2000. (ETSI TC MTS.)

5 ETSI. *TTCN-3 – Tabular Presentation Format.* EN00063-2 (provisional). Sophia-Antipolis, 2000. (ETSI TC MTS.)

6 ETSI. *TTCN-3 – MSC Presentation Format. EN00063-3 (provisional).* Sophia-Antipolis, 2000. (ETSI TC MTS.)

7 ITU. *Information Technology – Abstract Syntax Notation One (ASN.1).* Geneva, 1994. (ITU-T Recommendations X.680-683.)

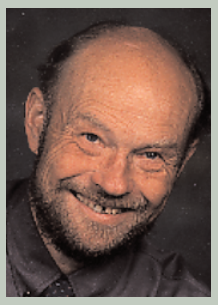
8 ITU. *Message Sequence Chart (MSC).* Geneva, 2000. (ITU-T Recommendation Z.120.)

9 Walter, T, Schieferdecker, I, Grabowski, J. Test Architectures for Distributed Systems – State of the Art and Beyond (Invited Paper). In: *Testing of Communicating Systems.* Petrenko, A, Yevtuschenko, N (eds.). Dordrect, Kluwer, 1998, 149–174. (Volume 11.)

10 Grabowski, J et al. On the Design of the new Testing Language TTCN-3. In: *Testing of Communicating Systems.* Ural, H, Probert, R L, von Bochmann, G (eds.). Dordrect, Kluwer, 2000, 161–176. (Volume 13.)

On Methodology Using the ITU-T Languages and UML

ROLV BRÆK



Rolv Bræk (56) received his Siv.ing. degree (M.S.E.E.) in 1969 from the Norwegian University of Science and Technology (NTNU) and is currently Professor in the Department of Telematics at NTNU. Rolv Bræk has extensive experience from application development using formal methods as well as from teaching, consulting and introducing systems engineering methodologies to industry. He is co-author of the book "Engineering Real Time Systems – An Object Oriented Methodology using SDL", and "TIMe The Integrated Method" published on CD-ROM by SINTEF. His current research interest is rapid service development.

Rolv.Braek@item.ntnu.no

A methodology is a system of methods and principles, where each method defines a systematic way to produce given results. The general goal of a systems development methodology is to support development of systems of a high quality through a controlled process that is as efficient as possible. To this end a methodology seeks to prescribe what are good practices and what are not so good practices. Although they share the same overall goal, there are considerable differences among methodologies. A framework will be presented in the following that enables some principles to be identified and some important differences to be highlighted. One interesting question is how the ITU-T language family and UML stand in relation to each other from a methodology point of view.

Why Methods?

Nobody believes that knowing a natural language like English or Chinese is sufficient to write great novels. Nevertheless, some people believe that knowing a programming language like C or Java is all you need to write great software. Surprisingly many believe that knowing analysis and design languages such as the ITU-T languages or UML is sufficient to design great systems. By simply introducing UML, with supporting tools, they believe their development teams will design better systems at lower cost. This is of course not true! Like writing novels or making programs, system design takes a lot more than language knowledge.

Part of this additional knowledge, when written down and presented systematically, is generally known as *methods* and *methodology*. Methods define a systematic way to produce given results, and therefore, constitute the core of a discipline like systems engineering. Without any methods there would be no discipline! In the context of systems engineering, methods prescribe how to go about producing specific results such as requirement specifications, design descriptions and test plans. Methodology is a system of methods and principles, put together to cover a larger part of the systems engineering process than a single method. The goal of systems engineering methodology is to help make better systems more efficiently and with better control. Ideally, knowing a methodology should be the same as knowing how to make great systems. In reality, good methodology helps people make sound solutions by bringing order into chaos and providing guidelines, but it does not replace human capabilities (creativity, experience, theoretical knowledge, etc.). It is a kind of condensed and generalised experience presented in a systematic way.

To understand the scope of systems engineering methods, it is necessary to step back and consider what it takes to develop successful products.

The Macro Cycle

The bottom line is that successful systems satisfy real needs existing in some domain (market). The key to success is therefore first and foremost to understand the needs that are present in the domain (both those that are explicitly expressed and those implicitly present). The second issue is to analyse the needs and define (specify, design, implement) systems that can satisfy (some of) the needs in a cost-effective way. The last issue is to manufacture and install systems into the domain such that the needs are really satisfied and a maximal market share is created. This picture is not static. In the next run, the domain is changed by the installed systems, experience is gained and new needs arise that cause the cycle to repeat in a spiral-like way as illustrated in Figure 1.

Figure 1 also illustrates that systems development deals with a reality consisting of domains and systems on the one hand and descriptions of the reality on the other. Traditional crafts use few descriptions[1)] if any, and most effort is focused on the reality itself. In systems engineering it is the opposite. It is an intellectual process where most effort is focused on descriptions, and comparatively little on the reality itself[2)]. The main reason is that descriptions (e.g. mathemati-

---

1) The term "description" is used here in the general sense of a symbolic representation of something as it is also used in the term "Formal Description Technique" (FDT). We do not distinguish between prescriptions, inscriptions and descriptions here.

2) One must of course always have the reality in mind and ensure that descriptions are valid.

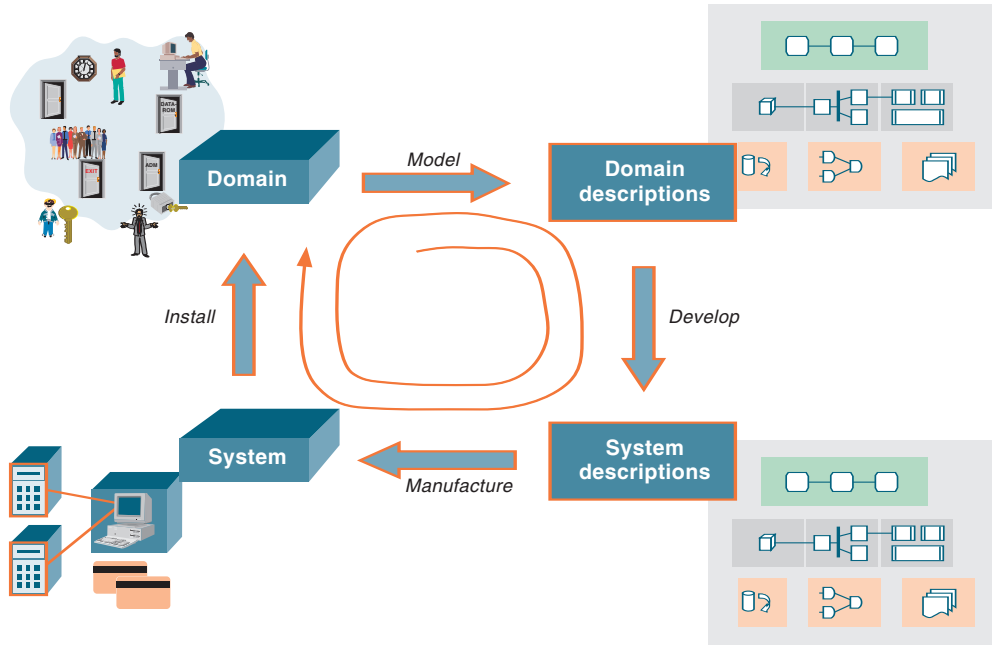


Figure 1 The macro cycle/spiral

cal models, SDL diagrams, Java programs[3]) are the only means by which the reality can be properly understood, communicated and analysed by human beings and by machines. Another reason is that, in many cases, one cannot afford to experiment and make mistakes with the real thing. Consequently, descriptions are the main objects of systems and software engineering.

Having established the importance of descriptions, the next issue is to decide what descriptions to make and how to make them. These are the primary questions any systems engineering methodology should answer[4].

Most methodologies will agree with the macroscopic picture in Figure 1, that there are at least domain descriptions and system descriptions (although domain descriptions may not be emphasised by every methodology). Domain descriptions are developed in order to enable the people involved to understand an existing reality, assess needs and plan new systems. System descriptions, on the other hand, are used to create a new reality satisfying the needs.

Although they are used for two different purposes (the domain descriptions to understand and analyse; the system description to produce and document), they both describe a reality. In both areas we are looking for suitable ways to describe complex realities. The main difference between domain descriptions and system descriptions is that system descriptions are used constructively when manufacturing systems, and must describe technical solutions in more detail and more precisely than domain descriptions. In addition, they must be organised in a way that supports an efficient and controlled development process. Domain descriptions, in contrast, should not go into system specific detail, but rather provide a common conceptual frame of reference for product planning and system development. They should have a wider scope than system descriptions, in order to capture the needs and future usage context of systems in the domain. Once they are made, domain descriptions can be re-used in all systems developments targeting that domain, as long as the domain remains stable. It should be noted however, that many alternative domain descriptions may be developed for the same domain, and that the domain descriptions are likely to evolve as more insight and experience is gained even if the domain itself is not changing.

---

[3] *Note that programs are descriptions with the special property that a machine in the real world may execute them.*

[4] *But not every methodology has done that. Some have focused primarily on activities and treated descriptions as secondary issues.*

It is important to understand that the full macro cycle need not be performed for each system instance. Typically many system instances will be produced from the same system descriptions, and a given domain description may hold for several system descriptions. Each macro step produces results that can be maintained separately and can be the responsibility of different departments in a company, if so desired.

Note that the descriptions (of domain and system) are the basis for communication between all the parties involved. Tradition has often caused different groups (e.g. marketing people and developers) to develop and use very different description techniques, which means that different groups communicate poorly and that the same information is repeated in different forms. An opportunity for improvement on the enterprise level is to introduce common languages and methods reducing the number of separate descriptions and increasing the value of each.

The macro cycle in Figure 1 illustrates the golden principle of all methodologies, which is *separation of concerns*; in this case, to describe independent aspects in separate descriptions. A consequence of this principle is that each description focuses on aspects that are dependent.

How to Describe Complex Realities?

In this section we do not distinguish between domain descriptions and system descriptions, but focus on the general problem of describing a complex reality[5)]. We shall try to establish some basic principles and principal solutions before discussing language and method issues.

First of all, it is important to realise that reality is so complex that a human being is unable to keep it all in mind at the same time. It is absolutely necessary to factor out aspects that can be understood one at a time and then be combined into an understanding of the whole. Two golden rules should be applied in combination:

- *Separation of concerns*. Identify aspects that are as independent as possible and describe them separately. In this way, the complexity of each description is reduced, and may therefore be expressed more clearly. Separation also helps to increase the modularity of a set of descriptions by allowing descriptions concerned with independent aspects to be changed independently. Finally, if different aspects require different skills and knowledge, separation helps to utilise different kinds of expertise better. Note that separation is not a goal in itself. Little is gained if the separated parts are too dependent.

- *Conceptual abstraction*. The general idea with abstraction is to remove irrelevant detail in order to focus on the essential. Conceptual abstraction means to replace low level concepts representing technical detail by more abstract concepts that are better suited to describe and study some aspects, i.e. by some kind of model. Mathematical models used for performance analysis and strength calculations are well known examples of conceptual abstraction.

In order to be useful, a conceptual abstraction must capture the nature of the phenomena in a way that helps to improve understanding, communication and/or analysis. The essential purpose of ICT systems is to perform some *logical behaviour*[6)] and handle some *information*. This sets them apart from systems where physical strength, power or movement is part of the purpose. Physical devices like computers, cables, screens and cabinets are merely the means to realise ICT systems while their purpose is to provide some functionality.

In order to focus on the essential, we need to separate the *functionality*[7)] in terms of logical behaviour and information handling from the accidental way it is implemented, and to model the functionality using a suitable conceptual abstraction.

Three main aspects that are largely independent can be identified and separated in different descriptions[8)]:

- *Functionality*. This is a conceptual abstraction of logical behaviour and information. The purpose is to describe logical behaviour and information as clearly as possible. And to do so

---

5) In order to describe the "reality" we may apply generalisation into types as well as conceptual abstractions. Descriptions may therefore be organised as description of general concepts (types, classes) and descriptions of particular phenomena (objects, instances).

6) We distinguish between logical behaviour and physical behaviour. Logical behaviour is the idealised behaviour that provides the system services. It is usually discrete, while physical behaviour involves physical properties and is usually continuous [14].

7) Note that "functionality" is used here as a term to represent information handling and logical behaviour in a general sense. The mathematical term "function" has a more restricted meaning.

8) A further refinement is normally needed, but this serves to illustrate the main points.

in terms that enable users and developers to communicate precisely, to establish a common understanding, and to ensure that the descriptions of functionality correctly represent the existing domain and/or the system being developed. It provides a view where the system may be seen as a whole, independent of realisation and technology. Functionality is normally described in terms of structures of active and passive objects with associated object behaviours.

- *Realisation*. This is a precise technical definition of the realisation in terms of the different technologies used, such as mechanics, electronics and software. This view is necessary to actually produce a working system. A large number of realisations will normally be possible for a given functionality, and the choice will depend on what properties are desired from the realisation itself (often called non-functional properties). If properly separated, a given description of functionality may hold for several realisations.

- *Deployment*. This defines a mapping between functionality and realisation by describing the realisation (the physical system) on a high level, by identifying the technologies used and by describing how and where the functionality is realised. It should focus on aspects that come in addition to the functionality, such as distribution, hardware/software allocation and use of middleware. Deployment and Functionality together may constitute the main design documentation.

As indicated in Figure 2, the separations may be considered as different viewpoints of the same reality.

The Reference Model for Open Distributed Processing (RM-ODP), identifies a different set of viewpoints: Enterprise, Information, Computational, Engineering and Technology, where the Computational and Information viewpoints cover Functionality [1], while the Engineering and Technology viewpoints (roughly) cover Deployment.

Indeed, most contemporary methods for systems engineering identify these aspects, but the way that the aspects are represented in descriptions vary considerably, and so does the terminology used. Some will claim that it does not matter which method is used as long as some methods are used. This may well be true for methods based on similar conceptual abstractions, e.g. methods based on communicating state machines, but there are significant differences between conceptual abstractions and some of these differences have a strong impact on methodology. The most important differences are found in the conceptual abstractions and languages used to express functionality.

How to Describe Functionality?

It is desirable that the languages for functionality satisfy at least three essential requirements:

- *Human comprehension*. Functionality should be represented in a way that enables human beings to fully understand it and to communicate precisely about it. To this end the concepts of the language must be well defined, and easy to understand.

- *Analytical possibilities*. It should be possible to reason about behaviours in order to compare systems, to validate interfaces, and to verify properties. This requires a semantic foundation suitable for analysis.

- *Realism*. The language should build on concepts that can be effectively realised in the real world. Although overlooked in many cases, this requirement is essential for two main reasons: 1. That it shall be possible to implement the functionality, 2. That the description of the functionality can serve as valid documentation of the real system.

The choice of conceptual abstraction is the main key to all this. One reason why methods for communicating systems have developed differently from general software engineering methods is that they deal with different domains. In communicating systems, interactions between concurrently operating, physically distributed objects have been central, while data structures and algorithms have been more central in other areas. For this reason the conceptual abstractions developed for communicating systems have put concurrency and communication up front, and have treated data and algorithms in that context. This is clearly reflected in SDL where a system must be defined in terms of communicating state machines with a sequential behaviour that encapsulates data and algorithms. UML, in con-

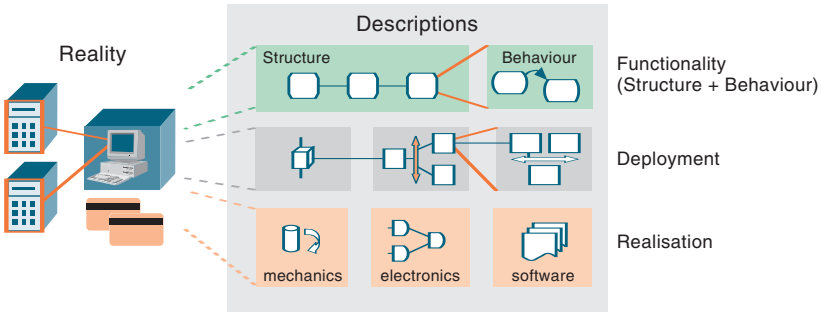


Figure 2 Three main aspects or viewpoints

trast, treats concurrency and State Machines as options, while Classes with Attributes and Operations are put up front. One may say that SDL puts the logical behaviour up front, while UML puts information up front.

When different domains now are merged into modern ICT systems, there must be ways to combine the approaches. The well-known problems from the communication domain do not disappear. Logical behaviour involving concurrency and interactions will still be an important aspect to factor out, as it determines the way that services are provided to users and thereby how the users will judge the system quality. Logical behaviour is both complex and difficult to understand due to its dynamic (transient) nature. Therefore a conceptual abstraction of behaviour that facilitates understanding and communication about logical behaviour is extremely important.

The drawback with programming languages in this area is that realism is achieved at the expense of human comprehension and analytical possibilities. Although programming languages (like CHILL, Java or C++) can describe logical behaviour, they do it by specifying action sequences. These are better suited to instruct the machine *how* to do things than to explain for the human being *what* is done. Humans need to understand the external interactions and the resulting evolution of state transitions that take place when the behaviour executes.

Some of the more mathematical languages, like LOTOS and Z, emphasise analytical possibilities at the expense of realism (and to some extent human comprehension). This is one reason why they have not gained wider popularity.

Communicating state machines of different forms are presently the best known conceptual abstractions used to describe logical behaviour, because they satisfy all three requirements above. State machines can be implemented in many ways, and combine realism with human comprehension and analytical possibilities. For this reason state machines are part of most contemporary approaches, including UML and SDL. But in UML they are optional.

Object and Property Descriptions

There is a general trend now to use object orientation as a common approach to describe both functionality, deployment and realisation. It is also a trend to use interaction scenarios to describe interactions between users and systems (use cases) and between objects within systems and among systems. Finally, state-transition based specifications are used to define the behaviour of individual objects.

This is not very surprising, considering that more and more applications tend to be distributed and reactive. Object orientation helps to master complexity by providing a structure in terms of objects, and by factoring out common features in general classes or types. Objects do not live alone, but communicate with other objects. Describing the behaviour of each object in terms of states and transitions that are triggered by incoming signals from other objects has proven to be of great value, and is now adopted in one form or another in most system engineering approaches.

A system generally consists of a structure of *objects*. Systems and objects are defined by means of *object descriptions* that represent how objects are related to other objects, how they are composed from attributes and other objects, and how they behave. Objects may be defined using Types or Classes, and inheritance may be used to define the Types/Classes. UML Class Diagrams, SDL diagrams and Java classes are all examples of object descriptions. Object descriptions provide the perspective of designers.

Systems and objects generally have *properties*. Properties are defined by means of *property descriptions* that state external properties of a system or object without prescribing their internal construction. Property descriptions are not constructive, but are used to characterize an object from the outside. There are many kinds of properties: behaviour properties, performance properties, maintenance properties, etc. This is the perspective that is normally used in specifications. UML collaboration diagrams, MSC diagrams, test cases in TTCN, and performance figures are all examples of property descriptions.

As illustrated in Figure 3, the separation between objects and properties is orthogonal to the three main viewpoints identified in Figure 2, leading to six different aspects to describe.

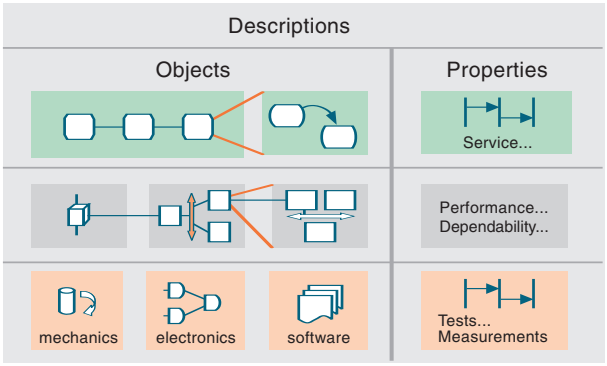


Figure 3 Objects and properties

Properties associated with functionality are often called *functional properties*, while properties associated with deployment and realization are called *non-functional properties*.

The interplay between property descriptions and object descriptions is central to systems development. Normally the first step is to specify the required properties, then to use the properties as input to synthesize a design (defined by means of object descriptions); and finally to verify that the design satisfies the properties. In principle these three steps are performed for each viewpoint. An important advantage of the ITU-T languages MSC (for properties) and SDL (for objects) is that the relationships between properties and objects can be formally defined and supported by tools.

It is important to note that a given property description may hold for many different object descriptions. A given set of MSC diagrams may, for instance be executed by several different SDL systems, and many alternative realizations may be able to pass the same set of TTCN test cases, and satisfy the same performance requirements.

Coverage by the ITU-T Language Family and UML

Figure 4 shows how the ITU-T languages and UML cover the description aspects identified in Figure 3. Apparently, the coverage is similar, with deployment as one exception. Here UML offers component diagrams and deployment diagrams, while the ITU-T languages have no notation at all[9)] (but the issue is addressed in a new question for the study period 2001-2004: question Q11/10 Deployment and configuration language). The ITU-T languages must therefore be combined with other notations in this area, such as the one presented in [2] that was also used in the SDL method guidelines [3], or by UML.

In the functionality area, the overlap is considerable after SDL-2000 has adopted (some of) the notation from UML Class Diagrams and the concept of composite states from UML State Machines. The need to use UML in combination with SDL has been greatly reduced by this, since the ITU-T languages now cover most aspects that were previously the strongpoint of UML.

When comparing UML with SDL, we may note some important conceptual differences:

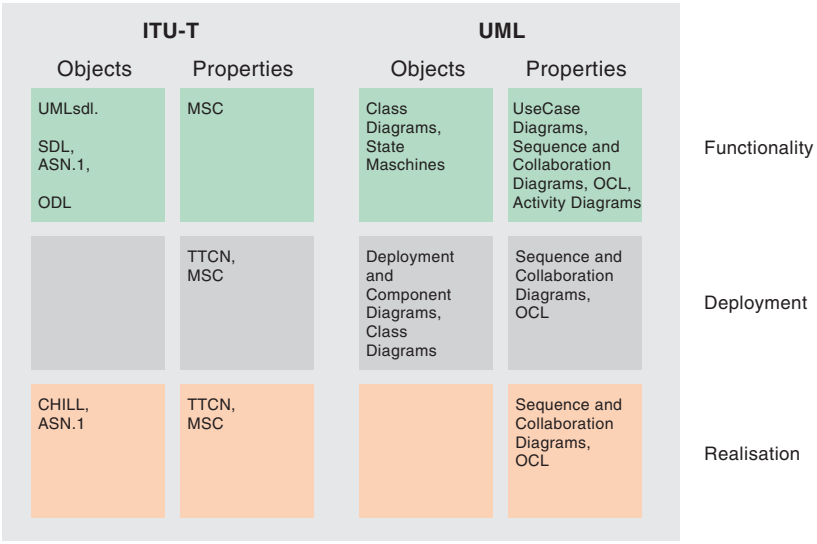


Figure 4 The ITU-T languages compared with the UML notations

- In SDL, the concept of a system is central. A system has well defined interfaces and is composed from agents that operate concurrently and communicate by asynchronous message passing. A system models an executing part of the real world, but may be defined with reference to a system type. In UML there is no such system concept. There are only classes and associations between classes. Object structures may be described, but only for illustrative purposes.

- In SDL, an agent may be composed from a possibly complex structure of agents and can be defined as an agent type. In this way, a composite type can be defined and used as one entity with well-defined interfaces. UML lacks a corresponding concept of composite types. Composition is just a special association between classes in UML.

- In SDL, concurrency and asynchronous communication between agents is the norm. State machines define the sequential behaviour of agents. In UML, concurrency and asynchronous communication is an option.

Some of these differences may disappear in UML 2.0, but presently UML is not really a system modelling language. It is an open ended and flexible language for modelling classes without internal structure (apart from attributes). This makes it well suited for visual modelling of object oriented software, but not really for system modelling – as yet. For system modelling SDL provides more powerful conceptual abstractions, and in addition an action semantics that makes

9) *One could argue that the ITU-T Object Definition Language (ODL) belongs to deployment, but the main focus of the language is objects and interfaces providing functionality. One may also argue that it is a property language, as the main focus is on interface definitions.*

its behaviour well defined. A methodology using UML may compensate for this to some extent by providing guidelines and rules and more precise interpretation of UML.

MSC and UML Sequence Diagrams are basically the same, but UML Sequence Diagrams lack the decomposition/composition mechanisms that MSC provides, and this gives MSC clear advantages when dealing with more complex cases. UML Collaboration Diagrams offer a view on interactions that many find useful, but this view is missing in the ITU-T language family.

TTCN and ASN.1 cover aspects that UML is missing entirely.

Note that Class Diagrams are mentioned also in the Deployment compartment for UML. One reason for this is that UML classes may be useful to visualise high-level realisation classes. A more important reason is that many people actually use UML Class Diagrams on a very low level just to visualise realisation classes, such as Java classes. Considerable method support is required to use UML with sufficient conceptual abstraction to be useful as Functionality description. Many UML users are not aware of this, but keep the UML use as a kind of visual programming. This is especially true for people with a strong programming background and little modelling experience.

Figure 5 The elaboration approach vs. the translation approach

Elaboration vs. Translation

We may distinguish between two fundamentally different approaches to systems development:

- *The elaboration approach*. Here the functionality is described using informal languages with incomplete semantics, and therefore the functionality description is incomplete. Details have to be added by elaboration during deployment design and realisation. As a result, the realisation description ends up as the only complete view of the system and is often the only one that is maintained. This gradually reduces the value of the other views, and makes the documentation very realisation dependent. When the technology and platforms change, as they do fast these days, more than necessary must be redone, because it is hard to factor out and reuse functionality that is not changed. Mainstream software engineering has followed the elaboration approach, and this has also been the case for most UML use including the Rational Unified process, RUP.

- *The translation approach*. Here the functionality is described as completely as possible using a language with well-defined and realistic semantics. The deployment is kept as orthogonal as possible to the functionality (using the principle of distribution transparency), and realisation of functionality is carried out by (manual or automatic) transformation of the functionality description. Advantages of this approach are that the functionality can be completely defined, analysed and simulated before implementation, and that the functionality description can remain valid for the realisation and serve as documentation. It may even be possible to generate the (application part of) realisations automatically, but this depends on the language and available tools. Functionality descriptions can apply to several implementations and survive technology and platform changes, and thereby give better return of investment, which is desirable, since functionality tends to last longer than realisations.

The impact that this choice of approach has on methodology and on the potential for process improvement should be obvious. But it has been a popular opinion that the translation approach is infeasible in a real industrial setting, and therefore the take-up of the translation approach has been slow. This may be partially caused by reluctance to introduce formal methods, especially among programmers. However, the translation approach does work, and many companies are now using it routinely in their product developments with good results. Experience from using the translation approach with SDL has shown that the approach helps to reduce the

number of errors in systems considerably[10], even when the translation to implementation is done manually. With automatic code generation, the figures are even better, because the translation errors are removed; see the article by Richard Sanders on implementing from SDL in this issue.

Separating the description of functionality from the realisation gives some additional benefits. First, that it provides a view where the system may be seen as a whole, independent of realisation technology. Second, that it can remain stable and survive technology changes. Finally, that it provides a solid foundation for technology trade-off and optimisation.

Figure 5 is an illustration of the differences between the two approaches. In the translation approach the viewpoints/aspects are well separated, while in the elaboration approach the separations are not so clear. Comparatively more effort is spent on functionality in the translation approach and less effort on realisation. Since functionality is expressed using formal languages, the quality of functionality can be assured separately from the quality of realisation. Provided that the realisation of functionality is automatically generated, the testing may concentrate on non-functional properties like performance and response times. An important point illustrated in the figure, is that the functionality description need not be changed if the realisation platform or programming language is changed. When using the translation approach, it is sufficient to change the deployment and realisation. The functionality description can be reused and thereby provide a better return of investment than possible when using the elaboration approach. One Norwegian company for instance, migrated from CHILL to C++ by changing their code generator and generate the new C++ code from the same SDL source as they previously used to generate CHILL.

Quality Assurance

Quality assurance techniques come in two main categories:

- Corrective techniques that focus primarily on *defect detection*, with subsequent correction. All the traditional verification and validation techniques including simulation, testing and inspection are in this category.

- Constructive techniques that focus primarily on *defect avoidance*, i.e. to avoid introducing errors in the first place. Tools for design synthesis and automatic program generation are obvious examples, but languages and methods that help to improve understanding and communication are clearly in this category too.

Corrective techniques may be seen as iterative, while constructive techniques are formative.

The quality of ICT systems can be split into two main aspects as suggested by Figure 2:

- *Quality of functionality*, that is related to the main purposes, i.e. the needs of the domain, and is concerned with the information handling and logical behaviour.

- *Quality of realisation*, that is related to the way the functionality is realised.

These aspects are quite independent and should as much as practically possible be separated (according to the golden rule on separation of concerns). With the elaboration approach, a clear separation of concerns is more difficult than in the translation approach. The ITU-T languages offer many advantages here, due to their conceptual abstractions, their well-defined semantics, and the possibility to define formal relationships between descriptions. An SDL model can for instance be extensively simulated and validated before implementation takes place, and implementation errors can be avoided by automated translation to realisation code. SDL models may be formally verified against properties specified in MSCs, and TTCN test cases can be derived from MSCs and thereby ensure that the realisation is conformant with the specification.

Method Maturity Level

Some companies using the translation approach have made the transition from *implementation oriented* development to *design oriented* development. They no longer treat the implementation code, in e.g. C++, as their primary documentation, but as secondary, derived documentation. Application designs expressed in languages such as SDL and MSC have taken over the role as primary documentation. On this level the application is understood and maintained in terms that are closer to user understanding. However, there is a snag: the amount of detail required to enable automatic code generation may clutter the description and reduce readability to a level where human errors are likely to increase. Some sort of layering is required to avoid this. Steve Randall describes one approach in his paper in this issue. Another approach is used in TIMe [4] where the functionality is split into an application part and an infrastructure part that are combined in a framework.

---

[10] *At least by 50 %.*

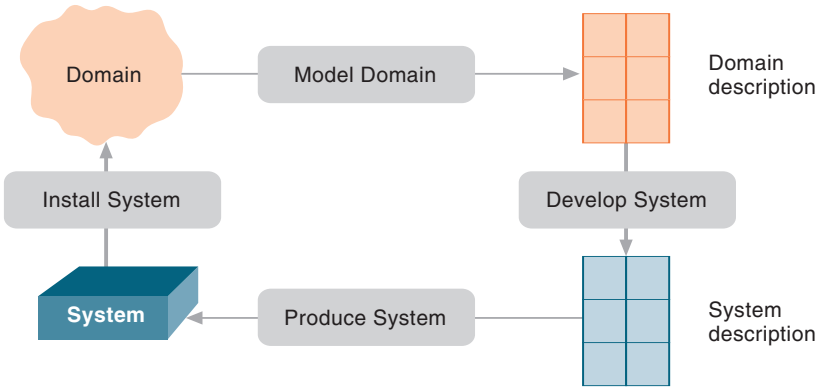


Figure 6 Macro methodology[12)]

Those that have moved to design oriented development, naturally seek further improvements. In a competitive market place, the ability to add new services (or modify existing services) with short time to market, while keeping the quality stable at a high level, is often sought as the next improvement. In practice, this means to focus more on the domain and the properties. Issues that emerge then are how to model properties separately and how to compose and map properties to designs. The ideal situation would be *property oriented development*, where designs are derived automatically from property descriptions. Although this vision cannot be realised today, some small steps in that direction have been made using existing languages and tools.

Using MSC to describe behaviour properties, for instance, offers some possibilities. First, because MSCs provide a readable and precise way to describe interaction behaviour and thus help to avoid errors, and secondly because MSCs can be used both constructively to (partially) synthesise application designs, and correctively to verify that application designs satisfy the properties specified in an MSC.

Methodology

The macro cycle/spiral in Figure 1 suggests a macro methodology as represented in Figure 6. Although it is very simplified, almost naïve, it illustrates the scope of methods and methodology: a set of *results* and *activities*. Note that all the activities in Figure 6 may go on at the same time, and normally will do so, by working on different individual results. Four main result areas with associated activities are identified in Figure 6, and a set of methods (a sub-methodology) can be defined for each of them[11)]. A central issue of methodology is to specify the results:

- What results should be achieved? At least *descriptions* covering the six different aspects we have identified in Figure 3 should be developed. The description icons in Figure 6 illustrate this.
- What should the results be like? The language/notation to use in each description must be specified, and *rules* must be provided for language use, both concerning the appearance (syntax) and the meaning (semantics).
- What relationships should hold between results? What are the rules for consistency between descriptions that can be used for verification and validation? What rules ensure a consistent documentation?

In principle, the activities are just means to achieve the results. In practice, they need to be broken down and explained both to provide guidelines and to facilitate project control. But the decomposition should focus primarily on steps in the result space, and not on activities *per se*. For each activity a method may specify:

- *Guidelines* for performing the activity;
- Preconditions for starting the activity;
- Intermediate results that should be produced;
- The final results (outputs) of the activity;
- The sub-activities that should be performed.

We may illustrate this by the system development activity. The goal here is to develop the six aspects of system descriptions such that the complete result satisfies all the rules and the relationships. This is achieved by decomposing the "Develop system" activity in Figure 6. A natural progression through intermediate results using the translation approach is illustrated in Figure 7. Here we have illustrated the state of the system description by colouring the fields of the icon that represent a description aspect that is ready. This shows a normal progression of description states. The figure may seem to describe a pure "waterfall" approach where each description aspect must be completed before starting on the next, but this is not necessarily the case. Each description aspect may consist of parts that may be developed separately, either in parallel or in sequence. By allowing activities to start with partial input and to run in parallel, the methodology can support alternative ways to order the activities over time, such as an incremental process or a spiral process.

---

11) *Note that methods and methodology are concerned with how to produce the best possible results, not actually producing them.*

12) *Note that the domain and the system are represented by separate descriptions that each may cover all six aspects.*

Sometimes, such figures as Figure 7 are mixed up with descriptions of *development processes*. A development process is a generalised project plan that specifies an ordering of milestones/ results and activity executions that every project should follow. Contrary to a methodology, a process specifies the activity executions in phases that actually produce results. A methodology should be more general and describe rules and guidelines in a way that can be applied by several processes. Therefore, a methodology should focus on activity types rather than activity instances (executions of activities). It should be possible to enact the activities in many ways and still produce results according to the guidelines and rules of the methodology.

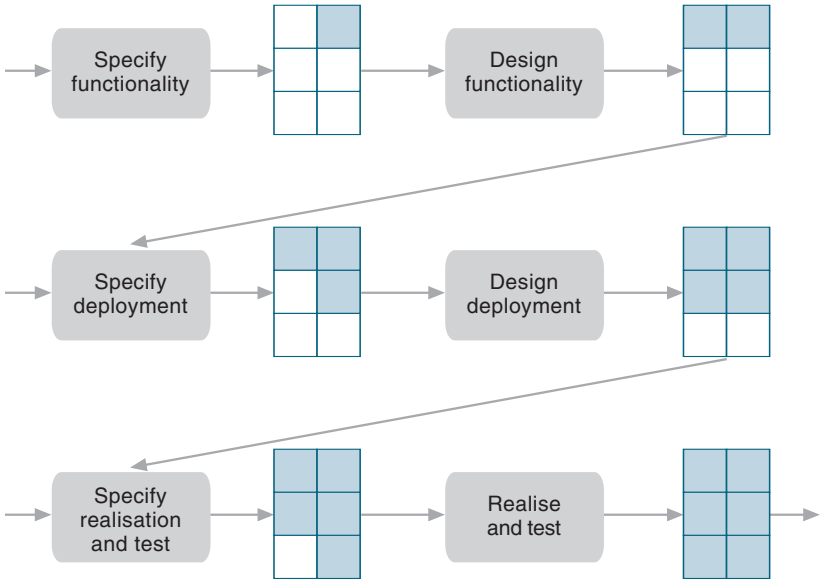


Figure 7 Sub-activities and intermediate descriptions of "Develop system"

Languages, or description techniques, determine the way that the reality can be described and understood. It is not so much the syntax of the languages that matters as the underlying concepts, or meaning. It is conceptual differences that really make methods and methodologies different. Many guidelines and rules focus on the meaning of descriptions rather than the syntax they are expressed with. Such guidelines and rules may often be applied to a range of languages using different syntax as long as the underlying concepts are similar.

Summing up, the core of systems engineering methodology is a set of descriptions, rules for well-formed descriptions, defined relationships between descriptions and guidelines for activities.

Some Methods

The first methods using the ITU-T language SDL were developed more than 20 years ago. Several industrial product developments at that time demonstrated that the quality of complex real time systems could be managed by means of the conceptual abstraction that the language provided, combined with a translation approach. At that time hardly any tools existed, so the products were developed using hand drawings in SDL and manual code generation. The author was responsible for one early method, called SOM that was used on an industrial scale by several companies during the late 1970s – early 1980s. Experiences from its first five years of use can be found in [5].

Based on positive experiences with SOM and similar methodologies, several Norwegian companies joined forces in the SISU project aiming to further develop and disseminate such methodologies. It was realised that the best way to ensure wide acceptance and adequate tools would be to use internationally standardised languages and work to make these as good as possible. SISU therefore funded parts of the Norwegian contributions towards the MSC-92 language and to make SDL-92 object oriented. This effort on standardisation was complemented with development of a methodology using SDL-92 and MSC-92 in combination with a notation called SOON[13)] for general object modelling. The method was documented in [2]. This methodology was further developed into TIMe, The Integrated Method [4], [6], based on SDL-96 and MSC-96 combined with UML.

During the same period, many companies developed methods using the ITU-T languages for their own internal use, see e.g. [7], and the tool vendors Telelogic and Verilog defined methods to help their customers use the languages and tools to their advantage [8], [9]. Method guidelines were also defined by the ITU-T [3], [10], [11], and ETSI developed method guidelines as explained in the paper by Steve Randall in this issue.

Lately the Rational Unified Process, RUP [12], [13], has been receiving considerable attention. It should be noted that RUP is not a methodology, but a web-enabled software engineering process using UML and Rational tools, such as Rational Rose. RUP is based on the elaboration approach.

13) *SOON was used for the same purposes as UML are used for when combined with SDL: general conceptual descriptions and deployment descriptions. The main reason for using SOON was that there were no UML available at that time.*

References

1 ITU-T. *Basic Reference Model of Open Distributed Processing – Part 1: Overview.* Geneva, 1997. (ITU-T Recommendation X.901.)

2 Bræk, R, Haugen, Ø. *Engineering Real Time Systems. An object-oriented methodology using SDL.* Prentice Hall, 1993. (ISBN 0-13-034448-6.)

3 ITU-T. *SDL methodology Guidelines*. Geneva, 1992. (ITU-T Recommendation Z.100, annex I.)

4 *TIMe: The Integrated Method; Electronic textbook*. 2000, November 15 [online] – URL: http://www.informatics.sintef.no/projects/time/

5 Bræk, R, Helle, O, Sandvik, F. SOM : An SDL Compatible Specification and Design Methodology. Experiences from 5 years of extensive use. *4th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS),* 1981. (IEE conference publication no. 198.)

6 Bræk, R et al. Quality by Construction Exemplified by TIMe – The Integrated Method. *Telektronikk*, 95 (1), 73–82, 1999.

7 Robnik, A, Dolenc, M, Alcin, M. Industrial Experience Using SDL in IskraTEL. In: *Proceeedings of Seventh SDL Forum. SDL'95.* Bræk, R, Sarma, A (eds.). Elsevier, 1995. (ISBN 0 444 82269 0.)

8 Ek, A. *The SOMT Method.* Malmö, Telelogic, 1995. (http://www.telelogic.com/download/papers/SOMT.pdf)

9 *ObjectGEODE – Method Guidelines*. Toulouse, Verilog, 1997.

10 Reed, R. Methodology for Real Time Systems. *Computer Networks and ISDN Systems,* 28, 1996.

11 ITU-T. *SDL+ Methodology: use of MSC and SDL [with ASN.1], Supplement 1 (05/97).* Geneva, 1997. (ITU-T Recommendation Z.100.)

12 Kruchten, P. *Rational Unified Process – An Introduction.* Addison-Wesley, 2000. (ISBN 0-201-70710-1.)

13 *Rational Unified Process*. 2000, November 15 [online] – URL: http://www.rational.com/products/rup/index.jsp

14 Shakeri, A. *A Methodology for Development of Mechatronic Systems.* Trondheim, Norwegian University of Science and Technology, Department of Telematics, 1998. (PhD thesis 1998:104.)

Descriptive SDL

STEVE RANDALL

SDL has been used informally by protocol standards writers for many years. Although informal SDL of this type is often very expressive, it lacks the formality necessary for ambiguities to be avoided and it cannot be simulated or validated. The “Methods for Testing and Specification” Technical Committee (TC-MTS) at the European Telecommunications Standards Institute, ETSI, have produced a set of guidelines which aim to help standards writers to produce SDL which is syntactically and semantically correct but which lacks none of the benefits of expression that have been achieved previously with informal use of the language.

Steve Randall (52) has worked in software and systems engineering and design for over 30 years. He started in Mitel Telecom Ltd in 1979 as UK Software Manager responsible for the development of software for PABX products. In 1991, Steve Randall and Alex Hardisty established PQM Consultants, now a successful partnership specialising in International standardisation for Corporate Networks. Steve has led projects at the European Telecommunications Standards Institute (ETSI) involving standards for the QSIG private network signalling system and handbooks on the use of formal languages in protocol standards. For 1 1/2 years he led the devleopment and validation of the set of guidelines satisfying the need for accessible and understandable protocol standards and which is now commonly referred to as “Descriptive SDL”.

steve.randall
@pqmconsultants.com

1 Introduction

For many years now, writers of telecommunications protocol standards have used the ITU-T Specification and Description Language, SDL, defined in Recommendation Z.100 [1], to provide a pictorial representation of the standardized protocols. Unfortunately, few, if any, of these standards employed SDL as anything other than a convenient notation for illustrating the flow of control within a unit implementing the protocol. In most cases, this approach resulted in a very readable specification – due mainly to its informality – but one which was syntactically and semantically flawed. Such SDL often contains ambiguities which make compatible implementations difficult to achieve. In recent years there have been some attempts to produce standards as formal SDL models (an example is the INAP CS2 specification to be found in EN 301 140-1 [5]) and although these are accurate and almost directly implementable, they tend to be complex and quite difficult for the human reader to assimilate and comprehend.

The benefits of specifying a protocol in formal SDL, even within an international standard, are great as automatic tools can be used to check the syntax and semantics of the specification, to simulate the interactions between signals and to validate the overall structure of the model. In addition, modern Computer Assisted Test Generation (CATG) techniques are such that a reasonably usable TTCN test suite can be generated directly from the SDL model. However, the benefits of readability that can be derived from an informal specification should not be underestimated.

In 1996, ETSI’s Technical Committee “Methods for Testing and Specification” (TC-MTS) decided that it would be worthwhile to investigate means of assisting standards rapporteurs to develop SDL models which combined the benefits of both formal and informal specification but which had few of the disadvantages. The result of TC-MTS’s efforts is a set of guidelines which define an approach to protocol specification which has subsequently been labelled, “Descriptive SDL” and is the subject of this article.

2 A Definition of Descriptive SDL

Before describing how it can be achieved, it is worth considering exactly what Descriptive SDL actually is and is not. First and foremost, it is not a new language or even a rigid subset of the existing SDL. It is, in fact, little more than SDL which:

- is expressed simply enough to make it easy for a human to read;
- follows SDL’s rules of syntax;
- has correct static semantics (for example, consistent use of data items);
- could easily be extended into a complete model for simulation and validation purposes.

One other characteristic of Descriptive SDL that is particularly important in the development of protocol standards is that the normative requirements (those aspects of the specification with which an implementation must conform) should clearly be identified.

The Descriptive SDL concept originally grew from the belief that there is a core of the language which is almost intuitive to any engineer who has ever had to draw or read a flow-chart. The majority of these engineers know instinctively that a rectangular symbol means “Do something” and that a diamond-shape means “Make a decision”. In addition, input symbols, output symbols, procedure calls and states can usually be interpreted correctly when viewed in the context of a complete diagram. On the other hand, most non-experts would find it difficult to understand the meaning of symbols such as the continuous input even with a copy of Z.100 [1] to hand. Having come to this realisation, it was

not difficult to decide that it would be useful to provide some guidance to protocol standards writers to enable them to make the most of this easily-understood core symbology.

When work began on EG 202 106 [4], the expectation was that following the guidelines would not necessarily result in SDL which would be complete enough to simulate and validate but experience has shown that in most cases it is possible to produce executable and, therefore, testable SDL which can genuinely be regarded as "Descriptive SDL". Although the guidelines were originally developed to help improve the SDL included in telecommunication protocol standards, they could equally well be used to improve the readability of models in "real" systems. With the powerful automatic tools that are available today for analysing SDL and compiling working models from it, there is a strong, and understandable, temptation to write SDL that only the tools can understand. All too often we forget that SDL stands for Specification and Description Language and that, as its name implies, it also has considerable value in communicating concepts and ideas to other human beings.

3 The Guidelines

When work began on the development of the "Guidelines for the use of SDL for Descriptive Purposes", the first step was to look at a number of existing standards and to identify common mistakes in the use of SDL, particularly where these caused ambiguities or made the specification difficult to comprehend. These mistakes included such things as the use of free text (without quotes) in task symbols, the use of ASN.1 data types directly as variables without the inclusion of a **dcl** statement and the use of message names which had not been defined as signals. The next step was to use common sense and established software engineering techniques to develop some reasonably simple methods for overcoming many of these problems in the use of SDL. These methods are reflected in the guidelines which have been collected together in the ETSI Guide, EG 202 106 [4].

It was clearly necessary for the guidelines to be separated into natural groups so that the right advice for any particular situation could be found easily. A number of different groupings were considered but the following seems to be the most appropriate:

- Naming Conventions;
- Presentation and Layout of Process Diagrams;
- Logical Structuring of Diagrams;
- The Use of Procedures and Operators;
- The Use of Decisions;
- Communication and Addressing (including System structures);
- The Specification and Use of Data.

It is not practical to describe every one of the guidelines here as this information can be obtained from EG 202 106 itself. However, it may be valuable to provide an example from each of the above categories.

4 Naming Conventions

When composing names and identifiers in any modern programming language, it is often tempting to use short combinations of letters and numbers. Indeed, for some of us, it is difficult to forget the lessons learned in writing assembler code and FORTRAN IV. Unfortunately, although these cryptic identifiers may have meaning to the writer, they are likely to be indecipherable to other readers. On the other hand, SDL allows names and identifiers that are almost unrestricted in length and even has rules that permit names to be wrapped across more than one line. Without care, these names can easily become unwieldy and meaningless.

Descriptive SDL suggests (rather than mandates) some useful constraints that software developers may wish to apply in order to ensure that names are meaningful and easy to read.

Like most of the guidelines, the following example, which relates to the number of characters in a name, is little more than common sense, but it was felt that it was worth saying anyway.

Example of a naming convention guideline:

> *Names of less than 6 characters may be too cryptic and names of more than 30 characters may be too difficult to read and assimilate.*

As an illustration of the use of this guideline, the SDL name for a variable that holds Input from the User could be abbreviated to something like "Uip" or, by taking the easy way out, it could be identified as "User_Input" or it could very accurately describe exactly what user input is by the use of "Collection_Of_Keystrokes_From_The_User". It is clear that the first one is too short to be really meaningful and that the last one is too long for easy use. That just leaves the term "User_Input" which clearly expresses the purpose of the variable while being short enough for easy manipulation within SDL symbols.

5 Presentation and Layout of Process Diagrams

Although the layout of an SDL diagram does not affect its correctness, it can have a major impact on its readability and interpretability. On a cramped page it is easy to miss a vital connection or even to see one that is not there.

Descriptive SDL includes some useful suggestions for laying out SDL process diagrams in a way that avoids these problems.

Example of a presentation and layout guideline:

> *The flow of SDL process diagrams should be from the top of the page towards the bottom.*

Again, this guideline is no more than simple common sense, but SDL that flows from one point on the page to all points of the compass is not uncommon but it is remarkably difficult to read.

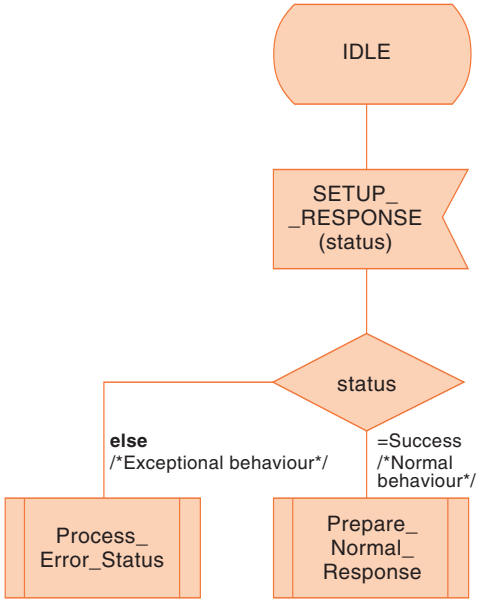


Figure 1 Segregation of normal and exceptional behaviour

6 Logical Structuring of Diagrams

It is surprising how easy it is to take a programming language based on logic and produce a specification that lacks any logical structure at all. Within any specification, it is necessary to be able to move quickly and easily between logically associated points without the need to review every page. Paragraphs and chapters are used intuitively to give a readable logical structure to prose but doing the same in a graphical language does not seem to come so naturally.

Within the Descriptive SDL guidelines there is useful advice on how to structure the SDL in a specification, particularly a standard, to make the most logical sense.

Example of a logical structuring guideline:

> *Process diagrams should segregate normal behaviour from exceptional behaviour.*

In its textual description, a protocol standard usually separates normal behaviour and exceptional behaviour under different section headings. Figure 1 shows a simple example of how making a similar separation within a process diagram can improve the readability of the SDL and simplify testing.

Although this guideline applies specifically to the use of SDL in protocol standards, the concept of splitting logically distinct behaviour flows is valid in any application.

7 The Use of Procedures and Operators

One of the greatest barriers to the easy comprehension of a complex protocol specification is the inclusion of too much detail in the process graphs. It is not always useful to see in minute detail how a particular function is achieved when all that is needed to understand the overall process is an indication that the function must take place. SDL procedures and operators provide an excellent mechanism for separating the "How" from the "What". Layering a model in this way allows the reader to choose to concentrate only on the functions performed and not on the methods for implementing them.

Example of a guideline on the use of procedures and operators:

> *Behaviour that could be considered a side-effect to its defined purposes, should not be specified in a procedure.*

This, once again, appears to be a statement of the obvious but it is not unusual for an SDL writer to assign a meaningful name to a procedure and then, as time goes by, to add more and more functionality to it without considering what was originally implied by its name. As an example, a procedure identified as "Get_User_Name" should do nothing more than get and return the name of the user. Readers would quite reasonably expect other operations, such as getting the user's email address or saving the user's telephone number in the appropriate record in a database, to be specified elsewhere. This guideline is particularly important in the context of layering a model.

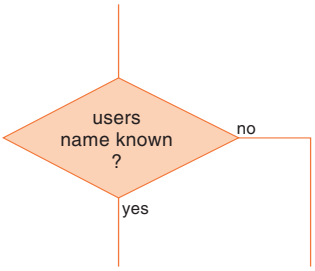


Figure 2 Informal text used in a decision

8 The Use of Decisions

Although decisions have to be used correctly in an SDL implementation model in order for the model to compile and build, it is very unusual to see a correctly formulated expression in a decision symbol within the SDL that is found in standards. However, the use of informal text usually makes the meaning of the decision extremely clear as can be seen in Figure 2.

The Descriptive SDL guidelines show how it is possible to write syntactically correct decisions without losing any clarity that comes from the informal approach that can be seen in the above example. They also show how decisions can be used more extensively than is currently normal within protocol standards to improve the overall structure of process graphs.

Example of a guideline on the use of decisions:

> *Identifiers used in decisions should clearly reflect to a reader the 'question' and 'answer' nature of the conditions being expressed.*

It is not unusual to find decisions in SDL which, although syntactically and semantically correct, make it very difficult to determine what question is being asked and what possible answers there are. After all, that is exactly what a decision symbol should convey – a question and a range of possible answers. By carefully choosing identifiers for variables, operators and procedures and by using synonyms to assign logical names to constants such as **true** and **false**, it is possible to construct decisions which do, in fact, clearly identify the question and the possible answers. Figure 3 shows how this principle can be applied to the example shown previously in Figure 2 to make it syntactically correct without losing any implicit meaning.

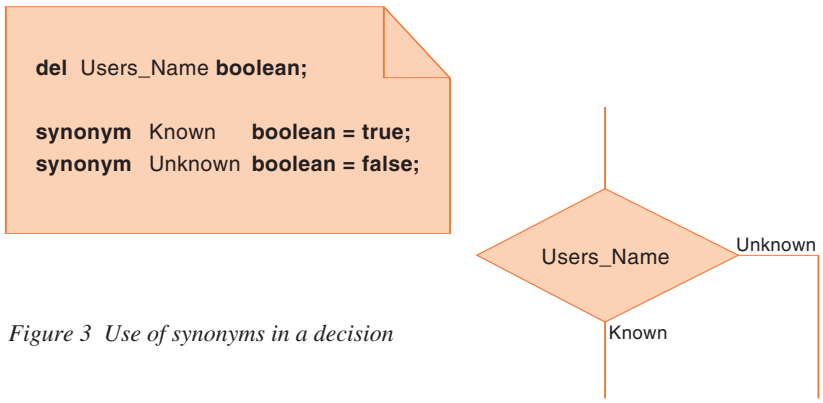


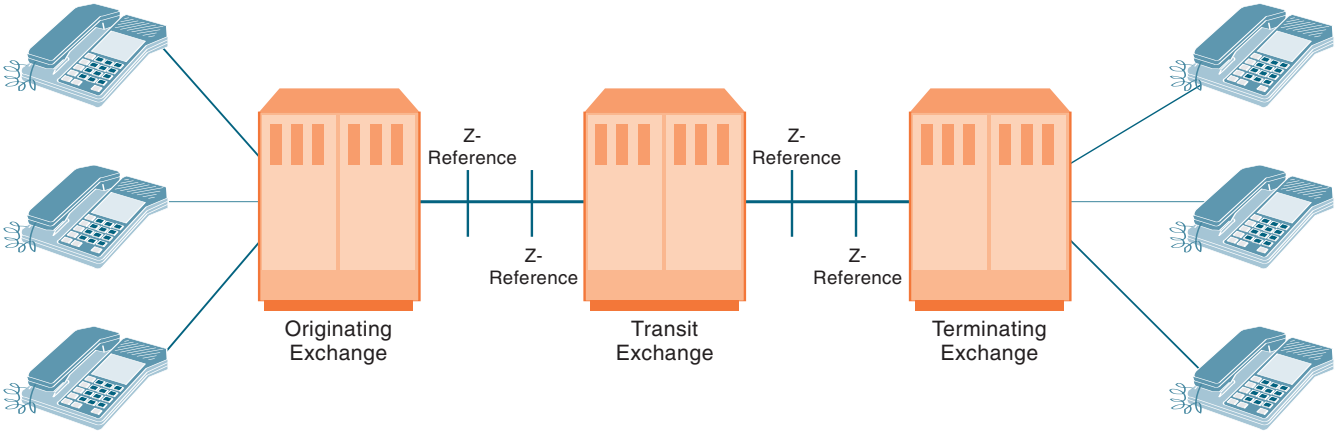
Figure 3 Use of synonyms in a decision

9 Communication and Addressing

Within a telecommunication protocol standard, perhaps the most important aspect of the SDL is the sending and receiving of signals by individual entities (processes) within the system. The contents, format and timing of these signals are the essence of the protocol. It is, therefore, important to ensure that the use of signals is clear and accurate within a specification. In addition, the relationships between blocks and processes within the system clearly identify where these protocol messages can be expected to appear.

A problem that is probably peculiar to the use of SDL in standards relates to the inclusion, or, rather, the omission, of system and block graphs to present the underlying architecture of a signalling protocol. That is not to say that the architecture is not described graphically but it is rare for SDL to be used for this purpose. Other diagrams, which are usually informal, are used instead. Descriptive SDL addresses this issue by showing how the same information can be specified using the formal, checkable graphs available in SDL.

Figure 4 Informal representation of a communication system



Example of a guideline on the use of formal structure:

> *SDL should be used to show the structure of a system as well as its behaviour.*

To illustrate this guideline, Figure 4 shows the type of informal diagram that is sometimes used in a standard to describe the overall system architecture. Although it is quite informative in terms of the generic system topography, it does not identify any aspect of the protocol between the individual exchanges. There is no formal meaning to the lines connecting the units so there is no possibility of testing their validity or of developing a protocol description based on this architecture.

Although not as attractive pictorially, the SDL system graph shown in Figure 5 conveys useful additional information and forms a solid formal base from which the process graphs can be developed. By placing the telephones (terminals) in the environment and by commenting Z_REF_A and Z_REF_B as "NORMATIVE", it makes it clear that only the relationships between the exchanges are important within the context of the standard. The use of **signallists** and communication path names adds information which is not available in the diagram in Figure 4.

Although there is generally little or no ambiguity regarding the destination of a signal at its source (process) or the source of the signal at its destination, this does not necessarily mean that these useful items of information will be obvious to the reader without referring to other diagrams and text. An SDL process graph is very much easier to read and understand if the destination of all output signals and the source of all received signals are made explicit. The **to** and **via** constructs, even when not strictly necessary, positively identify where a signal is to be sent or, at least, the route it will take. A comment added to each input symbol is an effective way of showing where the associated signal originated. Examples of these approaches are shown in Figure 6.

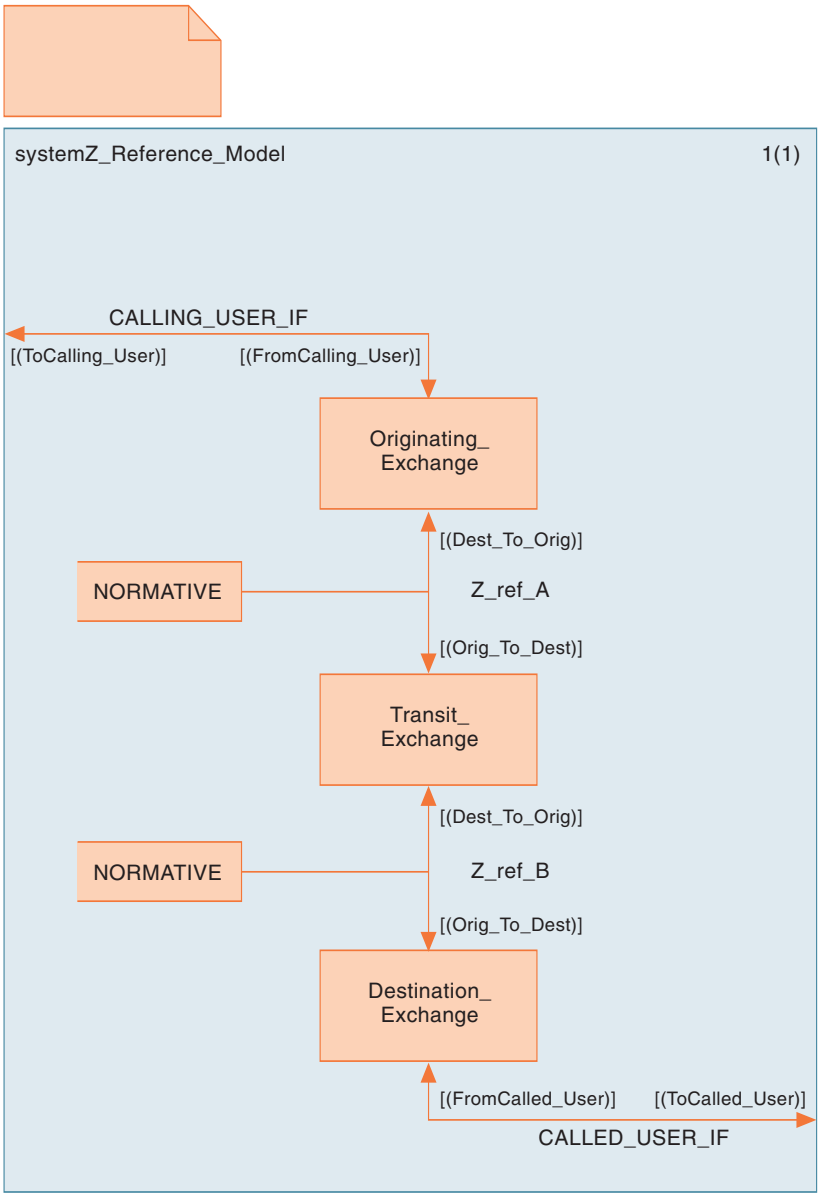


Figure 5 Formal SDL representation of a communication system

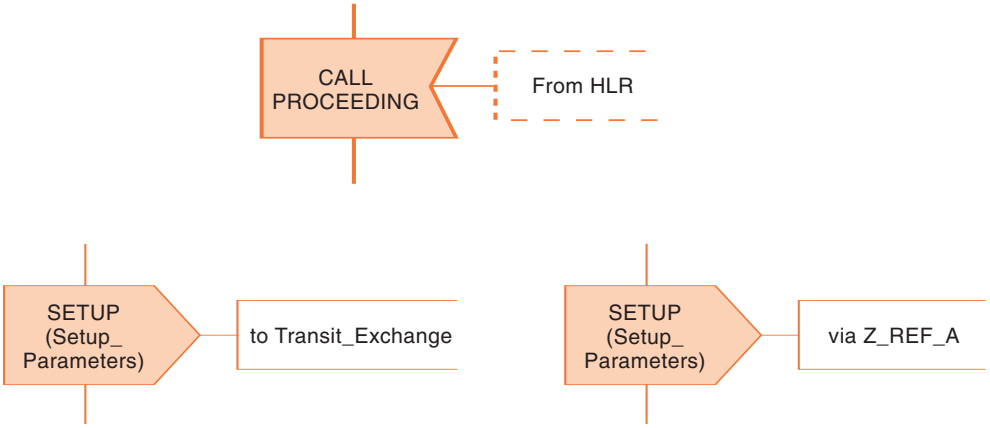


Figure 6 Identification of signal source and destination

Example of a guideline on communication and addressing:

There should be only one signal in each output symbol.

Although the syntax of SDL permits multiple signals to be included in a single output symbol, limiting the number of signals to one per symbol helps to improve the clarity of specification – one output, one signal.

10 The Specification and Use of Data

Descriptive SDL provides only a small amount of guidance on the specification and use of data in SDL models. Much of the advice that might be necessary here has already been covered in other sections of the guidelines. For example, the readability of data items is greatly improved by careful choice of identifier according to the naming conventions discussed earlier.

As the Abstract Syntax Notation, ASN.1 [3], is used extensively for defining the contents and structure of protocol message parameters in telecommunication standards, the guidelines concentrate primarily on the combined use of both ASN.1 and SDL. It is not possible to provide an exact mapping between the two languages (as examples, SDL has no **choice** concept and ASN.1 does not have the range-limiting possibilities offered by **syntype**) although ITU-T Recommendation Z.105 [2] makes a good attempt at this. Following the Descriptive SDL guidelines should help to avoid many of the problems in this area.

Generally, the guidelines related to specifying and using data in an SDL model are concerned with simplifications such as hiding the structures of complex data sorts by using **sequence**, **set** and **struct**.

Example of a guideline on the specification and use of data:

The top-level parameters of messages should be contained in a single structured type (e.g. ASN.1 SEQUENCE or SET) rather than specified as a list of simple types.

Here is a clear example of a guideline aimed at simplifying the specification by suggesting that if the parameters of a message contain more than one information element, they should be specified in a separate data structure. This avoids overloading the associate output and input symbols with lists of individual parameter items.

11 Conclusion

This article has only been able to give a flavour of Descriptive SDL and it is necessary to read EG 202 106 [5] for the complete picture. This can be obtained free of charge from the ETSI web site at *http://webapp.etsi.org/pda/*. Probably the most important thing to remember is that Descriptive SDL is a pragmatic way of using existing SDL rather than a new language in itself. It would be wrong to try to suggest that the guidelines would be applicable in their entirety outside the area of protocol standards development but they do contain some very valuable common sense which any software engineer designing in SDL would do well to be reminded of. The more advanced features of the language should not, of course, be avoided just because some people find them difficult to interpret but well structured SDL that is expressed clearly enough for a human to read and understand will always be much easier to review and maintain.

Acknowledgements

The development of the Descriptive SDL guidelines took place as a project led by the author and funded by the European Telecommunications Standards Institute (ETSI). Significant technical contributions were made by Milan Zoric and Anthony Wiles of the ETSI Secretariat, Rick Reed of TSE Limited, Juhana Britschgi of Nokia and André Wendling of Clemessy.

References

1 ITU. *Specification and Description Language (SDL).* Geneva, 1993. (ITU-T Recommendation Z.100.)

2 ITU. *SDL combined with ASN.1*. Geneva, 1994. (ITU-T Recommendation Z.105.)

3 ITU. *Information technology – Open Systems Interconnection – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Geneva, 1994. (ITU-T Recommendation X.680.)

4 ETSI. *Guidelines for the use of formal SDL as a descriptive tool*. Sophia Antipolis, 1999. (ETSI EG 202 106.)

5 ETSI. *Intelligent Network (IN); Intelligent Network Application Protocol (INAP); Capability Set 2 (CS2); Part 1: Protocol specification*. Sophia Antipolis, 1999. (ETSI EN 301 140-1.)

Combined Use of SDL, ASN.1, MSC and TTCN

ANTHONY WILES AND MILAN ZORIC

The purpose of most telecommunication standards is to promote interoperable products. Many techniques are used in order to make the standards more precise. In some cases controlled use of the natural language can make specifications less ambiguous. However, for complex systems, this is often not sufficient, and therefore, specifications using formal languages with their simulation and error detection capabilities are employed. Interoperability is further improved if an organised process of test specification development accompanies the base standard development. This paper describes the way combined use of SDL (Specification and Description Language) [1], MSCs (Message Sequence Charts) [6] and ASN.1 (Abstract Syntax Notation One) [2, 3, 4, 5, 8, 9] along with test development in TTCN (Tree and Tabular Notation) [7] affect the standards and the product implementation. While there are many elements that are specific for standardisation domain, some experiences should be relevant in general use of these languages.

Anthony Wiles (51) received his MSc in Physics and Computer Science from Uppsala University, Sweden in 1984. He has been involved in the development of TTCN since the language first originated late 1980s and was the ISO editor of the original TTCN language specification. He has published several papers and tutorials on the practical and theoretical aspects of protocol specification, automatic test generation, testing and the use of TTCN. Anthony Wiles is currently manager of the Protocol and Testing Competence Centre at ETSI, which advises and provides support to many of ETSI's technical bodies.

Anthony.Wiles@etsi.fr

Milan Zoric (51) received his Dipl.Ing. diploma in Electrical Engineering and his MSc in Telecommunications and Information Technology from the Univ. of Zagreb, Croatia in 1972 and 1982 respectively. His work has been related to SDL since the early 1980s. He participated in the validation of SDL formal semantics in 1988, published numerous papers related to SDL and led a team developing an SDL tool. Since 1995 he has participated in a number of ETSI projects related to the use of SDL in standardisation, and worked as consultant for other international companies. In 1998 Milan Zoric joined the Protocol and Testing Competence Centre at ETSI.

Milan.Zoric@etsi.fr

1 Introduction

In ETSI the PTCC (PEX and Testing Competence Centre) exists with a mission to assist ETSI Technical Bodies in the application of state-of-the-art specification and testing techniques in a pragmatic and flexible manner to help ensure the technical quality of ETSI deliverables in an efficient and economic manner.

The presentation in this paper is based on PEX experience gained during the development of the Radio Link Control (RLC) protocol standard of HIPERLAN/2 system developed by the ETSI European Project Broadband Radio Access Networks, EP BRAN

2 The HIPERLAN/2 System

The increasing demand for 'anywhere, anytime' communications and the merging of voice, video and data communications create a demand for broadband wireless networks. ETSI has created the BRAN project to develop standards and specifications for broadband radio access networks that cover a wide range of applications and are intended for different frequency bands.

HIPERLAN/2 is one of the systems being standardised by EP BRAN. HIPERLAN/2 is capable of supporting multi-media applications by providing mechanisms to handle QoS. The typical operating environment is indoors. Restricted user mobility is supported within the local service area; wide area mobility (e.g. roaming) may be supported by standards outside the scope of the BRAN project. HIPERLAN/2 systems are intended for operation in the 5 GHz band.

HIPERLAN/2 provides transport services with data rates up to 54 Mbit/s. Integration into different network types is achieved by convergence layers.

HIPERLAN/2 has the potential to become the technology in the 5 GHz range. It is technologically superior to other wireless LAN standards because of the highly efficient use of the available resources, possible support of QoS, and efficient means for the support of security functions (authentication and encryption) well integrated into association procedure.

Figure 1 shows the HIPERLAN/2 protocol stack and the functions in it. The left part contains the Radio Link Control Sub layer (RLC), which delivers a transport service to the DLC Connection Control (DCC), the Radio Resource Control (RRC) and the Association Control Function (ACF). Note that only the RLC entity is standardised which defines implicitly the behaviour of the DCC, ACF and RRC. The RLC protocol is operating in a radio environment with significant probability of messages not received by the peer entity. Since there is no underlying layer that would assure reliable communication, the RLC has to provide retransmission mechanisms itself.

3 Supporting the Protocol Development Process

3.1 Specification

The main goal behind the use of SDL in standards has been to improve the functional specification in the standard by making it more precise. However, the use varies from highly informal to very formal models that are useful for simulation and validation aimed at error detection. The former appears to be readable, but is not precise

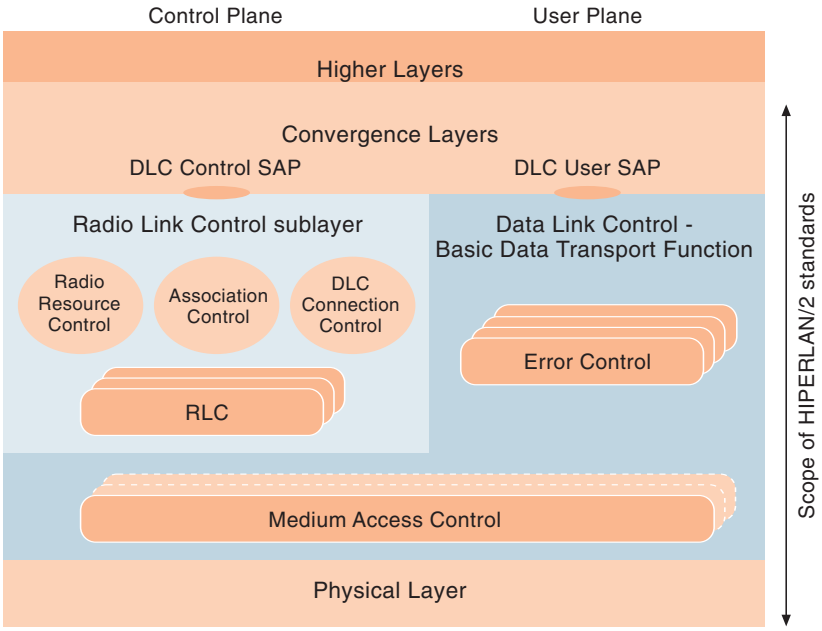


Figure 1 HIPERLAN/2 protocol stack and functions

enough and often hides ambiguities; see the paper by Steve Randall on *Use of SDL for descriptive purposes* in this issue. The latter may be precise and correct, but looks in many cases like program code, which hinders the main purpose of the SDL specification: to communicate the desired functionality to the readers.

Fortunately, it was agreed at an early stage in the RLC development, when only some rough ideas about the protocol existed, to use formal SDL in RLC protocol development supported by PTCC expertise. It should also be noted that the protocol development group had BRAN specific expertise, but had modest knowledge about SDL, MSCs and ASN.1.

The main goal was to arrive at a correct specification that would be readable, especially for readers that are experts in writing test specifications and in implementing the standard. In accordance with this goal, the protocol development group initially agreed to use the following approach:

a) Develop SDL models that specify only the parts of the system which are most important for achieving interoperability. At the same time it was planned to cover these parts by test specifications.

b) Use MSCs to specify the normal behaviour.

c) Use ASN.1 to specify at an abstract level the data used in messages. That was most important for Protocol Data Units, but there was no reason not to use ASN.1 for other data specification.

d) Use Descriptive SDL (see the paper by Steve Randall on *Use of SDL for descriptive purposes* in this issue) for modelling of full (normal and exceptional) behaviour where:

- normal behaviour is clearly distinguished from exceptional behaviour;
- naming rules are identified and followed;
- operators are used for any data manipulation.

During the protocol development many aspects were refined. The two most notable refinements were:

e) High Level MSCs were introduced;

f) The MSCs were refined to include description of options, alternatives and MSC references.

When the specifications were well advanced, the decision was taken not to use standardised encoding rules but to define the encoding of RLC PDUs in a tabular format.

3.2 Testing

ETSI members have long recognised the important role that standardised test specifications play in the development of products based on ETSI standards. Comprehensive conformance test specifications exist for technologies such as GSM, DECT, INAP, N-ISDN, B-ISDN, Q-Sig, TETRA, VB-5 and HIPERLAN/1. Plans for year 2000 include tests for 3GPP (terminals), HIPERLAN/2, HiperAccess and possibly some TIPHON protocols.

Even in a climate where time-to-market is paramount, the importance of testing has not been marginalized. In fact, the reverse seems to have occurred. Vendors know that a tested product is a quality product and are prepared to put time, effort and valuable expertise into this activity.

Nonetheless, it makes good economic sense to rationalise the testing process as much as possible. While certain key areas quite rightly require regulatory testing, the policy today is to keep this to a minimum. As a consequence there has been a subtle but fundamental change in the development and application of ETSI test suites over the last few years. Manufacturers are increasingly using these test specifications for their internal product testing, and the tests themselves are concentrating on a specific purpose: conformance testing for interoperability.

For HIPERLAN/2, two sets of tests were planned. One set, written manually in TTCN,

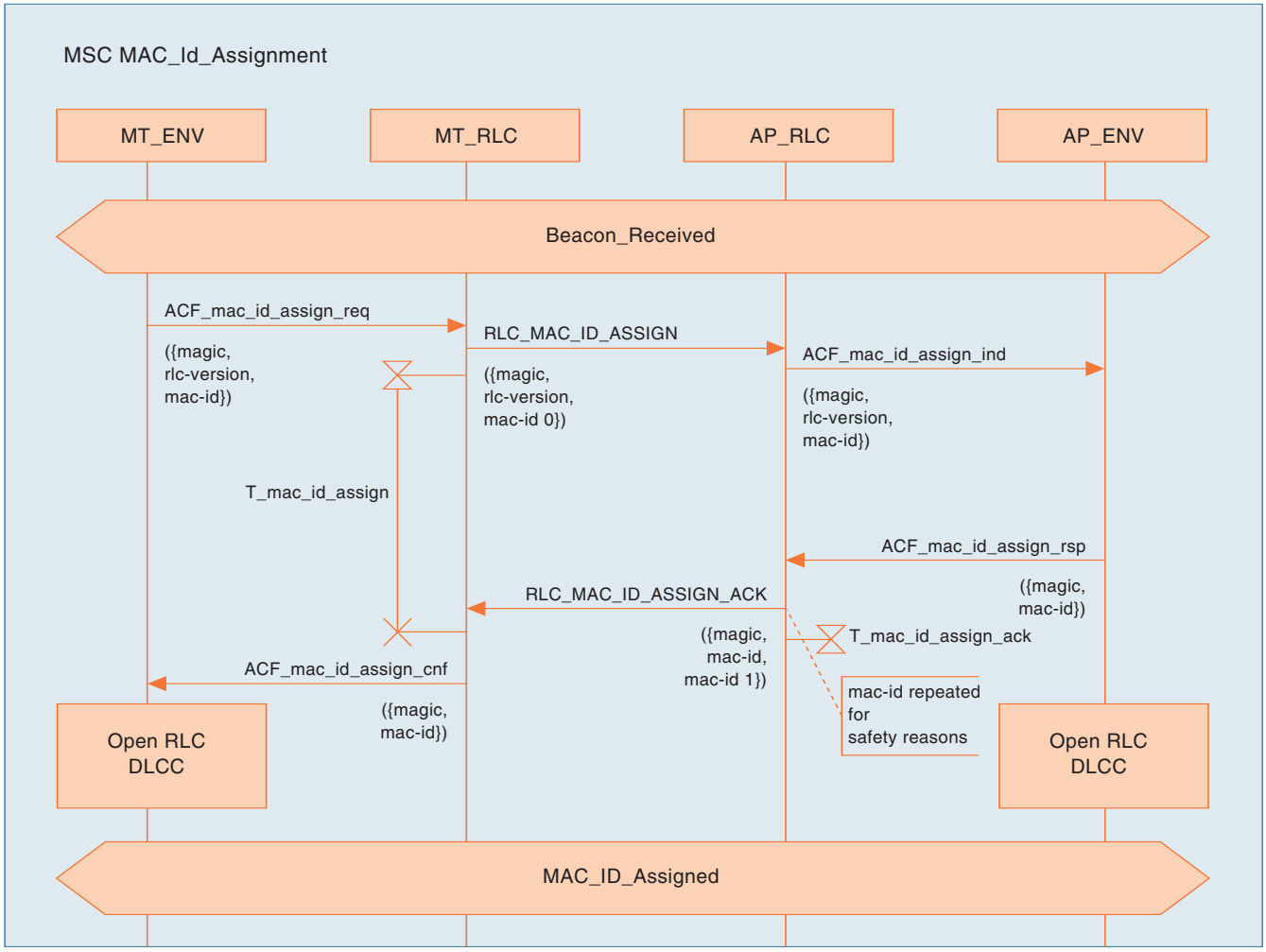


Figure 2 The MSC diagram of MAC-ID-ASSIGN procedure

was intended to represent the normative part of the testing standard. The other set was to be derived semi-automatically from the SDL models using CATG (Computer Aided Test Generation). The purpose of using CATG tools was to increase the understanding of the benefits that might be achieved in the context of standardisation.

4 Practical Experiences

4.1 Specification

4.1.1 MSC and HMSC Diagrams

In the early stages of the RLC protocol development, MSC diagrams were used to define the message interchange for normal protocol behaviour. MSC diagrams are very intuitive and require almost no training. They are much easier to develop and change than the SDL diagrams, which means they are well suited for initial phases of the work.

Rather than showing only the messages, the diagrams included additional information, such as starting and ending conditions and timer setting and resetting actions. Initially the diagrams had only some informal information about parameters of messages, but the final version contains the specification of parameters using ASN.1 syntax. One example MSC diagram is shown in Figure 2.

During the work it became obvious that the MSC diagrams for some procedures were becoming very complex, difficult to manage and difficult to read. At that stage, High-level MSCs (HMSC) were introduced. The first impression was that HMSCs just helped to improve the graphical presentation, but it soon became apparent that HMSCs bring many additional benefits that change both the way of working and the final result a great deal. As an illustration, the association procedure of the RLC protocol is given in Figure 3.

The following are the most important benefits of using HMSCs:

- *Segregation of description levels*
  While the HMSC specifies what needs to be done, the MSCs show how it should be done.

- *Improved organisation of specification work*
  The work first concentrates on what is required, and then on the details completed in a how part.

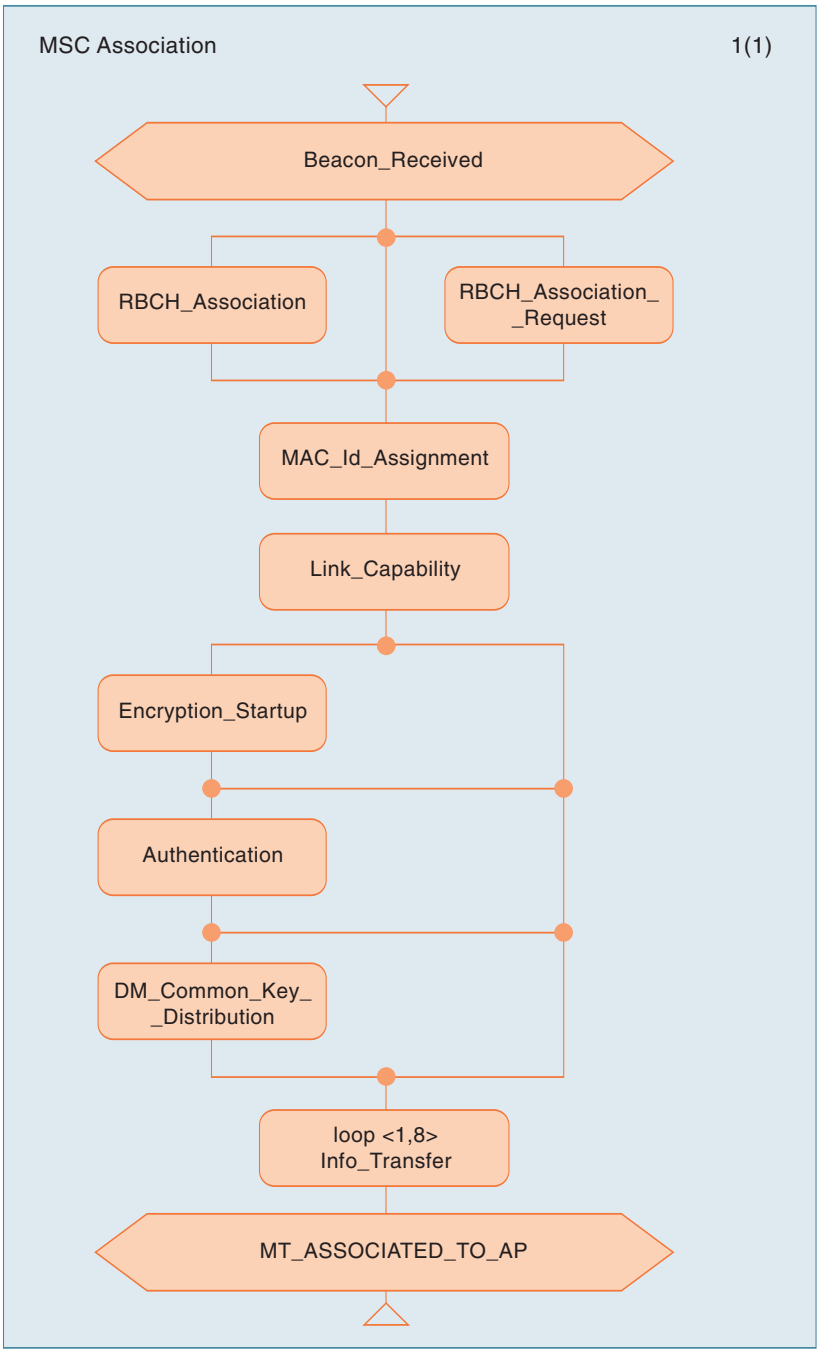


Figure 3 HMSC of the RLC association procedure

- *Additional power of expressing requirements*
  HMSC can conveniently express options and alternatives in using parts of a complex procedure. In the HMSC a line that bypasses the MSC reference says that a procedure behind the bypassed MSC reference does not always have to be used.

- *Improved documentation*
  The documentation is better organised, segmented in manageable units and much more readable and understandable.

- *Reduced possibility of ambiguities*
  This kind of complex procedure description is less open to ambiguities in contrast to plain text.

- *Better input for SDL development*
  The MSC diagrams, when placed in the wider context of a HMSC diagram, provide a significantly better input to SDL model development.

No major problems were encountered during the work on MSCs and HMSCs. However, some minor problems were identified where improved understanding and guidance were needed. For example, it is not obvious how the MSC-conditions should be understood so that appropriate naming could be selected. Once defined, some care has to be given to updating the information on conditions. Since things change as initial ideas mature towards the final design decision, the updates are needed along the way.

A limit on how far the development of MSC diagrams should proceed must be set. In this case, it was decided that the MSCs were to cover normal behaviour only. In other applications, important exception cases could also be added. For detailed definition of exception cases, SDL is better suited than MSCs.

The testing experts' reaction to the use of MSC and HMSC diagrams (without any training or prior explanation) was very positive, in fact they asked whether additional HMSC diagrams could be developed for parts of protocol behaviour not initially covered by HMSC diagrams. We witnessed that HMSC and MSC diagrams were used very much during the test development. The MSC and HMSC diagrams were subject to numerous reviews in member companies. The stable releases are always made available to engineers implementing the system in member companies. For this purpose, pdf format of those diagrams was very frequently requested.

4.1.2 SDL

The scope of the SDL modelling was restricted to parts that are essential for interoperability. This means that the SDL is describing how the implementation of a standard should behave at the normative interface and not how it should be implemented and built in order to achieve this. We may express this even more strongly by saying that the SDL model in a standard should never be such that it can be directly implemented as a product. It can serve as a reference, parts of it could be used directly in implementations (for example ASN.1 specification of PDUs), but implementation dependent details should never be introduced in the standard.

The fact that the model can simulate the behaviour at the normative interface brings many benefits but would not in itself earn acceptability by the readers. Readers are engineers trying to understand the SDL specification with a view to

implementation. For them, human readability is the ultimate criterion that determines whether the model is acceptable or not. Once they have an understanding of the model, they can start examining what it does or does not cover and judge whether the model is complete and correct. The possibility of simulation and validation is good for detection and removal of undesired properties (deadlock, lost signals, etc.). However, human inspection is very important in order to determine whether the valid model fulfils its purpose.

Our practical experience is that the combined use of HMSCs, MSCs and descriptive SDL makes the SDL model readable. Once the readers understand the requirements of the normal behaviour expressed in MSC and HMSC diagrams, they start reading the SDL model by examining where this behaviour can be found in the SDL model. Clear separation of normal behaviour from exceptional behaviour in the SDL model helps them to easily find what they are looking for. Identification of the normal behaviour description in the SDL model, and its consistency with the MSC description, builds the initial confidence that the model is sound. Having a firm understanding of the normal behaviour, the reader can proceed to examine the exceptions.

It has to be noted that in this scenario many readability issues are determined already when working on MSC diagrams. For example, the naming chosen in MSC diagrams needs to be followed in SDL diagrams as well.

Our experience in the SDL review process was that very valuable comments were received from engineers who claimed having no expertise in SDL, but who were very familiar with the protocol.

As an illustration, a page from the SDL specification is given in Figure 4. This SDL diagram could be examined against the MSC diagram given in Figure 2.

Data are in the SDL diagrams strictly handled by operators defined for this purpose. Because most operators are very simple we used operators defined in textual SDL syntax. This does not change the substance of the specification but reduces the number of operator definition pages by more than 50 %, compared to graphical operator diagrams.

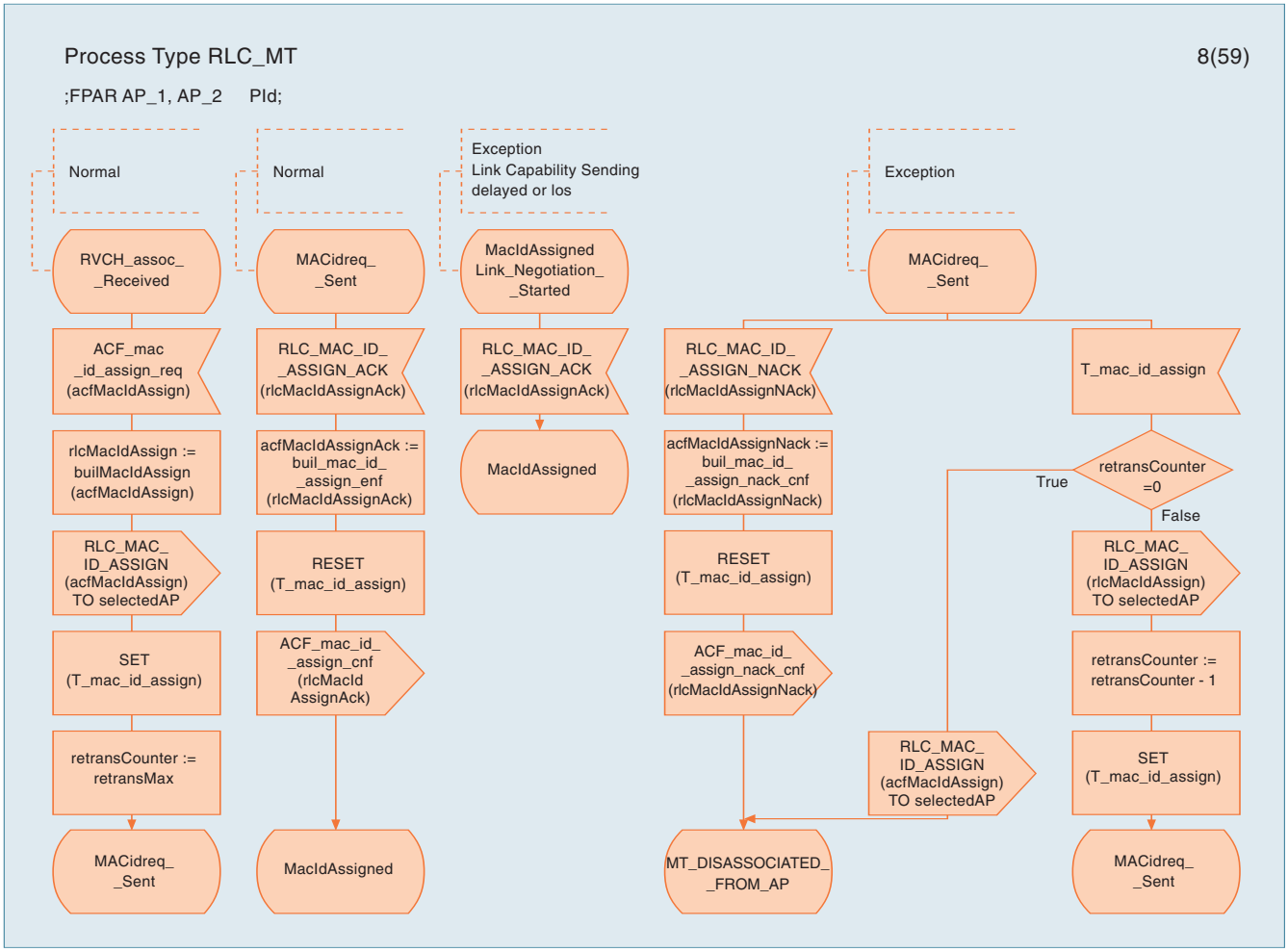


Figure 4 Part of the SDL specification where the MAC-ID-ASSIGN procedure is "implemented"

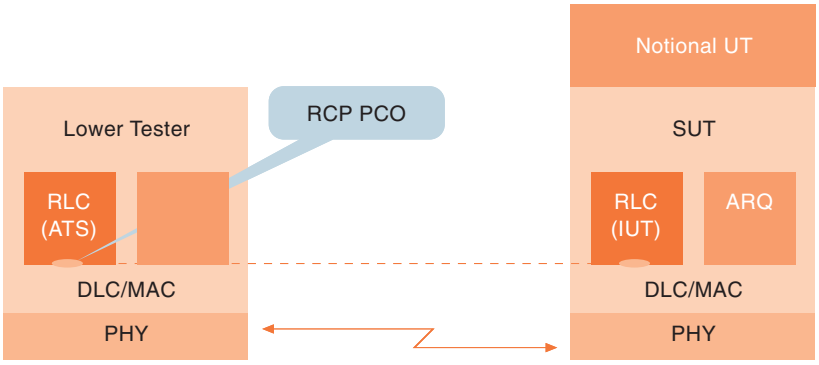


Figure 5 Test architecture for RLC

The main benefit of using operators is that operator names indicate what needs to be done, whereas the operator definition hides away the details of how it is done, and this helps to simplify the diagrams where operators are used, i.e. the process graphs. The added benefit is the improved type checking. In SDL2000 methods could be used in the same way instead of operators.

4.1.3 ASN.1

The use of ASN.1 allowed the group to start defining the protocol data units (PDU) independently of the transfer syntax. Individual data were specified along with the respective constraints in value range and/or length. It was possible to specify optional fields and to define optional field groups. ASN.1 data specification is easy to integrate into the SDL model and the TTCN test specification.

Since the decision was to use the manual encoding tables, the abstract specification was used to develop the concrete encoding tables. It would be possible to demonstrate that the existence of the abstract data specification leads to better structured transfer syntax tables than could have been achieved without abstract specification. However, the benefits of abstract syntax are likely to be forgotten, because the transfer tables are those that need to be implemented.

4.2 Testing

The HIPERLAN/2 testing strategy is as follows. Two sets of conformance test specifications are currently under development: those for testing the radio aspects, which are necessary for regulatory purposes, and those for testing the protocol aspects. In the latter case the tests concentrate on specific parts that are critical to interworking, such as the RLC protocol, error-handling mechanisms (ARQ) and, of course, the various convergence layers.

The HIPERLAN/2 protocol conformance test suites are being developed at ETSI. These test suites will be used by the members of EP BRAN to self-test their HIPERLAN/2 products. In addition, the HIPERLAN/2 Global Forum will use the conformance specifications as a basis on which to perform further interoperability testing.

One of the positive side effects of producing test specifications is that their development, if done in a timely manner, can give valuable feedback to the base standards. A classic example of this is DECT, where the test suite development team worked very closely with the EP DECT specification group as well as with the developers of real DECT test systems. This activity validated both the test suites and the relevant DECT standards.

4.2.1 HIPERLAN/2 Test Architecture

A single-party testing concept is used, which consists of the following abstract testing functions:

Lower Tester
A Lower Tester (LT) is located in the remote BRAN H/2 test system. It controls and observes the behaviour of the Implementation Under Test (IUT).

RLC ATS
A RLC Abstract Test Suite (ATS) is located in the remote BRAN H/2 test system.

RCP PCO
The Point of Control and Observation (PCO) for RLC testing is located at a Service Access Point (SAP) between the RLC layer and the MAC layer. All test events at the PCO are specified in terms of Abstract Service Primitives containing complete PDU. To avoid the complexity of data fragmentation and recombination testing, the SAP is defined below these functions.

Notional UT
No explicit upper tester (UT) exists in the system under test. Nevertheless, some specific actions to cover implicit send events and to obtain feedback information are necessary for the need of the test procedures. A black box covering these requirements is used in the SUT as a notional UT according to ISO 9646. This notional UT is part of the test system.

4.2.2 Producing Test Specifications

A core set of test cases, concentrating on features critical to interoperability, were developed manually. The HMSCs and MSCs provided valuable and precise input to the development of these test specifications. The HMSCs gave a good framework for defining groups of tests, while the MSCs were useful in providing the main flows for individual test purposes, often clarifying difficult parts of the base standard. For complex parts of the behaviour, also the

SDL specification was consulted during manual test specification. In all, approximately 160 test purposes with corresponding test cases written in TTCN were produced (80 for mobile terminal (MT) side, 80 for the access point (AP) side).

In parallel, the SDL model was used to generate some additional test cases. This required some extra detail to be added to the model, but generally not too much. For the MT side, the tool generated over 260 individual test cases and a similar number for the AP side. On analysis it was seen that in many cases several simple generated test cases mapped to one more complex test case manually produced. When this was taken into account, it was found that 44 of the generated test cases mapped exactly to the manual test cases, and 80 new test cases, not specified manually, were created. Again, similar figures were found for the AP side. At this stage it is too early to say if the generated test cases are of the same quality as the manually produced test cases, but the results look promising. The generated test cases will be included as an informative annex in the testing standard when it is finally published.

This work also provided a useful side effect by validating the SDL model at the same time. Useful feedback was provided to the SDL team from the testing activity.

5 Conclusions

On the whole, experience showed that the use of these techniques led to better quality in the relevant standards, for both the base standards and the testing standards. Early feedback from implementers indicates this to be true in the product development process, as well.

While development of the models took a significant amount of commitment and effort, their production did not delay delivery of the standard. Experience shows that it is important to maintain a systematic way of working, where a good co-operation between the protocol experts and PEX is vital. While each ETSI project has its own unique problems and needs, it is expected that the pragmatic use of methodology, similar to that applied in HIPERLAN/2, will continue to play an increasingly important role in the timely development of high-quality standards and test specifications. This in turn should play an important role in ensuring that HIPERLAN/2 products will interoperate.

6 Acknowledgements

The authors would like to acknowledge the good and devoted work of the EP BRAN groups described here, and thank the Chairman of EP BRAN Mr Jamshid Khun-Jush for supporting the publication of this paper.

References

1 ITU-T. *Languages for Telecommunication Applications – Specification and Description Language (SDL-2000)*. Geneva, ITU, 1999. (ITU-T Recommendation Z.100 (11/99)).

2 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.* Geneva, 1998. (ITU-T Recommendation X.680 (1997). ISO/IEC 8824-1:1998.)

3 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.* Geneva, 1998. (ITU-T Recommendation X.681 (1997). ISO/IEC 8824-2:1998.)

4 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.* Geneva, 1998. (ITU-T Recommendation X.682 (1997). ISO/IEC 8824-3:1998.)

5 ITU-T. *Information technology – Abstract Syntax Notation One (ASN.1): Parameterisation of ASN.1 specifications.* Geneva, 1998. (ITU-T Recommendation X.683 (1997). ISO/IEC 8824-4:1998.)

6 ITU-T. *Languages for Telecommunication Applications – Message Sequence Chart (MSC).* Geneva, ITU, 1999. (ITU-T Recommendation Z.120 (11/99).)

7 *Information technology – Open systems interconnection – Conformance testing methodology and framework – Part 3 : The tree and tabular combined notation.* Geneva, 1998. (ISO/IEC 9646-3:1998 (ITU-T Rec. X.291, ETSI TR 101 666).)

8 Larmouth, J. *ASN.1 Complete*. Morgan Kaufmann, 1999. ISBN 0-12233-435-3. (http://www.oss.com/asn1/larmouth.html)

9 Dubuisson, O. *ASN.1 – Communication between heterogeneous systems*. Morgan Kaufmann, 2000. ISBN 0-12-6333361-0. (http://asn1.elibel.tm.fr)

Implementing from SDL

RICHARD SANDERS

Richard Sanders (41) is Research Scientist at SINTEF Telecom and Informatics, and lecturer at the Norwegian University of Science and Technology (NTNU), from which he graduated in 1984. He has previously worked as a consultant with CAP Gemini and as a software designer and manager at Stento, developing intercom systems and participating in the SISU project. He joined SINTEF in 1995, and helps introduce new development methods in companies, holds courses and carries out research in the field of system development methodologies. He is currently undertaking a PhD at the Institute of Telematics at NTNU.

richard.sanders @informatics.sintef.no

SDL is rated as a fine modelling language by researchers and industry alike. But can it be used to derive efficient implementations? This article sheds light on some of the issues involved.

1 Introduction

One of the positioning features of SDL, and possibly the main reason for its adoption by an increasing number of users outside the traditional telecom domain, is the ease in which SDL models can be transformed to a wide range of popular programming languages. From the early days where such transformations were done manually, to the present situation with good tool support for automated transformations, the transformation from a formal model in SDL to an actual system running on a computer has become relatively short and well paved.

Newcomers to the SDL world quickly adapt to the state-of-the-art of the SDL CASE tools that are available. Given good tools users can experience the true benefits of design-oriented development. These benefits include an executable model of a system, enabling simulation at different levels of granularity, and intuitive traces of the system's actual or intended behaviour expressed in Message Sequence Charts. Advanced model checking facilities are available, and a fairly straightforward means of generating implementations both for host and target. All this can be achieved while maintaining much of the tool functionality even in the target implementation environment.

In the earlier days of SDL, hand coding from specifications (e.g. protocol standards) and system designs was commonplace. Engineers found practical solutions to the manual transformation from SDL to running systems, and reported good results, such as fewer errors and easier maintenance. Despite the manual work involved, the implementation process was efficient, and typically counted for less than 30 % of the total development effort, testing included. Many products where developed this way, and resulted in an increase in the number of SDL users. Since the end of the 1980s this manual coding scheme has increasingly been replaced with automatic generation of code by tools.

Design orientation

Design orientation is characterised by a development process where systems are understood and maintained mainly in terms of abstract design descriptions, e.g. in SDL and MSC. This is considered to be a more mature process than *Implementation orientation*, where the actual programming code is the only true description of the system, and any design models that exist act as illustrations only. The step up from Design orientation is *Property orientation*, where the focus is on desired properties.

SDL encourages the translational approach[1)] to design, as opposed to the elaborational approach favoured by methods like Booch and OMT, and which to a certain degree is still true of many UML methods. The benefits of a translational approach with automatic code generation are obvious and imminent:

- The SDL system and the implementation are by nature consistent, which was often not the case with manual coding from informal SDL;
- Systems can to a large extent be maintained solely on the SDL level, keeping complicated behaviour expressed in a form more easily understood by humans (i.e. MSC and SDL), in contrast to machine-friendly programming languages (e.g. C++, Java, Chill, etc.).

In this article we shall dwell slightly on these translation issues, that is, how the SDL primitives can be turned into software running on hardware. Newcomers to SDL and automated program generation often consider such issues uninteresting, trusting tool support to cater for the technicalities involved. And while the latter attitude generally speaking is viable, some words about the transformation from the ideal

[1)] *The translational approach involves transformation between different models at different levels of detail, e.g. from an implementation-independent model in SDL to code in C++. This in contrast to the elaborational approach, which is distinguished by re-working and adding details into models as you progress down the development life cycle, until the model is to some extent complete.*

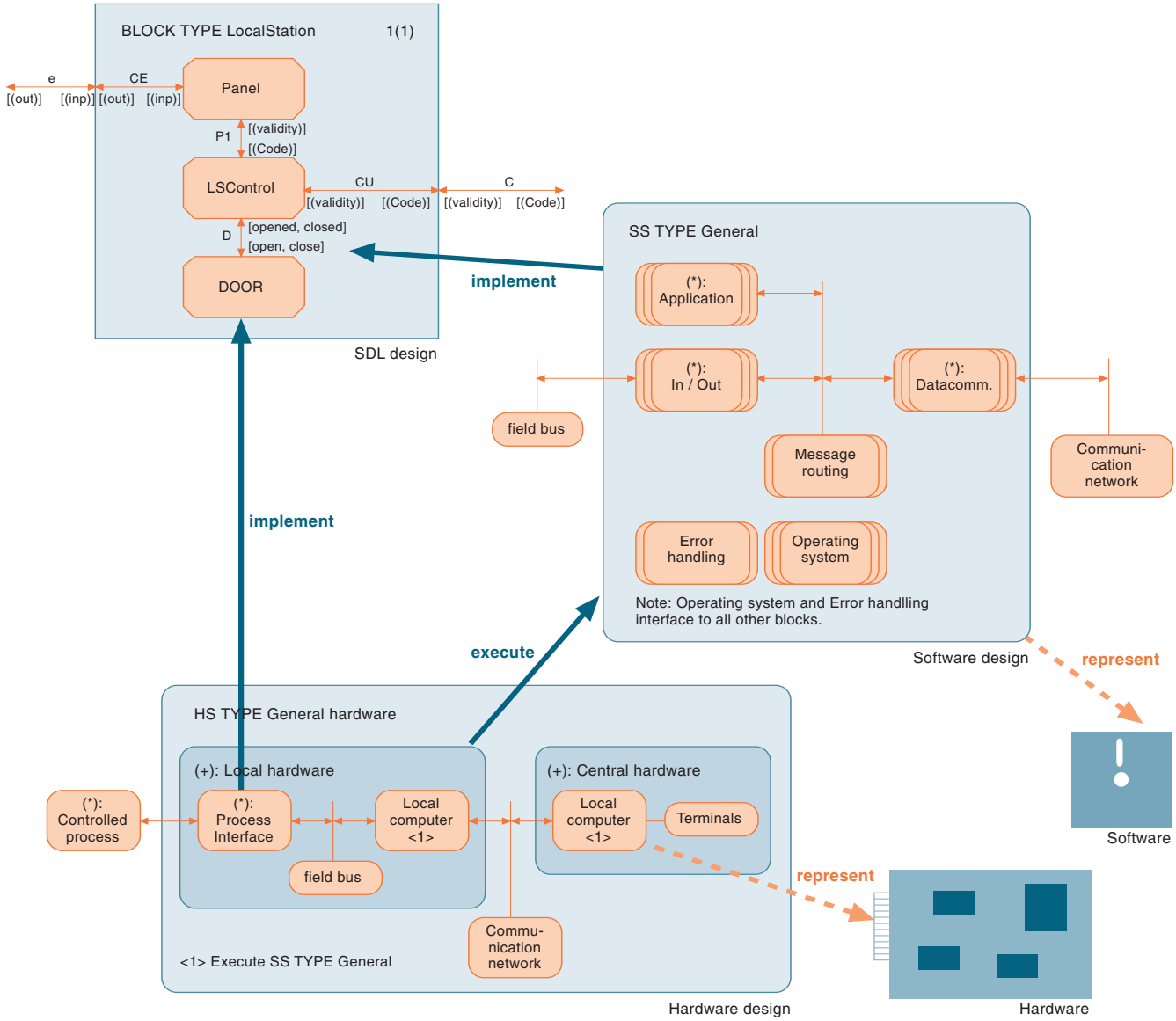


Figure 1 Implementation descriptions illustrate the coupling between functional design and implementation design[2)]

world of formal definition techniques to the accidental world of programming languages and computers can be worthwhile, since many implementation choices are available. Indeed, knowing and controlling them is crucial for anyone needing to remove bottlenecks and optimise performance. This activity is here called Implementation Design.

2 The Ideal World of SDL

SDL is a language with an ideal view of the world. It is based on the concept of finite state machines communicating via asynchronous[3)] messages. The underlying SDL machine of each process contains a single FIFO queue of infinite length where all signals to the process arrive. SDL processes consume one signal at a time from the FIFO queue, and perform a transition on that signal. Transitions take an "insignificant" amount of time, and run to completion once they are started. Note that processes operate in parallel, so the transitions of many processes may run at the same time. All data belong to processes[4)], and are described by abstract data types of possibly infinite range. SDL channels are secure; SDL signals never disappear in a signal transmission.

Additional SDL constructs include:

[2)] *This figure uses the SOON notation [1] to depict Software design and Hardware design. See also [4, 8, 9, 10]. See [11] to see to what extent UML stereotypes can model the same as SOON.*

[3)] *A kind of synchronous communication is supported in remote procedure calls.*

[4)] *SDL 2000 now allows data to belong to blocks, though access to the data is controlled by the state machine associated with the block.*

Implementation design

The goal of Implementation Design is to define the mapping between the abstract system defined in SDL, and a concrete system made up of hardware and software components. [1]

- Process identity values (PId) used to identify communicating peers (no specific implementation is presumed by SDL);
- Priority input;
- Enabling conditions;
- Remote procedure calls;
- A save queue for signals to be treated after the next change of state;
- Importing and exporting variables from/to other process;
- Timers that can be started or stopped, and that signal timeouts in the form of signals in the FIFO queue. SDL recognises time ticks, but does not define the interval between two ticks.

For space and simplicity, these and other SDL mechanisms are not elaborated any further in this article. An overview of the interior of an SDL state machine is shown in Figure 2, see also the paper by Rick Reed on SDL-2000 for new millennium systems in this issue.

3 The Not-So-Ideal Real World

SDL is different from the real world in many respects. Real systems have finite resources like limited queue lengths and maximum value ranges for data variables, and programs on a single computer do not run truly parallel. Real systems are subject to errors, e.g. the malfunctioning of hardware, and the corruption of signals on unreliable communication paths. In addition there may be different synchronisation needs between a system and its environment than the asynchronous, time-independent world of SDL messages.

There is a design gap between SDL and the world of programming languages [1]:

- **Concurrency**. Sequential programming languages like C or Java give no support to the concurrency of SDL. Some languages like CHILL and ADA support concurrency but do so differently from SDL.
- **Timing**. Very few programming languages support time at all. SDL-like timing is not directly supported by any language, not even by CHILL.
- **Communication**. SDL-like signal communication is not directly supported by any programming language.
- **Sequential behaviour**. An SDL process graph specifies state-transition behaviour in the fashion of an extended finite state machine. Programming languages specify action sequences.
- **Data**. SDL data are abstract and possibly infinite. The implementation in a programming language has to be concrete and finite.

But many of these differences can be overcome with affordable measures. Some of these measures and techniques will be introduced in the following sections.

3.1 Implementing Signal Transfer

Message transfer in the ideal SDL world can most generally be implemented using an asynchronous messaging technique, or in special

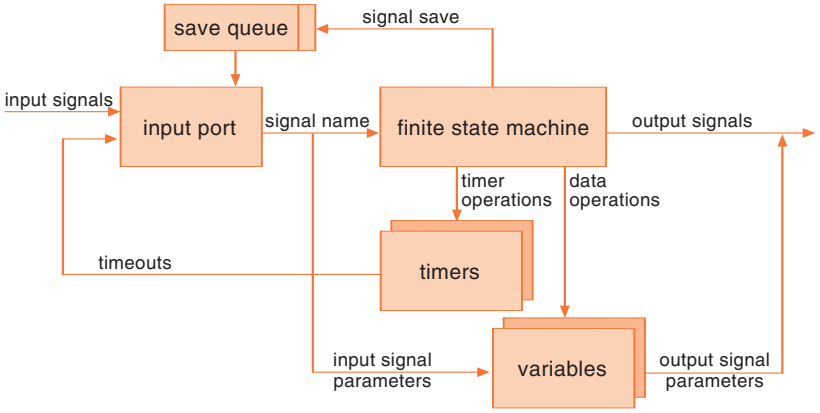


Figure 2 The SDL machine

cases as continuous values, or procedure calls or method calls in the programming language.

3.1.1 Signals Implemented as Method Calls

Using method or procedure calls, the signal sending is implemented by calling a method implemented in the receiving process. That method contains the transition of the receiving process. This synchronous technique is the most time efficient, but can only be used in special cases where the following conditions are satisfied [1]:

- The SDL behaviour must have the nature of a procedure call tree (e.g. A calls C who calls E, who returns to C, who calls F, who returns to C, who returns to A; see Figure 3).
- Only the SDL process at the root of the tree can receive signals from the environment or run SDL timers (process P1 in the example).
- Any response modelled as SDL signals must be implemented as return values in the procedure call, which limits the number of return signals to exactly one, which must be the last SDL action in the transition of the called process.
- The maximum processing time for any such call tree should not exceed the arriving rate of external signals to the root process.

This implementation scheme[5)] is time and space efficient in software, since it relies on some of the most basic primitives of programming languages and processors (e.g. using stack registers). The scheme is a common technique used for message calls in traditional designs of information systems modelled in e.g. OMT or UML. It implies synchronous communication semantics, and has inherent limitations that make it inappropriate for communication structures that form a network. The scheme is only viable when the mentioned set of criteria is fulfilled. These criteria are seldom present in reactive real-time systems, hence schemes involving asynchronous message communication and flexible scheduling are used.

The method call scheme can be used with advantage in parts of an SDL design where these criteria are satisfied, while other parts use the more general implementation technique of buffered messaging mentioned below. Optimisation performed during manual coding typically involved

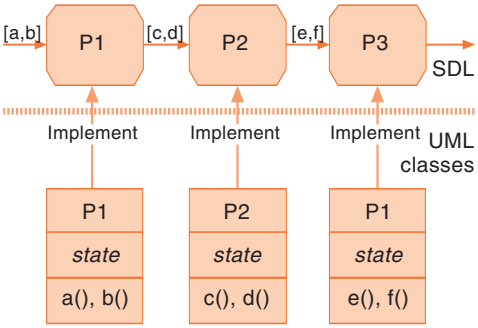


Figure 3 Implementing communication with method calls

```
Pseudo code of class P2:
    int state;
    void P2_c () {
       switch (state)
       {  case 1: P3_e (); P3_f (); State: = 2; return:
          case 2: .....; return;}
       }
    }

    void P2_d () {
       switch (state)
       {  case 1: P3_f (); State: = 3; return:
          case 2: .....; return;}
       }
    }
```

changing the SDL model so that these criteria were met within components faced with high performance requirements. Commercial tools for automatic code generation do not seem to address this issue very well (although the criteria analysis could well be performed by the tools).

3.1.2 Signals Implemented by Message Buffers

As stated above, the most general implementation of message sending involves buffered communication. This separates communication from activation, but requires additional memory (the buffers) and processing (buffer handling) compared to procedure calls. A very general approach involving parallel software processes (here denoted *Sender* and *Receiver*) that communicate using a general semaphore *S* is shown in Figure 4 [1].

The infinite message queues of SDL do not exist in the real world, but by employing techniques like semaphores providing back pressure, or exception handling at buffer overflow, the sending processes may be temporarily suspended. Then the receiving process can consume a signal from the limited sized input buffer, thus making room for additional messages. Protection using semaphores is shown in Figure 4, where *Sender* must fetch a free buffer from the freepool *F* prior to

5) Note that this technique departs from the SDL semantics of run-to-completion for transitions that applies between SDL processes.

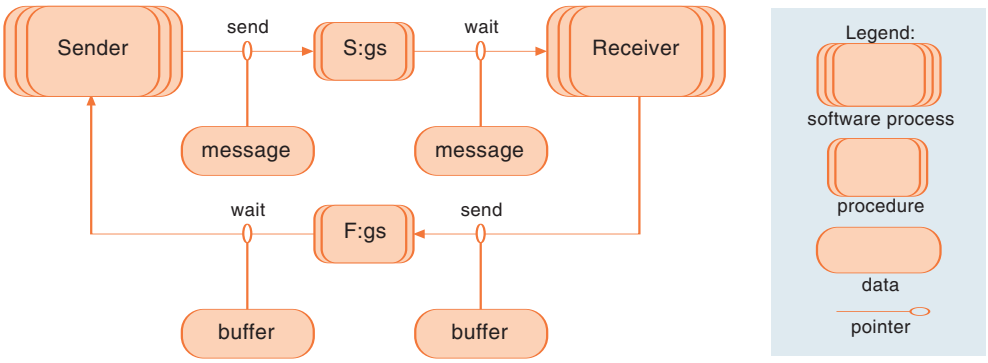


Figure 4 Concurrent buffered communication

sending a message to *Receiver* via *S*. *Receiver* will return the buffer to the freepool *F* after consuming the message. *S* and *F* are general semaphores containing queues, which are typically well supported by operating system kernels.

3.1.3 Optimising Parameter Passing

A final remark on the implementation of signal transfer is concerned with the parameters of SDL signals. In SDL, a process can only get hold of the input parameters by copying them into process variables. Likewise, parameters have to be explicitly set in output signals. There is no built-in mechanism to forward a received signal to some other receiver, complete with parameters.

This typically leads to a lot of copying in and out of parameters, and large amounts of process data required to hold values that are of no interest to the process as such. In designs of protocol stacks, this is particularly apparent. Many newcomers to SDL react to this, and view it as a waste of time and space.

However, this need not be the case in the implementation. An analysis of the SDL can show that parameters are e.g. forwarded largely unchanged, and an implementation can e.g. reuse the message buffer without returning it to the freepool in Figure 4. Or analysis can reveal that process data are only used to hold values for the duration of a transition, and never longer. In an implementation, such process data need not be stored in non-volatile memory, but instead placed on the stack. These are examples of optimisations that are common-place in good compilers, and should be found for SDL compilers too. See also [2].

Note, though, that there are limits to what a general code generator can analyse; if a process calls external code, the code generator may not be able to perform a complete analysis. If so, other mechanisms must be used, e.g. special annotation or commenting of signals and data.

3.2 Implementing SDL Processes

The implementation scheme most close to the ideal SDL world is realising each SDL process

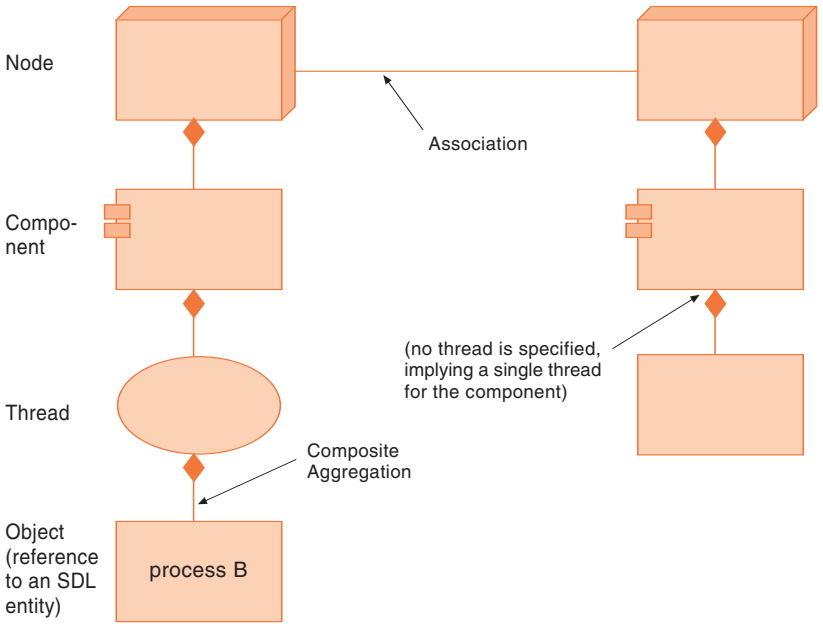


Figure 5 UML deployment diagrams used to express software structure (from Telelogic deployment editor)

instance on its own hardware processor. Thus every process runs truly parallel. This is seldom feasible, and the most common technique is to collect a number of SDL process instances in one software process (also known as operating system process or task), and have a number of such software processes running at different levels of priority.

SDL processes are typically allocated to high priority tasks (like timing and I/O processes), medium priority tasks (the application processes), and low priority tasks (SDL processes that perform background jobs performing statistics, monitoring, logging and error reporting).

This type of software design decision is outside the scope of SDL, and cannot be formally captured in the SDL model. Instead, additional descriptions are necessary, see Figure 1. Design methodologies like TIMe [3, 16] treat implementation design as a topic of its own, and SDL tools typically have additional functionality to help such grouping and code distribution in the program generation process. An example is the deployment editor in Telelogic TAU, see Figure 5. See also the paper by Philippe Leblanc, Anders Ek and Thomas Hjelm on Telelogic SDL and MSC Tool Families in this issue.

3.3 Implementing State Transition Diagrams

The term *process* has two distinct meanings; an *SDL process* is a Finite State Machine, while a *software process* (also known as operating system process or task) is a unit of scheduling and priority in a running software system. One of the important decisions during Implementation design is deciding the mapping from SDL process to software process; this can be decisive for the properties of the resulting system.

The state diagrams of SDL that model the actual behaviour of the SDL process can be implemented in a number of ways, with various results regarding execution time, space required, and support for range of SDL constructs, see [1].

The *state-oriented method* uses the CPU's program counter to store the state of the SDL process, with GOTO jumps for state changes. On the one hand, this scheme saves time and space in the implemented software process, but on the other hand wastes time and space by requiring more software processes, since it limits the mapping to a 1:1 relationship between SDL process instances and software process instances.

The most general scheme is to use a *table form of the state diagram* combined with common programming language constructs. The current state of each process is typically stored in a process variable along with the process data, and the state diagrams are stored in table rows for fast look-up at transition time. Multiple instances of a single process type or of different process types can be grouped together to run in the same software process, providing a N:1 mapping.

3.4 Implementing Abstract Data Types – and Adding Legacy Code

Not all detailed working of a system deserves to be visible at the SDL level. Examples include aspects like device drivers, complex algorithms and complex operations on data. The SDL way of concealing some of this detail is by extended use of operations on user defined data types. Data types are defined on the SDL level, typically by building upon the predefined SDL data types, or by making new types. New data types are defined from the predefined types, e.g. by restricting the value range, thereby providing a mapping to programming language constructs.

```
syntype BYTE integer constants (0:255);
```

For example, the ideally infinite range *integer* of SDL can be used to make an 8-bit *BYTE* type:

The BYTE variables can then easily be mapped to e.g. "short unsigned int" in C, for a given compiler for a given processor.

SDL-96 [4] allows operations on data to be described with axioms. Axioms are not constructive, and are seldom used in practice[6]. Instead, operations are defined by their signature at the SDL level, and are implemented in the programming languages of your choice. An example of this is given in Figure 6.

Hand coding SDL operations is one method of interfacing manually written or legacy code into the SDL world. Another way of interfacing legacy code is by concealing a legacy part in dedicated SDL processes, e.g. a database system or a GUI. SDL tool vendors provide good support for such code integration, although reverse engineering is not yet generally available from commercial CASE tools[7] (many contend that reverse engineering is not applicable to SDL systems engineering). One of the benefits of

---

[6] *Axioms were removed from the SDL-2000 standard [9].*

[7] *Work within reverse engineering to SDL is currently in progress, e.g. at the Software Engineering Institute at Moscow [12].*

Figure 6 Abstract data type in SDL-96 with corresponding C code for an operator

```
newtype dir_no /* Directory number */

struct
    digits      integer; /* Number dialled */
    no_digits   integer; /* Number of digits */
adding
operators
    append_digit : dir_no, integer -> dir_no;
endnewtype;
```

```
dir_no append_digit(dir_no dno, int digit)
{
    if ((digit>=0) && (digit<=9))
    {
        dno.digits=dno.digits * 10 + digit;
        dno.no_digits ++;
    } else {
    error(PARAM_OUT_RANGE, digit); }
return (dno);
}
```

interfacing legacy code in dedicated SDL processes, is that this maintains a loose coupling between the SDL part and the legacy part.

3.5 SDL Runtime Support

A common technique employed when implementing SDL systems is introducing a layer of software called the SDL runtime support between the application processes and the operating system that runs on the hardware. Novice SDL users may not be aware of this layer, since program generation tools typically offer code generation to both naked processors and a range of operating system kernels like PSOS, VRTX, etc. The SDL runtime support implements many concepts of the SDL machine (see Figure 2), with some limitations due to the nature of the real world, thereby making the task of implementing the actual SDL processes much simpler, see Figure 7 [1].

4 Support for Program Generation from CASE Tools

Generally speaking, the commercially available tools give good support for program generation, and the common SDL application developer does not need to spend much time analysing what schemes are used in the program generation. The dedicated application developer probably never needs to read the generated code, which may be of interest during initial integration testing only. The resulting programming

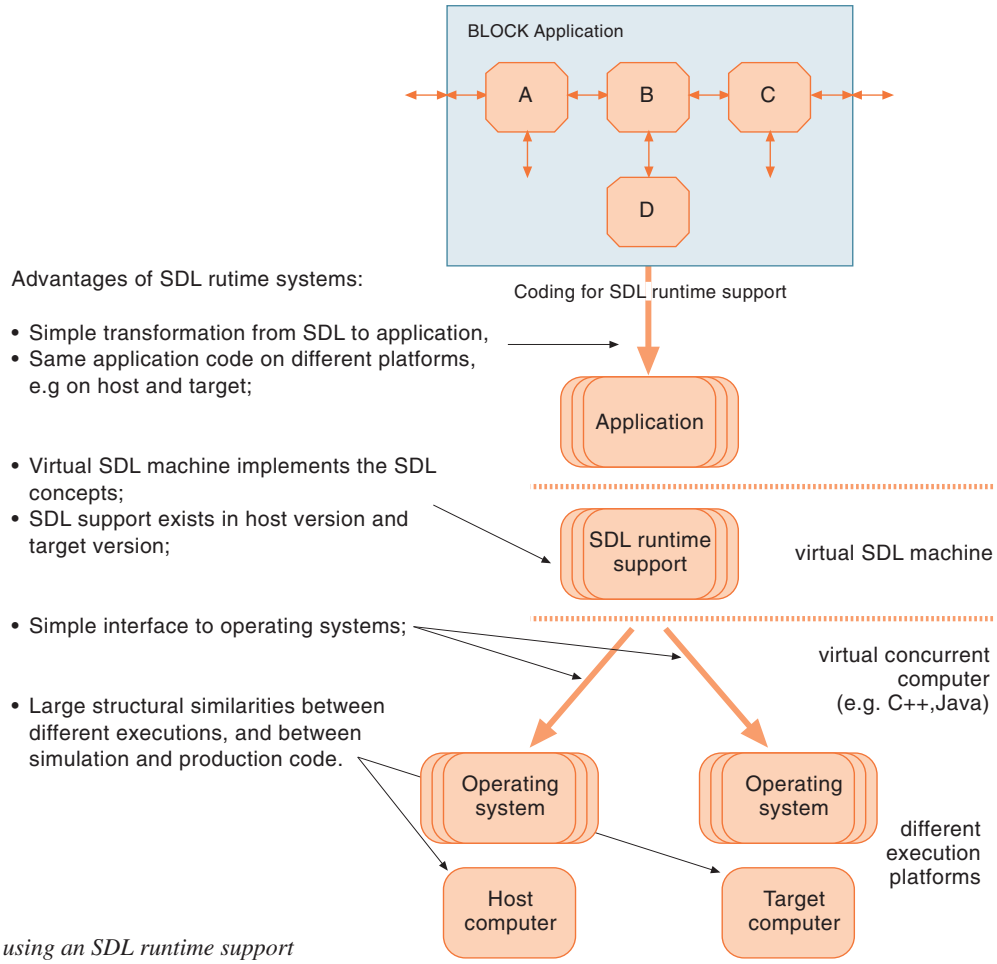


Figure 7 Layered implementation using an SDL runtime support

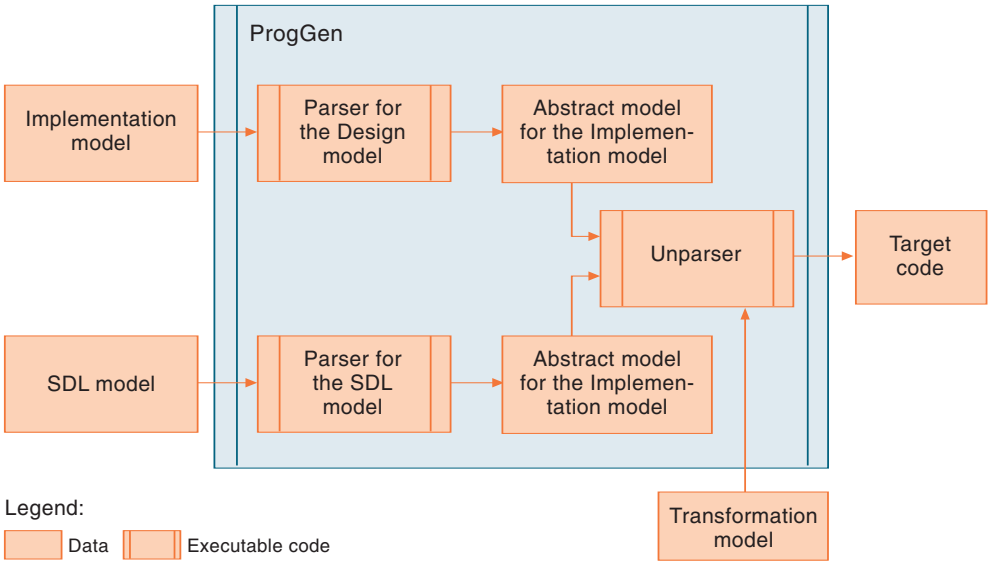


Figure 8 ProgGen's architecture

code can in general be untouched by human hand – and unread by human eye.

But the same does not apply to the necessary few who are responsible for mapping the ideal design onto the real world. And this is not always a trivial task, as this article hints at. To get optimised code for a new product, blood, sweat and tears may still be required. And although tool vendors have and are investing considerable effort in facilitating the program generation process (a fact that is reflected in the considerable rise in price of the toolkit when program generation is added), there is still much left to be desired.

It is not yet the case that the implementation designer can choose different implementation techniques for different SDL components in a system, e.g. to optimise for space in one instance, and size in another. Or that implementation speed and size are automatically adjusted depending on which SDL constructs are used in an actual design.

SINTEF has been working with program generation for over 15 years, and has developed a very flexible program generator named ProgGen [5, 6]. ProgGen takes as input the textual version of an SDL design, and outputs code in a variety of formats and programming languages, depending on the skeleton files defined by the implementation design. This tool has recently been extended with the capability of selecting different mapping strategies for different parts of an SDL design, according to implementation design decisions fed into the program generator in a textual form [7], see Figure 8. Although ProgGen is not a fully-fledged tool (e.g. it currently only supports SDL-88 and SDL-92 [8]), it shows what can be obtained from using flexible tools.

5 Consequences of Program Generation to the Development Process

Working in a design-oriented paradigm, users report higher quality and higher productivity, especially when maintaining or enhancing systems designed using SDL. Many companies have over a decade of experience with automatic code generation from SDL, and are constantly increasing the penetration of SDL in their reactive real-time system designs.

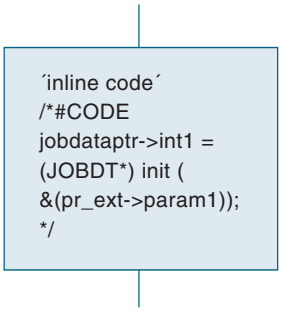
Industry typically reports that errors delivered to the market are reduced by at least 50 % compared to a pre-SDL development process, and that an increased productivity of 20–30 % in software design has been measured[8]. Even higher yields in the area of test generation and execution can be expected when TTCN[9] is introduced. The added cost of investing in a new development process is typically earned within the first project, with subsequent projects reaping the full benefits. Developers can more easily move between projects, projects become less person-dependent, and project progress is easier to define and monitor (less of the "90 % finished" syndrome). This is good news for project leaders, management and designers alike.

---

[8] *Data from the Norwegian SISU project in 1989 – 1996 [13]. CASE vendors boast of even higher earnings [14].*

[9] *Test suites in Tree and Tabular Combined Notation (TTCN) can be semi-automatically derived from SDL and MSC models [15].*

Figure 9 Example of inline code in an SDL process graph



Some words of caution should be mentioned, though. One direct consequence of automatic code generation is that SDL is used much as a graphical programming language, substituting textual languages such as C++ and Java[10]. SDL was initially not designed to be a programming language, and has until SDL-2000 left much to be desired in that respect. Many constructs expected by programmers have been absent or were cumbersome, e.g. expressing behavioural loops[11].

The CASE tools supporting inline-programming directives in native programming languages have introduced another drawback. It is easy to understand the motivation for inline code, since it simplifies the task of generating complete applications in some chosen programming language. But it defies the objective of building implementation-independent models; one can no longer change implementation language without going through the whole model and altering the inline code.

One aspect to be kept in mind by SDL designers wanting to generate code automatically, is the need to be formal. If one has the habit of using SDL slightly informally, and wants to start generating code, the need to be precise and detailed becomes apparent. Loose ends must be tied up, and informal use of SDL constructs revised. These revisions also apply when simulation and formal analysis of SDL system is performed, so this extra work is not a weighty argument against code generation, since simulation gives a high return on investment. There are reports from industry that an additional 30 % effort is required to fully formalise SDL designs, compared to designs in informal SDL. This increase in design effort, though, is quickly earned in the later maintenance and enhancement phases.

Note that some SDL tools support the simulation of informal SDL, a feature that is attractive for high level designs.

Some companies evaluating automated code generation have become aware of the penalty involved in the way of memory requirements and processing overhead introduced by some of the implementation platforms. Although these requirements vary depending on e.g. the level of debugging requested, and the fact that different target operating environments are supported by CASE tools[12], one can still find examples of manual coding that are more efficient.

With manual coding you can take advantage of your knowledge of the environment, using shortcuts to save time and space. This can work fine, as long as the assumptions are valid and are understood. One can single out particularly demanding parts to be hand coded or realised in hardware, while other parts are generated automatically.

There is a strong pressure on tool vendors to improve the code optimisations, like there used to be when 3rd generation programming languages were introduced. Programmers now generally tend to rely on the quality of compilers, and do not write assembly code by hand unless they really have to. Hopefully, the same will soon apply for SDL code generation, if it does not apply already.

6 Impact of SDL 2000

Many of the new features added in the 2000 version of SDL [9] have taken into account the fact that SDL has become popular as a graphical programming language. For instance, it is now possible to introduce temporary data in transitions, that easily can be implemented as stack variables, thereby limiting the static memory requirements of an SDL process implementation; only data with a life span longer than a transition needs to be stored in persistent memory.

Exception handling, similar to what you find in C++ and Java (and CORBA), is defined in SDL

---

[10] *Interestingly enough, designers of digital hardware have moved in the opposite direction. Textual hardware description languages like VHDL and Verilog substitute the earlier graphical hardware expressions of the schematics and block diagrams. In general the goal is to strike a good balance between pictures and words. SDL is possibly close to this goal within its realm of description.*

[11] *Many of the changes to SDL introduced in SDL 2000 were motivated by the current use of SDL as a programming language, and has resulted in many requested programming features, like loop constructs and stack variables (see later).*

[12] *E.g. C-micro from Telelogic.*

2000. This is depicted in special graphics in the process graphs.

The action language introduced in SDL 2000 supports typical programming constructs such as loops and conditions, features which "SDL programmers" have missed. This results in a textual representation of what previously was space-consuming graphics, or was hidden in operators or, worse yet, as inline code in some programming language, see Figure 10. The action language should remove the need for much of the inline code that is all too often found in SDL designs, and should bring SDL design back toward the implementation-independent track where it originally was and should be.

We conclude that SDL is moving in a direction that is favourable for those wanting to derive their code from SDL, and who wish to maintain systems at the SDL level. It is possible to combine designs in SDL with models in UML that cover other system aspects (like database systems and GUI). Tool vendors are working hard to implement the aspects of SDL 2000 most needed by SDL users, especially in the field of program generation. This, along with the other aspects of SDL treated in other articles in this issue, will continue to increase the adoption of SDL in product development world-wide.

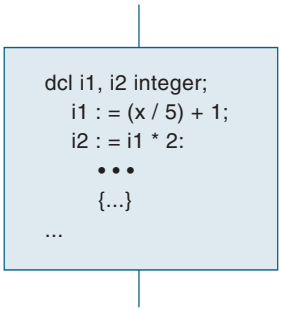


Figure 10 Textual tasks in SDL 2000

7 References

1 Bræk, R, Haugen, Ø. *Engineering Real Time Systems.* Hemel Hempstead, Prentice Hall International, 1993.

2 Henke, R, König, H, Andreas Mitschele-Thiel, A. Derivation of Efficient Implementations from SDL Specifications employing Data Referencing, Integrated Packet Framing and Activity Threads. In: *SDL'97. Time for Testing, Proceedings of the Eighth SDL Forum,* 397–414. Amsterdam, Elsevier Science, 1997.

3 Bræk, R et al. *TIMe : The Integrated Method.* 2000, October 12 [online]. – URL: www.sintef.no/time.

4 ITU-T. *CCITT Specification and Description Language.* Geneva, ITU-T, 1996. (Recommendation Z.100 (SDL'96).)

5 Floch, J. Supporting evolution and maintenance by using a flexible automatic code generator. In: *Proceedings of ICSE-17 – 17th International Conference on Software Engineering,* Seattle, 1995.

6 *The ProgGen program generator.* 2000, October 12 [online]. – URL: www.informatics.sintef.no/projects/proggen/.

7 Johansen, U. *SEP sluttrapport : Designstyrt oversetter fra SDL.* Trondheim, SINTEF, 1999. (Sintef report STF40 A99040.) (ISBN 82-14-01729-7)

8 ITU-T. *CCITT Specification and Description Language.* Geneva, ITU-T, 1994. (Recommendation Z.100 (SDL'92).)

9 ITU-T. *CCITT Specification and Description Language.* Geneva, ITU-T, 1999. (Recommendation Z.100 (11/99).)

10 ITU-T. *SDL Methodology Guidelines.* Geneva, ITU-T, 1993. (Recommendation Z.100 (03/93) Appendix I.)

11 Floch, J. *Using UML for architectural design of SDL systems.* Trondheim, SINTEF, 2000. (SINTEF report STF 40 A00009.)

12 Mansurov, N, Probert, R L. Dynamic scenario-based approach to re-engineering of legacy telecommunication software. In: *SDL'99 The Next Millennium, Proceedings of the Ninth SDL Forum.* Amsterdam, Elsevier Science, 1999.

13 *The SISU project.* 2000, October 12 [online]. – URL: www.sintef.no/sisu.

14 Crossing the fjord. *Signals.* Malmø, Telelogic, 1999. (Telelogic Newsletter No 2.) (www.telelogic.se)

15 Kerbrat, A, Jéron, T, Groz, R. Automated test generation from SDL specifications. In: *SDL'99 The Next Millennium, Proceedings of the Ninth SDL Forum.* Amsterdam, Elsevier Science, 1999.

16 Bræk, R et al. Quality by construction exemplified by TIMe – The Integrated Methodology. *Telektronikk*, 95 (1), 73–82, 1999.

Validation and Testing

DIETER HOGREFE, BEAT KOCH AND HELMUT NEUKIRCHEN

Use of formal specifications provides the basis for allowing validation of the specification towards expected behaviour, and it allows testing of an implementation according to the formal specification. This paper introduces and provides an overview of techniques for validation and testing.

Dieter Hogrefe (42) graduated with a diploma degree in Computer Science and Mathematics from the University of Hannover where he later received his PhD. His research activities are directed towards Computer Networks and Software Engineering. He has published numerous papers and two books on analysis, simulation and testing of formally specified communication systems. He has worked for SIEMENS research centre, the University of Hamburg, and the University of Berne. Since 1996 he has been director of the Telematics Institute in Lübeck and full professor at the University of Lübeck.

hogrefe@itm.mu-luebeck.de

Beat Koch (35) studied Computer Science at the University of Berne, Switzerland. After his graduation in 1994, he worked as a software engineer in industry. Since 1996, he has been a research assistant at the Intitute for Telematics of the Medical University in Lübeck, Germany. His research focuses on automatic test generation and he is one of the developers of Telelogic's Autolink tool. Currently, he is finishing his PhD thesis on "Test-purpose-based Test Generation for Distributed Test Architectures".

bkoch@itm.mu-luebeck.de

Introduction

When formal description techniques (FDTs) such as the *Specification and Description Language SDL* [17] were developed for computer communication systems, the idea of precise and rigorous specification were the driving force. Prose specifications of complex technical matters are not always sufficient to guarantee interworking of products from different vendors. Also, hidden ambiguities are sometimes only discovered after the respective systems have been put into operation. Errors of this kind tend to be very expensive. FDTs have promised some help in this matter, because they make possible rigor, precision, completeness, and so on.

However, it did not take long before these promises proved to be only partly true. While FDTs provide the basis for rigor, precision and completeness, they do not guarantee these qualities. In fact, using FDTs is very similar to programming, from which it is known that errors occur easily. Hence, *validation* techniques are very important and will be explained in the second section of this paper.

Even if we have a perfectly valid specification, its value is somehow limited if we cannot relate it formally to an implementation. In some cases, this relationship comes for free: If the specification is directly translated into an executable implementation without any manual steps. Unfortunately, this is not the common case. Usually, the SDL specification abstracts from a number of details which later appear in an implementation. These abstractions are filled with life during the design and implementation process. That is of course a source for errors. We will therefore devote the third section of this paper to the aspect of conformance testing, i.e. to the question on how to show that an implementation really conforms to a specification.

We conclude this paper with a summary and a view on current research activities.

Validation

Specifying a system with a formal specification language like SDL [16] is in some respects similar to implementing a system with an ordinary programming language. In both cases, the result will probably contain errors. Therefore, formal specifications as well as system implementations have to be tested in some way. However, although it is basically the same problem, the testing of an implementation is different from the testing of a formal specification.

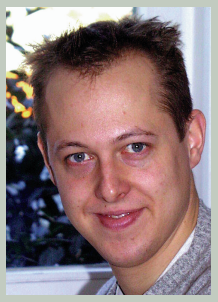
On a terminology level, implementation testing is also called *verification* and it involves activities concerning the question "Am I building the system right?". In contrast, *validation* deals with the question "Did I build the right system?" [21].

In the domain of designing and testing computer protocols using formal methods, both terms are used in a mixed way. This is due to the fact that a strict distinction of these two testing terms is not always possible.

Hence, according to [12], the term validation is used to describe all activities "used to check that the formal specification itself is logically consistent".

Another difference between the testing of an implementation and a specification is the underlying machine. The model specified by the formal language is usually executed on an abstract machine with potentially infinite resources. Furthermore, several views of one system may be specified using different models or even different specification languages. This is in contrast to the testing of an implementation module, which comprises all the different models and runs on a physical machine.

There are several methods to check the consistency of a formal specification. Static techniques analyse the formal specification without executing the model described by the specification. Dynamic analysis techniques build an executable model from the formal specification in order to validate it.



Helmut Neukirchen (29) studied Computer Science at the University of Technology Aachen with a focus on distributed systems and software construction. He graduated with a diploma degree in 1999. Since January 2000 Helmut Neukirchen has been research assistant at the Institute for Telematics at the Medical University of Lübeck on the EU project INTERVAL. His main research interests are formal methods in specification and testing of communication and real time systems.

neukirchen@itm.mu-luebeck.de

Static analysis

Static analysis techniques evaluate a specified model without executing it. This can be achieved manually by reviews, inspections or walk-throughs, and in an automated way by syntax and semantic analysis.

Depending on the kind of manual static analysis, the author, other experts or professional inspectors analyse the documents by following a checklist. Besides general questions regarding completeness and consistency, such checklists may contain questions concerning properties specific to the formal language used for specification.

In the domain of automated static analysis, syntax analysis will just check whether the specification is in accordance with the rules describing correct statements in the particular language. Semantic analysis uncovers the use of variables without declaration, data-type clashes or references to unspecified parts. In general, syntax and semantic analysis assure that the specified system is well-defined, complete and self-consistent from a language point of view. A system that has passed these checks is comparable to a program that has been successfully compiled [11]. However, these techniques are not powerful enough to reveal whether the functionality of a system is correctly specified or not.

Dynamic Analysis

Dynamic analysis techniques execute the model described by the behavioural parts of the formal specification. This allows different approaches of validation. First of all, it is possible to test the executable model like any other executable program. Test scenarios with suites of test cases can be applied to the model. Comparing the expected behaviour with the observed behaviour as a result of a stimulus sent to the system allows validation of the specification on a black box basis.

A white box driven approach is to explore the functional behaviour of the model step by step. During such an exploration, two classes of properties can be checked [11]: General properties that apply to nearly any system independent of the particular requirements. They can be defined without reference to a particular system. Examples of such properties are the absence of deadlocks and livelocks, no reading of variables before assignment, or range violations. The second class of properties is system specific properties. They concern the special dynamic behaviour of the model depending on user requirements. By specifying assertions, invariants or other observable conditions which are particular to the modeled system, a validation tool can check whether these conditions hold during the execution of the model.

While the general properties can be checked automatically, checking the system specific properties requires an explicit description of the properties. Making these property descriptions is usually time consuming and may in turn be error-prone.

Figure 1 shows the relationship between model, formal specification and the corresponding implementation.

Simulation Based Validation

The most common dynamic analysis technique used in practice is based on a simulation-like state space exploration [12]. Another application for simulation is performance validation. This topic concerns non-functional timed properties of the examined system and is not covered here.

Exploring the state space of a model allows a white box based validation of the specification. Starting from the initial state, the executable transitions leading to successor states can be calculated. Analysing each visited state with respect to general and user-defined properties enables us to validate the specified model. Two classes of exploration algorithms have to be distinguished: exhaustive and non-exhaustive exploration algorithms.

Exhaustive validation performs an analysis of the complete model. Thus, definite statements about properties such as the absence of deadlocks can be made. The most common exhaustive exploration approach builds the reachability graph of a system by visiting all reachable states. The visited states in this graph are used during the exploration to avoid multiple visits of states. This ensures the termination of the algorithm. A reachability graph comprises all execution paths the system is able to perform. Deadlocks, livelocks or dead code can be discovered in this way. The state diagram of a small sample system and the corresponding reachability graph with a sample execution path is given in Figure 2. Circles depict states, arrows correspond to transitions.

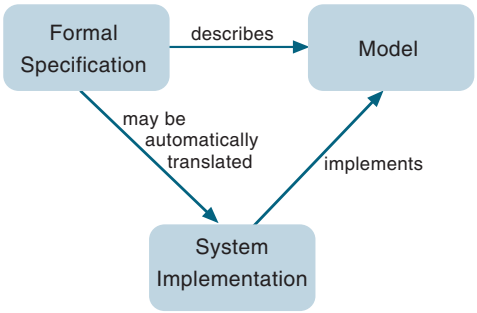
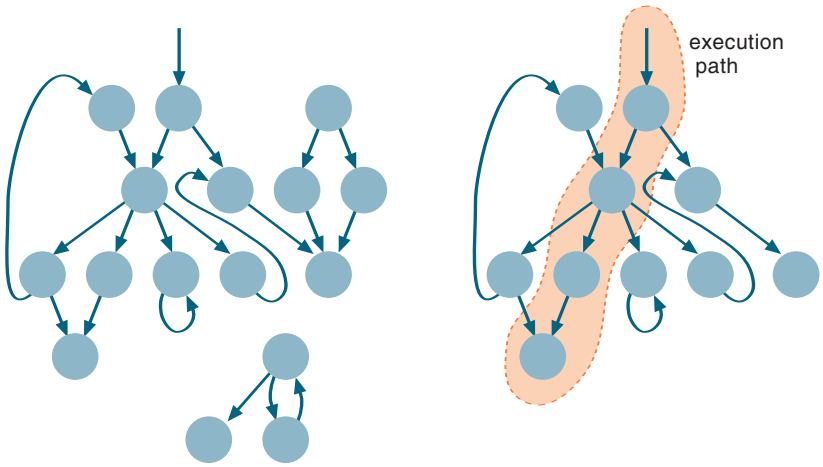


Figure 1 Relationship between model, specification and implementation



(a) State diagram (b) Reachability graph

Figure 2 A state diagram and its corresponding reachability graph

Unfortunately, this method is only applicable to small systems. The number of states grows exponentially with the complexity of a system. Non-exhaustive exploration algorithms cope with this state explosion problem by exploring only parts of a system. Certainly, this method cannot prove error-freeness for the whole system, but experience has shown that specification errors manifest themselves in many different states. Therefore, it is not necessary to cover all execution paths of the system to spot errors. The problem is to find a sufficient subset of all paths.

One practical technique is to choose a more or less random subset of depth-bound paths. Most tools provide a *random walk* exploration: Transitions to be executed are chosen randomly, yielding a random exploration of the state space [24]. By repeating random walks, the desired coverage of the specification is obtained.

A more elaborate non-exhaustive technique is the bit-state exploration [12]. Each state is encoded via a hash function to represent an array index. In this way, each state identifies a slot in a hash table used to indicate whether that state has already been visited or not. This enables tools to approximate an exhaustive search of considerably larger systems, because less memory is needed for each state – instead of storing visited states completely, just a one-bit slot per state is needed. Because of the risk of hash value conflicts, bit-state exploration is nevertheless a non-exhaustive validation technique.

Independently of these techniques, the number of states can be reduced by a controlled partial search. SDL tools like *SDT* and *Object*GEODE offer options to let the user limit the state space which has to be validated. For example, validation tools can be guided by *Message Sequence Charts (MSC)* [15] describing scenarios for the model under analysis. Such a guidance is a specialization of the notion of a *validation model* [12]. A validation model is a small, formally specified executable model describing a certain aspect of the system being validated. The complete system specification is checked against the validation model.

In the SDL domain, MSCs and SDL-based observer processes are used to validate a specification with regard to user-defined properties. By giving an MSC that describes some desirable behaviour of the system, the simulator checks for execution paths satisfying the following properties: The execution trace must include all events that exist in the MSC and must not contain any observable event that is not part of the MSC. The sequence of observable events must be consistent with the partial ordering of the events that is defined by the MSC.

Other Validation Techniques

If a system is specified using a more abstract specification language than SDL, more sophisticated validation techniques can be employed. Such languages are not widely used in the commercial area, because they differ from the popular imperative state machine based specification languages. An example is the μ-calculus which is based on temporal logic.

An approach well studied in the academic area is model checking. The idea behind this is to express states and possible transitions by means of logical predicates. Thus, questions concerning reachability can be answered by automatically solving logical equations. Symbolic model checking is still based on state exploration, but algebraic transformations are used and states are represented symbolically, not explicitly in a reachability graph. These symbolic logical formulas can be transformed into boolean expressions which in turn can be represented very efficiently by binary decision diagrams (BDDs) [3]. This allows us to explore larger systems [4]. The *Xeve/Estérelle* toolkit is an example of a symbolic model checking tool [1]. Another widely used model checker is the *Spin* tool [12]. It uses a partial order reduction technique to cut down the state explosion. In [19], a study of the interconnection of *Object*GEODE with a model checker is presented.

The *Las* tool introduces a new validation approach [7]. Linear programming is used to prove properties on communicating automata yielding a polynomial complexity. Linear programming is an efficient technique to solve certain optimization problems. This approach avoids the construction of the reachable state space of a model. Instead, it calculates directly on the formal model whether or not a certain

property holds by searching an execution path satisfying the property. The practical value of this approach is not yet clear and further studies are necessary.

Testing

Testing is an important aspect of today's product development cycle. The complexity of new telecommunications systems increases constantly; the amount of time needed for testing and its cost grow accordingly. In order to reduce time and cost for testing, research institutes, standardization organizations, tool providers and industry are actively developing formal methods, languages and tools for test generation and test execution. In this section, we will focus on test generation for software systems.

In the domain of software testing, many categories are distinguished: Domain, risk, load, stress, scenario, feature, integration or user testing are just some of the common testing methods. We will concentrate on conformance testing, where the conformance of a system implementation (the *Implementation Under Test (IUT)*) with regard to a (formal) system specification is checked. The *Conformance Testing Methodology and Framework (CTMF)* has been standardized with the ISO/IEC 9646 multipart standard [13]; it provides the foundations for the methods and tools discussed here.

Figure 3 shows the relationship between specification, *test suite* and implementation. A test suite consists of a set of test cases. Each test case describes sequences of signals which are exchanged between the tester and the IUT through *Points of Control and Observation (PCO)*. At the end of each sequence, one of the three possible test verdicts *pass*, *inconclusive* and *fail* is assigned. During test execution, the execution of each test case should end with a pass verdict. If this is the case, then a *conformance statement* can be made about the implementation.

There are two main problems which have to be solved in order to get high-quality conformance test suites: First, a set of test cases has to be identified which as a whole guarantees a certain level of conformance. Second, a test suite has to be generated which contains the information necessary to derive executable tests.

Test Case Generation

There are two approaches to test case generation: Automatic (exhaustive) test case generation, and test-purpose-based test case generation.

Exhaustive Test Generation

Exhaustive test generation methods aim at identifying and generating complete test suites automatically. The input is a *Finite State Machine (FSM)* or some variation, e.g. an *Extended FSM (EFSM)*, *Communicating FSM (CFSM)* or *Communicating Extended FSM (CEFSM)*. EFSM is the underlying model of a one-process SDL specification; CEFSMs correspond to SDL specifications with multiple communicating processes.

An important aspect of exhaustive test generation methods is the definition of the test suite goal. One commonly used goal is the establishment of a guaranteed *fault* coverage, e.g. 95 %: If a test suite is executed completely and no error is detected, then the statement can be made that the implementation is guaranteed to be free of 95 % of all possible faults (it may contain any number of the remaining five percent of possible faults, though). There are several types of faults, the main ones being *output* and *transfer* faults [20]. Output faults occur if the output of a transition does not match the expected output; transfer faults occur if a transition ends in the wrong tail state.

Another common test suite goal is to obtain a certain *code* coverage. Taking an SDL specification as an example, a code coverage of 90 % would mean that 90 % of all SDL symbols in the specification are covered by at least one test case. In general, test suites which obtain a high fault coverage are considered to be of higher quality than the ones which rely on code coverage.

Unfortunately, due to the state space explosion problem, current methods for exhaustive test generation can only handle specifications which are very limited in size and often restricted with regard to the specification possibilities offered by languages such as SDL. Nevertheless, methods and tools have been developed which produce test cases to test the components of a CEFSM in isolation or in context (embedded testing); examples can be found in [2] and [5].

Test-Purpose-Based Test Generation

Exhaustive test generation as described in the previous section has two major drawbacks: First, it can only be applied to small systems or parts of systems; it cannot be used for today's real-world complex systems, because of state space explosion and infinite state spaces. The second, less technical but nevertheless important drawback is the lack of test case documentation.

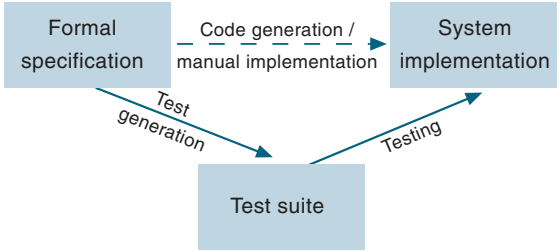


Figure 3 Relationship between specification, test suite and implementation

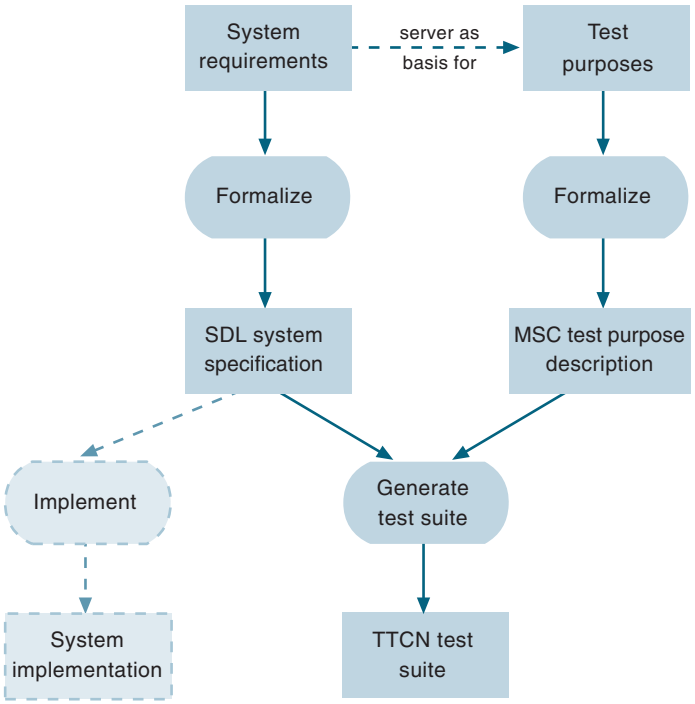


Figure 4 Test-purpose-base test generation

Automatically generated test cases tend to be rather non-descriptive sequences of test events which do not make much sense to the human reader. Test-purpose-based test generation methods alleviate both these problems.

The main idea behind test-purpose-based test generation is the following: Before a system is specified formally and implemented with some programming language, there are usually requirements capture and analysis phases. In these phases, engineers define the important signal flows. Later on, the system is specified with a formal specification language. It is obvious that the specification should exhibit the behavior defined during requirements capture, so the signal flows defined there can now be used as test purpose descriptions. To generate test cases, the system specification can be simulated against the test purpose descriptions. During the simulation run, the signal exchange at predefined *Points of Control and Observation (PCO)* is observed. Signal sequences which correspond to the test purpose description lead to a *pass* verdict; observed signals which are correct according to the specification but which are not expected in the test purpose are marked with an *inconclusive* verdict. This method has originally been proposed in [10] using SDL as the system specification language and MSCs for test purpose description (Figure 4).

Obviously, with the test-purpose-based approach, the quality of a test suite cannot be measured with fault coverage criteria.[1)] The completeness and quality of the test suite mainly depends on the experience of the persons defining the test purposes. However, the generated test cases are guaranteed to be consistent with the formal specification. This consistency is not given by default, since test designers are not necessarily aware of the formal specification. Furthermore, efficient algorithms and tools exist which can generate test cases even for very complex systems [9]. Last but not least, test purpose descriptions can also be used as documentation for the system. The applicability of this method to various real-world systems and protocols has been shown [22], [23].

Test Generation without a Formal System Specification

For many existing systems in industry, formal system specifications have never been developed and it would not be cost-effective to do a specification just for test generation purposes. In other cases, only partial specifications exist. Nevertheless, the idea of being able to specify test purposes with MSCs instead of directly writing test cases in a specialized test language has been widely accepted. For this reason, industry requires tools which are able to translate test purpose descriptions directly into test cases.

Test Suite Generation

The road to executable test suites does not end with the identification and generation of test cases. Test suites must be saved either in a proprietary language defined by the test equipment vendor or in the standardized *Tree and Tabular Combined Notation (TTCN)* [14]. If TTCN is chosen and the test suite is supposed to be easily readable and understandable by humans, then the following optimizations should be done automatically by the test generation tool:

- All declarations should be generated;
- The number of constraints should be minimized by merging identical constraints and by supporting constraints parameterization;
- Constraints should get meaningful names;
- The number of test steps should be minimized through parameterization.

If Concurrent TTCN is used to specify tests for a distributed test architecture, then the tool has to support additional features: Test case descriptions have to be split into descriptions for all test components and synchronization messages should be generated automatically.

[1)] *It is possible to measure the code coverage during simulation, though.*

The ultimate goal of any industrial-strength TTCN test generation tool must be to generate test suites which require no manual postprocessing.

Tools

There are two SDL-based test generation tools which are available commercially: *Autolink* [9] has been developed in a joint project with Telelogic AB by the Institute for Telematics of the University of Lübeck; it is part of the Telelogic Tau tool suite. *TestComposer* [18] has been developed by Verilog as part of the *Object-*GEODE tool set. Both tools are based on the test-purpose-based test generation methodology and they contain many similarities. Below, we just give a summary of the distinguishing features of the tools.

Autolink has been available since 1997 and it has been evaluated and used in projects of the European Telecommunications Standards Institute (ETSI), as well as telecommunications companies. Because of the incorporation of many features requested by users, its strengths lie in the readability of generated test suites and in its adaption to industry realities. Therefore, besides offering state exploration based test generation, Autolink also supports the direct translation of MSCs into test cases without the need of a complete formal system specification. Other unique features are:

- Generation of Concurrent TTCN output including the automatic generation of coordination messages;
- Inclusion of a simple configuration language which allows to define rules for automatic constraints naming, constraints parameterization and test suite variables (PIXIT);
- Support for distributed test generation.

Although TestComposer has been developed from scratch, it is based on a relatively long history of research and tools in the domain of automatic test generation. Its strengths lie in its state space exploration techniques. For example, TestComposer is able to generate test cases from partially defined test purpose descriptions; the missing parts are filled in automatically. Other distinguishing features are:

- Automatic postamble[2)] computation;
- Comprehensive timer support;
- Output of test suites in a user-definable format.

Conclusions

Although it may not seem obvious at first sight, there are several similarities between validation and automatic test case generation. Both techniques require searching the state space of the system under investigation. During validation, the tools look for peculiarities in the state space such as unspecified reception or deadlocks. During exhaustive test case generation, the whole state space is searched for those test cases which can detect faults in an implementation. In the test purpose based method, the state space is explored to check if a trace exists which matches a predefined and formally specified test purpose. This way, test cases are generated which lead to pass and inconclusive test execution verdicts. Due to these similarities, tools such as Autolink [9] and TestComposer [18] make use of techniques originally developed for validation, such as described in [12].

Validation and test case generation both suffer from the state space explosion problem which makes it impossible to exhaustively validate or test systems of practical size. However, the use of available tools is already beneficial and highly recommendable: For validation, advanced state space exploration algorithms have been developed which allow to explore significant parts of the state space of real-world system specifications in a feasible amount of time. For test generation, support of the pragmatic approach of using (human specified) test purposes combined with state space exploration has made tool-assisted development of high-quality test suites a reality.

Experience in industry and ETSI has shown that the initial effort to develop a formal specification can be quite high. However, this effort is offset by the ability to detect design errors at an early development stage through validation, by the possibility of automatic code generation and the ability to easily develop test suites through automatic test generation. All in all, relevant reductions in time-to-market and development cost can be expected through the use of formal techniques.

A lot of research is going on in the field of formal specification, validation and test generation. At the end of 1999, new versions of SDL and MSC have been standardized by the *International Telecommunication Union (ITU)*. With the *Unified Modeling Language (UML)*, a new notation for object-oriented software development has been standardized by the *Object Management Group (OMG)*. At ETSI, guidelines are developed on how to use object-orientation in the standardization and specification process of telecommunication systems. Also at ETSI, a

2) *A postamble is a sequence of test events to bring the IUT into a stable, well-defined state.*

completely new version of the testing notation TTCN is in the final stages of development. Meanwhile, research institutes and tool providers continue to develop enhanced methods and tools for validation and test generation.

References

1 Bouali, A. *Xeve : an estérel verification environment.* Sophia Antipolis, Inria, 1997. (Technical Report 0214.)

2 Bourhfir, C et al. A test case generation tool for conformance testing of SDL systems. In: Dssouli, E et al. [8], 405–419.

3 Bryant, R E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers,* 35 (8), 667–691, 1986.

4 Burch, J R et al. Symbolic model checking : 10^{20} states and beyond. In: *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science,* Philadelphia. Los Alamitos, IEEE Computer Society Press, 1990, 428–439.

5 Cavalli, A et al. Hit-or-jump : An algorithm for embedded testing with applications to in services. In: *Proceedings of the Joint International Conference FORTE/PSTV'99. IFIP TC6 WG6.1.* Boston, Kluwer, 1999.

6 Cavalli, A, Sarma, A (eds.). SDL'97 – time for Testing. *Proceedings of the Eighth SDL Forum,* Evry, France, September 1997. Amsterdam, Elsevier, 1997.

7 Devulder, S. A comparison of lpv with other validation methods. In: *Proceedings of ASE-99: The 14th IEEE Conference on Automated Software Engineering,* Cocoa Beach. Los Alamitos, IEEE computer Society Press, 1999.

8 Dssouli, R, Bochmann, G v, Lahav, Y (eds.). *SDL'99 – the Next Millennium, Montréal. Proceedings of the Ninth SDL Forum.* Amsterdam, Elsevier, 1999.

9 Ek, A et al. Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. In: Cavalli and Sarma [6], 245–259.

10 Grabowski, J, Hogrefe, D, Nahm, R. Test case generation with test purpose specification by MSCs. In: *SDL'93: Using Objects. Proceedings of the Sixth SDL Forum*. Amsterdam, North-Holland, 253–265.

11 Hogrefe, D. Validation of SDL systems. *Computer Networks and ISDN Systems,* 28 (12), 1659–1667, 1996.

12 Holzmann, G. *Design and Validation of Computer Protocols.* Englewood Cliffs, Prentice-Hall, 1991.

13 ISO/IEC. *Information technology – Open systems Interconnection – conformance testing methodology and framework*. 1994. (International ISO/IEC multipart standard No. 9646.)

14 ISO/IEC. *Information technology – Open systems Interconnection – conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. 1997. (International ISO/IEC standard No.9646-3.)

15 ITU-T. *Message Sequence Charts.* Geneva, 1999. (ITU-T Recommendation Z.120.)

16 ITU-T. *Specification and Description Language (SDL)*. Geneva, 1999. (ITU-T Recommendation Z.100.)

17 Sarma, A, Ellsberger, J, Hogrefe, D. *SDL – Formal object-oriented language for communication systems.* London, Prentice-Hall, 1997.

18 Kerbrat, A, Jéron, T, Groz, R. Automated test generation from SDL specifications. In: Dssouli et al. [8], pages 135–151. Proceedings of the Ninth SDL Forum.

19 Kerbrat, A, Rodriguez-Salazar, C, Leujeune, Y. Interconnecting the objectgeode and caesar-aldeberan toolsets. In: Cavalli and Sarma [6], 475–490.

20 Petrenko, A, Bochmann, G v, Yao, M. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems,* 29 (1), 81–106, 1996.

21 Rakitin, S. *Software Verification and Validation.* Norwood, Artech House, 1997.

22 Scheurer, R. *Demonstrating the Applicability of Automatic Test Case Generation Methods.* PhD thesis. Berne, University of Berne, 1997.

23 Schmitt, M et al. Autolink – putting SDL-based test generation into practice. In: *Proceedings of the 11th International Workshop on Testing of Communicating Systems (IWTCS'98),* Tomsk, Russia, 1998. IFIP TC6. Amsterdam, Kluwer, 1998, 227–243.

24 West, C. Protocol validation. *Computer Networks and ISDN Systems,* 24, 1992.

Distributed Platform for Telecommunications Applications

ANASTASIUS GAVRAS

Anastasius Gavras (37) works for EURESCOM GmbH as Project Supervisor in the area of security middleware and management of networks and systems. After his studies he worked five years in the area of broadband networks and ATM at GMD-FOKUS in Berlin. In 1994 he joined Deutsche Telekom AG as an engineer and project leader in research and development. His technical work was focused on advanced distributed software architectures and technologies for telecommunication applications. In late 1996 and 1997 he was appointed to work in the international core team of the TINA Consortium. His final international experience before joining EURESCOM was at Sprint Inc. where he carried out work of common interest to Sprint and Deutsche Telekom in the area of middleware.

Gavras@eurescom.de

Recent advances in distributed computing are based on the paradigm of object orientation in which functionality is encapsulated in objects. Objects offer their functionality at their interfaces. In distributed object computing, it has become irrelevant on which computer the objects reside and the interactions between the objects can transparently cross physical computer boundaries, by utilising computer networks for communication. A set of services, collectively termed *middleware*, has been introduced to hide the complexity and heterogeneity of computers and computer networks. The Object Management Group (OMG) has developed the Common Object Request Broker Architecture (CORBA) specification, providing an open standard for the implementation of interoperable middleware products. The Distributed Processing Environment (DPE) is based on CORBA, and consolidates additional requirements that arise from the nature of large-scale telecommunications applications. The DPE recognises the increased need for flexibility in the architecture arising from the great diversity of systems found in a telecommunication environment. Furthermore it recognises the need for automated tools to support a number of common management tasks typical for large-scale systems. Finally the DPE acknowledges the value of the Reference Model for Open Distributed Processing (RM-ODP) as a guideline for dealing with the complexity of very large systems. The DPE uses the RM-ODP concepts.

1 Introduction

The intention to standardise *Open Architectures for Distributed Systems* was one of the most challenging undertakings in computer science. The need to establish a common understanding of the variety of aspects that characterise distributed systems, was the reason that brought together the leading standards bodies ISO and ITU-T to standardise the *Reference Model of Open Distributed Processing* (RM-ODP), which has been standardised in the X.900 series of ITU-T Recommendations. In itself the Reference Model is a conceptual framework for architectures for distributed systems and needs refinement to be applicable in a certain domain.

For this purpose a number of further standardisation activities were started, one of which is the work in ITU-T SG10 to define a distributed platform for telecommunication applications, namely the draft ITU Distributed Processing Environment Architecture (ITU DPE [1]). The DPE focuses on the aspects of the RM-ODP related to the run-time environment for telecommunication and information services and applications. The purpose of the DPE is to provide detailed technical requirements that should lead to specifications, both to help the DPE vendors to develop their products and the application developer to understand the infrastructure support that the DPE provides. Before detailing some of the most important aspects of the DPE this article elaborates on the RM-ODP, explaining some of the basic concepts, such as viewpoints, functions and transparencies. The main part of this article summarises the requirements and functionality for the DPE to support the execution of distributed telecommunications applications and discusses the main issues for the construction of a DPE.

2 The Reference Model for Open Distributed Processing

The elements of a distributed system (computers, networks, operating systems, etc.) may in general form a heterogeneous landscape, since they may be built by a diversity of hardware and software vendors. Co-operation is an inherent characteristic of distributed systems. RM-ODP enables co-operation in heterogeneous systems by imposing certain commonalties, which transform the heterogeneous system into an *open* system. In order to accommodate future technological advancements the Reference Model is completely independent of technology characteristics. Thus RM-ODP provides one of the most important properties of open systems, namely the separation of a system specification from its implementation. Furthermore, it provides the formalism for the specification of *server*- and *client*-components in an open distributed system.

2.1 ODP Documents

The RM-ODP standard consists of the following four documents:

- *Overview and Guide to Use* [2] introduces the Reference Model and provides an informal description of the concepts, such as the object model, the viewpoints, etc. This document explains the application of the Reference

Model for the definition of new ODP standards and architectures.

- *Descriptive model* [3] defines the concepts, which are needed to perform the modelling of ODP systems, and defines the principles of conformance to ODP systems.

- *Prescriptive model* [4] describes the required properties of open distributed systems. These properties must be fulfilled by all ODP compliant systems.

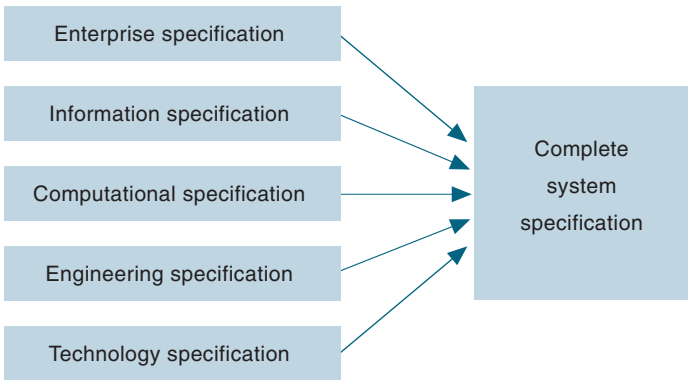
- *Architectural semantics* [5] provides the formal specification of some of the concepts of the descriptive model.

2.2 Concepts of the Reference Model

RM-ODP defines an object model, which can be used for the analysis of arbitrary distributed systems. RM-ODP defines an *object* as a model of an entity. An object is characterised by its behaviour and, dually, by its state. An object is distinct from any other object. An object is encapsulated, i.e. any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment. For a more detailed elaboration on the RM-ODP object model the reader is referred to [3].

A complete specification of a distributed system often contains more information than could fit into a single comprehensive description. In order to describe a complete system, the Reference Model uses a number of interrelated descriptions, namely the *viewpoints*, each describing a certain facet of the whole system. Each viewpoint describes the system from a certain angle and uses a set of rules specific to that angle, namely the *viewpoint language*. Related aspects of the system can be described in different viewpoints, which raise the issue of *consistency* among different viewpoint specifications. The Reference Model defines five viewpoints, namely *Enterprise, Information, Computational, Engineering* and *Technology*. See Figure 1.

Figure 1 RM-ODP viewpoints



- The *enterprise viewpoint* describes the terms and conditions that follow the business requirements under which a distributed system operates. The role of the user and its interactions with the system are described here. The used term *enterprise* does not imply that a system is constrained to be contained within a single company. The enterprise viewpoint language is suitable to describe also distributed systems spanning several organisations.

- The *information viewpoint* defines the semantics of information and the semantics of information processing in an open system in terms of a configuration of information objects, the behaviour of those objects and environment contracts for the system.

- The *computational viewpoint* describes a distributed system as a collection of objects (computational objects), which interact with each other according to the *client-server* principle. Here the interfaces of the computational objects are defined.

- In the *engineering viewpoint* the detailed distribution aspects of the system are exposed. These are defined as a collection of infrastructure-objects supporting the client-server communication of computational objects. Here the so-called *distribution transparencies* are described.

- The *technology viewpoint* describes the components and the used technologies for the implementation of the system.

The enterprise and information specifications do not expose the distribution of the system. The computational viewpoint defines computational objects and interrelations between them. The engineering and technology viewpoints specify the distribution of the components and the means for their implementation.

2.3 Transparencies and Functions

The programmer of distributed applications should ideally not need to care about the details of the communication between application components in a network. The programming environment that keeps the concerns about communication away from the programmer is called a *distributed platform*. ODP defines a number of functions that can be offered by the distributed platform and that can significantly improve overall system robustness and fault tolerance, and furthermore reduce the complexity of programming a distributed application. Some ODP functions are shown in Box 1.

Each ODP function shields the programmer from the distribution concerns and provides sim-

Box 1 Example ODP functions

- The *node management function* controls processing, storage and communication functions within a node. This function can be used to configure communication channels.
- The *group function* provides the necessary mechanisms to co-ordinate the interactions of objects in a multi-party binding. This function can for example facilitate concurrent information exchange with multiple receivers (e.g. multicast or broadcast).
- The *replication function* is a specialisation of the group function. All interfaces of a group offer the same service, establishing redundancies (e.g. replicas) for the purpose of constructing fault tolerant systems.
- The *migration function* co-ordinates the migration of objects from one (source) node to another (target) node. Migration is accomplished by establishing a replica object at the target node and activating it according to some predefined rule or condition. Thus, the migration function uses the replication function.
- The *relocation function* facilitates an uninterrupted client-server interaction during system changes. These system changes can for example be caused by communication domain management activities (change of node's network address) or other management activities such as migration or replication.
- The *type repository function* maintains a database of service type specifications and type relationships in a type hierarchy tree.
- The *trading function* mediates between service requests and service offers. A server can export its service, normally in the form of an interface identifier, into a database. A client can then query this database to receive the interface identifier for a particular needed service.

plifications for the construction of distributed systems. These simplifications are called ODP *transparencies*. In other words, ODP functions provide ODP transparencies. Some important transparencies are shown in Box 2.

3 Distributed Processing Environment

The Distributed Processing Environment (DPE) is the *distributed platform* that provides the ODP transparencies. One or more transparencies may be provided according to the requirements of the distributed telecommunications applications that shall be supported. The DPE can be viewed as the telecommunications middleware, which supports the execution of distributed telecommunication applications. The DPE is the infrastructure on which distributed telecommunications applications such as multimedia and real-time applications can execute.

Any distributed telecommunications application designed according to RM-ODP will benefit from using the DPE as a distributed platform. The DPE itself is based on the RM-ODP concepts and principles and is positioned at the engineering viewpoint. It uses the engineering functions as described in RM-ODP as a basis and, where appropriate, refines them or specialises them for the DPE. The DPE adopts

Box 2 Important ODP transparencies

- The *access transparency* is the most fundamental one, since it overcomes the hurdle of heterogeneity. It provides a common access mechanism for all objects of a distributed system. Its importance is obvious since it enables communications between objects running on different hardware systems.
- The replication function can be used to replace a failed object service by a running replica without the application noticing the replacement and not even noticing the failure of the original service. This behaviour is called *failure transparency*.
- When an object service is migrated to another node (e.g. for load balancing reasons) while the application remains unaware of this migration, this behaviour is called *migration transparency*.
- A migrated object is assigned new interface identifiers. The relocation function allows clients to still communicate with the migrated object as if a migration never happened. This behaviour is called *relocation transparency*.

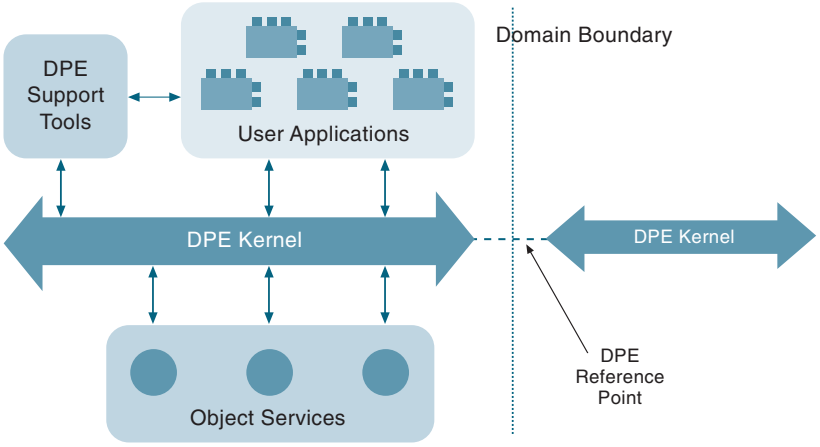


Figure 2 DPE Architecture

OMG CORBA [7] as the prime technology base, and thus the CORBA object services are incorporated where appropriate.

The DPE consists of a collection of DPE nodes that are interconnected. The DPE includes a DPE kernel, which provides support for object life cycle control and inter-object communication. Object life cycle control includes capabilities to create and delete objects during run-time. Inter-object communication provides the mechanisms to support the invocation of operations provided by operational interfaces of remote objects. The DPE kernel provides basic, and technology independent functions that represent the capabilities of most operating systems, i.e. the ability to execute applications and the ability to support the communication of applications with each other.

3.1 The Architecture

The DPE architecture as illustrated in Figure 2 is the unifying entity that encompasses all the required functionality, into a standardised implementation. It does this by presenting to applications a view of a well-structured heterogeneous distributed environment, where applications and services interact through standardised open interfaces. The architecture hides the heterogeneity of the underlying systems, e.g. programming languages, operating systems, computing systems and network protocols.

The DPE architecture provides the middleware for interaction between engineering computational objects (eCO[1)]) on remote DPE nodes. It also provides tools for diagnosis and configuration for use by system and network administrators. The components of the DPE Architecture are the *Kernel*, the *Object Services* and the *Support Tools*.

A DPE node is controlled by one DPE kernel. There is a DPE reference point between any two DPE nodes. Communication between distinct DPE nodes takes place over a DPE reference point.

OMG CORBA is a technology mapping of the DPE Architecture. Although the DPE architecture is based to a great extent on CORBA, it does not repeat CORBA functionality and describes the additional requirements that arise in the telecommunications domain. At the level of object services, many OMG adopted object services as specified in OMG CORBA Services [8] can be reused to build a DPE.

3.2 Computational to Engineering Mapping

Although RM-ODP describes the engineering modelling concepts, it does not provide a framework for the mapping of a computational specification, provided in ODL (*Object Definition Language* [6], see the paper on Object Definition Language by Born and Fischer in this issue) or IDL (*Interface Definition Language* [7]) onto the engineering modelling concepts. There is clearly a lack of methodology for supporting this mapping. Direct mapping of ODL and IDL specifications to a specific language (i.e. C++, Java, etc.) is available today ([6] and [7]) bypassing the engineering modelling concepts. The direct language mapping introduces a direct dependency of the computational specification on the target programming environment, so that at least the deployment of applications onto several nodes is depending on the capabilities of the target environment. To efficiently apply a complete object life cycle including deployment, creation/deletion activation/deactivation of objects, and to support additional functionality such as run-time object migration (for load balancing or high availability reasons) etc., an engineering methodology is required. For simplicity reasons we assume that there exists a methodology for mapping computational objects to engineering objects.

3.3 Communication in the DPE

A computational object interacts with other computational objects by invoking the computational operations they offer, or by exchanging stream flows with these other objects. *Operational interfaces* and *stream interfaces* are distinguished. The interactions that occur at an operational interface are structured in terms of invocations of one or more operations and responses to these invocations. Operations are classified into interrogations and announcements. Unlike an interaction via an interrogation, in an interaction via an announcement, no result is passed back

1) *The engineering representation of a computational object.*

from the server to the client, and the client is not informed of the outcome (success or failure) of the invocation.

Computational objects define *computational interfaces* as interaction points for other objects. The engineering interfaces of the engineering objects reflect the interfaces of the respective computational objects. Interfaces of engineering objects are described by engineering interface references that specify the information needed to uniquely identify an engineering location of an interface and to bind to this interface. Engineering interface references are capable of being passed across heterogeneous DPE nodes and are comparable for equality (for enabling the identification of interfaces after migration of an object instance for example).

A *stream interface* is an abstraction that represents a communication end-point that is a source, a sink or both a source/sink for information flows. When objects interact via stream interfaces, the information exchange occurs in the form of stream flows between the objects, where each stream flow is unidirectional and is a bit sequence with a certain frame structure (data format and coding) and quality of service parameters. A stream message consists of infinite-length information.

Binding is a contractual context, resulting from a given establishing behavior. In order for two objects to communicate via their interfaces the execution environment (i.e. the DPE) must provide the mechanisms to bind them. Bindings can be categorized according to the interface kind they bind, into *operational* or *stream* binding. Bindings can be further categorised according to the establishing behaviour. *Explicit* bindings result from the interactions of the objects that will take part in the binding, i.e. when the user makes use and controls explicit binding actions, and *implicit* bindings are performed by an external party, i.e. when the user does not express binding actions.

In the DPE implicit binding for computational interfaces is supported through the GIOP protocol that is a mandatory requirement for every DPE. In this case only the Interface Object Reference (IOR) is known to the client object that invokes operations on this interface without having control of the binding. How the implicit binding is implemented is extensively elaborated in the CORBA specification.

In order for two objects to interact by means of stream flows between them, each object has to offer a stream interface and the two interfaces must be bound.

The basic difference between interactions via operational interfaces and interactions via stream interfaces is that interactions via an operational interface are structured in terms of operation invocations and responses, whereas no such structure is imposed on interactions via stream interfaces. Once the stream interfaces of two objects have been bound, a set of *information flows* has been set up between the objects with some specific quality of service parameters. Thereafter, the producer of a flow inserts information into the flow, and the consumer retrieves information from the flow and consumes it. No explicit interaction occurs between the producer and the consumer during this information exchange. The control of these flows is achieved using operational interfaces. Eventually, either the producer, the consumer, or some third party object releases the binding between the producer and the consumer. Thus, the paradigm for interaction via stream interfaces is that of asynchronous message passing objects, while the paradigm for interaction via operational interfaces can be that of a remote procedure call, or that of asynchronous message passing.

4 Kernel Transport Network

The Kernel Transport Network (KTN) is the network that transports invocations and responses between different DPE nodes. Guidelines for the construction of a KTN are given, however no restriction is implied on the way the KTN actually maps onto the underlying network technology.

The transportation of the invocations between different DPE nodes is done via some standardised messaging protocol. In order to guarantee interoperability between DPE nodes a mandatory messaging protocol is specified. This protocol is the Generic Inter-ORB Protocol (GIOP) as specified in CORBA. In specific environments the use of GIOP is either not reasonable or not possible. In this case CORBA describes the approach for the specification of an Environment Specific Inter-ORB Protocol (ESIOP). The same approach is adopted for the DPE Architecture.

A number of different transport protocols may be used to transport GIOP messages. In order to guarantee interoperability between DPE nodes a GIOP mapping onto a mandatory transport protocol is specified. The mandatory transport protocol is TCP/IP and the mapping of GIOP onto it is call Internet Inter-ORB Protocol (IIOP) and is specified in CORBA.

There is a clear requirement for the DPE to support protocols other than IIOP for the KTN. Within the telecommunications domain, two of the most popular protocol stacks that require support are the SS7 protocol stack (i.e. the

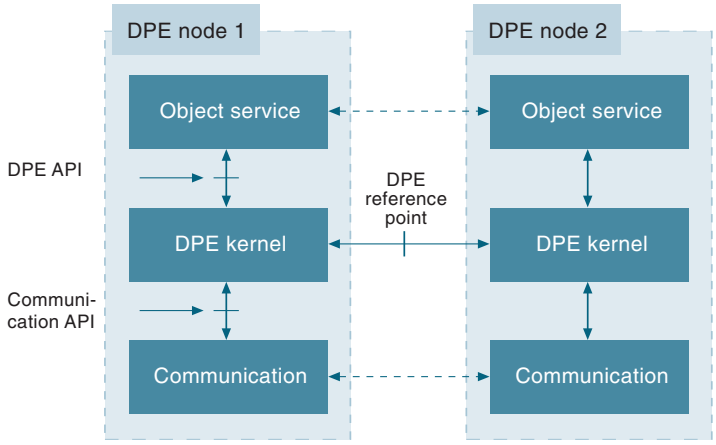


Figure 3 Interoperability framework

TCAP or SCCP layers) and the ATM protocols (i.e. the AAL5 protocol). To support these protocols and guarantee interoperability, GIOP mappings for protocols such as SS7 and ATM-AAL5 need to be specified.

The issue of interoperability over multiple protocols raises the broader problem of multiple protocol support in the DPE. The DPE must support the ability to communicate over multiple protocols, whereas the actual mechanism must be transparent to the application developer. Nevertheless, it must be possible for the application developer to set the parameters associated with a KTN connection. For example the application developer must be able to specify bandwidth requirements for a KTN connection if the underlying protocol supports this parameter.

5 DPE Interoperability Framework

Adopting the ODP viewpoints raises a number of issues with respect to interoperability mainly in the transition from the computational to the engineering viewpoint. At the computational level objects interact with each other without being concerned about how this interaction is implemented in a real system. The DPE, which is positioned at the engineering viewpoint, provides the mechanisms to enable the interactions specified at the computational viewpoint.

DPE interoperability specifies an approach to support the seamless interoperation of objects running on distinct DPE nodes. The approach is flexible in the sense that it allows several different combinations to support the specific needs of different environments. The basic idea of a homogeneous executing environment in which objects (services and applications) execute is maintained.

The interoperability framework for the DPE is based on the interoperability approach taken in CORBA and enriches it with a solution for interoperability in a heterogeneous environment. When observing two distinct DPE nodes that need to inter-operate, three different levels of interoperation can be identified (see also Figure 3):

- **Object service**: When service interfaces are specified in IDL or ODL no information is available about the way peer objects perceive and process information. While the semantics of basic services (i.e. common object services) is generally well understood, the application designer only knows the internal semantics of specialised applications. At this level the interoperability concern lies in the internal semantics of the services or applications. This level of interoperability is outside the scope of the DPE.

- **DPE kernel**: At this level service requests and service replies are transported between peer objects by mechanisms provided by the DPE. The details of these mechanisms involve binding of the object interfaces (implicitly or explicitly) through functions provided by the DPE Application Programming Interface (API) (see Figure 4) and the DPE reference point, and data transfer involving the conversion of operation parameters and operation results into a common format for transmission.

- **Communication**: This level is responsible for the interoperability of the transport protocols, which are used to transmit the service requests and service replies at the DPE level. The communication level is also out of scope of the DPE.

For the DPE a clear separation between the network dependent parts of the DPE and network independent parts is required. This separation allows developing an adaptation to a networking protocol independent of the core of the DPE. Through this adaptation the development of networking protocols for inter-DPE communication becomes de-coupled from the rest of the DPE. The mechanism of plug-able protocols enables the integration of the DPE core and network adaptation modules of different vendors. More specifically it allows the use of IIOP as a plug-able transport protocol. In the case of message protocol like TCAP the message protocol actually replaces the CORBA GIOP message layer according to the CORBA ESIOP approach.

Box 3 DPE kernel engineering services

- *Flexible DPE architecture* allowing the introduction of new binding and communications mechanisms and the ability to incrementally add transparency services (security, transaction, persistence, migration, etc.).
- *Multi-protocol support* allowing the introduction and simultaneous execution of multiple communication protocol stacks.
- *Generic communication scheme* to allow the exploitation of different communication resource multiplexing policies and the construction of any protocol.
- *Flexible binding* to provide support for any binding between objects.
- *Support for stream interfaces*, including the definition of strongly typed stream interfaces, the ability to bind together multiple stream interfaces and application-level processing of streams.
- *Multithreading support*.
- A *flexible event-to-thread mapping*.
- Support for *memory management*.
- Support for native *monitoring functionality*.
- Support for *interceptors* (e.g. filters) to allow for monitoring of events.
- *Generic scheduling schemes*, supporting both priority and deadline-based scheduling policies.
- *Time service*.
- Support for different levels of *object granularity* in both space (memory size) and time (object lifetime and duration).
- Small *memory footprint* (if necessary for deployment in embedded systems).
- *Documented time behaviour*.
- Support for computational objects with *multiple interfaces* (ODL).

6 Engineering Services Provided by the DPE Kernel

Box 3 lists the engineering features supported by engineering services that a DPE kernel may provide. A detailed description of the kernel services is omitted here due to space constrains.

Not all kernel services are required for all applications. Profiles that support different kinds of services and applications must be defined. These profiles need to specify which kernel services are mandatory for a given profile.

7 Object Services

This chapter describes object services needed to support the execution of telecommunication applications. Some of these object services are generic enough in nature to be considered as general purpose computing services, while other object services are specific to the telecommunications domain.

The following object services are considered as DPE object services:

- *Life Cycle Service:* The life cycle service provides functions for creating, deleting, copying, and moving individual objects or collections of objects. It also provides capabilities for deactivation, reactivation, replication, recovery, and migration.

- *Naming Service:* The naming service is a fundamental DPE Service. Its objective is the localisation of interfaces, especially of object implementations. The naming service provides a mapping between a human readable name and an interface reference.

- *Trading Service:* The trading service supports late binding between two objects, the exporter and the importer. To do so, it administers information about service offers, the associated interface references and service attributes. The exporter offers its services (interfaces); the importer seeks services and uses the trading service in order to get hold of them.

- *Security Service:* The security service manages confidentiality, integrity, accountability, and availability within the DPE. The security service counteracts threats of disclosure, deception, disruption, and usurpation of telecommunication data and services.

- *Notification Service:* The notification service enables objects to emit or receive notifications without being aware of the set of objects with which they are communicating. Similarly, it enables objects to receive notifications without having to interact with emitter objects. The service acts as a broker between emitters and recipients.

- *Transaction Service:* The transaction service provides transactional communication between objects guaranteeing consistency of applications with properties collectively referred to as ACID properties: atomicity, consistency, isolation, and durability. Open nested transactions are provided for real-time applications.

- *Concurrency Control Service:* Concurrency is a paramount concern in a real-time system, where a trade-off exists between high availability on the one hand and application consistency on the other. A Concurrency Control Service enables multiple threads to co-ordinate their access to shared objects. When many concurrent threads access a shared object, any conflicting operations by the threads are reconciled so as to preserve the consistency of the object state.

- *Persistence Service:* The persistence service allows for the management of the persistent state of objects. It ensures the integrity of the data of a given database by ensuring the consistency of the objects.

- *Interrogation Service:* The interrogation service provides interrogation operations on collections of objects. It can be used to return collections of objects that are either selected from a source query-able collection or produced by a query evaluator. Interrogations are specified using a query language and may perform general manipulation operations such as selection, updating, insertion and deletion on collections of objects.

- *Messaging Service:* In order to achieve their aims of high performance, scalability and throughput telecommunications applications may use asynchronous communication in combination with an event based programming model. The messaging service satisfies the requirements for a truly asynchronous method invocation model.

- *Migration Service:* A migration service as part of the DPE object life cycle service requires realisation of two distribution transparencies:
  - Relocation transparency masks a migration of an object from other objects bound to it;
  - Migration transparency masks location changes from the object being relocated.

- *Licensing Service:* The licensing service enables accounting of access and use of telecommunication services and software applications.

- *Event Logging:* For the purpose of managing any system, it is necessary to have the ability to trace the activity history of the system. A service for making event notifications persistent and storing them (logging) is a basic requirement.

- *Topology:* The Topology service provides a general service for managing the topological relationships (associations) between distributed objects. Its purpose is to relieve application objects from the burden of managing associations by providing a service for storing topological information independent of a specific application.

- *Software Distribution/Installation:* The Software Distribution/Installation service provides run-time support when software needs to be distributed and/or installed on a large number of nodes. For example when deploying new telecommunications services, support by the DPE is needed in order to properly co-ordinate the deployment.

- *Software Configuration Management:* The Software Configuration Management service provides the functionality to configure various services running over the DPE, in the same way as management functionality is used in telecommunication networks to set up and modify parameters of the physical equipment. When such functionality is added to the DPE, the users can manage both software and hardware entities in a seamless fashion. This service may be integrated with the Integrated Management Service (see below).

- *Integrated Management:* The Integrated Management Service targets the integration of user application management, platform management and network management.

- *Control & Management of Audio/Video Streams:* The Control and Management of Audio/Video Streams Object Service adopted the OMG [9] specifies in detail this service.

8 A Final Word

The open services market is a vision that is rapidly emerging and that will have a major impact on the business of telecommunication operators. Several business roles are expected to operate in this market. Crucial for the success of this market is the availability of distributed platforms and open interfaces between the parties. Standard open interfaces will enable telecommunications operators to provide services to a larger group of customers and also to provide a larger set of services to their customers. The open service market is enabled by distributed object technology and is founded on distributed platforms such as the DPE. State-of-the-art object middleware, such as defined by OMG is an enabling technology for the open services market. Several projects (prototypes and real world deployments) demonstrate already that distributed platforms are ready to accept the challenges of a new way of service provisioning. The telecommunications operators and the middleware industry are working diligently to further enhance the standards and products of distributed platforms.

9 References

1 ITU-T. *ITU-T COM10-R4 Report of the third meeting of the Study Period.* Geneva, 1999.

2 ITU-T. *Information Technology – Open distributed processing – Reference Model: Overview.* Geneva, 1997. (Recommendation X.901 (08/97).)

3 ITU-T. *Information Technology – Open distributed processing – Reference Model: Foundations.* Geneva, 1995. (Recommendation X.902 (11/95).)

4 ITU-T. *Information Technology – Open distributed processing – Reference Model: Architecture*. Geneva, 1995. (Recommendation X.903 (11/95).)

5 ITU-T. *Information Technology – Open distributed processing – Reference Model: Architectural semantics*. Geneva, 1997. (Recommendation X.904 (12/97).)

6 ITU-T. *ITU Object Definition Language.* Geneva, 1999. (Recommendation Z.130 (02/99.)

7 Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.4.* October 2000. (Available electronically from http://www.omg.org)

8 Object Management Group. *CORBA Services, 1991–2000.* (Available electronically from http://www.omg.org)

9 Object Management Group. *Audio/Video Stream Specification, New Edition*. January 2000. (OMG Document Number formal/00-01-03.) (Available electronically from http://www.omg.org)

Formal Semantics of Specification Languages

ANDREAS PRINZ

Andreas Prinz (37) studied mathematics and computer science at the Humboldt-University in Berlin and received his MSc in mathematics (1988) and PhD in computer science (1990) there. From 1990 until 1993 he was a post-doctoral fellow at the Humboldt-University. From 1993 to 1994 he worked at the Software Verification Research Centre (SVRC) in Brisbane, Australia before returning to the Humboldt-University in 1994. Since 1997 he has been working at the Berlin-based company, DResearch GmbH. His research interests include formal methods together with their application and use in tools, so he also has a strong interest in software technology and compiler construction. Dr. Prinz has worked in several projects dealing with the development of modern telecommunication systems using advanced technology.

prinz@informatik.hu-berlin.de

While many specification languages today have a formally defined syntax, they often lack a formally defined semantics. The ITU languages have both. This paper provides motivation for defining both the syntax and the semantics, and explains how they are provided.

Introduction

Formal languages play an important role in mathematics and in computer science. In contrast to natural languages, formal languages have two important properties. Firstly, formal languages always make clear whether a particular sentence (in mathematics also called "formula", in computer science called "program" or "specification") belongs to the language or not. This property is usually called the *syntax* of the language. Secondly, the meaning of the valid sentences is clear. This is called the *semantics* of the language.

Nowadays the description of a specification language typically contains a formal syntax, but only an informal semantics description. However, it is important also to state the semantics formally in order to know the implications of statements in that language.

The specification languages SDL and MSC, from the International Telecommunication Union (ITU), benefit from being formally defined. This article explains the benefits and how formal definitions are provided. Box 1 provides a simple Glossary of the main terms used in this paper.

Motivation

There is a need to provide formal definitions of specification languages. This need comes from a desire to have better possibilities to check properties of specifications as well as to provide better means to check the correctness of tools supporting the specification language.

The idea is simple: If the formal semantics of a specification language is given without referencing any implementation details, one can then check a concrete implementation against this description for correctness. The implementation details can be chosen by the implementation and are not prescribed by the specification. If a formal definition of the specification language semantics is provided, you can derive properties of the system without even implementing it.

In the following, we survey some areas where formal methods are beneficial.

Box 1: Glossary

Syntax: The syntax of a specification language is the set of constraints on the formulation of specifications using the language.

Semantics: The semantics of a specification is the set of the constraints it describes. The semantics of a specification language is the semantics of all its syntactically valid specifications.

Grammar: A grammar is a method of syntax formalisation.

Formal semantics: The semantics of a specification language is called formal if it is defined by means of mathematics.

Specification: A specification is a set of constraints on a system. An SDL specification is a set of functions provided by a set of communicating processes.

Implementation: An implementation (within a certain resource context) is an algorithm which satisfies the specification.

Formal Grammars

Formal grammars have proved most useful to formally define syntactical structures. The Backus-Naur Form (BNF) is an example of a formal grammar. BNF consists of a set of production rules that recursively define a set of valid character sequences. Backus and Naur originally developed the BNF grammar for the prescription of the syntax of the ALGOL 60 programming language.

Box 2 shows a characterisation of different formal grammar types.

Please find below the grammar for simple arithmetic expressions.

```
T={"0", "1", "+", "-", "*"}
N={expr}
s=expr
R={<expr,"0">, <expr,"1">,
        <expr, expr "+" expr>,
        <expr, expr "*" expr>,
        <expr, "-" expr> }
```

Box 2: Grammars

Four parts give a grammar: a set of *terminal* symbols T, a set of *non-terminal* symbols N, a *start symbol* s ∈ N and a set of *grammar rules* R. Each grammar rule has a left-hand side LHS and a right-hand side RHS which is depicted as LHS ::= RHS. Both LHS and RHS are sequences of terminals and/or non-terminals, i.e. LHS ∈ $(T \cup N)^*$, RHS ∈ $(T \cup N)^*$.

The linguist Chomsky developed a hierarchy of four types of grammar. More restrictions on the grammar rules mean simpler grammars and less syntax expressible. However, simpler grammars are easier to analyse, see the table below.

type	grammar	sample production	restrictions on grammar rule
0	unrestricted	$t_1N_1t_2t_3N_3 ::= t_3N_4N_5t_1t_2$	none
1	context sensitive	$t_1N_1t_2N_3 ::= t_3N_4N_5t_1t_2$	RHS is longer than LHS
2	context free	$N_1 ::= t_3N_4N_5t_1t_2$	LHS is one non-terminal
3	regular	$N_1 ::= N_5t_1$	LHS is one non-terminal, at most one non-terminal on RHS, at most one terminal on RHS

Most often a context free grammar formalism (e.g. BNF) is used to define the syntax of a specification language.

The set R could be expressed by the following BNF rule.

```
expr ::= "0" | "1"
       | expr "+" expr
       | expr "*" expr
       | "-" expr
```

The success of formal grammars lead to the current situation that specification languages have a formally defined syntax, most often based on some variant of BNF. The language grammar is used by tool developers to build tools and by language users to understand language constructs.

Formal Semantics

The definition of the semantics of some specification languages (e.g. UML) is given in ordinary prose. In order to define the syntax of a language construct, a production from a BNF or other formal grammar is provided. Added to this formal syntax is a few paragraphs and (hopefully) a number of examples to define the semantics. Unfortunately it has been the case that the meaning of the prose is ambiguous, leading to different interpretations of the semantics of a language construct. This may affect both users of the language and tool developers. Firstly, a language user may misunderstand the specification, and also tool developers may implement a specification construct in a manner different from other tool developers of the same specification language. Hence, as with syntax, methods are required to provide a precise, readable and concise definition of the semantics of a specification language.

Language Design

When the semantics of a specification language is defined formally, some interesting questions can be asked:

- What relations exist between language constructs?
- Can some language constructs be derived from other language constructs?
- Can combined use of language constructs cause problems?

Although many of these questions were asked without a formal semantics, it is now possible to examine them by formal means. Moreover, already the formalisation process will uncover omissions and inconsistencies in the language definition.

Checking the Specification

Formal semantics can be used to mathematically check properties of the specification. This way, every possible behaviour of the specification can be covered and not only those that are considered during tests or during use. The following questions can be addressed:

- Does the specification contain a deadlock?
- Does the specification contain an infinite loop (livelock)?
- Will the specified algorithm always terminate?
- How long will it take to compute the result?

In order to formulate such questions formally, it is necessary to have an appropriate mathematical description of the language (and thus also for the individual specification).

Please note, that some of the questions above are in fact not decidable, i.e. no algorithm can automatically check such a property for any specification.

Type Safety

A formal definition of the semantics of a language allows to define typing. A correctly typed specification provides constraints on language usage, e.g.: In this specification, all data will behave such that no data flows into a place that is not capable of holding it.

Typing is a property that is statically decidable, i.e. it is decidable at compilation time. However, typing states constraints that are satisfied by the specification when interpreted. Most modern specification languages introduce polymorphic typing. Using a formalisation it is possible to prove the correctness of the typing rules, i.e. that static type correctness implies dynamic type correctness.

Semantics Definition Styles

The problem of language semantics definition has been a research topic for a considerable period. However, unlike the area of syntactical definition, satisfactory solutions have been rare. Although semantics definitions for mathematical languages are well-known, defining the semantics for specification languages turned out to be more difficult. Specification languages are often larger than the mathematics languages, they have lots of special cases, and they have dynamic semantics, i.e. the meaning of a construct depends on the state of the whole system.

Many different methods of formal definitions have been developed, and these may be subdivided into three general classes:

- operational techniques;
- denotational or functional techniques; and
- axiomatic techniques.

Unfortunately, no single method is appropriate for both users and tool developers. Sometimes a semantics does not follow purely one of the above styles, but is in fact a mixture of them. In these cases it is often valuable to identify the parts that are covered by a certain style, in order to gain a better overview of the semantics. In the following, the different styles are explained in more depth using the simplified arithmetic expressions as defined in the formal grammar section as an example.

Axiomatic Semantics

Axiomatic techniques for specification language semantics were derived from mathematical logic, logical equations and *model theory* out of a desire to perform program correctness proofs. The entities of the language and their relations to each other are identified. For the example we have the entities expr, “0”, “1”, “+”, “-“, “*” as indicated by the following declarations.

domain expr
expr is the only domain set

constants “0”, “1”: expr
0 and 1 denote elements of the domain expr

functions “+”: expr × expr → expr
this defines the signature, i.e. type of the addition function

“*”: expr × expr → expr
this defines the signature of the multiplication function

“–“: expr → expr
this defines the signature of the minus sign

Their relations are captured with the following axioms.

axioms for all x,y,z: expr

1.	x	≠	1+x
2.	0+x	=	x
3.	x+y	=	y+x
4.	x+(y+z)	=	(x+y)+z
5.	1*x	=	x
6.	x*y	=	y*x
7.	x*(y*z)	=	(x*y)*z
8.	x*(y+z)	=	(x*y) + (x*z)
9.	x+(–x)	=	0
10.	–(x+y)	=	–x + –y
11.	–(x*y)	=	–x * y

From the axioms above we can for example derive that 0*x=0 as follows.

0*x	= (a+(-a))*x	(by 9)
	= x*(a+(-a))	(by 6)
	= (x*a) + (x*(-a))	(by 8)
	= (x*a) + ((–a)*x)	(by 6)
	= (x*a) + (–(a*x))	(by 11)
	= (x*a) + (–(x*a))	(by 6)
	= 0	(by 9)

We can also derive that $1 \neq 0$, because $1 = 1+0 \neq 0$.

The benefits of the axiomatic method are the following:

- It provides a very abstract semantics definition;

- There is no impact on an implementation;
- Mathematical methods (proofs, model checking) can easily be used;
- The axioms are concise and understandable.

But there are also problems:

- No or only little guidance to implementers is provided;
- The description tends to be very large when many basic constructs are considered;
- It is very complex for real languages;
- It is difficult to formalise operations and states;
- There is no easy overview of the implications of the definition.

The sample language illustrates the implications problem. The intention of the definitions above was to provide a description of the integers. So we would like to conclude $0 \neq 1 + 1$. However, this is not implied. It would be perfectly valid if the domain expr contained only the elements 0 and 1 with $1 + 1 = 0$. All axioms hold in this case.

To avoid writing too many axioms, the axiomatic semantics is often restricted to only so-called initial models. This means that we only want to consider the most general models matching the definition. This is often stated with two conditions: *no junk* and *no confusion*. 'No junk' means that we do not want elements in a domain that are not really implied by the language definition, e.g. we do not want to have π in the domain expr. 'No confusion' states that we do not want to regard elements to be the same unless explicitly stated. In the above example, this would mean that we have an implicit axiom $1 + 1 \neq 0$.

Denotational Semantics

The basic idea is to give a denotation to every element of the language. This means to map the syntactical expressions of the language to a well-known domain. For the sample language we define a mapping from the language entities to the integers. The denotation function is often called [_] as in the definitions below.

```
[expr]      =  integer
["0"]       =  0
["1"]       =  1
[x "+" y]   =  [x] + [y]
[x "*" y]   =  [x] * [y]
["–" x]     =  – [x]
```

Please note that there is always an implicit axiomatic semantics hidden in this approach, in the example the semantics of integer, which is considered to be predefined in ordinary mathematics. The general concept of the denotational semantics is to map the unknown language to a known language. This basic known language should itself have a formal semantics given with one of the three styles.

The benefits of the denotational semantics are the following:

- It resembles the syntax structure;
- It builds on known domains;
- The semantics description is fairly abstract.

However, there are also problems:

- It provides only little guidance to tool developers;
- It is usually too complex for users;
- For complex languages, the target domains are not readily available;
- There are again difficulties to express states and operations.

The denotational approach works better the easier the mapping is. In the example we see that the mapping is one-to-one and therefore it is easy to read. However, it is much more difficult to map a real language such as C to basic mathematical domains. In such a mapping, various auxiliary functions must be introduced, and one element of the source language is mapped to many elements of the target language. Therefore, a common approach is to first define a specialised target language axiomatically, and then to give a denotational semantics based on this special language.

Operational Semantics

The operational approach is the most concrete one, and it is very near to implementation. The idea is to interpret the specification in an abstract interpreter. The abstract interpreter is a program of an abstract computer (e.g. a Turing Machine). The operational semantics of the sample language in the previous sections is given below using an abstract Pascal style for the interpreter program:

```
procedure compute(e: expr) returns integer is
  case e of
    "1": return 1;
    "0": return 0;
    "+": return compute(e.first) + compute(e.second)
```

```
    "*": return compute(e.first) * compute(e.second)
    "–": return – compute(e.first)
  endcase
endprocedure
```

The operational method is also using a predefined semantics, namely the semantics of the abstract computer. In fact, this abstract computer semantics may again be given using any of the three semantics definition styles. However, the semantics of the abstract computer need not be complete, because only one program – namely the interpreter program – is interpreted. It suffices if the machine can handle this single program. Using the operational method, one could even define the semantics of a language in (a restricted version of) itself using some kind of bootstrapping.

There are the following benefits of the operational method:

- It provides a good formalisation of implementation;
- It is easily understandable for tool developers;
- It is well suited for state-based languages.

Again, we have some problems:

- Operational descriptions tend to be very detailed;
- It is very difficult to derive formal proofs from an operational semantics;
- The operational approach needs an underlying semantics of an abstract computer.

A similar remark as for the denotational semantics is in place here. The operational approach is easier to understand when the underlying abstract computer matches the paradigm of the source language. Therefore, it is quite common to first build a special abstract computer which is tailored to the source language to provide an easy interpretation.

Formal Semantics for SDL: Overview

Static Semantics

The static formal language definition consists of the following parts as shown in Figure 1:

- a concrete syntax;
- a set of well-formedness conditions;
- a set of transformation rules; and
- an abstract syntax as basis for the dynamic semantics.

The *syntax* defines the set of syntactically correct SDL specifications. For SDL we distinguish between a concrete and an abstract syntax. The concrete textual syntax (SDL-PR), is formally defined in BNF. Some extensions are used to capture the concrete graphical syntax (SDL-GR). Both the concrete and abstract grammar each prescribe a syntax tree. Each specification written in SDL is considered to make up a syntax tree, both for the concrete and abstract syntax. The abstract syntax is obtained from the concrete syntax by removing irrelevant details such as separators and lexical rules. Moreover, shorthand notations are not represented within the abstract syntax. They are replaced by their corresponding basic constructs (see also transformations below).

The *well-formedness* conditions define additional conditions that must be satisfied by a well-formed SDL specification and that can be checked without interpreting an instance. An SDL specification is valid if and only if it satisfies the syntactic rules and the static conditions of SDL. In fact, the static conditions refer to the syntax, but for conciseness reasons they have not been stated in the concrete syntax because they are not expressible in a context free grammar.

There are basically five kinds of well-formedness conditions:

1. *Scope/visibility rules:* The definition of an entity introduces an identifier that may be used as the reference to the entity. Only visible identifiers may be used. The scope/visibility rules are applied to determine whether the corresponding definition of an identifier is visible or not.

2. *Disambiguation rules:* Sometimes a name might refer to several definitions. Rules are applied to find the correct identifier.

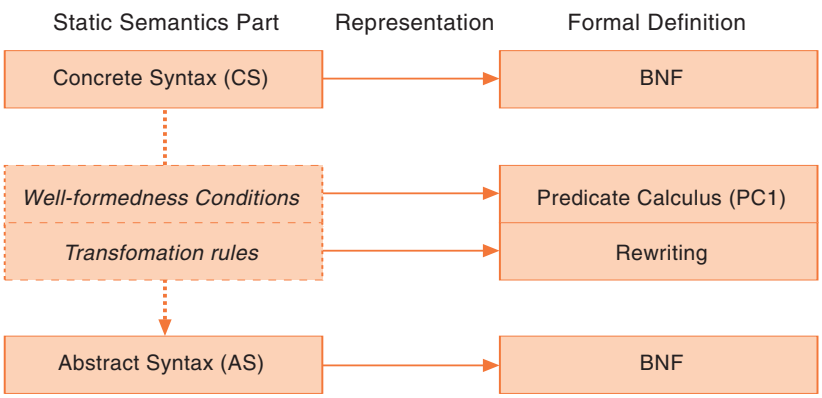


Figure 1 Static Semantics Overview of SDL

3. *Data type consistency rules:* These rules guarantee that at interpretation time no operation is applied to operands that do not match their argument types. More specifically, the data type of an actual parameter must be compatible with that of the corresponding formal parameter; the data type of an expression must be compatible with that of the variable to which the expression is assigned.

4. *Special rules:* There are some rules applicable to specific entities. For example, an agent must contain local agents and/or a (composite) state.

5. *Concrete syntax rules:* There are some rules that refer to the correctness of the concrete syntax and that do not get transformed into the abstract syntax, e.g. the name at the beginning and at the end of a definition match.

The well-formedness conditions of SDL are formalised by first order predicate calculus (PC1).

Furthermore, some language constructs appearing in the concrete syntax are replaced by other language elements in the concrete syntax using *transformation rules* to keep the set of semantic core concepts small. For instance the SDL remote procedure notion is a shorthand notation for sending a request signal and receiving a reply signal. The transformations are given in the language description. Formally they are represented as rewrite rules. A single transformation is realised by the application of a rewrite rule to the concrete specification, which essentially means to replace parts of the specification by new parts as defined by the rule.

It is important to identify appropriate core notions matching the intuitions behind the language design in order to facilitate easy transformation. If there are too many notions, the semantics will be unnecessarily complicated. If there are too few or inappropriate notions, the transformations tend to become very complex and their meaning is no longer easily understood.

Dynamic Semantics

The *dynamic semantics* is given only to syntactically correct SDL specifications that satisfy the well-formedness conditions. The dynamic semantics defines the behaviour associated with a specification.

The dynamic semantics starts with the result of the static description, i.e. with the *abstract syntax*. In order to better show the structure of the dynamic description, we identify three parts of the abstract syntax, namely *structure*, *behaviour* and *data*. The dynamic semantics is based on a mathematical theory called Abstract State Machines (ASM). Each SDL specification is associated with a particular multi-agent, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents co-operatively performing concurrent machine runs. The behaviour of agents is determined by ASM programs, each consisting of a transition rule, which defines the set of possible runs. Each agent has its own partial view on a global state, which is defined by a set of functions and a set of domains. By having non-empty intersections of partial views, interaction among agents can be defined.

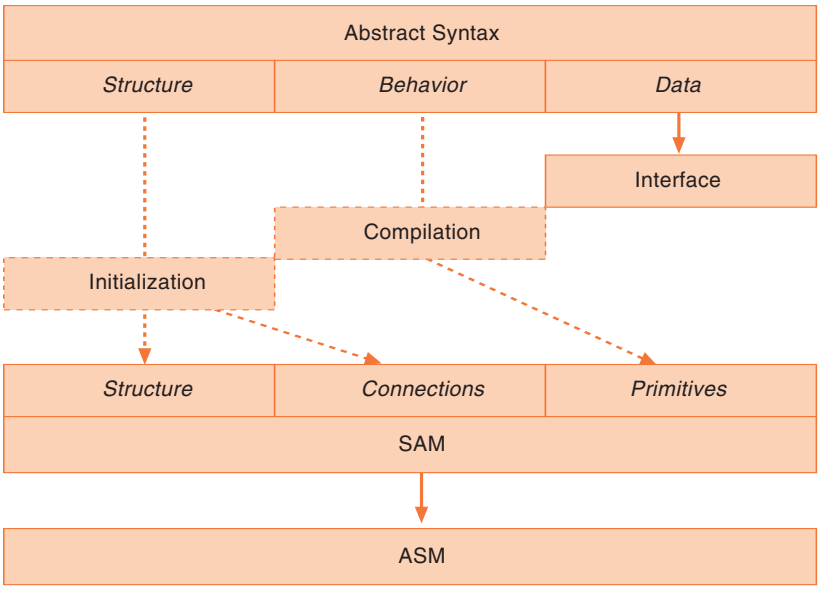


Figure 2 Structure of the Dynamic Semantics

All parts between the abstract syntax and ASM belong to the dynamic description. There are four parts, as can be seen in Figure 2.

- An *SDL Abstract Machine (SAM)* is defined using ASM. For better match with the abstract syntax, we identify three parts of the SAM, namely (1) basic features to express *structural* properties, (2) *connections* (SDL channels and other SDL connections) and (3) *behaviour primitives*. The last feature can be considered as the instruction set of the abstract machine.

- An *initialisation* is necessary to handle static structural properties of the specification. The initialisation provides a recursive unfolding of all the static objects of the specification. The same process will be initiated at interpretation time when new SDL agents are created. Therefore, the initialisation is merely the instantiation of the outermost SDL agent.

- A *compilation* function that maps behaviour representations into the SAM primitives. This function amounts to an abstract compiler taking the abstract syntax of the state machines

Box 3: History

This box provides an overview of the history of the formal semantics of SDL and MSC.

Year	SDL	MSC
1988	First formal semantics for SDL: The static semantics is an axiomatic description of the conditions and the result of the transformations; the dynamic semantics is operational using Meta IV and a variant of CCS.	
1992	Due to requests from tool builders, the static semantics, especially the transformations were described operationally. The dynamic semantics approach was not changed.	First formal semantics of MSC: The static semantics is an axiomatic description of the conditions; The dynamic semantics is denotational for the mapping and axiomatic for the process algebra. An operational semantics for the process algebra is also provided for courtesy.
1996	The language changes were very small, only minor changes to the semantics were necessary.	Changes in the MSC language lead to the disposal of the axiomatic semantics. The process algebra is formalised operationally.
2000	The old formal semantics approach could not be pursued further due to problems to adequately express timing aspects and problems with the size of the formal description. A new semantics as presented here is developed.	No new formal semantics for MSC-2000 as yet.

as input and transforming it into the abstract machine instructions (see SAM).

- A *data* semantics, which is separated from the rest of the semantics by a data interface. The use of an interface is intentional at this place. It will allow exchange of the data model, if for some domain another data model is more appropriate than the built-in model. Moreover, the built-in model can be changed this way without affecting the rest of the semantics.

The new formal semantics of SDL is defined starting from the abstract syntax of SDL, which is documented in Z.100. A behaviour definition corresponding to this abstract syntax is defined. The approach chosen uses all three basic semantics definition styles. The SAM is an extension of ASM and both SAM and ASM have a mathematical definition (axiomatic semantics). The compilation defines an abstract compiler mapping the behaviour parts of SDL to abstract code in ASM programs (denotational semantics). Finally, the initialisation describes an interpretation of the abstract syntax tree to build the initial system structure (operational semantics).

Abstract State Machines – ASM

The dynamic semantics associates a particular multi-agent real-time ASM with each SDL specification. The specification is represented as an abstract syntax tree. The ASM consists of a dynamically growing and shrinking set of *autonomous agents* that perform *concurrent machine runs*. The behaviour of ASM agents is determined by *ASM programs*, each consisting of some finite collection of *transition rules* which defines the set of possible machine runs. Each ASM agent has its own *partial view* on a given global state, on which it fires the rules of its program. According to this view, ASM agents have a *local state* and a *shared state*, through which they can interact.

We point out that there are strong structural similarities between SDL systems and multi-agent ASMs, which is a prerequisite for intelligibility and maintainability. Conceptually, these structural similarities are utilised by identifying ASM agents with active SDL objects, as follows. Various classes of ASM agents (with distinguished behavioural properties) will be introduced for modelling the behaviour of SDL agents, SDL agent sets, and SDL channels. ASM agents are either created during the system initialisation phase or dynamically according to dynamic process creation in the underlying model.

The execution of a system starts from a pre-initial system state with the creation of a single ASM agent for the SDL unit “system”. Performing a stepwise unfolding of the system under consideration, this distinguished agent then creates further ASM agents according to the system substructure and associates a view and an ASM program with each of them. ASM programs are determined by the kind of the SDL unit modelled by a given ASM agent. This way, starting the system amounts to instantiating the SDL unit “system”, just like it would have happened during system execution.

Once an initial abstract machine configuration reflecting the initial system structure has been built up, the actual system execution phase starts. Similar to the system initialisation, the behaviour of ASM agents during system execution is determined by corresponding ASM programs. During a run, further ASM agents may be created, and existing ASM agents be terminated, according to the transitions of the SDL agents.

Tools, Executability, Implementation

When the new formal SDL semantics was designed, we were taking the needs for implementing the semantics into account. These needs are easily understandable when you realise that the complete formal semantics definition will span more than 200 pages. Despite all efforts to have a good structuring and an understandable presentation of the semantics, it will be hard to gain a complete overview of the semantics. Therefore, the formal semantics is automatically transformed into an SDL-to-ASM compiler. To make the implementation possible, it was planned to use existing tools and to use meta tools as much as possible. For this reason, yacc and lex and kimwitu were used to implement the formal semantics.

The following characteristics make an implementation possible:

- ASM is an operational technique: The basic state change primitive defined in ASM is an update, which is basically the same as an assignment. All the other constructors define just finite sets of updates.

- ASM is supported by tools: There are freely available implementations of ASM. We use the ASM workbench of Paderborn with the ASM dialect ASM-SL [13]. A close contact with the developers of the ASM workbench ensures that the SDL semantics can be processed by this tool.

- A special style of using ASM is applied for the definition of the SDL semantics: Although ASMs are executable in principle, there are some instructions that could cause problems. In the SDL semantics all set constructions are computable, i.e. they use only computable functions over finite domains and not the general notion of sets supported by First Order Predicate Calculus.

Formal Semantics for MSC: Overview

Static Semantics

The structure of the MSC static semantics is similar to the SDL static semantics. There is also a distinction between a concrete grammar and an abstract grammar. Moreover, there are static well-formedness conditions that are formalised using predicate calculus formulae (PC1). However, MSC does not contain as many constructs as SDL, so it is possible to give the semantics without transformations. This is possible because the mapping between the MSC concrete syntax and abstract syntax is almost one-to-one, such that it need not be formalised. See Figure 3 for an overview of the MSC static semantics.

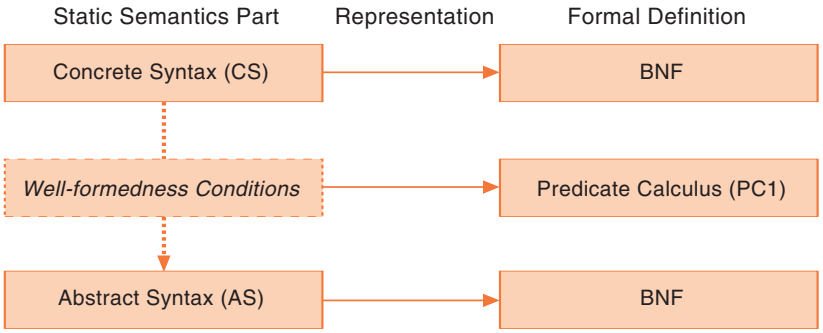


Figure 3 Static Semantics Overview of MSC

Dynamic Semantics

A dynamic semantics is given only to syntactically correct MSC specifications that satisfy the well-formedness conditions. The dynamic semantics defines the behaviour associated with a specification.

Each MSC specification is mapped to a particular process-algebraic term. The process algebra used to represent an MSC specification is tailored to the language constructs of MSC. The mapping function amounts to an abstract compiler taking the MSC syntax as input and producing a process algebraic term. See also the SDL compilation function for comparison. According to the semantics classification scheme, the mapping amounts to a denotational semantics.

The second part of the MSC semantics is the formal semantics definition for the MSC process algebra. As indicated in Figure 5, there are two ways of giving this semantics: an axiomatic semantics or an operational semantics.

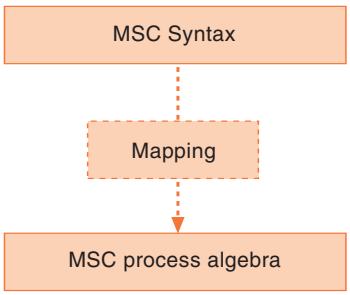


Figure 4 Structure of the MSC Dynamic Semantics: Part 1

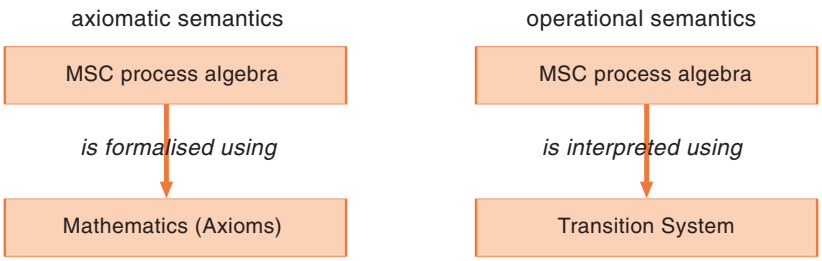


Figure 5 Structure of the MSC Dynamic Semantics: Part 2

In the axiomatic variant, the MSC process algebra is formalised using axioms. This means, the equalities between two MSCs can be decided using the axioms of the process algebra formalisation.

The operational semantics means that the process algebra of an MSC is interpreted using a transition system. The transition system defines which moves are possible for the process algebraic term.

Concluding Remarks

In this article, we have presented an overview of the SDL and the MSC formal semantics definitions. The general distinction between axiomatic, denotational and operational methods was used to identify corresponding parts in the semantics definitions of the two languages.

The subdivision of the semantics into the parts presented above has proved very useful, as it is essential to be able to adjust the semantics to the development of the language. This is achieved by using two layers of abstraction: (1) abstract the concrete syntax into an abstract syntax, (2) transform the abstract syntax to an abstract machine / an abstract process algebra.

It should be noted that the formal SDL semantics has been conceived in parallel with the language definition itself. During this process, several substantial changes to the language definition, which turned out to be a "moving target", were made. These changes of course affected the formal semantics definition, but usually did not touch the SAM.

The formal MSC semantics was last updated for MSC-96. It is contained in Annex B and C of Recommendation Z.120. It covers the static semantics as well as the dynamic semantics. Unfortunately, no new update is made for MSC-2000, due to lack of human resources.

The SDL-2000 formal semantics will cover the whole language SDL-2000. As with previous versions, it will be contained in Annex F of recommendation Z.100. The approval of the complete Annex F is scheduled for November 2000. Currently the dynamic semantics and the static conditions are almost ready, the majority of the transformations still needs to be formalised.

Bibliography

1 ITU. *Specification and Description Language (SDL).* Geneva, International Telecommunication Union (ITU), 2000. (ITU-T Recommendation Z.100.)

2 ITU. *Specification and Description Language (SDL).* Geneva, International Telecommunication Union (ITU), 1993. (ITU-T Recommendation Z.100 Annex F.)

3 Glässer, U. ASM semantics of SDL : Concepts, methods, tools. In: Lahav, Y et al. (eds). *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC Berlin,* 29 June – 1 July 1998, 2, 271–280, 1998.

4 Lau, S, Prinz, A. *BSDL : The Language – Version 0.2.* Berlin, Department of Computer Science, Humboldt University Berlin, 1995.

5 Gotzhein, R et al. Towards a new formal SDL semantics. In: Lahav, Y et al. (eds). *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC Berlin,* 29 June – 1 July 1998, 1, 55–64, 1998.

6 ITU. *Message Sequence Charts (MSC).* Geneva, International Telecommunication Union (ITU), 2000. (ITU-T Recommendation Z.120.)

7 ITU. *Message Sequence Charts (MSC).* Geneva, International Telecommunication Union (ITU), 1998. (ITU-T Recommendation Z.120 Annex B and C.)

8 Mauw, S, Reniers, M A. Operational semantics for MSC'96. *Computer Networks and ISDN Systems,* 31, 17, 1785–1799, 1999.

9 Glässer, U, Gotzhein, R, Prinz, A. Towards a New Formal SDL Semantics based on

Abstract State Machines. In: Dssouli, R, Bochmann, G v, Lahav, Y. *SDL'99 : The Next Millenium.* Amsterdam, Elsevier, 1999.

10 Bo, A et al. *SDL Formal Semantics – preliminary draft.* Beijing, BUPT, 1999. (Technical Report of BUPT.) See also http://tseg.bupt.edu.cn.

11 Gurevich, Y. Evolving Algebra 1993 : Lipari Guide. In: Börger, E (ed). *Specification and Validation Methods*. Oxford University Press, 1995, 9–36.

12 Gurevich, Y. *ASM Guide 97.* Ann Arbor, EECS Department, University of Michigan, 1997. (CSE Technical Report CSE-TR-336-97.)

13 Del Castillo, G. *ASM-SL, a Specification Language based on Gurevich's Abstract State Machines : Introduction and Tutorial*. Department of Mathematics and Computer Science, Paderborn University. (Technical Report (to appear).)

14 van Eijk, P, Belinfante, A. *The term processor Kimwitu – Manual and cookbook (version 4.0).* Enschede, University of Twente, 1996. (Technical report.)

Telelogic SDL and MSC Tool Families

PHILIPPE LEBLANC, ANDERS EK AND THOMAS HJELM

Philippe Leblanc (41) graduated from the Ecole Nationale Supérieure de l'Aéronautique et de l'Espace in 1981. He has worked in the field of real-time software engineering for 16 years, taking various positions, from software developer to project manager and senior consultant. He has helped major Real-Time application developers deploy formal description techniques and object-oriented modelling techniques on industrial projects. He is ITU-T SG10 Rapporteur for Methodology linked to the ITU-T modelling languages. He now works as Product Maketing Manager at Telelogic on SDL and UML technologies.

philippe.leblanc@csverilog.com

Anders Ek (40) graduated from the University of Lund in 1986 with an MSc in Computer Science. He has since been working with artificial intelligence applications, tools based on formal methods and software design languages and environments. He has been working both as a programmer, software analyst, project manager, requirements engineer and team manager. He has worked with language design and been active in the experts groups developing MSC-96 and SDL-2000, two graphical design languages standardized by ITU. He joined Telelogic 1992 and is now head of a development team focusing on language design and future software environments.

anders.ek@telelogic.se

Tool support is required for an efficient use of description techniques in industrial contexts. The objective of this paper is to present the SDL/MSC-based Telelogic tools for analysis and design of real-time systems. The specification and design languages supported by these tools are SDL and MSC which are international standards, defined in the ITU-T recommendations in the Z.100 series and Z.120, that are widely used within the real-time software community, and more particularly for the engineering of telecom applications. Telelogic supplies two integrated tool suites in this area: Telelogic Tau mainly for the communication industry and ObjectGeode for specific application areas such as aerospace and defence systems. These tool suites give also a partial support to ASN.1 and TTCN. Both help to design complex systems and produce reliable and maintainable software applications.

These SDL and MSC tools are complemented with the Telelogic Tau UML Suite that provides full support to UML. The UML Suite is not further presented in this paper, only its relationship with the SDL and MSC tools are detailed.

1 Scope

First we present the generic activities supported by the Telelogic tools (Section 2). Section 3 describes the iterative engineering process that can be applied by project teams working with these tools, whereas Section 4 presents some typical industrial applications that are successfully developed with the help of these tools.

More details are given on the packaging of the tools (Section 5). New features are now being delivered with the most recent versions of the tools, regarding application deployment, support of UMTS and support for C++ development, they are described in Section 6 with a special focus on the application deployment process.

The SDL and MSC modelling is strongly connected to the UML modelling. Section 7 presents the mapping rules that are implemented in the tools. They are based on the real-time specific UML profile defined in the ITU-T Z.109 Recommendation.

We conclude the paper on the future of these technologies. There is a strong trend from the standardisation bodies and language experts, as well as from users and tool vendors, to merge SDL and UML. Telelogic has active participation in the ITU-T and the OMG to speed up this process. A tool development roadmap has been defined and technical teams have been reorganised in order to deliver in a near future a single UML, SDL and MSC toolset unifying the different SDL, MSC and UML tools, and supplying concrete benefits for real-time software development such as design-level debugging, model validation, code generation, assisted deployment and test case generation.

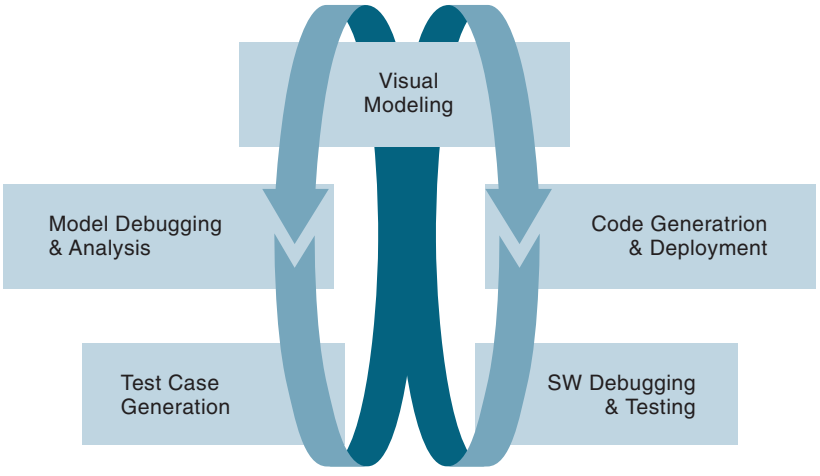


Figure 1 Generic Architecture of the Telelogic Tool Families

Thomas Hjelm (37) holds an engineering degree and an MSc of Computer Science from Oxford University, as well as telecom-oriented company MBA. He has over 10 years of senior experience in software methodologies, including use of formal methods for development of security systems, model-checking of protocols, practical application of the Telelogic Tau tools, object-oriented methods and CMM based software process improvement. He has worked in the areas of development of real-time software CASE tools, train security software and Mobile Communications. He has been working as Product Manager at Telelogic since 1998.

thomas.hjelm@telelogic.com

2 Supported Activities

Telelogic SDL and MSC tools support the generic activities depicted in Figure 1. Tool suites comprise:

- Visual modelling tools to build analysis and design models according to the UML, SDL and MSC notations.
- Model debugging and analysis tools to debug models and to explore their behaviour in automatic mode for early error detection and conformance checking.
- Test case generation tools to build interactively or in assisted mode test plans for software conformance testing.
- Code generation and deployment tools to convert models into executable applications and to deploy the generated software onto target platforms according to various configurations.
- Software debugging and testing tools to help debugging software applications at design-level and testing them in accordance with test plans either produced from models or provided externally.

3 Engineering Process

The SDL and MSC tools allow developers to set up an iterative and architecture-centric engineering process depicted in Figure 2. First the development project is divided into successive iterations, each developing an increment of the application. The application is completed at the last iteration. This means that the system architecture is continuously refined along the process and that it guides the sequence of iterations.

Within all iterations, two micro-cycles are carried out subsequently. The first micro-cycle (left-hand side circular arrow in Figure 1, shortest circular arrows in Figure 2) consists in enriching the application models using visual modelling tools, and verifying and validating these changes using simulation tools. The second micro-cycle (right-hand side circular arrow in Figure 1, longest circular arrows in Figure 2) consists in converting the changes validated during the first micro-cycle into executable code and deploying the generated software on the target platform; possibly testing these changes on the host platform first.

4 Typical Industrial Applications

Software applications targeted by the Telelogic tool families, range over the whole real-time software domain, and more particularly: telecommunications, aeronautics, space, defence and automotive applications. Historically, the telecommunication sector is the most important one, representing more than two thirds of the customer base.

Typical examples of systems that have been successfully developed with the Telelogic tools are:

- Public and private fixed networks and terminals, based on various technologies such as ATM, Internet, VoIP, ISDN, IN, etc.;
- Mobile networks and mobile and wireless terminals, based on technologies such as: UMTS, GSM, GPRS, Bluetooth, CDMA, WLAN, IEEE 802.11, TETRA, etc.;
- On-board communication software embedded on aircraft, e.g. for the Future Air Navigation System (FANS);
- On-board management software embedded on satellites or on the International Space Station (ISS);

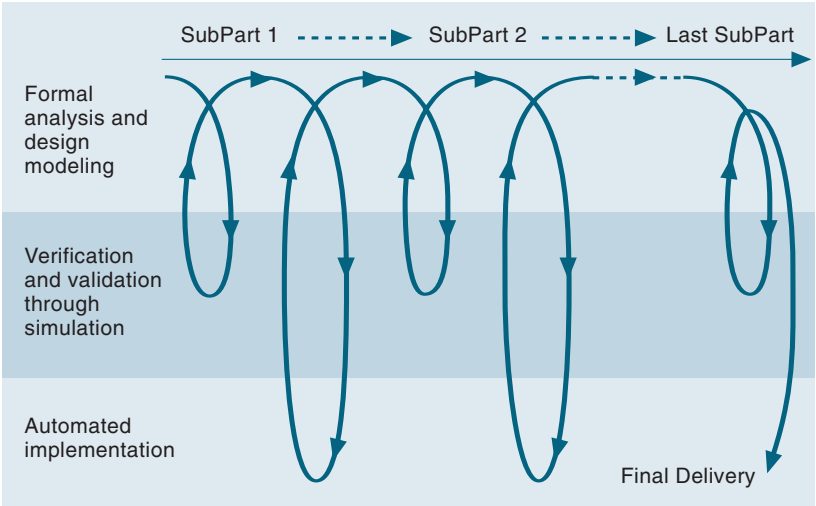


Figure 2 Iterative Engineering Process

Tau Tools for Visual Modelling	
Organiser	To define personal workspaces and browse SDL descriptions
SDL Editor	To create SDL descriptions (see Figure 3)
MSC Editor	To create MSC requirements and test specifications, and to visualise test execution traces
HMSC Editor	To create high-level MSCs, in order to structure large sets of basic MSCs (see Figure 3)
State Chart Editor	To create UML state charts and to visualise overview of SDL state machines
Deployment Editor	To create UML component diagrams describing physical structure of applications
SDL Overview Viewer	To display entire SDL hierarchies in one diagram
Type Viewer	To display type hierarchies of SDL descriptions
C/C++ to SDL Translator	To reuse existing C and C++ code in SDL applications
ASN.1 Import	To reuse existing ASN.1 data definitions in SDL applications
UML to SDL/MSC Translator	To produce preliminary SDL descriptions based on UML analysis models developed using the Telelogic Tau UML Suite

Tau Tools for Model Debugging and Advanced Analysis	
Analyser	To perform syntactic and static semantic check of SDL descriptions
Simulator	To perform host debugging of SDL descriptions and regression testing either using a script language or using MSCs as test descriptions (see Figure 4)
Validator	To perform automatic testing of SDL descriptions based on formal verification techniques and state graph exploration
SDL-TTCN Co-Simulation	To test SDL applications using TTCN test suites

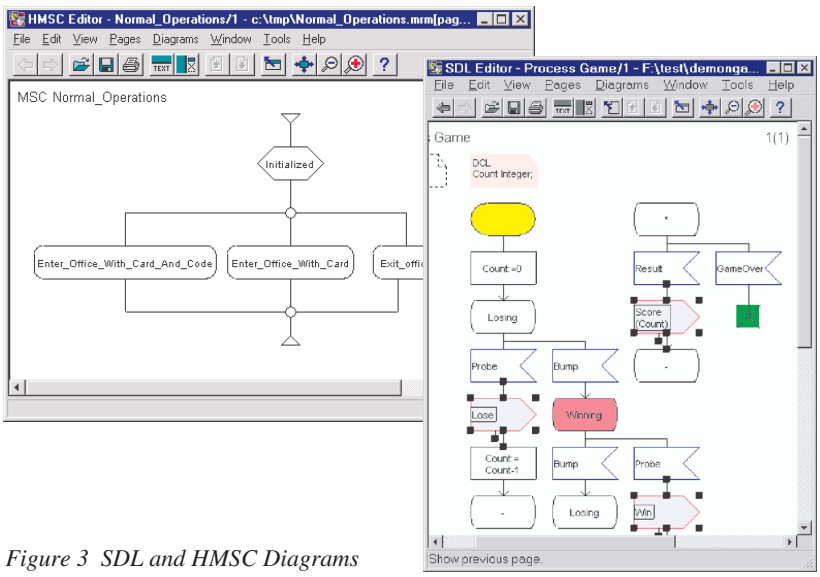


Figure 3 SDL and HMSC Diagrams

- Vehicle electronics such as Embedded Control Unit (ECU), driver assistance systems, etc.

In terms of project size and software complexity, Telelogic tools are able to manage a large range of applications: from large-size projects with up to 300 developers (e.g. in the case of switching systems), down to small footprint applications with 8 kB of ROM (for code) and 256 b of RAM (for data) running on a micro-controller (e.g. for military on-board software).

Software produced with Telelogic's technologies can be deployed on most of the real-time operating systems available commercially, such as VxWorks-Tornado, pSOS+, VRTX, OSE, QNX, Nucleus Plus, POSIX, Chorus, OSEK, and workstations running UNIX, Windows or Linux. Applications can also be generated for bare systems without any RTOS. In that case, a compact and optimised SDL virtual machine is included in the generated application.

Telelogic is the biggest supplier of real-time software engineering tools with 32 % of the market share. Specifically in the telecommunication sector, Telelogic owns more than 50 % of the CASE tool market for the development of real-time systems.

5 Toolset Packaging

5.1 Telelogic Tau SDL Suite Packaging

The different modules composing the Telelogic Tau SDL Suite are presented in the following tables, as well as some tool screenshots.

Telelogic Tau SDL Suite is available on UNIX and Windows machines.

5.2 ObjectGeode Packaging

The ObjectGeode packaging is similar to the Tau SDL Suite packaging:

- Visual modelling tools include: SDL Editor, MSC Editor, SDL&MSC API, Editor API.
- Simulation tools include: Model Debugger, Model Advanced Analyzer, Simulator API.
- Test case generation tools include: Test Composer and TTCN Test Suite Publisher.
- Tools for application generation, deployment and software debugging include: SDL C Code Generator, Run-Time Libraries for various RTOS, Design Tracer.

ObjectGeode is available on UNIX and Windows machines.

6 New Tool Features

New versions of the Telelogic SDL tools have been recently delivered, in particular the 4th generation of Telelogic Tau SDL Suite. This recent tool generation integrates new features providing significant help for industrial software development. In particular, it gives full support for: application deployment using visual models, UMTS development including import of ASN.1 UMTS modules, and C++ development enabling reuse of legacy C++ code.

6.1 Deployment Editor and Targeting Expert

The transformation of logical SDL models into efficient implementations is a key-activity for SDL-based application development. The purpose of the Deployment Editor and Targeting Expert is precisely to provide support in this activity. Deployment diagrams in Telelogic Tau SDL Suite describe the physical architecture of the application and the relation between the logical SDL entities like blocks and processes and this physical architecture. This is essentially a model based on the following main concepts:

- Nodes, i.e. the hardware platforms on which the application will run;
- Components, i.e. the executable files that make up the application;
- Logical SDL entities that run in the threads, i.e. component property set either to light (only one thread for the entire SDL system), or tight (one thread per instance) or instance-set (one thread per instance set).

An example of a deployment diagram is shown in Figure 5. The aggregation structure illustrates the relations between the different entities, which components can execute on which node type and which SDL entities can run in each component. The UML stereotypes used in this diagram are defined in the UML profile elaborated by Telelogic for SDL-96 (see Section 7).

The SDL model and the deployment diagrams cover two of the most important aspects of an application, the logical behaviour and the physical structure. The purpose of the Targeting Expert tool is to further define the details of each component in the physical structure. It guides the user through implementation choices that are available for each component and makes it possible to fine-tune the execution of the generated applications and to define the details of the build process. It is also the tool of choice for building the application and can be executed both in interactive mode, mainly for setting up the generation and build properties, and in batch mode, to build the final application. The Targeting

Tau Tools for Test Case Generation	
TTCN Link	To perform semi-automatic development of TTCN test suites based on SDL descriptions
Autolink	To perform automatic generation of TTCN test suites based on SDL descriptions and MSC test purpose specifications

Tau Tools for Code Generation and Deployment	
C Advanced code generator	To generate C code for applications with soft real-time constraints
C Micro code generator	To generate C code for applications with hard real-time constraints in term of memory and performance
Chill code generator	To generate Chill ITU-T language
Targeting Expert	To perform fine-tuning and optimisation of generated code, and to control the build process
Real-Time OS adaptations	To make the generated code executable on various real-time operating systems including ChorusOS, OSE, pSOS, QNX, VRTX, VxWorks, Tornado, Nucleus PLUS, Win32 and Solaris

Tau Tools for Software Debugging & Testing	
Target Tester	To test the generated applications on target platforms

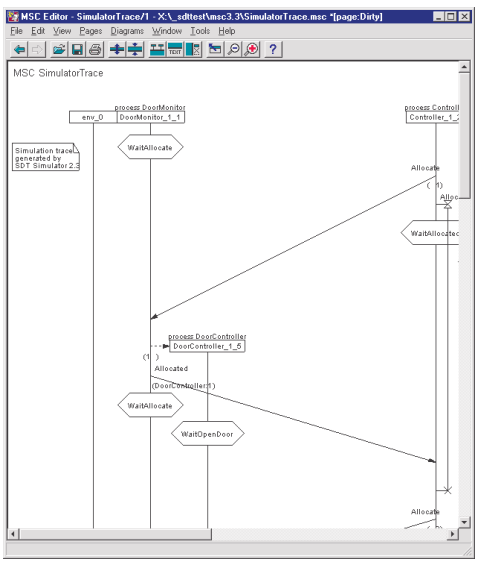


Figure 4 MSC Generation during SDL Simulation

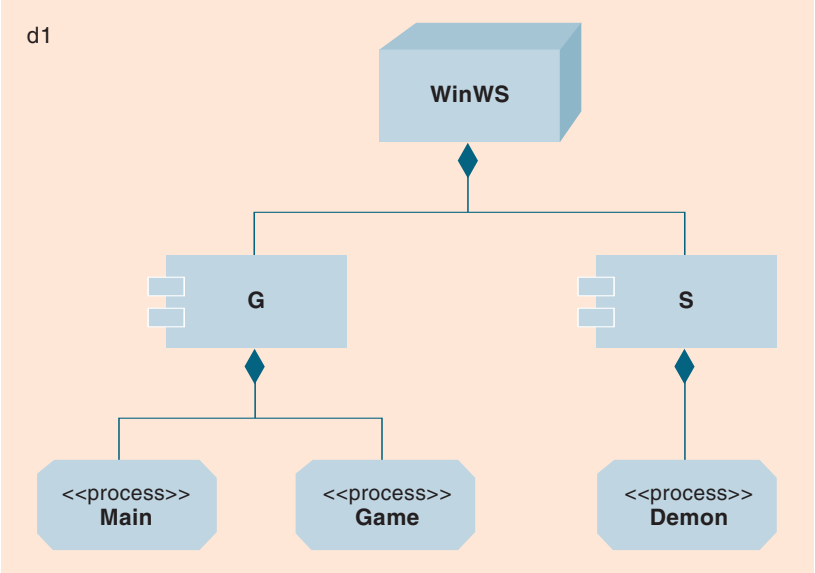


Figure 5 A Typical Component Diagram

Expert can also export a generated make file for inclusion in an external make process.

Typical properties that are described using Targeting Expert are:

- Details of the code generator to be used;
- Details of the compiler that is used, e.g. optimisation issues like whether the compiler most efficiently handles characters or integers, the command line parameters needed and the environment variable settings required;
- Definition of what communication links will be used within the application, defining e.g. what encoders/decoders to use;
- Description of what tool has to be used to download the generated application onto the target platform.

The properties can be set either for a complete application, for all components running on specific nodes or for specific components.

6.2 Use in UMTS Projects

Many parts of the UMTS standards are written using SDL, ASN.1 (e.g. the RLC PDU definitions) and TTCN (conformance tests).

Telelogic Tau has been updated to allow immediate reuse of these parts of the standards. For example the ASN.1 Import has been extended to import the full ASN.1 standard covering X.680, X.681, X.682 and X.683, in concordance with the new version of the Z.105 standard. In addition, it is possible to automatically generate PER encoders and decoders for the UMTS interfaces, which is an easy way to create application simulators, prototypes or the final application code.

The UMTS support brings several benefits to 3G projects. It is possible to jump-start the development by importing the SDL specifications, which can then be simulated to help understanding their dynamic behaviour, or they can be used as the starting point for the application development. By importing the ASN.1 specification, the SDL design will by definition have the right interfaces with no risk of interpretation or coding mistakes.

6.3 C++ Support: Effective Integration of UML and SDL

More and more Application Programming Interfaces are written in C++ and it can be very useful to be able to use these directly in SDL applications. The C++ Access application allows C++ interfaces such as Microsoft Foundation Classes or database interfaces to be imported into SDL and used directly in the SDL diagrams, for example in Task or Decision symbols. The code generators will generate the corresponding C++ calls.

The benefit of this approach is that the full SDL syntactic and semantic checks are available (contrary to when in-line code is used). It means that if the C++ interface is changed, the SDL analysis will immediately detect any inconsistencies between the new interface and the calls written in the SDL code, thus reducing the risk of late discovery of coding errors.

The C++ interface allows an effective development approach where UML can be used to design the overall system, where after certain classes are translated into SDL and others are developed in C++. By using the C++ Access, the results of the parallel developments are smoothly combined. The original UML design ensures that both parallel development tracks work from a common well-defined base.

7 Mapping Rules between UML 1.3 and SDL-96, and Tool Support

7.1 Principles

In the Telelogic tools, SDL and MSC modelling is tightly connected to UML modelling. The recommended engineering process is to start system modelling with UML, since UML is widely accepted as a general-purpose modelling notation. When moving to the architectural and behavioural design activities, SDL and MSC are recommended instead as they include: structuring and decoupled communication mechanisms for building modular architectures, an action language to fully describe behaviour, and an execution model (formal dynamic semantics) making behavioural models executable.

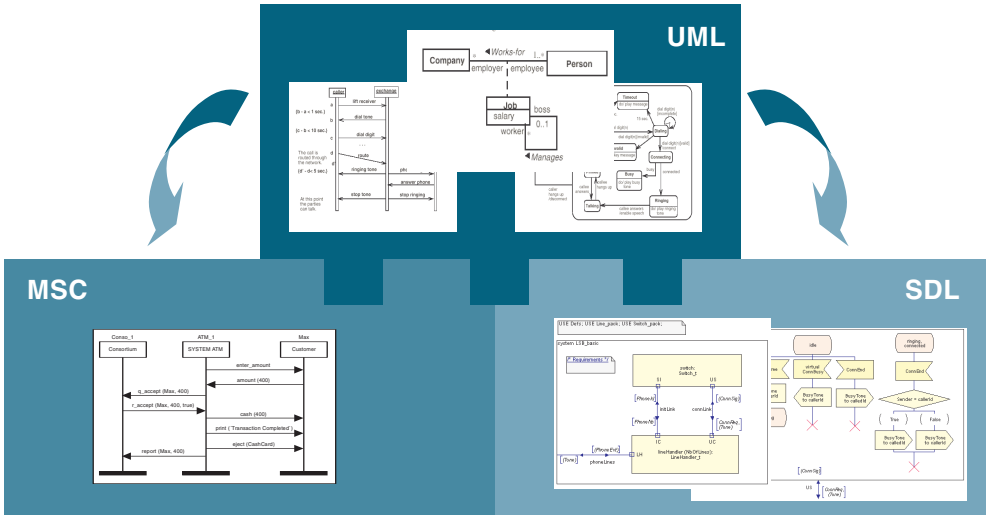


Figure 6 From UML to SDL and MSC

The joint use of these three different notations must be as smooth as possible from an end-user point of view. The ITU-T has made significant progress in this area with SDL-2000, which directly includes UML constructs and the well-known UML graphical representation, and with Z.109 that describes the mapping rules between UML and SDL, see the paper by Birger Møller-Pedersen on *UML combined with SDL* in this issue.

These achievements constitute the theoretical foundation for the UML to SDL-MSC mapping supported in the current versions of the Telelogic tools. Adaptations were required to map UML to SDL-96, since Z.109 relates to SDL-2000. These adaptations have led to limitations that will be removed as soon as the tools will be SDL-2000 compliant.

In this mapping, see Figure 6, the three most important UML diagrams are considered: Class diagram, Statechart diagram and Sequence diagram. They are respectively converted into: SDL Structure diagrams, SDL State machine diagrams and basic MSC. Use-case diagrams in UML cannot be taken into account, as they do not contain semantics. The following sections detail the different mapping rules as defined by Telelogic for SDL-96. Note that they do not strictly comply with Z.109 and would be different for SDL-2000, since they are applicable to SDL-96.

7.2 UML Class Diagram and SDL Structure Diagram

Class diagrams are converted into type definitions in SDL Structure diagrams in accordance with a set of stereotypes that must be used to enable the conversion. Stereotypes are «actor», «block», «process», «signal» and «newtype». They are applicable to classes.

The hierarchical structure generated for the SDL model is based on the active classes stereotyped with «block» and «process» and on their aggregation links. For each block/process class, two SDL elements are generated: the corresponding block/process type and the corresponding block/process instance set. Associations between active classes are then converted into gates for the block/process types and channels/routes between the block/process instance sets.

Classes stereotyped «actor» are converted into channels connected to the environment.

Attributes to classes stereotyped «process» are converted into local variables, as well as associations pointing to classes stereotyped «newtype». Operations of classes stereotyped «process» and «block» are converted into signals or remote procedures if the operations return values.

Classes stereotyped «signal» are converted into SDL signal declarations placed at the system level, and classes stereotyped «newtype» are converted into SDL newtypes placed also at the system level.

Inheritance is also supported, using the native SDL concepts of block/process type and inheritance.

7.3 UML Statechart and SDL Transition Diagram

UML Statechart diagrams are converted into SDL state machine diagrams, according to the general rule: one UML state machine generates one SDL state machine.

For most of the Statechart constructs, there is a direct mapping to SDL. For example, State, Reception, ChangeEvent, Guard, TimeEvent, SendAction, AssignmentAction, CallAction and

TerminateAction are respectively converted into State, Input, Continuous Signal, Enabling Condition, Timeout, Output, Assignment, Procedure Call and Stop.

UML entry and exit actions are mapped to duplicated actions in all the SDL transitions leading to or leaving the corresponding state.

The main difference comes from the state composition mechanism allowed in Statechart and not in SDL. Hierarchical UML states are flattened, and intermediate entry / exit actions are duplicated in the SDL transitions. The resulting SDL state machine is a flat set of transitions going from simple states to simple states. Note that this will be different in SDL-2000.

7.4 UML Sequence Diagram and Basic MSC

The mapping from UML Sequence diagrams to MSC is straightforward, as a UML Sequence diagram is a subset of Z.120/96, except for two constructs, activation flow and method invocation, which have no correspondence to constructs in MSC-1996.

The mapping is one-to-one: a UML Sequence diagram is converted into a single MSC; a UML class instance (vertical bar) is converted into an MSC instance; a UML message is converted into an MSC message.

Note that MSC has many other features, such as structural mechanisms, that are not supported in UML.

7.5 Tool Support with Telelogic Tau UML Suite

These mapping rules described above are supported in the Telelogic tool families: models made with the UML Suite can be converted into SDL/MSC models processable by the SDL/MSC tools. Therefore a project can start the modelling activities with the UML Suite, and then move to SDL for detailed design with neither time loss nor information loss.

Practically, a 3-step approach is recommended:

1. The UML model is built using Class diagrams, Sequence diagrams and State diagrams, without any constraints on the UML used. This allows pure UML analysts to perform this step without any preconception of what the system architecture will be.

2. The “informal” UML model made in the first step is reworked: SDL stereotypes are introduced to formalise the UML model, and the transition actions are reformulated using an SDL-based concrete syntax. We are still at the UML level, but the designer must be aware of the modelling constructs of SDL required for real-time applications.

3. The “formalised” UML model is ready for conversion into SDL. The SDL model (and MSCs if any) is generated. Most of the time, the model will have to be reworked in order to have a correct and complete SDL design containing type definitions, architecture and state machines. This step is similar to the formalisation activity as defined in the SDL+ Methodology (defined in supplement 1 (04/96) to Z.100): the informal SDL model is reworked in order to conform to the static and dynamic semantics of SDL. Statically, it means that usage of types must conform to their declarations; connections must be coherent; state machines must be consistent with the type declarations and connections, etc. From a dynamic point of view, dynamic creation of process instances must be compatible with the state machine definitions; signal routing must be based on a correct use of Pids and channels, etc. It does not seem appropriate to work at the UML level to complete the SDL modelling, even if it is possible, as the SDL tools give a very efficient support to do that (static checker and simulation tools run directly on SDL models).

8 Future of Notations and Technologies

UML is an excellent general-purpose modeling technique, but needs to be complemented to efficiently address the development of real-time applications. UML needs:

- Structuring and loosely coupled communication mechanisms for building modular reusable architectures;

- An action language for specifying transition actions;

- An execution model for executing models.

These requirements are generally recognised by software engineering experts including the OMG, and enhancements are already proposed by the OMG, through two Requests For Proposal “Action Semantics” and “UML Profile for Scheduling, Performance and Time” and the UML 2.0 Request For Information (RFI).

Facing the increasing use of UML for software engineering and considering these needs, Telelogic has set up a 3-step convergence plan, initiated in 1999 (see also Figure 7):

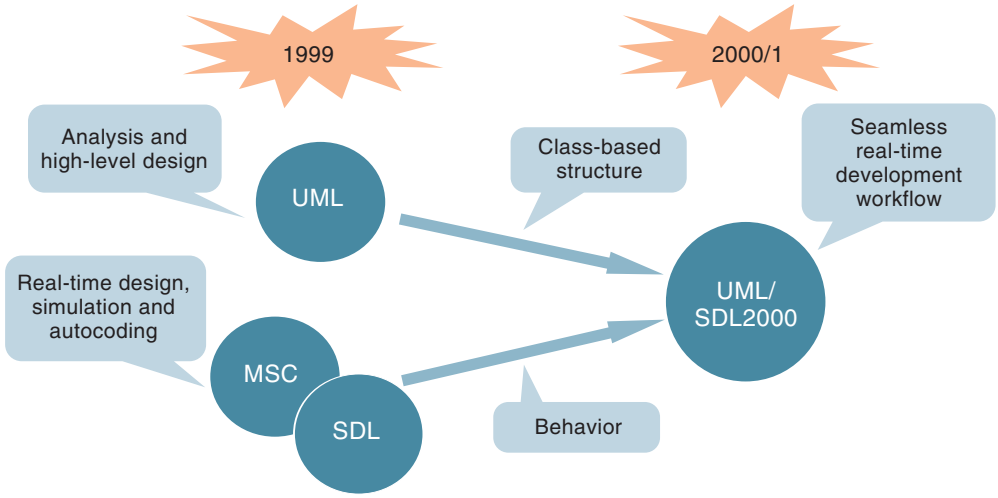


Figure 7 Technology Trends

Step #1:

In 1999, theoretical studies on a mapping between UML 1.3 and SDL-96 have defined an informal UML profile for real-time applications. This allows designers to build consistent UML and SDL models but separately.

In parallel, Telelogic has collaborated with the ITU-T to define SDL-2000, in order to incorporate in SDL the UML constructs that were missing in SDL-96, mainly a class-based data model and Statechart-specific concepts, and to add the UML class representation to the SDL Structure diagrams.

Step #2:

In 2000, the UML 1.3 – SDL-96 mapping is automated in the Telelogic tools enabling designers to generate SDL models from UML models. This tool support allows a joint and consistent use of both notations within a project, without loss of information. Thus, UML and SDL end-users benefit from all the powerful features provided by the Telelogic Tau and ObjectGeode Suites, such as model debugging, model validation, test case generation, code generation and assisted deployment.

In parallel, Telelogic participates to the OMG RFP working groups and UML 2.0 RFI, in order to promote the SDL solution as the reference UML profile for real-time. Submissions consist in extending UML with the SDL-2000 concepts in order to solve the current UML 1.3 limitations: structure and communication based on block, process, gate and channel; SDL syntax as concrete action language; SDL execution model. Major industrial users are also supporting this approach, such as Ericsson, Motorola and Alcatel.

The standardisation work is continuing in the ITU-T to consolidate SDL-2000, and in particular on methodological aspects.

Internally, all the Telelogic technical teams involved in the development of the Telelogic Tau and ObjectGeode Suites have been reorganised, putting together their experience and their technologies.

Step #3:

In 2001-2002, Telelogic will deliver a coherent and powerful support for SDL-2000 and UML within a single tool suite, including advanced simulation and code generation facilities.

Compatibility between current tool families and the next product generation will be sought, in order to provide customers with an enhanced UML/SDL tool support without disruption.

This 3-step roadmap is compatible with the UML evolution as we can anticipate from the OMG. Within two years, the SDL and UML notations and technologies will be merged, without sacrificing the benefits of model simulation and autocoding.

9 More Information

ITU Recommendations, in particular the Z.100 series including Z.100, Z.105, Z.109 and Z.120, can be accessed from the ITU Web site: www.itu.int.

The SDL Forum Society, www.sdl-forum.org, will also provide the readers with valuable information regarding SDL and MSC: tutorials, latest versions of SDL and MSC Recommendations. A mailing list is also available for members.

The OMG Web site, www.omg.org/uml, gives you access to the UML Specification as well as recent information on the UML working groups.

More information on the Telelogic company and on these products is accessible from www.telelogic.com.

Cinderella SDL – A Case tool for System Analysis and Design

ANDERS OLSEN AND FINN KRISTOFFERSEN

This paper describes the CASE tool Cinderella SDL that supports system development using SDL and UML. Current and planned features of Cinderella SDL are described in the context of the system development activities. For each of the activities, system analysis, design, test, and implementation, it is illustrated how Cinderella SDL may support this activity. The paper describes how the tool enables the combined use of SDL and UML, supports the system design process, can be used for design validation, and provides a basis for code generation from an SDL/UML system model.

Anders Olsen (44) received his MSc from the Technical University of Denmark in 1983 and has worked with SDL ever since. In the International Telecommunication Union (ITU) he was responsible for – and main author of – the Formal Definition of SDL, first in 1988 and then again in 1992. He has played a central role in numerous SDL related EU projects, most recently in the project SCREEN 1996 – 1998, where he was technical leader. In 1995 he was co-founder of Cinderella (www.cinderella.dk). Apart from the work for Cinderella, he has also established the company Cinderella Consult.

anders@cinderella.dk

Finn Kristoffersen (40) holds an MSc E.E. from the Technical University of Denmark, and has been working at Tele Danmark Research in the Software department since 1986. He has been involved in ITU and ETSI standardisation of formal methods in conformance testing, and in mobile protocol specification and validation projects. He has worked with WAP service development, and he is co-founder of Cinderella. His current interests are formal specification methods and their combined usage in system development.

finn@cinderella.dk

1 Introduction

Cinderella SDL is a CASE tool, based on UML and SDL, for system development. Cinderella SDL supports the main activities of development:

- *System analysis*, i.e. the definition of a conceptual model for the system to be developed.
- *System design*, i.e. the definition of an executable model for the system to be implemented.
- *System tests*, i.e. verifying that the system fulfils the user requirements.
- *System Implementation*, i.e. generation of executable code.

Ideally, system development starts with analysis, followed by system design, followed by test of the system design, followed by generation of implementation code from the design.

However, in practice, software development is an iterative process where it is often necessary to go back to a previous activity due to changed requirements or due to new knowledge of the system.

That is, software development follows the so-called spiral model shown in Figure 1 (as opposed to the 'waterfall' model):

For Cinderella, the challenge is to support these activities in such a way that:

- The amount of redundancy is minimal. That is, properties that have already been specified during e.g. analysis should not be repeated during design.
- The properties of parts belonging to different activities are kept consistent. That is, there should be a mechanism to assure that changes, e.g. in the design, do not invalidate the analysis part or make it obsolete.
- The user is not forced to think in terms of analysis and design. Rather, the user should be free to choose views of their own choice.

The freedom in choice of views (or abstractions) is an essential point[1)]. The choice of view should not be restricted to the activities above, because abstraction should provide the ability to deal with exactly the properties you want, and not to be concerned with any other properties.

The upcoming version of Cinderella SDL, version 2.0, gives a first approach to fulfil this vision, by providing a tight integration of UML and SDL. Cinderella SDL is a UML tool for UML users, an SDL tool for SDL users and an

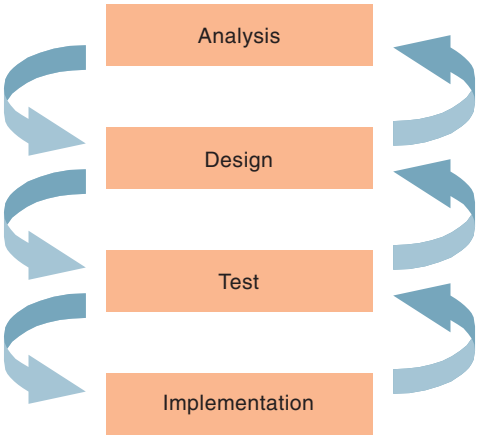


Figure 1 The Spiral Development Model

[1)] *The four activities can be regarded as dealing with the conceptual view, the semantic view, the external view and the physical view – or in terms of ODP – the informational viewpoint, the computational viewpoint, the enterprise viewpoint and the engineering viewpoint.*

SDL/UML tool for those users who want to benefit from both technologies. In this version of Cinderella SDL, a specification can at one moment be viewed and edited as UML and the next moment as SDL. For example, a state machine can be viewed and edited as a UML state chart (with few details) or as an SDL process diagram (with all details).

In the following, the Cinderella approach to system development, based on the four activities, is described:

- Section 2 describes the UML support and how the user can switch view between UML and SDL;
- Section 3 describes the SDL features;
- Section 4 describes the methodology for testing;
- Section 5 describes implementation issues;
- Section 6 concludes the paper.

2 Combining SDL and UML

In the planned version of Cinderella SDL a user can make a complete system design

- by using UML only; or
- by using SDL only; or
- by combining SDL and UML.

Behind the scene, SDL structures are used no matter which approach is taken. Therefore, a system can at any time be viewed in the form of SDL.

The parts of UML that are supported are those which form the central part of UML and which have clear resemblance to SDL-2000, namely the class diagram and the state charts.

To illustrate how it works, let us consider a very small game where players get the message *Win* or *Lose* when they trigger *(Probe)* the system. A user can at any time ask the system of the score, i.e. the number of times they got *Win* more than *Lose*.

The system consists of one class: *Toss*. In addition, we specify the *user*, even though the *user* is not part of the system. This is partly in order to include the communication scheme, partly because the *user* (the environment) becomes important during the test phase.

The UML classes are shown in Figure 2.

There is an implicit SDL package with system name followed by 'Package' for every system. This package makes visible all the types that are defined directly in the system. Thus, the above two classes correspond to the system as shown in Figure 3, here represented by the process instance *Toss*.

Note that the *User* is not part of the system. It is anyway specified on the class level, partly for explanatory reasons, partly because the *User* class will be important during the system test.

The *tossPackage* is defined as shown in Figure 4.

The process Toss can be defined either with a state chart or by using an SDL process diagram.

A state chart for the Toss process contains two states *Odd* and *Even*. When the game is in state *Even* and it handles the event *Probe,* it will respond with Win. When it is in the state *Odd* and handles *Probe*, it will respond with *Lose*. The system can randomly change state. In both states, it can handle the triggered operation *GetCount*.

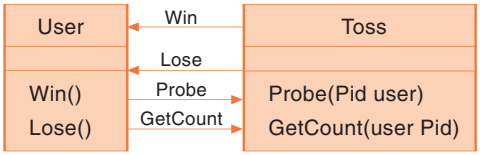


Figure 2 UML classes for the environment (user) and the Toss process

Figure 3 SDL system based on the Toss class

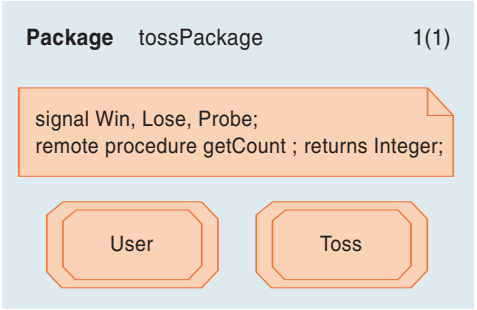


Figure 4 The implicit SDL package

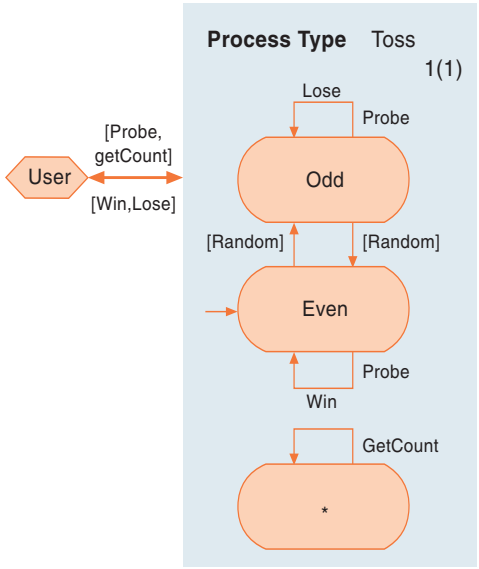


Figure 5 Overview of the state machine

The diagram in Figure 5 is a mixture of a UML state chart and SDL process diagram, but this is just to show the flexibility. Alternatively, the state chart can be shown using UML notation solely, or as SDL using a state machine diagram.

To define the behaviour of the transitions, there are three options:

1. The user can switch to SDL view where the user will see an SDL state machine diagram with empty transitions. The transitions can be inserted using the SDL editor. Afterwards, the user can switch to the state chart view without losing the information.

2. The user can click on the actions (e.g. Win) and a dialog-based wizard will assist the user (typically a UML user) through the process of defining actions.

3. The behaviour is defined in terms of a set of procedures, one for each combination of state and input. For example, the calls of the four procedures, *Probe_even_even*, *none_even_odd*, *Probe_odd_odd* and *none_odd_even* (shown in Figure 6), can be generated by the tool, while the contents of the procedures is defined by means of one of the above approaches.

The SDL graph for the toss process with procedures is shown in Figure 6.

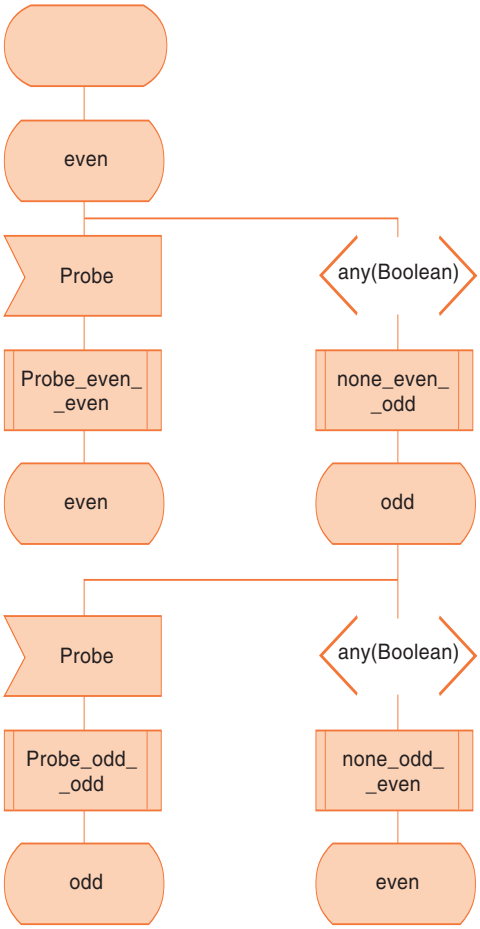


Figure 6 SDL state machine for the Toss process

3 The SDL Part

SDL does not enforce a particular system development methodology, so an SDL tool should support different methodologies. Methodology independent features offered by an SDL tool may include the following aspects:

- editing and analysis;
- navigation features;
- organising the parts of a specification in different files;
- simulation.

In this section we will describe how Cinderella SDL provides support for these aspects. In Figure 7 the graphical user interface to Cinderella SDL is shown. The three main areas of the tool are *the editing area* containing the SDL windows, *the specification explorer* containing the specification structure and the *properties area* providing information about a selected item.

Editing and Analysis

When developing a system specification in SDL using Cinderella SDL, a number of basic editing features ease the design process.

The basic editing features include: diagram type dependent symbol bars offering only the valid symbols according to the diagram type and when new graph symbols are added a connecting flowline is automatically inserted. As Cinderella SDL is a Windows application, all basic editing functions and keyboard shortcuts for cut, copy and paste are available. Selection of symbols may be done either by selecting all symbols

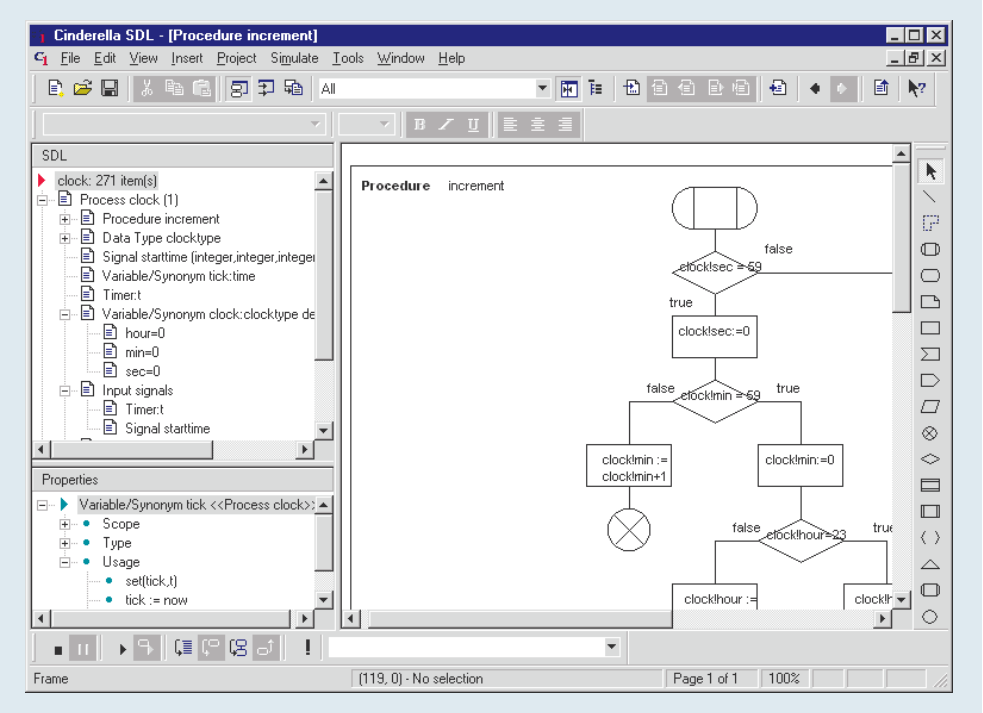


Figure 7 The Cinderella SDL user interface

within an indicated area, by selecting all symbols belonging to the subtree of a selected symbol, or by selecting all symbols in a diagram page. Editing text in symbols may be done either directly in the symbol or using a separate text-editing window. Text search and replace functions are available. The search and replace functions have user defined scope settings to limit the operation to the current page, current diagram, current and nested diagrams or the complete specification.

Although the SDL editor includes immediate checks for which symbols may be connected, it is the syntactical and semantical analyser that provides the basis for most of the other tool features, including advanced navigation and simulation.

Cinderella SDL allows the user to select which SDL-recommendation shall be used by the analyser, i.e. SDL-92 or SDL-2000. The analysis level can also be controlled to include only syntax analysis or complete syntax and semantical analysis. The combined use of the datatype language ASN.1 and SDL is also an option that can be set by the user. The incremental analysis in Cinderella SDL may be done in the background while at the same time constructing the information needed to offer the advanced navigation features as described in the following section. When working on large specifications, the background analysis may sometimes slow down the response time of the editor, so this feature can be disabled and analysis is then performed only upon user request.

Navigation Features

Navigation is a key issue when developing system specifications using SDL. One reason for this is that an SDL specification typically consists of a number of separate entities like packages and diagrams, where definition and usage occur in different places. Cinderella SDL provides the following features to ease navigation in a specification:

- Forward/Back functions to browse the list of diagram pages shown in an editor window;

Figure 8 Right mouse button functions for a variable name

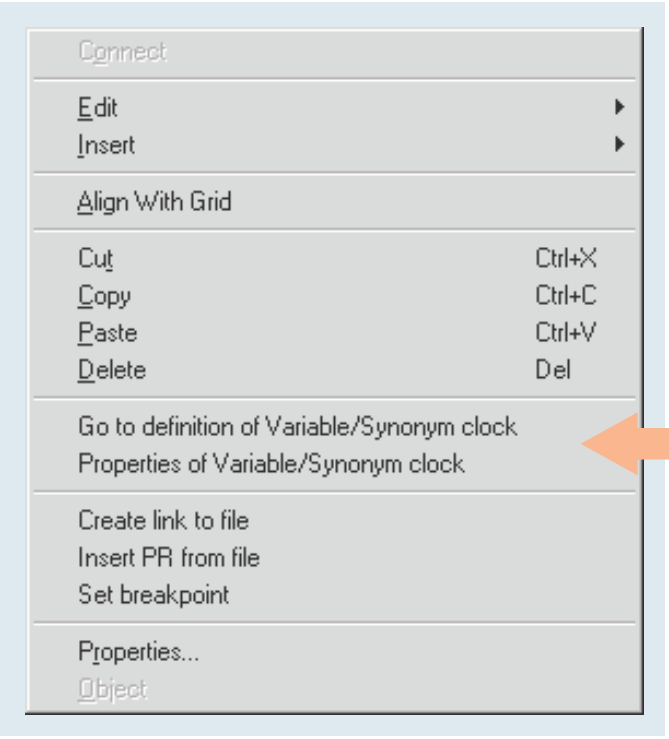


Figure 9 The 'usage' list makes browsing to occurrences of a name easy

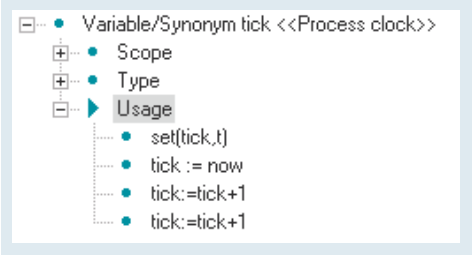
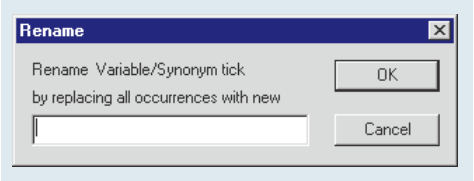


Figure 10 The rename feature provides a safe way to rename entities in a specification



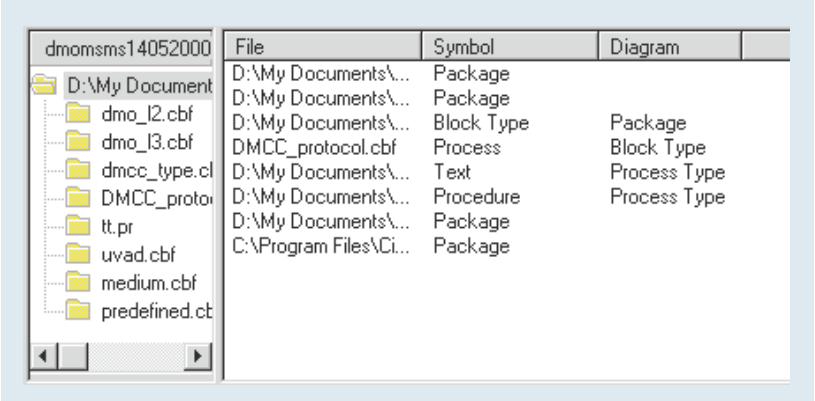


Figure 11 The Linked file view shows the file structure of a specification

- Diagram page navigation functions to show first, previous, next or last page. Also selection of a specified diagram page number is supported;

- Selecting an entity in the specification explorer and double click on this entity will activate the entity in the editor window;

- More windows of the specification can be open simultaneously in the editing area, and thereby provide the means for a more convenient overview of different entities in a specification.

As an alternative means to browse a specification, it is possible to select a name or symbol in the specification and use the right mouse button to get access to all basic operations and information available for the selected item. Figure 8 shows an example of the functions available to a variable name via the right mouse button. For a variable name, two primary features for browsing are to go to the definition or to view the properties of the variable in the property pane.

In the property pane all derived information about a selected entity are shown. As a new feature it also lists all occurrences of the name in the properties pane. The usage list, see Figure 9, is convenient when browsing a specification as you can activate a specific occurrence of the name in the editor window by selecting the name from the usage list. Another feature that is available from the properties pane is the rename operation.

The rename feature allows you to replace a specific name in the specification independently of the way that name is formatted, and independently of whether the same name is used in other scope units. Figure 10 shows the rename dialog box for the replacement of a variable name. The rename feature is only available when a specification has been analysed and the selected analysis level is set to a level sufficiently high to support this operation. The rename operation only replaces occurrences of the selected name belonging to the same entity class and in this way offers a safe way to do renaming in a specification.

Specification Architecture

Cinderella SDL supports keeping a specification in a single file, which may be convenient if porting the specification between different PCs; as well as dividing a specification into several files. Dividing a specification can be done dynamically and at many levels. In Cinderella SDL packages, systems, blocks, processes, procedures and text symbols can be saved to separate files. The format of the different parts may also be chosen according to the current needs as the tool allows mixing of graphical and textual SDL. The current structure of a specification can be shown in the "linked files" view as illustrated in Figure 11.

Simulation

During development of the system specification, verifying the system requirements through simulation is a useful way to check the correctness of the system design. To support this kind of system validation, the simulator should be an integrated part of the tool. In Cinderella SDL the simulator directly interprets the SDL model, and hence can be used on partial system specifications. This allows validation through simulation to be used very early in the design process. The simulator also provides a complete dynamic range check that supports the validation of constrained data types.

The simulator supports the operations: single step, step over, run to a selected symbol, and to set and remove breakpoints. In addition the simulator allows you to manipulate the state of the system, e.g. by inserting signals in the input

queue of a process, creating new instances of a process, or change the value of a variable. The result of a simulation can be viewed as an MSC, see Figure 12.

Tool Configuration

In order for a specification tool to be useful in different system development projects, it must be easy to configure to comply with the methodology required by the company and with developer preferences. For the described features of Cinderella SDL, they can all be controlled by the user. Figure 13 shows a single tab for the configuration of the explorer pane. A configuration may be given a name and saved so that it can be made available when starting the tool again.

To support the portability between different system development tools, Cinderella SDL can also load and save in the standardised exchange format CIF, as well as the textual representation of SDL.

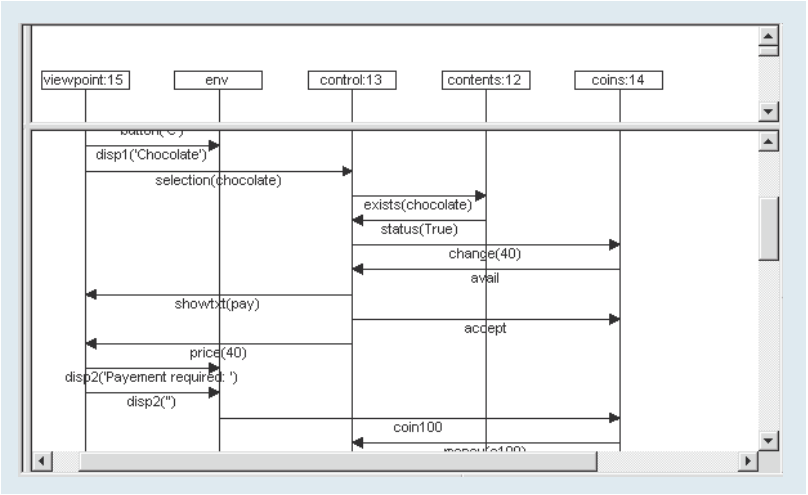


Figure 12 MSC view available during simulation

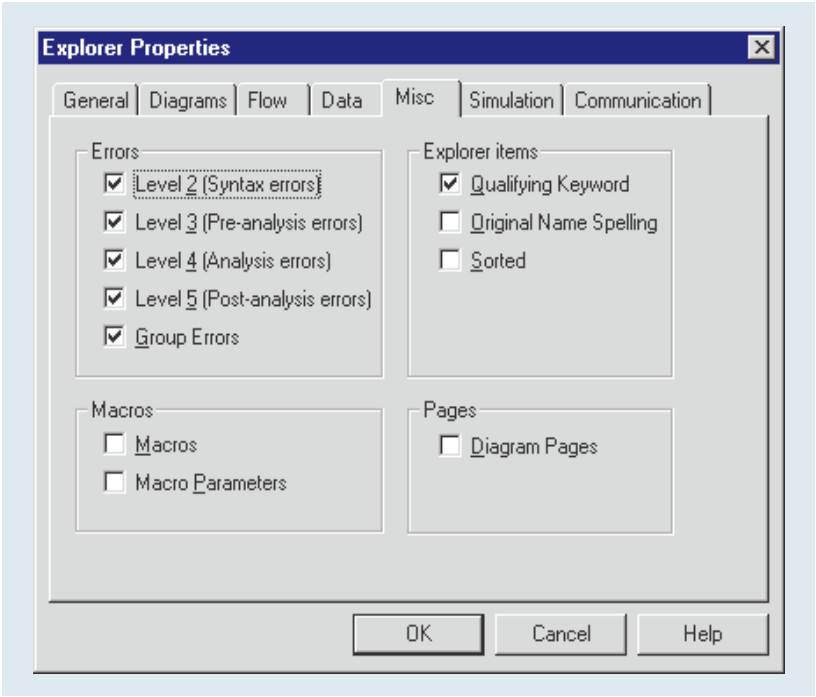


Figure 13 User controlled settings must be available to support different methodologies

4 Design Testing

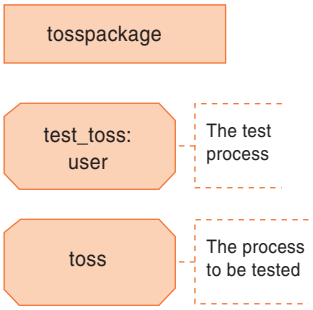
Design testing is the activity to validate that the developed system model satisfies the system requirements identified during analysis. Cinderella SDL allows you to define SDL test processes to validate your SDL design and use the simulator to check the system requirements. In order to increase the value of the validation the definition of the SDL design model and the test processes ideally should not be performed by the same group of system/test designers.

Although SDL is not a dedicated test notation like e.g. TTCN, it may still be useful for SDL design testing. Some of the benefits when using SDL for design testing are:

- The same notation is used for system design and for test definition.
- There are no requirements for an explicitly defined test architecture, as the SDL-recommendation defines the rules for communication between an SDL system and the environment of the SDL system.
- The testing can be used for 'black-box' testing as well as 'white-box' (module) testing.
- It is easy to check exchanged signals and variable values as the same notation is used for the system specification and the test specification.
- Test development and design validation can be done as an integral part of system specification.
- The test specification serves as documentation for the design validation performed, and for the system tests they may be the basis for the test specification to be performed on the implementation.

The SDL test process is defined in a separate specification, which includes the system specification or the selected parts of it. When the test process is simulated in combination with the

Figure 14 The test system for testing the toss process



(partial) system specification, the result may be checked using the MSC view. As the system specification file(s) are linked to the test specification, it is easy to apply the test again if the system has been updated.

In the following example, we illustrate the principles of test processes using the *toss* system described in Section 2. We want to define a test case that checks that the score counter is correctly updated according to the result of sending an initial *Probe* signal. If the Win signal is received the getCount procedure shall return 1 and –1 if the *Lose* signal is received. The test specification is shown in Figure 14. The test process *test_toss* is an instance of the user process that resides in a linked file. In this way the test specification will always use the latest version of the toss specification.

The test process instance *test_toss* consists of a single test procedure *tc_1* that tests the counter. The process and procedure diagrams are shown in Figure 15. The variable verdict will have the value true if the simulation of the test is success-

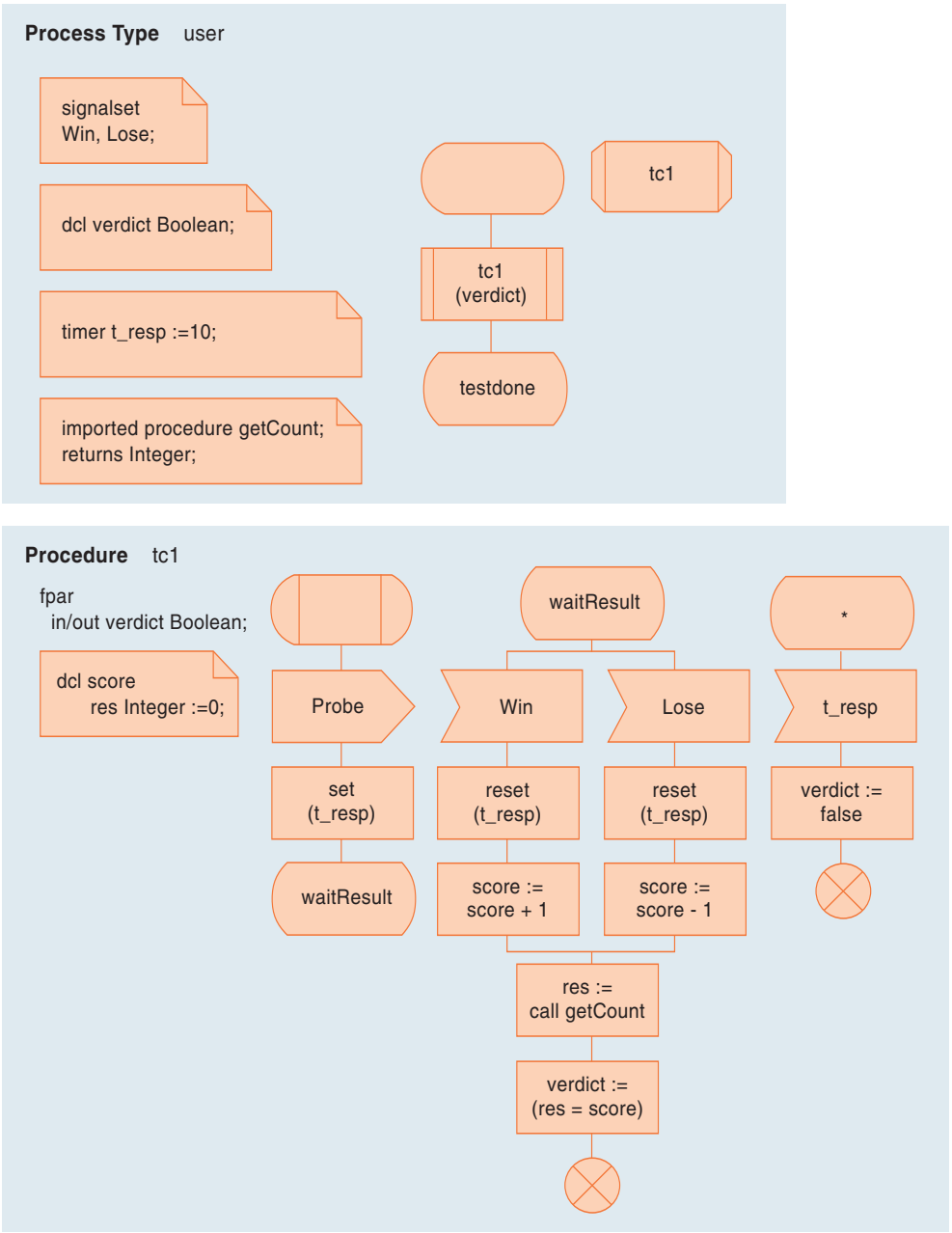


Figure 15 The process type user that test_toss is an instance of and the test procedure tc_1

ful. The timer *t_resp* is introduced to ensure progress of the simulation if the tested system does not respond with an expected behaviour. The test process behaviour may be extended with call to additional test procedures, allowing for a complete test suite to be executed in one simulation.

When a simulation of the test system is performed the signal exchange may be viewed as an MSC. Figure 16 shows the MSC view of a successful test simulation of test system for the *toss* process.

This example only illustrates one approach to use SDL for testing an SDL specification. Obviously, there is a number of ways in which the support for simulating combined SDL systems and test processes can be utilised to validate a system specification and document the validation performed. For example, different parts of an SDL specification may be included in different test systems to do module testing of these parts. The designed SDL test processes may also be used as a basis for implementation of the test for the implementation of the system.

5 Implementation Generation

After finalizing the system design, defining the test environment and testing the system design against the test environment, the next step is to generate a system implementation. An implementation for the test environment may also be generated and used for testing the system implementation after it has been deployed.

To generate an implementation, you need a code generator that is customized to meet your requirements to the target environment. Due to target environment dependencies (e.g. C++/Java, CORBA/threads/DCOM/RMI), code generation is not a built-in feature in Cinderella SDL. To have code generation support, you should either:

- Generate SDL-PR and use one of the existing commercial PR-based code generators;
- Use one of the existing commercial code generators which can be “plugged-into” Cinderella SDL by means of the API;
- Build your own “plug-in” code generator, based on the source text skeletons that come with Cinderella SDL.

You can make your own plug-in by utilizing the SDL API, which comes with Cinderella SDL. Through the API you have access to all the internal SDL structures in the tool – both those representing the SDL syntax tree, and also the entity descriptors that contain all necessary information about SDL entities.

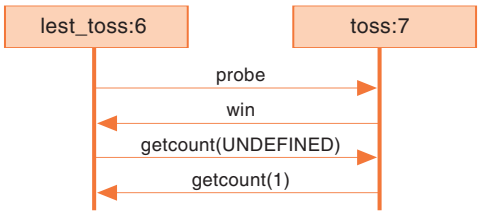


Figure 16 An MSC view of the result of a successful test simulation

6 Conclusion

Cinderella, which was founded in 1995, announced the first release of Cinderella SDL in April 1998. Since then, a large number of users have downloaded Cinderella SDL, evaluated it and given valuable feedback. A considerable number of developers were not familiar with SDL beforehand. They used Cinderella SDL to get familiar with SDL and the tool in a low-cost way, only investing their time.

Due to the amount of new features in SDL-2000, a similar situation may occur when Cinderella SDL version 2.0 is released – existing SDL-92 users and new users will use Cinderella SDL to get familiar with SDL-2000.

However, since the tool will support the new SDL-2000 standard in a way, which is appealing both to traditional SDL users, UML users and to users new to formal specification languages, the target of Cinderella SDL is much broadened.

The tool is expected to be available in the second quarter of 2001, together with a supplementary MSC-2000 editor. These future releases will be based on the same principles as for the current release of Cinderella SDL. This means support for different system development methodologies and options to configure the tool according to user preferences.

However, evolution does not stop here. In a few years, it is likely that a merge of SDL, MSC and UML will take place. It is the responsibility of the tool vendors to assure that new evolutions do not create new problems in terms of backward compatibility or restrict the development methodologies that may be used for system development.

For more information on Cinderella SDL and other Cinderella products, please visit www.cinderella.dk.

The Evolution of SDL-2000

RICK REED

In November 1999 SDL-2000 became the latest international Recommendation of the ITU-T in force for specification and description of telecommunications systems and standards replacing the previous version. SDL is a language that has evolved to meet changing years over a period of more than a quarter of a century. This article recounts the history of SDL-2000, tracking the previous versions from 1976. An account is given of the latest changes to the language, followed by the author's opinions of the direction of evolution for the future.

Rick Reed's (53) work in software support systems (see page 20) led to his participation in the ITU group on CHILL 1977–80. His work on software methods from 1982 onwards led to his participation in the ITU SDL group while still in ITU-T SG11. He was associate rapporteur for data 1984–1988 and for methodology 1992–1996, and rapporteur for the whole of SDL 1996–2000. For the 2001–2004 study period he is chairman of the Modelling Languages Working Party, which includes SDL and MSC. Rick Reed has been head of the UK delegation since 1988. He has been involved in several projects at ETSI concerning SDL. He was a founding member of the SDL Forum Society, its first Treasurer, and as Secretary he is now actively organising the 10th SDL Forum for 26–29 June, 2001.

rickreed@tseng.co.uk

1 History of SDL

In 1968, "stored program control" (SPC) systems were just coming into use for telecommunications. CCITT (replaced by ITU-T in 1993) decided that its Study Group 11, which dealt with signalling and switching, should assess the impact of SPC on telecommunications standards. At the end of the 4 year study period in 1972, the result was to launch studies on languages for human machine interaction (called at that time "man-machine language" – MML), specification and description, and programming.

1.1 SDL-84 Evolution

The study continued on the specification and description language (SDL) and the first Recommendation (23 pages containing some symbols for "state transition diagrams", definition of concepts and a few examples) was published in the CCITT Orange book in 1976. It was assumed that the rules for use and semantics were intuitively understood, as many organisations were using different forms of state transition diagrams. The 1980 CCITT Yellow book had the first real description of today's SDL in 72 pages. This was still very informal but recognised that the need to find errors at the specification stage and to provide good tool support, the SDL drawings needed their meaning to be defined more formally than in the 1976 Recommendation.

The metamorphosis from a graphical drawing technique with loose semantics to a formal description technique took place in the following four years, so that the 1984 CCITT Red Book contained an interpretation model that treated the drawings as the basis of mathematical graphs. Additional language features were added, so that the drawings represented process graphs (with several concurrent process instances, which might be created or stopped dynamically), and such that branching in the graphs and passing information by signals between processes depended on data. The basic language of states and transitions triggered by signals between processes did not change, but the 1984 version enriched the language with many features. Data and a textual form were added to the language. User guidelines were published.

1.2 SDL-88

The increasing importance of software was reflected in the creation of Study Group 10 (Languages for telecommunication applications) which between 1984 and 1988 improved the formal basis for SDL. An added incentive was given by work on support tools. The semantics were defined in an abstract way independent of whether the concrete language was the textual Phrase Representation (PR) form, or the Graphical Representation (GR) form. The infrastructure (abstract syntax, textual and graphical grammars, semantics and models) of the current SDL-2000 [1] and previous SDL-92 [2] Recommendations were established. Work was accelerated to complete the formalisation in two years by early 1987. The SDL-88 Recommendation [3] in the CCITT Blue book was about 200 pages (not including the formal definition in VDM or the user guidelines). During this period the data definitions part was aligned with the evolving ISO standard LOTOS [4] using the same algebraic model. This required few changes to the syntax of the language, and gave data a sound mathematical basis.

SDL-88 is the foundation of all the subsequent versions. Articles [5], [6], [7] or [8] provide tutorial material.

1.3 SDL-92

At the start of the next period, 1988 to 1992, there was a requirement to keep the language stable, but it was also realised that there were major user benefits if the language could allow for re-usable objects. The main evolution in this period was the addition of object features as an extension. A **block** (or **process**) **type** can be used to define a different **block** (or **process** respectively) at each place where it is used.

Remote procedure calls and non-determinism were also added to meet user needs. The SDL-92 Recommendation [2] was about 10 % larger than the SDL-88 standard [3]. Significant work was also done in the period on methodology, which is a difficult area to handle within the framework of standardisation since many aspects are organisation dependent. The result was issued as methodology guidelines to replace the user guidelines of the Blue book. There is an overview of SDL-92 in [9].

1.4 SDL Combined with ASN.1

After 1993, ITU encouraged publication of Recommendations at any time. Work was progressed rapidly on using SDL in combination with ASN.1 in Z.105 [10].

ASN.1 was widely used for defining data structures on protocol interfaces: that is protocol data units conveyed by messages between systems: in standards and sometimes in real systems.

ASN.1 allows data values to be defined, but does not define any operators between those values. For example, ASN.1 defines the Integer values 1 and 2, but does not define "+" or any way of writing expressions and therefore cannot be used to describe behaviour. SDL can be used to describe structure and behaviour, and in particular can define operators for data. Z.105 brings these two worlds together, and an overview was published in [11].

Z.105 was approved in March 1995 [12], and as far as possible was an extension of SDL-92.

Z.105 allowed data used within SDL processes and in signals to be defined using a subset of the ASN.1 notation. This then also implied that ASN.1 encoding can be used, at least for signals to and from the SDL system. This is an advantage over SDL, because SDL does not define encoding of data.

Another advantage is that ASN.1 has been widely used to define protocol data units conveyed by messages between systems: in standards and sometimes in real systems.

1.5 SDL-96

Apart from the development of Z.105, stability was the major objective in the 1992 to 1996 period, and allowed tools to catch up with the Recommendation.

A common graphical interface between tools was standardized in Z.106 [13], and a new methodology framework document was produced as Supplement 1 to Z.100 [14], based to some extent on work done (but never published) by ETSI. Some extensions to SDL-92 were agreed to meet user needs, generally relaxing rules of SDL-92 to make it easier to use and some corrections were made to the language definition where it was unclear, ambiguous or inconsistent. These were collected into the 35 pages of Addendum 1 to Z.100 [15] which together with SDL-92 defines what became known as "SDL-96". The texts of Z.100, Z.106, Addendum 1, and Methodology Supplement were approved for publication in 1996.

SDL-96 was essentially a superset of SDL-92 which in turn was a superset of SDL-88, so that valid SDL-88 was still valid in SDL-96, except in a few obscure cases that were unlikely to arise in practice. SDL-96 can therefore be considered as SDL-92 with some corrections and a few extensions. The extensions simplified SDL by harmonizing concepts and improved the expressive power of SDL. There were no changes to the underlying models.

1.5.1 Harmonized Communication

Remote procedure definitions, remote variable definitions all defined communication primitives, but in SDL-92 remote procedures and remote variables could not be used:

- to show the communication on channels and gates;
- for communication with the environment of the system.

The use of remote procedures and remote variables was harmonized with signals, such that whenever a signal can be mentioned, a **remote** procedure or **remote** variable can also be mentioned (with a few exceptions that do not make sense). If a **remote** procedure or **remote** variable is mentioned on a gate or communication path of a diagram then the diagram need not include an **imported** specification. To use **remote** procedures and variables for communication with the environment, they must be mentioned on a channel connected to the environment.

A related improvement was to allow a signal list identifier (in brackets) in input.

The possible clash of names between signals, **remote** procedures and **remote** variables is resolved by taking first a visible signal with the name. A procedure (or variable) with the same name as a signal must be preceded by **procedure** (or **remote** respectively).

1.5.2 External Behaviour

SDL-96 enabled the use of procedures and operators defined outside the SDL-description by adding the keyword **external** at the end of the operator or procedure heading to show that the body of the procedure is defined external to the SDL.

1.5.3 Simplified Paths

Channels have names that are used in **output via**, and to connect a path outside a diagram when the inner diagram is drawn separately (that is it is **referenced** – the term "linked" is used in this article). When an inner diagram is the instantiation of a **type**, the outer path is connected to the inner gate name and the outer path name is not needed. SDL-96 allows the names to be dropped when not needed, and similarly signal lists do not have to be repeated on both the outer and inner path.

In SDL-96 the use of channel and signallist names in diagrams can be restricted to just those places where they are needed (that is if they are referred to in some part of the description), or if they serve a useful explanatory function.

1.5.4 Paths to Self

In SDL-92 it was not allowed to draw a communication path from one process instance-set back to the same one, or from a block-set and back to itself. Therefore, it was not possible to denote the communication between one member of the set and another. This restriction was removed in SDL-96. So that the construct is not ambiguous, it must be a one-way communication path, but two or more distinct one-way paths are allowed so by this means two-way communication is possible.

1.5.5 Agents as Systems

SDL-96 allows a service, process, or block to be considered as a system with the surrounding constructs (that is for a service the process, block and system) implied. This simplifies the specification of simple systems, and makes it easier to consider a larger system as a number of communicating smaller system

1.5.6 Extended Use of Packages

The SDL-92 restriction that packages can only be attached to packages or the system diagram was removed. Packages can be attached to any diagram. This has the advantage that the visibility of the items in the package can be restricted to just those diagrams where they are used.

1.5.7 State Expression

An additional imperative operator, STATE, was added, which returns the name of the current state as a charstring expression.

1.5.8 Nullary Operators

These are operators without arguments. The implicit ordering rules of SDL for ordering literals do not apply to nullary operators, therefore they are useful for defining sorts of data with the **ordering** property. Otherwise, they are the same as data literal values.

1.5.9 Common Interchange Format (CIF)

SDL has had both a textual phrase representation (SDL/PR) and a graphical representation (SDL/GR) since 1984. The two forms have a large textual part that is common to both, but graphical constructs also have an equivalent textual representation. Originally, the objective was to provide a form like a programming language to enable SDL to be more easily processed and analysed by computers. At that time processing of graphics was slow and suitable equipment was expensive. SDL/PR was designed to be machine processed, and therefore although it is readable, its grammar is not very "user-friendly".

SDL/PR found other roles as enabling SDL to be interchanged between tools, and as an intermediate language within tools supporting SDL/GR. However, if SDL is transferred from one tool to another using SDL/PR, the diagrams are often unrecognisable because graphical layout information is lost.

The objective of the CIF is to enable SDL to be transferred between tools and be "recognisably the same". CIF is an extension of SDL/PR that contains the additional graphical information in comments of SDL/PR.

1.5.10 Methodology

This is not part of the Z.100 standard: the 1992 guidelines are in an appendix and are therefore *informative* (as compared to an *Annex* such as Annex F Formal definition which is normative). The methodology associated with Z.100 can only suggest approaches, but SDL can be used validly in many other ways. The work done in the 1992–96 period on methodology is published in Supplement 1 to Z.100 – "SDL+ methodology: use of MSC and SDL (with ASN.1)".

The SDL+ methodology did not replace the 1992 guidelines, but provides a more detailed framework that can be elaborated for a particular use to develop a formal specification in SDL. It covers to some extent MSC and use of ASN.1, and can be applied from the level of requirement capture. It suggests the use of the OMT object model notation for forming the initial classes. An overview, published in [16], was presented at the SDL'95 Forum.

1.6 MSC and UML

The Message Sequence Chart (MSC) language was first standardized in Z.120 in 1992 as MSC-92, although they had been suggested since 1984 as auxiliary diagrams to be used with SDL. MSC tools now exist that are linked to SDL tools.

MSC has continued to evolve, and a variant has appeared as part of the UML set of notations in OMG.

Ivar Jacobson of Rational, who is well known for use cases and promoting UML, participated in the SDL group in the early 1980s and acknowledges the common origins for ideas in his work and in SDL. However, he ceased to participate in the group before the creation of the MSC standard and the development of SDL as it is today. In fact, SDL-2000 and MSC-2000 could easily be taken as UML profiles by OMG.

1.7 Recommendations in Force and Availability

The standards in force at the end of 1999 related to SDL were:

- SDL: Z.100 (11/99) [1] and Supplement 1 (10/96); [15];
- SDL combined with ASN.1 modules: Z.105 (11/99); [17];
- Common interchange format: Z.106 [13];
- SDL with embedded ASN.1: Z.107 [18];
- SDL combined with UML: Z.109 [19];
- Use of FDT's: Z.110 (11/99) [20].

The 11/99 standards were available as the English drafts immediately to members of the Study Group and the SDL Forum Society, and in pre-publication form via the ITU subscription service (ITU-T Recommendations Online) mid-December 1999. The published English version may differ in page numbers and minor changes in wording, but the technical content should be the same.

2 The SDL-2000 Standard

SDL is a living language, which means it is used frequently with new applications in new areas. If SDL were a natural language, new applications, features and vocabulary would arise, while some language constructs would become unused, unfamiliar or even unknown. Similarly SDL users have new needs, and have innovative ideas for changing SDL. However, SDL is a formal language that needs to be well defined and machine processable, so that all users and tools can understand and manipulate designs in SDL. Changes to SDL need to be carefully managed. Formally, this was the task of ITU-T Study Group 10 under Q.6/10 in the period 1996 to 2000, but the work was done in close collaboration with the SDL Forum Society.

SDL is quite complex. This means that there was plenty of scope for clarifying the language definition so that it is not misinterpreted, and there were still some ambiguities, inconsistencies or just defects in the language (that is "bugs") that needed correcting. The study therefore specifically included defect correction. Maintenance also includes the possibility to change the language, and for SDL the rules for maintenance (including the change procedure) were defined in Addendum 1 of Z.100. Similar rules are now part of SDL-2000.

The scope of the study question for SDL-2000 was (phrases copied direct from Q.6/10):

> *... corrections, ease of use and ease of maintainability, new uses of SDL – especially in conjunction with other languages, and simplification – both by decommitting non-used features of SDL and by combining similar concepts of the language.*

SDL is mainly used for implementing systems. More focus was therefore given to constructs for using SDL for design and implementation.

2.1 SDL Simplification and Maintenance

There were differing opinions on the precise meaning and scope of "simplification" objective for the 1996–2000 study period: it could just be deletion of some language concepts, or it could extend to modelling blocks and processes (and the SDL-92 *services*) all as variants of some building block object. The definition was adopted that change meant "removal of unnecessary restrictions and differences in language concepts and perhaps unnecessary, unused features". Simplification has encompassed both the above changes.

Z.100 for SDL-2000 consolidates Z.100 (03/93) [2] with Z.100 Addendum 1 (10/96) [15] and many features of Z.105 (03/95) [12] into one Recommendation [1]. This also incorporates the Master List of Changes maintained by the Q.6/10 experts' group according to the rules Recommended in [15].

As noted above, the maintenance rules allowed features to be removed and the following features were deleted:

- Alternative block partitioning (that is the possibility to give two different versions of a block, one where the block is described using processes and one where the block is described using a block substructure);
- Structural graphical macros (that is macros in block diagrams);
- Behavioural graphical macros;
- View/reveal;
- Channel substructure;
- Signal refinement;
- Axiomatic data type definition.

Taking such a step is difficult because there is always a call for backward compatibility. A procedure for deleting features was defined and followed. Essentially, it was possible to remove these features because they were seldom used.

The service feature was not deleted from the language but was harmonized with blocks, processes and composite states, so the keyword service is no longer used.

2.2 New Features in Z.100

A long list of open items was considered for inclusion in SDL-2000, ranging from trivial items such as allowing the keyword **call** to be optional, to major changes such as the introduction of block identities similar to Pids.

The new features covered by Z.100 for SDL-2000 are:

- Exceptions;
- Textual algorithms;
- New data model;
- Nested packages;
- Mixing blocks and processes;
- Type based creation of processes;
- Typed Pid sorts;
- Interfaces;
- PR/GR harmonization;
- Composite states;
- Object modelling support.

These features are described briefly below, with the exception of object modelling support, which is related to UML and is treated in 2.4.

2.2.1 Exceptions

Exceptions were a specific topic of study and a short requirements list was:

- Handle run time errors (which implies there are language defined errors);
- Compatibility with Z.130 Object Definition Language [21];
- Avoid any conflicts between exceptions in SDL and in other languages (such as IDL/ODL, Java);
- User defined exceptions and causing exceptions;
- Provide a mechanism to enable non-responding **remote** procedures to be trapped.

It was also recognized that there could be some improvement in the handling of timers.

The new mechanism provides the ability to raise an exception, which can have a handler not considered to be part of the usual behaviour. This is an important feature for the use of SDL with the ITU ODL and with CORBA.

2.2.2 Textual Algorithms

The mechanism allows algorithms to be written textually within graphical diagrams. Most users prefer SDL/GR most of the time, but in some situations, SDL/GR leads to additional diagrams, unnecessary complexity, unnecessary indirection and lack of conciseness. SDL/PR is not a good alternative, because it is verbose, tool oriented and may still require indirection. Textual algorithms introduce a user-oriented mechanism for combining text with SDL/GR. It was already supported by one tool before the SDL-2000 was approved.

2.2.3 New Data Model

There have been a number of problems associated with the SDL data model based on the algebraic approach using ACT ONE. In addition, the SDL-92 inheritance model for data was not consistent with other types in SDL. The algebraic approach to define new kinds of data has not proved popular with users, and it is considered difficult to use and difficult to implement fully. Most users used only the language-defined kinds of data, and wanted data similar to object oriented programming languages. It was therefore agreed that the data model for SDL could be changed to support new features, to have typing similar to other types in SDL and to support most of the current use of data in SDL-92.

The specific rationale for change was:

- It had not been possible to properly incorporate error! in the existing ACT ONE model;

- The algebraic approach for defining new sorts of data was difficult to master and it is difficult to make efficient implementations from ACT ONE;

- The existing model produces some undesirable language rules (such as not permitting a struct field as an in/out parameter).

The new model brings SDL data more in line with the other object/type features of SDL (**block type**, **process type** etc.). It makes data in SDL easier to understand for someone who is familiar with data in a (so-called) object oriented programming language such as C++ or Java.

SDL-92 data models (**newtype** sorts of data in the old terminology; "**value types**" in the new terminology) are still valid in SDL-2000, except where they rely on full implementation of axiomatic descriptions.

value types in SDL-2000 are either the language-defined types, or composite **value types** (such as Array, Struct, String ...) constructed using other non-composite or composite types. Operators can still be defined with a **value** type (as is possible in SDL-96), with the body of the operator described in algorithmic SDL or as **external** (as allowed in SDL-96).

The SDL-2000 model introduces "**object types**" for data that can refer to "**value types**" and have polymorphic operators and methods.

Methods introduced with a type are applied to instances of the type using the dot notation familiar from other languages. For example:

counter.increment(10) /* counter is a variable and increment is a method */

Although user definition of axioms in the new model is not supported, the built-in data types still have a description that uses axioms. These data types therefore still have the formal description as before. Other data types are constructed from these types and the features of SDL-2000.

2.2.4 Nested Packages

Nested packages are allowed. This provides a more flexible packaging mechanism, and makes it easier to mix SDL with some other languages.

2.2.5 Mixed Blocks and Processes

The restriction that blocks and processes could not be mixed in one diagram is removed. Many users have requested this change, and there seems to be no good reason for maintaining it, especially without block partitioning.

The harmonization has gone further with agents as a concept that covers the system, blocks and processes. Processes can also be nested within processes, and the service concept is replaced by composite states that are state aggregations (see 2.2.10).

When a process is defined within another process definition, then each instance of the outer process contains a number of instances of the inner process. All these instances are scheduled to run in an alternating way, and the inner process instances can access the data of the outer process instance directly.

When a process definition is directly contained within a block, instances of the process run concurrently, and concurrently with any other instances in the block, including the instance that owns any variables of the block. Processes within a block access block variables by implicit remote procedure calls to the instance for the variables.

2.2.6 Type Based Creation

An agent instance may be created from an agent type. In SDL-2000, it is not necessary to have an explicit definition of the agent instances in the case that there is only one set in the context of the creation and there are no initial instances. Also because no instance name is needed in this case, create can be used in another process type which would be impossible otherwise.

2.2.7 Typed Pid Sorts

These are similar to the Pid sort, except that the actual Pid value must belong to the correct process type. The benefit is additional security and confidence that a Pid actually identifies a process of a specific process type when used as a destination after to for an output, **import** or a remote procedure call.

2.2.8 Interfaces

An interface is a type that contains the definition of a number of signals, remote variables and remote procedures and may be associated with channels, gates, connections or signal sets. Because an interface is a type it can have context parameters. An interface is a scope unit for the contained definitions, which nevertheless are visible outside the interface (like operators defined with a sort of data). An interface can inherit from one or more other interfaces.

2.2.9 PR/GR Harmonization

In SDL-92 there are some inconsistencies between the text in SDL/PR and the text in SDL/GR, where the same grammar could be used in both cases (for example punctuation and whether items are optional). These were made consistent wherever possible.

2.2.10 Composite States

This introduces a hierarchical state mechanism. One benefit is the ability to hide sub-states within a super-state, so that the details are hidden and it is easy to have an overview of process behaviour. Another benefit is that when a design is elaborated, a state may be split into sub-states without losing the original state.

A composite state can consist of just one set of sub-states.

2.3 Incorporation of Z.105 and the Data Model

Use of ASN.1 is strongly related to the data model of SDL. As it was agreed to change the underlying data model of SDL-92, the Z.105 Recommendation had to be updated, because it relied on the algebraic model.

The old Recommendation Z.105 introduced some inconsistencies between SDL with and without ASN.1. These inconsistencies are removed in Z.100 for SDL-2000. One consequence is that SDL-2000 is case sensitive, which also has some benefits when using SDL with other languages (for example, SDL is often translated by tools for implementation into C which is case sensitive).

Consideration was given to incorporating Z.105 completely into Z.100, but the view was taken that Z.105 should remain a separate document describing how the ASN.1 syntax can be used with SDL. However, it was also agreed that some functionality of Z.105 should be incorporated into Z.100 as SDL concepts so that choice, optional and default fields are now part of SDL.

Z.105 (11/99) is a mapping of ASN.1 to Z.100 (11/99) features, so that a data type specified in ASN.1 has an equivalent specification according to Z.100 (11/99). This is not the case between SDL-92 and Z.105 (03/95): for example, SDL-92 allows "[" and "]" in names whereas Z.105 (03/95) does not, and CHOICE required some change to the SDL model.

Another change has been to separate the use of ASN.1 modules with SDL, and the definition of ASN.1 embedded in SDL. The use of ASN.1 modules as SDL packages is still within the scope of Z.105 (11/99)[17], but ASN.1 embedded within SDL is now in the Z.107 (11/99) Recommendation [18].

2.4 SDL with UML

Partly because of some common heritage (see 1.6), it was always the case that UML and the ITU set of languages had similarities and complemented each other. Even before UML existed, it was recognised [16] that object modelling was needed in conjunction with SDL. The suggested technique (in [14]) was OMT, the basis of object modelling in UML. There are small differences between MSC and UML sequence diagrams.

UML defines language meaning allowing some semantic variations, and leaves open the notation, so that various different notations may be used. The ITU standard for SDL defines both the notation and the meaning. UML allows a set of notations to be considered together. There seems to be no reason why SDL cannot be considered to define state machines within UML.

On the other hand, the Z series Recommendations had no language for class diagrams and there was no reason not to adopt the widely accepted UML notation for this. By making this notation part of SDL, it binds the object models closely to SDL and also makes the notation readily available to SDL users not otherwise using UML.

The standard Z.109 goes one step further and details the use of SDL combined with UML. It defines a UML profile for SDL, and in that way defines how UML can be mapped to SDL. Of course, this mapping only applies for that part of UML that is relevant to SDL [9]. The sequence diagram part of UML would map to MSC.

One change to SDL has been to allow the reference symbols that link to SDL diagrams (such as block type, process type, and procedure) to be shown as generic class symbols: a three partition box for the object identity, its attributes and behaviour. These can be used wherever a linking symbol is valid. A further change is that multiple occurrences of a these linking symbols are allowed in the same context, provided they are consistent. This enables an object model at one level to be presented as a number of pages of an SDL diagram showing different associations on different pages.

SDL is also extended to show more relationships between objects. The communication paths and creation relationships have always been part of SDL. SDL-2000 has named association symbols (lines with various ends such as arrows and diamonds) with text at each end describing a relationship between two objects. The usual object models of UML are now part of the SDL syntax. The SDL standard [1] puts constraints on the syntax and the use of names, but does not further define the meaning of the relationships. The meaning otherwise defined by the SDL standard should be changed by the inclusion or deletion of associations: they do not have any impact on the SDL defined semantics.

3 Open Issues for Further Study

When the first version of SDL was produced, some of the originators thought that the language was finished and no changes would be needed. In retrospect, it is easy to understand why this was not the case, and why there has been user demand for more features and new paradigms in the language. On the other hand, the basic graphical notation and the extended finite state machine (EFSM) paradigm at the core of SDL have stood the test of time. Even in 1980 it was difficult to foresee the relative low cost and high performance of commodity computers available in 2000, that can run complex SDL tools. It is therefore difficult to predict precisely how SDL will develop through the first century of the third millennium.

The best guess for the future is that SDL will continue to exist with EFSM at its core. The language is still evolving and use is growing.

The ITU-T study programme in the short term is addressing for 2001 the open issues of revised formal definition of SDL-2000, revised Common Interchange Format (CIF) for SDL-2000 and a methodology update. There is an ongoing study of time and performance issues that could lead to changes to SDL, or some way of linking requirements such as time deadlines to SDL models. There needs to be a binding of MSC data to SDL, and there is a need to be able to define encoding of SDL data on interfaces.

One known area of difficulty is the composition of services on a mix and match basis during system implementation. For example, with three telecommunication services (A, B and C), it should be possible to define the base system, the service A, the service B and the service C. It should then be possible to generate an implementation for any combination. That is: base, base+A, base+B, base+C, base+A+B, base+A+C, base+B+C, base+A+B+C. It is a methodology issue to define how this can be done (if it can) in SDL-2000. If it cannot be done, it is an unresolved language issue.

The previous examples are known user needs, but even in the relatively short period of 1996 to 1999 the focus, general environment and expectation changed so that the issue of object modelling changed from being considered as an auxiliary notation, to requiring integration into SDL. It is therefore difficult to predict what SDL-2004 may or may not include. It is only certain that it should be user driven: it is proposed that language development is carried out by the SDL Forum Society (www.sdl-forum.org) and the results submitted to ITU-T for approval and publication. It is easy for individuals or organizations to participate in the SDL Forum Society, which has its objectives focussed on SDL/MSC and can be flexible. The ITU-T by comparison has authority and the infrastructure for publishing Recommendations and maintaining them over a long period. There is the opportunity to utilise the strengths of both organisations.

Other possible issues to be tackled for the future are the development of IP protocols in SDL and the adoption of MSC with SDL as a UML profile. IP in SDL is likely, because of the telecommunications over IP projects that are in progress, and MSC-2000 with SDL-2000 technically meets the UML profile requirements, whether officially recognised or not.

The SDL Forum Society

The SDL Forum Society is a non-profit making organization formed in order to promote the Specification and Description Language (SDL) and Message Sequence Chart (MSC).

The Society has existed since June 1990 and, as well as promoting the knowledge and usage of MSC/SDL and providing information on the development and use of SDL/MSC, one of its major functions is to promote and organise the SDL Forum which takes place once every two years.

The SDL Forum provides an opportunity for experts, users, toolmakers and even some critics of MSC and SDL to meet, engage in useful discussion and socialise. It is called a 'Forum' because it is a 'meeting place' for the exchange of ideas and a chance for people to meet others who are involved in the use of MSC and SDL. The plans for the Tenth SDL Forum are already under way, and it will be held in Copenhagen in June 2001.

The SDL Forum Society also supports the SDL and MSC Workshop – SAM – and this addresses three topics: language issues; the relation to other Languages or Techniques; Design, Methodology and Applications. Those involved in the workshop include researchers, users of the languages, and members of standardization bodies. The event is less formal than the SDL Forum. It enables intensive discussion of the languages, and evolution of ideas on the future development and application of SDL and MSC.

When joining the Society one of the benefits is discounted access to either the Forum or the Workshops, and discounted copies of the proceedings. There are also free updates on ITU-T activities related to MSC/SDL including access to ITU-T documents on studies in progress. The Society has recently been given permission from the ITU-T to distribute a version of the SDL-2000 and MSC-2000 Recommendations via its web site.

Other activities have included a Grant for student work on related research, and specific support for the continuing evolution of SDL and MSC, changing to meet current and future needs and technologies.

If you would like any further information on the Society, including information on how to join and details of the inexpensive membership fees, visit the web site at http://www.sdl-forum.org.

Some SDL models are available for sale as components and this market may increase. One factor is that the major SDL tool supplier has about one third of the world real time development market (in all fields, not just telecommunications). SDL-2000 should give excellent support for producing components. With the continued growth of the telecommunications market and increase in SDL use in other areas, an SDL component market seems likely.

References

1 ITU-T. *Specification and Description Language (SDL).* Geneva, ITU-T, 2000. (Z.100 (11/99).)

2 ITU-T. *CCITT Specification and Description Language (SDL).* Geneva, ITU-T, 1994. (Z.100 (03/93).)

3 ITU-T. *Functional Specification and Description Language (SDL).* Geneva, ITU-T, 1989. (Z.100(1989).)

4 ISO. *Information Processing Systems – Open Systems Interconnection – LOTOS – a Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* Geneva, ISO. (ISO/IEC 8807.)

5 *Introduction to SDL-88.* (2000, September 06) [online] – URL: http:///www.sdl-forum.org/sdl88tutorial/

6 Bræk, R. SDL Basics. *Computer Networks and ISDN Systems,* 28, 1585–1602, 1996.

7 Færgemand, O, Olsen, A. Introduction to SDL-92. *Computer Networks and ISDN Systems,* 26, 1143–1167, 1994.

8 Sarma, A. An introduction to SDL-92. *Computer Networks and ISDN Systems,* 28, 1603–1615, 1996.

9 Møller-Pedersen, B. SDL-92 as an object oriented notation. *Telektronikk*, 89 (2/3), 71–83, 1993.

10 ITU-T. *Data networks and open system communications – OSI networking and system aspects – Abstract Syntax Notation One (ASN.1)*. Geneva, ITU-T. (X.680-681.)

11 Verhaard, L. An introduction to Z.105. *Computer Networks and ISDN Systems,* 28, 1617–1628, 1996.

12 ITU-T. *SDL Combined with ASN.1 (SDL/ASN.1).* Geneva, ITU-T, 1995. (Z.105 (03/95).)

13 ITU-T. *Common Interchange Format for SDL.* Geneva, ITU-T, 1997. (Z.106 (10/96).)

14 ITU-T. *Use of MSC and SDL (with ASN.1).* Geneva, ITU-T, 1997. (Supplement 1 (04/96) to Rec. Z.100 (03/93) SDL+ Methodology.)

15 ITU-T. *Corrections to Recommendation Z.100 (03/93).* Geneva, ITU-T, 1997. (Z.100 Addendum 1 (10/96).)

16 Reed, R. Methodology for Real Time Systems. *Computer Networks and ISDN Systems,* 28, 1685–1701, 1996.

17 ITU-T. *SDL combined with ASN.1 modules.* Geneva, ITU-T, 2000. (Z.105 (11/99).)

18 ITU-T. *SDL with embedded ASN.1.* Geneva, ITU-T, 2000. (Z.107 (11/99).)

19 ITU-T. *SDL combined with UML.* Geneva, ITU-T, 2000. (Z.109 (11/99).)

20 ITU-T. *Criteria for the use of formal description techniques by ITU-T.* Geneva, ITU-T, 2000. (Z.110 (11/99).)

21 ITU-T. *Object definition Language.* Geneva, ITU-T, 1999. (Z.130 (02/99).)

Perspectives on Language and Software Standardisation

AMARDEO SARMA

Amardeo Sarma (45) received his B.Tech. degree from the Indian Institute of Technology, Delhi, in 1977 and his Master's degree (Dipl.Ing.) from the Technical University of Darmstadt in 1980, both in Electrical Engineering. In 1981 he joined the Research Institute of Deutsche Telekom AG's Technology Centre, where he was Head of Research Groups 1990 – 1995. He participated in several projects dealing with signalling, protocols and specification techniques. In 1995 he joined EURESCOM as Project Supervisor in the area of software technologies, middleware and ATM. In 1999 he returned to Deutsche Telekom as Head of Department for Technology and Methods Management, focusing on strategic selection and promotion of technologies and methods. Within ITU-T he has been involved in Study Group 10 since 1987. His current special interests lie in R&D strategies and business development, and his focus lies in the areas of broadband networks and services, specification languages and formal methods.

sarma@telekom.de

This paper presents the history of formal languages and software standardisation in the International Telecommunication Union (ITU) and their use in various application areas. It looks both into applications within standardisation bodies, and into the increasing use in industry. By looking at other, sometimes competing, languages, a view is developed into what the key differentiating features of standard ITU languages are. Special emphasis is placed on the Specification and Description Language (SDL) and Message Sequence Charts (MSC), as the two driving forces in the ITU language family. Finally, a perspective is offered on which critical factors may determine future development and use.

Introduction

Recently, someone approached me about techniques and tools for specification, testing and validation. I asked whether he had considered TTCN or knew about SDL and MSC, and the response was that these were old-fashioned concepts for the non-IT world. This was not the first time that I sensed the underlying opinion that we should dump the old telecom stuff and look at these great things from the IT side instead, as it was they that would provide the solutions of the future.

I agree with one side of the story. Cheap computers, Information Technology and the Internet are opening a whole new world of opportunities and ideas. There are great new ideas from the IT industry that we should keenly look at. So on this count the opinions may well be justified. On the other hand, telecommunications has produced a large number of solutions and products that do not need to be reinvented in a new context. This also applies to languages and software. Organisations such as OMG recognise the contributions that individuals and companies from the Telecom side can make in the software area.

History

The era of telecommunication software began for the predecessor of ITU-T in the late sixties, CCITT, when computer-controlled switches began to emerge as a replacement for earlier electro-mechanical switches. Whereas the latter were hardwired and less troubled by logical errors, a need was seen to use techniques to make the software of the new switches as reliable as the old hardware. Industry and the operators required the same sort of robustness for programme-controlled switches and for the protocols that influenced the behaviour of the switches. CCITT embarked on defining languages that would guarantee these quality requirements.

Initially, three domains were identified, where the use of languages and software techniques should apply:

- For the specification and description of what these switches and protocols were supposed to do. The aim was to reduce error and increase robustness. SDL (current version [1]) was the answer to this, though the SDL of the time was quite different from the language today.

- For programming switches able to handle complex protocols and functions, for which the existing languages were considered inadequate for distributed switches able to process complex protocols. The answer to this was CHILL (current version [2]).

- To describe what one would call a GUI these days. For this, the Man Machine Language (MML) [3], [4], was considered the appropriate answer.

These languages were developed initially in a Working Group of Study Group XI, which later became a separate Study Group, Study Group X, and finally Study Group 10.

In parallel, the languages ASN.1 [5] for describing protocol formats and TTCN [6] as a testing language were developed jointly by ITU and ISO. Here Study Group 7 was the driving force.

In the 90s, Study Group 10 developed Message Sequence Charts [7] as a formalised and much more extended version of time sequence diagrams. This language was especially useful to show scenarios and to show examples of the flow of signals between parts of a system or different systems. In 1999 finally, the ITU Object Definition Language (ODL) was defined as an extension of the Interface Definition Language IDL from the Object Management Group (OMG).

Coverage of Languages by ITU-T

SDL, MSC, TTCN and ASN.1 enjoy success and wide areas of application. While MML seems dwarfed by modern GUI techniques, it should be recognised that MML were used in many products and provided a lot of valuable guidelines for user interface design. Z.361 [8] from this Study Period can be considered a continuation of the MML work. MML provides line-based dialogues, menus, form-filling, windowing (i.e. GUI) and online help. CHILL has been successfully applied by telecom vendors to develop many products and had a large installation base during the 1980s, but has not spread much outside the telecom domain. Since most companies used proprietary CHILL compilers and tools the market for commercial CHILL tools became too small to face the competition from languages like C, PASCAL and C++. However, CHILL-2000 is a modern object-oriented language for real-time systems, see the article by Jürgen Winkler in this issue.

Most of the above mentioned ITU-T languages are either already object-oriented or plan to become so soon, thereby being part of the contemporary object-oriented paradigm. The move towards object-orientation was not always easy, since the large number of existing users always placed a strong requirement on backwards compatibility.

Study Group 10 has also dealt with areas such as Human Machine Interface (HMI) data, Quality issues, Methodology, Guidelines for the use of languages, etc.

Since the mid-90s, ITU-T Study Group 10 has intensified its co-operation with ETSI MTS (Methods for Testing and Specification) and the SDL Forum Society. With human resources for standardisation activities becoming increasingly scarce, this co-operation has led to fruitful results for all parties. ETSI MTS has been especially active in the area of ASN.1 and TTCN, and ITU has adopted much of the work done there. The SDL Forum Society has been able to address a wide range of users of SDL and MSC and has thus been able to promote the languages in ways normally not open to an organisation such as ITU. The SDL Forum Society is organising Workshops and Conferences on the ITU-T languages.

Use of ITU-T Languages

Standardisation Bodies

Use in other standards was the main motive for developing SDL, ASN.1 and TTCN. A glance into many of the existing Q series Recommendations by ITU shows the widespread use of SDL, though this use has often been quite informal. Standardisation bodies, such as ETSI and ITU, offered tool support for their Rapporteurs to be able to create “good” SDL, MSC, TTCN and ASN.1. ETSI went even further, providing professional support for SDL, ASN.1 and TTCN via a team of employed experts, the permanent expert team, who provided invaluable help to the Rapporteurs.

The dual use of SDL and MSC as both informal graphical techniques and as formal languages has helped the acceptance of these languages. In a first step, they may be used as informal graphical techniques that help engineers to express their ideas in a systematic way. This in itself provides greater clarity than plain text could provide. Initial uses of SDL for service descriptions (I.200 series) or protocols (e.g. Q.931) were of this informal kind, allowed SDL to gain acceptance for use within the body of the Recommendations. This paved the way for more formal uses of SDL and MSC, which became apparent when tool support could be provided, not only to draw diagrams, but to check for consistency and help in validating the specifications.

The use of SDL in the Intelligent Network domain is an example of a success story in using “formal” SDL. With IN’s Capability Set 2, the formal use of SDL in standards began in ETSI with Frans Haerens recognising the value of the formalism available. His approach turned out to yield the results expected: Far from being a formal nuisance, the development of IN CS2 protocols allowed the concerned to reason about their specifications and detect errors early. The result was not only a machine-readable SDL specification, but also a reduction in time to develop the recommendation while simultaneously detecting more errors than was possible in the earlier version, CS1. The success has been taken up by ITU using SDL in a more formal way, as well.

Industry

In the 80s, manufacturers of telecommunications equipment realised the potential, not only of the programming language CHILL, but of other ITU-T languages as well. AT&T (now Lucent), SIEMENS, Nortel, ECI, Alcatel, Ericsson and Nokia are examples of companies that have ventured on their use. Some, such as SIEMENS and AT&T, developed their own tools. Others used commercially available tools, such as those from the market leaders Telelogic and Verilog, now merged into a single company. It turned out that standardisation bodies on their own would never have offered a market large enough for good, commercial tools to develop. It is this “other” industrial market, not the original focus of the language standardisation that provided the critical mass for the tool vendors.

Today, SDL and MSC are widely used in the telecommunications industry, especially in the mobile phone business, and have been shown to cut development time, improve quality and reduce cost. Beyond the telecommunications industry, the automobile and aircraft industry are further examples where SDL and MSC are used, especially when dealing with distributed, communicating systems.

The IP and Internet area provides potential large application areas for ITU languages. This use has been taken up in the ETSI project TIPHON, and here is a potential for the definition of IP-based protocols. Many of the requirements expressed for IP-based protocols are no different in principle from traditional telecommunications protocols.

The take up of the ITU-T languages by industry has been and will remain critical for their overall success. It therefore remains a crucial issue to identify new areas and domains where the unique strengths of the ITU-T languages can be used (see following section). Wherever distributed and interacting systems exist, there is an opportunity for ITU-T languages to play a role.

Comparison with Other Languages

An early motivation for developing programming and specification languages within ITU and ISO was to provide formal languages that were independent of short-term interest of specific big companies or vendors. The development of internationally standardised languages ensured stability and maintenance. This argument still remains a critical issue for some companies that state this as a reason for using ITU-T languages in preference over others.

Specifically, the ITU languages SDL and MSC also used a unique mix of features to address their particular market. These are:

- A graphical syntax in addition to a textual syntax, allowing users to visualise the behaviour of systems and processes in an intuitive way.

- Formal semantics that provides the basis for ensuring that the systems are well-defined, less susceptible to reasoning errors, and provides the means to ensure consistency and correctness.

- Intrinsic capability for verification and validation of the systems, that follows as a result of the formal semantics, which makes it possible to simulate the behaviour of systems before they are actually implemented and built.

- Conceptual suitability for distributed communicating systems, like those encountered in the telecommunications domain, and in other domains, as well.

- Implementability, in the sense that efficient implementations can be derived from specifications.

- Commercial tool support by independent vendors and a common interchange format allowing portability between tools.

Many of these features are provided by other languages, but not in totality. For example, LOTOS [9], Estelle and Petri Nets provide a formal basis with many of the advantages listed. But LOTOS and Estelle do not have a graphical basis that is easy to read and understand or they moved towards a graphical syntax too late, while none have the crucial commercial tool support. Even the graphical notation of Petri Nets does not follow the thinking pattern of engineers, the main users that are addressed by SDL and MSC. Tools provide translation facilities between ITU-T languages and implementation code, for example generating C or C++ from SDL. SDL thus becomes suitable as a specification, design and implementation language.

ASN.1 and TTCN, on the other hand, address very specific and unique needs encountered in the telecommunications domain. They address a clearly defined and sustainable niche that other techniques and languages do not address as well. They too have adequate tool support, and clearly address the needs of the involved engineers. ASN.1 is used to define the syntax of protocol data, while TTCN is used to specify test suits – which may be generated automatically from SDL specifications and MSCs.

ODL was developed within ITU as a superset of IDL because the organisation responsible for IDL (OMG) did not comply fully with the needs of the telecommunications domain (see also the paper on “Object Definition Language” by Marc Born and Joachim Fischer in this issue). The work in this area is however done in close co-operation with OMG to make sure that any maintenance of ODL does not come into conflict with the needs of the users.

GDMO [10] (Guidelines for the definition of managed objects) is developed by the International Standardization Organisation (JTC 1/ISO TC 97/SC 21/WG 4) and has been used within ITU-T in the TMN area. It provides the definition of information models for Telecommunications Management Networks (TMN), defining information at the Q3 interface. The original GDMO has an alphanumeric syntax only. The

methodology for the use of GDMO is contained in Recommendation M.3020 [11].

Within ITU, two lines of work were started on GDMO. One was to provide GDMO with a graphical syntax, and the other to enrich GDMO with behaviour based on SDL. These activities were stopped due to lack of commitment to use the approaches. Perhaps the added value of having a graphical syntax and a formal behaviour component for this language was not apparent enough. Today, CORBA IDL is increasingly being seen as an alternative to GDMO. For a while, GDMO was also seen as a possible candidate for data descriptions within SDL, as the abstract data type (ADT) formalism used in SDL-96 was seen as rather academic and was scarcely used in practice. Today, ASN.1, which is closely related to GDMO, is used jointly with SDL as described in Recommendation Z.105 [12] [13].

The Unified Modelling Language (UML) has received much attention. The language was initially created as a merger of various popular object-oriented techniques for analysis (e.g. OMT) and later standardised by OMG. It has gained wide acceptance and is now supported by commercial tools. Various large companies now use UML and the often-associated Unified Process as their internal development standard.

ITU-T encourages joint use of SDL and UML to make maximum use of the strengths of both. The paper *SDL Combined with UML* by Birger Møller-Pedersen in this issue, which corresponds to the title of the ITU Recommendation Z.109 [14], addresses this topic.

This joint use of SDL and UML is just one of the examples of using ITU-T languages in combination with each other or with non-ITU languages. Recommendations Z.105 and Z.107 that allow joint use of SDL and ASN.1, are further examples. Beyond these standardised combinations, it is up to the tool vendors to support further combinations, for example of SDL, MSC, ASN.1, TTCN and UML.

In future, joint use will be supported at three levels. Tool vendors will provide support for the joint use of languages as required due to the immediate demand of their customers, including the need for code generation to programming languages. Whenever the demand turns out to be of a more general nature, ITU-T will seek to provide standard guidelines as part of a methodology document or go even further and issue Recommendations for the support of further combinations, such as the joint use of SDL, MSC, ASN.1, TTCN and UML. The approach for the future should be pragmatic, catering for the level of the requirements of the users of the languages.

Tools

When we discuss the prospects of languages and software in standardisation, it helps to compare the success stories with the failures. The real key to success appears to me to lie in the commercial use and tool support. Users need a simple and efficient way to use the languages, and they need to be able to do so faster and better, to be convinced that the languages are useful. If they do not have efficient language-sensitive editors, checking and validation facilities, as well as the ability to eventually generate a running system, a technique is bound to gain little support.

This is in my view clearly the reason why CHILL remained a niche language, even though the niche consisted of several major companies. Companies using CHILL preferred to keep their "competitive advantage" over others by not sharing or selling their tools, and tool vendors did not see a perspective because their prospective users had their own tools. Non-proprietary tools were not able to make ground over the proprietary ones used by the major companies involved and failed to convince a broad base of users outside these companies. Tools for a language that intends to have a major impact must be available even to students at universities at competitive cost.

The same I believe is true for the ISO languages LOTOS and Estelle, as well as for Petri Pets. Though these languages were well conceived and may even have been partially superior from a theoretical point of view, their lack of commercial tools support have confined them to being an academic exercise. Also, LOTOS and Petri Nets have not been used in a distributed environment in the whole cycle from specifications to *efficient* implementations. Being thus unsuitable as a design and implementation language and not having intermediate supporting techniques towards implementations is the reason for their relative failure.

This brings me to a central point in the strategy for a language or a family of languages to be successful: The availability of tools for SDL, MSC, TTCN, ASN.1 and ODL is the issue for their future survival and well-being. The day a language fails to be supported by powerful and commercially available tools at reasonable price and with a perspective for the future, the death sentence for the language has been spoken. The more market and competition there is for providing tools to the customer, the brighter the prospect of success.

A Perspective on Standardised Languages in the Coming Years

At the start of a new Study Period, as now for the Period 2001 – 2004, it is now time to re-assess the need for activities on language and software standardisation. In the competitive environment of today, where clear needs must be identified, and nice-to-have is just not enough, a sober assessment is essential. The current situation is:

- With the growing IT orientation for telecommunications, ITU-T Recommendations will increasingly require the adoption of specification languages in the Recommendation text, especially SDL and ASN.1, but also TTCN and MSC.

- New areas, such as IP-related protocols, have come into focus for ITU languages, and this tendency will increase over the coming years.

- Many companies continue to prefer ITU languages, and the fact that the languages are standardised is seen as providing a substantial benefit.

- Results from the TINA Consortium need standardisation. ODL is an example of standardisation within ITU based on TINA and OMG results Other areas, such as the standardisation of DPE, are continuing and this has been identified as an important issue for the future activities in ITU. Whether or not other results from TINA are taken up will depend on the market demand for them.

- The vendors of tools supporting ITU languages, especially SDL tools at the moment, are experiencing unprecedented growth, indicating that there is a strong market.

As a result, there seems to be a clear case for continuing work on standardising languages and generic software. This work is in my view best done in a single Study Group within ITU-T. By bringing the work done on all ITU languages into a single Study Group, we will achieve the required critical mass to pursue effective work of high quality to the benefit of all the languages. The synergy by utilising the capacity of language experts of various fields has already been demonstrated by taking up TTCN within Study Group 10. Distributing this scarce resource over various standardisation groups would diminish the effectiveness of work.

However, the ITU community must also react and properly address current developments in other areas, such as on UML, IDL and XML. SDL and MSC experts must continue to maximise their efforts to make concepts from ITU-T languages a part of the next major UML release, UML 2.0. It has been very helpful for all sides that users of SDL and MSC as well as tool vendors have been active within OMG by addressing issues related to the strengths of ITU-T languages. This is often a very practical question. Several issues that now come up within UML have been addressed for a long time within SDL and MSC and become parts of the languages. They need not be invented again, but can rather be reused in the context of other languages.

At the same time, efforts to develop SDL 2004 and MSC 2004 should continue to keep pace with technology and contemporary trends. Whether SDL and MSC concepts at some future point become part of an all-embracing UML, or whether specialised languages continue to exist for a long time, is an open issue. Many previous efforts to find a single, all-embracing language that solves all problems, have failed. Some time ago, some enthusiastic proponents of the language thought that SDL might fulfil this role. It has turned out again and again that generalisation may reduce the key strengths of a language. Rather than diversify and try to solve all problems, the development in the last years have rather concentrated on the specific strengths of the language and made them the best solution for a particular application domain. The strategy for ITU-T languages thus remains: Concentrate on the strengths and remain the best solution for the particular domain, and at the same time co-operate with other languages that have their strengths in other areas. Do not try to beat them on their battlefield. That is bound to fail. Rather provide frameworks for the joint use of languages to give the customer, and not one or the other language, the maximum benefit.

As a result, ITU will continue the work and maintain its languages. As separate languages, SDL and MSC will be maintained to cater for use within and outside ITU-T, addressing all sides of their current customer base. The need remains to provide support for protocol specification within the telecommunications domain, considering that a large number of Recommendations and standards use SDL and MSC. Far from weakening, this use has gained momentum in the past few years, showing a growing need of customers for ITU-T languages.

At the same time, ITU-T should even more strongly take industrial needs into consideration, and thereby make use of other organisations that support these languages. One such example is the SDL Forum Society, which organises Conferences and Workshops on SDL and MSC and promotes these languages to a wider audience than ITU-T could access on its own. It is becoming clear that standardisation bodies as the only

users alone will no longer be able to provide the critical mass needed to keep work on SDL tools going. A clear focus on industrial customers is essential to keep a wide interest in the languages alive.

Apart from the users, tool support, especially commercial tool support, is the crucial item for ITU-T languages, and I think it is fair to say the statement applies to any language. Tools must be powerful and cheap at the same time, and must therefore address as wide an audience as possible. Here, a clear trade-off must be made. It is not too good to have too broad a scope, which means that the language is not particularly good in any specific domain. On the other hand, a language should not be so specific that it addresses a very small market only, which makes tools too expensive and does not allow for the economies of scale to provide high power and low price at the same time. The latter may be a major reason for the relative failure of the programming languages CHILL and ADA, as well as for the specification languages LOTOS and Estelle. All have very clear technical benefits and are perhaps even superior to today's success languages, but good and cheap tools that were widely available have never supported them. Specialisation alone has not been the dominant problem. The problem was to create a broad enough customer base to make commercial tools viable.

Questions for the Study Period 2001 – 2004

Q1/10:	Quality assurance, methodology and use of description techniques (revised)
Q2/10:	ODL: Object Definition Language (revised)
Q3/10:	Software Platforms and Middlewares for the Telecom Domain (revised)
Q4/10:	Unified Modelling Language (UML) Combined with ITU-T Languages (new)
Q5/10:	Encoding of data in SDL (new)
Q6/10:	Specification and Description Language – SDL (revised)
Q7/10:	Time Expressiveness and Performance Annotations in ITU-T Modelling Languages for SDL and MSC (new)
Q8/10:	Testing Languages and Validation based on Formal Models (new)
Q9/10:	Message Sequence Chart – MSC (revised)
Q10/10:	Data binding of Specification and Description Language (SDL) to Message Sequence Charts (MSC) (new)
Q11/10:	DCL: Deployment and configuration Language (new)
Q12/10:	URN: User Requirements Notation (new)
Q13/10:	Quality Aspects of Protocol-related Recommendations (new)

Outlook

Some strategic issues have already been addressed, such as the need for a focus on industrial customers, the availability of powerful and cheap commercially available tools and the concentration on the strengths, not the weaknesses of the languages. For the latter, the strategy should be co-operation with other languages that are strong in these areas. UML is the most important language in this context, and the combined use of UML with ITU-T languages is one of the new Questions identified within ITU-T Study Group 10.

In addition, some shortcomings of ITU-T languages – where other languages do not already have a technological advantage – have been identified for future work. An example of this is the area of time and performance, where activities have been identified for the coming years. Languages for user requirements as well as for configuration and deployment were identified as a new need. Many solutions for data in SDL have been standardised in the past, from abstract data types (ADT) to the integration of ASN.1 within SDL. A new Question will deal with data in SDL.

Middleware and Distributed Platforms remains to be an issue, especially with the prospect of being able to adopt and reuse much of the results emanating from work on TINA in the past.

Finally, Quality has received much attention in ITU during the last years, in particular in view of quality of protocol-related Recommendations. As quality is an area where ITU-T is seen by other organisations, such as the Internet Engineering Task Force (IETF), to have a high level of competence, and since formal languages is one method with which quality can be ensured, this topic has been taken up in two Questions, each with a different view on the subject.

Closing Remarks

The future for languages and software in standardisation I believe lies in the development and maintenance aligned to market needs in close co-operation with tool vendors. The ability to penetrate the IP market will be critical. And if the critical mass of software and language experts and tool designers work together in as close a co-operation within ITU as in the past, ITU will continue to support further work on languages.

Important web addresses that readers are encouraged to look up are:

- http://itu.int
- http://etsi.fr
- http://sdl-forum.org

References

1 ITU-T. *Specification and Description Language (SDL).* Geneva, International Telecommunication Union, 1999. (Recommendation Z.100.)

2 ITU-T. *CHILL – The ITU-T Programming Language.* Geneva, International Telecommunication Union, 1999. (Recommendation Z.200.)

3 ITU-T. *Man Machine Language (MML).* Geneva, International Telecommunication Union, 1988. (Recommendation Z.301 –Z.341.)

4 ITU-T. *Man Machine Language, Data Oriented Human-Machine Interface Specification Technique – Scope, Approach, and Reference Model.* Geneva, International Telecommunication Union, 1993. (Recommendation Z.352.)

5 ITU-T. *Abstract Syntax Notation One.* Geneva, International Telecommunication Union, 1997. (Recommendation X.680.)

6 ITU-T. *The Tree and Tabular Combined Notation (TTCN)Tree and Tabular Combined Notation*. Geneva, International Telecommunication Union, 1998. (Recommendation X.292.)

7 ITU-T. *Message Sequence Chart (MSC).* Geneva, International Telecommunication Union, 1999. (Recommendation Z.120.)

8 *Design guidelines for Human Computer Interfaces (HCL) for the management of telecommunications network.* Geneva, 1999.

9 ISO. *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour.* Geneva, International Organization for Standardization (ISO), 1989.

10 ITU-T. *Information technology – Open Systems Interconnection – Structure of management information: Guidelines for the definition of managed objects.* Geneva, International Telecommunication Union, 1992. (Recommendation X.722 (01/92)/ISO/IEC 10165-4:1992.)

11 ITU-T. *TMN interface specification methodology.* Geneva, ITU, 2000. (Recommendation M.3020.)

12 ITU-T. *SDL combined with ASN.1 Modules (SDL/ASN.1).* Geneva, International Telecommunication Union, 1999. (Recommendation Z.105.)

13 ITU-T. *SDL with embedded ASN.1.* Geneva, International Telecommunication Union, 1999. (Recommendation Z.107.)

14 ITU-T. *SDL Combined with UML.* Geneva, International Telecommunication Union, 1999. (Recommendation Z.109.)

Quality of Service in the ETSI TIPHON Project

MAGNUS KRAMPELL

Magnus Krampell (45) is Project Supervisor at EURESCOM GmbH, supervising projects in the areas of IP/Internet and QoS. He is presently on leave from Telia AB, where he has been employed since 1990. Experiences include: Development of TMN systems, TINA work at Bellcore in USA and research and development of broadband systems and services. Mr. Krampell received his BSc in Mathematics in 1984 and his LicSc in Computer Science in 1988.

krampell@eurescom.de

Introduction

Voice over IP (VoIP) is a service that captures the imagination of many people. It started out as a tool for hobbyists on the Internet, who could talk to each other without incurring long distance costs. And now there is probably no traditional telecommunications operator who does not offer VoIP services in some form, and to some community.

The ETSI TIPHON project has been working on standardisation of VoIP issues (the project is now on its fourth year) in the very changing environment of IP networks. An aspect that was included right from the beginning is Quality of Service (QoS), which was identified as a major factor in the VoIP service offer.

Very early, TIPHON defined four Quality Classes, each with a different requirement on the carrying IP network and each (presumably) being priced differently. However, over time there have been underlying problems, which has prevented these Quality Classes from being deployed in a large scale.

This paper discusses some of these problems and the efforts made by the TIPHON Working Group 5 to overcome them.

QoS Issues

For QoS to be a deployable (and sellable) property of a VoIP network, a number of issues need to be solved. Some of these are:

1. The customer must be able to decide which Quality Class to select. A possible solution would be to only let customers decide when they order the service. Several research projects are working on solutions where the customer selects in real-time. In a previous article in *Telektronikk* [1], some principles for how this can be done are outlined. A EURESCOM project has investigated solutions which are close to these ideas. Some documents describing results of that work can be found at [2]. The TIPHON group has not addressed this issue explicitly.

2. The VoIP application needs to be able to signal to the network its QoS requirements (and the network needs to be able to signal back the achieved QoS). The TIPHON project has done significant work on the first issue, as we shall see below. The second issue is targeted for the near future.

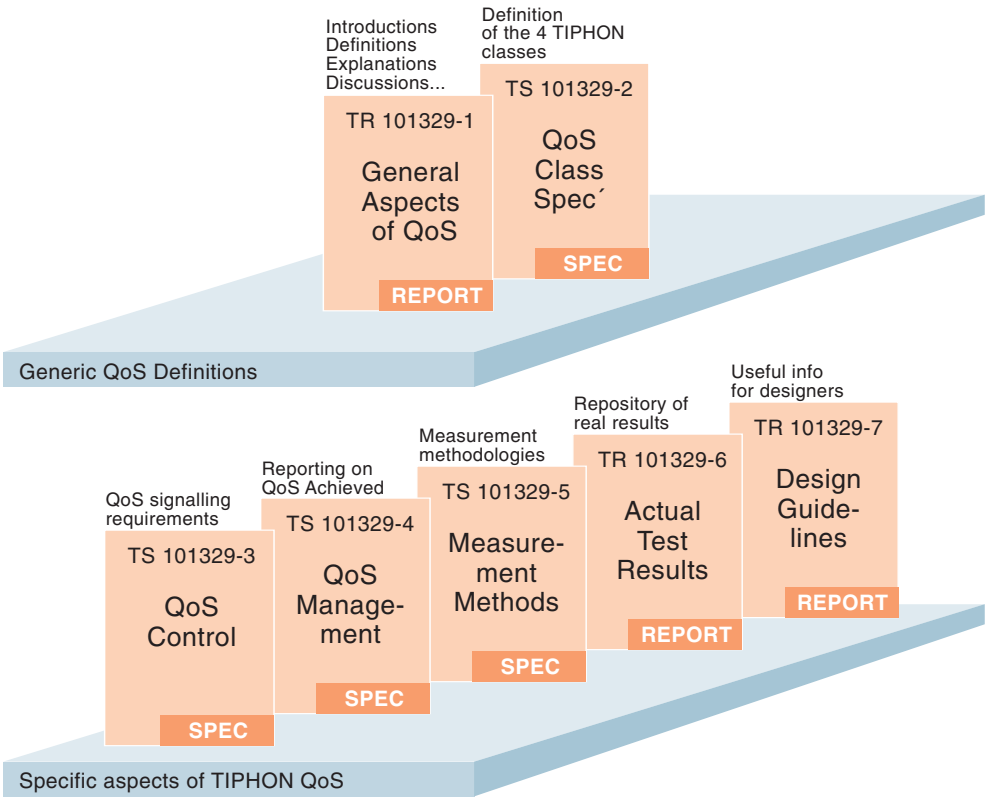
3. The Network needs to be able to provide differentiated levels of Quality. While TIPHON has been active, the IETF (Internet Engineering Task Force) has worked on solutions for this (e.g. the DiffServ architecture [3]), which is now becoming available in equipment from major manufacturers. However, the components of DiffServ need to be put together into services that are provided end-to-end. Work on this issue is ongoing in many places. The outcome of this work may affect the TIPHON solutions.

4. Networks belonging to different operators need to be able to agree on how traffic traversing from one network to another should be treated. The EURESCOM Project EQoS [4] has looked at this issue. (Results from that project related to TIPHON are presented in another place in this issue of *Telektronikk*). While related to the issue described in the item above, inter-provider issues are wider than the inter-network issues. Work on these issues is also ongoing in several places (also in EURESCOM).

5. The network and service need to be manageable, i.e. a management system must be in place, so that network management systems can perform traffic control actions and (when this fails) customers can be informed when the requested quality level cannot be reached. This is an important area and the TIPHON group will do some work here.

6. How to measure the different aspects of QoS is also an area where complete understanding is lacking. The TIPHON group provides valuable contribution to this subject. But, as is indicated below, there is much more to be done. Maybe there is a possibility for TIPHON to take onboard more efforts in this direction. If so, this would be appreciated. If there are no such possibilities, the outcome of work in other groups will hopefully complement the work done by the TIPHON group.

Figure 1 TIPHON WG 5 document structure



TIPHON Working Group 5 Documents

Those who have followed TIPHON WG 5 for some time may remember the first version of the document TR 101 329. This was a good document, but it contained many things. Because of this it was issued as a TR – Technical Report. However, an analysis revealed that some parts were actually specifications, while others were pure comments.

Table 1

TS/TR 101329	Title	First approved	Current version
Part 1	General Aspects of Quality of Service	**TIPHON18** (May 2000)	**V3.1.1**
Part 2	Definition of Quality of Service Classes	**TIPHON18** (May 2000)	**V1.1.1**
Part 3	The signalling and control of End-to-end Quality of Service in TIPHON Systems	**TIPHON21** (Dec 2000 [1])	**V0.9.5**
Part 4	Quality of Service Management in TIPHON Systems	– [2]	–
Part 5	Quality of Service measurement methodologies	**TIPHON20** (Sep 2000)	**V0.2.6**
Part 6	Actual measurements of network and terminal characteristics and performance parameters in TIPHON networks and their influence on voice quality	**TIPHON18** (May 2000)	**V1.1.1**
Part 7	Design Guide for Elements of a TIPHON connection from an End-to-end speech transmission performance point of view	**TIPHON20** (Sep 2000)	**V0.2.1**

[1] *Planned date.*
[2] *No approved document exists yet.*

	4 (Best)	3 (High)	2 (Medium)	1 (Best Effort)
Speech Quality	Better than G.711	Equivalent or better than G.726 at 32 kb/s	Equivalent or better than GSM-FR	Not defined
Reasoning behind Quality Class	Better than today's SCN, i.e. probably conference quality with 7 kHz Codecs	Equivalent to today's SCN	Equivalent to analogue wireless	Equivalent to today's Internet. Usable but possibly degraded
End-to-end Delay	< 100 ms	< 100 ms	< 150 ms	< 400 ms

(2, 3, and 4 are guaranteed; 1 is not.)

Table 2

The working group realised this and created a structure, which is shown in Figure 1. Document 101 329 now consists of seven parts. The first two parts relate to high level generic issues. The other five are more specific in their nature.

A reader should always start with parts 1 and 2, but may then select among the other parts, depending on interest.

Table 1 shows the status of the different documents and when they were first approved.

Overview of Document Contents

This section provides a short overview of the contents of the different documents.

Part 1: General Aspects of Quality of Service

Since QoS is a concept with different meanings to different people, the TIPHON group has gone back to some safe ground and provides in the first part some very basic definitions and descriptions of the scenarios that the project looks at. Even the experienced researcher/engineer should browse through this document.

Part 2: Definition of Quality of Service Classes

One of the highlights of the TIPHON project is that they have defined a set of concrete Quality Classes! Four classes are described in this document, together with some explanation on how they differ, and which parameters affect them mostly. The four Classes are presented briefly in Table 2 (see the document for details).

The concept of Terminal Modes and Network Classes also help to explain how the Quality Classes can be provided using different terminals and networks. The Network Classes are presented briefly in Table 3 (see the document for details).

The document contains a discussion on which of the three parameters (Delay, Jitter and Loss) plus Codec Type are associated with which component (i.e. terminal or network). The conclusion of this is that Codec Type is associated with the terminal only, while Loss is associated with the network only. The remaining parameters (Delay and Jitter) are associated with both the terminal and the network. The "budget" for these parameters must thus be split between the two end terminals and the intermediate network.

Part 3: The Signalling and Control of End-to-end Quality of Service in TIPHON Systems

This is an important document! The ideas presented here are perhaps not revolutionary, but provide a path towards a system architecture, where the application layer can negotiate the Quality Class end-to-end and then negotiate this information to the network. The descriptions of Reference points, Interfaces, Primitives and QoS Parameter Groups provide a very coherent framework. It remains to be seen if it holds true when put to the test in a real system.

As a bonus information, a conversion table is provided, mapping the TIPHON functional elements to functional elements used in other fora. The reader is thus relieved of the effort of trying to find the difference between e.g. a TRM (Transport Resource Manager) and a Bandwidth Broker.

Part 4: Quality of Service Management in TIPHON Systems

This document remains to be written! When part 3 is approved (assumed to happen in November/December 2000), work will commence on this work item. The work started over a year ago, but major changes to the TIPHON architec-

Table 3

Network Class	Packet Loss	Delay Variation (Jitter)
I	< 0.5 %	< 10 ms
II	< 1 %	< 20 ms
III	< 2 %	< 40 ms

ture has prevented work on Service Management to start from a stable foundation. Also the work going into the part 3 document has cleared up many of the open issues related to how inter-provider QoS will be handled. When discussing Service Management (which of course must be done by each provider along the chain of providers a call goes through) the responsibilities of each provider must be clear.

It seems likely that now that these issues are being removed, work on Service Management will provide useful results in a reasonably short time.

Part 5: Quality of Service Measurement Methodologies

Measurements are very much the basis for the management of a service and managing QoS is no exception. One can assume that SLAs (Service Level Agreements) will specify how measurements are being done, in order to avoid disputes when interpreting measurement data related to QoS.

As always, standards are useful in these cases. One would expect that a document like part 5 would contain a complete set of references to different standard definitions that leave no room for misunderstanding.

However, the document in its present version does not really live up to that goal! Regarding Voice Quality, if seems to fulfil the role (even if I am not an expert on Voice Quality matters). On measurement on the Transport Layer (i.e. measurements on Delay, Jitter/delay variation, and Loss) it should probably be regarded as a contribution to the ongoing discussion, rather than as a normative reference document (even if the sections are labelled “Normative”).

As a contribution to a general understanding of Measurements in IP networks it even contains some new aspects. One such aspect is the concept of Packet Loss Correlation, which is a measure of the “Burstiness” of packet loss. The document contains a comprehensive section with material on this subject.

In the cases where the proposed methodology differs from state-of-the-art in other fora (e.g. IETF – the Internet Engineering Task Force) it remains to be proven that the methodology proposed by the TIPHON group is of higher value. This can only be done through practical tests. Results from such tests will be met with great interest.

Part 6: Actual Measurements of Network and Terminal Characteristics and Performance Parameters in TIPHON Networks and their Influence on Voice Quality

As indicated already by the document title, the content is very focused on Voice Quality. A set of measurements has been performed (both subjective as well as objective), which report on the effects of Delay and Packet Loss on the perceived Voice Quality.

Since the nature of packet transportation in an IP network is somewhat different from packet transportation in a “traditional” PCM (Pulse Coded Modulation) system, these measurements are extremely valuable. If heuristics can be built up on how the three parameters (Delay, Jitter and Loss) affect Voice Quality, the perceived quality can be predicted in different situations. The provisioning of the four Quality Classes defined by TIPHON (see part 2 above) can perhaps be reduced to a matter of measuring/monitoring and managing the values of these three parameters in the transport network. The people behind these measurements also assisted in the development of the E-model (ITU-T Rec. G.107/G.109). A motivation for that model is to allow comparisons in a simpler manner.

However, the voice quality measurements are only one type of measurement that is interesting. As was indicated in the section above, results of measurements on the network level parameters will certainly be welcome.

Part 7: Design Guide for Elements of a TIPHON Connection from an End-to-end Speech Transmission Performance Point of View

The four Quality Classes of TIPHON are described in terms of Voice Quality equivalents, and are presented with some values for end-to-end Delay and Packet Loss. As was described in the section presenting part 2 above, the design of a terminal must take into account the target values for the main parameters, and how the budget for them can be split between the terminal and the network.

The part 7 document goes into some detail on e.g. how the delay is a function of codec type and how delay can be split into a set of realistic scenarios (a discussion on how each of the TIPHON Quality Classes can be reached is also given).

Being a designers guide, the documents contain a comprehensive bibliography.

Conclusion

When the TIPHON project started in the spring of 1997, the ambition was to profile (i.e. decide on how to use some of the options) the H.323 recommendation in order to enable interworking between H.323 terminals, H.323 Gatekeeper (where applicable) and terminals connected to the circuit switched network. This sounded like a not too difficult task and the estimated time-table was set at one year.

Now, three and a half years later, there are more open issues than ever. However, TIPHON has changed with the world around it, and has adopted a much more generic approach to how VoIP systems will be deployed on a broad scale.

The TIPHON architecture has changed several times (version 3 is now being finalised), but it is nice to see that the Quality Classes that were defined very early, still seem to be useful. The description of them has changed over the years, but the basic concept remains (i.e. having a discrete set of Quality Classes) and seems to gain increasing industry acceptance.

In Working Group 5 (QoS), the work has been concentrated around a limited set of work items. This approach, together with the division of the original document into seven parts, seems to have worked well.

Readers not acquainted with QoS work in TIPHON are recommended to read the different parts in the right order. Readers with knowledge on the Quality Classes and with an interest in a specific area can find useful material in the individual part documents. However, even experienced researchers/engineers should browse through the first part.

Since my primary interest is in architectures for QoS support in transport networks, I found the part 3 a very interesting document. From reading the other documents as well, I have the impression that the people behind the TIPHON QoS work should be commended for their good work!

References

1 Krampell, M. The ETSI project TIPHON as a way towards a Harmonised User-Oriented Quality Model. *Telektronikk*, 94 (2), 88–92, 1998.

2 *EURESCOM Project QUASIMODO – QUAlity of ServIce MethODOlogies and solutions within the service framework: Measuring, managing and charging QoS.* 2000, November 3 [online] – URL: http://www.eurescom.de/public/projects/p900-series/P906/P906.htm

3 Blake, S et al. *An Architecture for Differentiated Services*. 1998. (IETF RFC 2475 (12/98).)

4 *EURESCOM Project EQoS – A Common Framework for QoS/Network Performance in a multi-Provider Environment.* 2000, November 3 [online] – URL: http://www.eurescom.de/public/projects/p800-series/P806/P806.htm

5 ETSI. *Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON); End to End Quality of Service in TIPHON Systems.* (ETSI TR/TS 101 329 – parts 1 through 7.) [http://www.etsi.org/tiphon/]

QoS and SLA Structure in a VoIP Service Case

IRENA GRGIC, OLA ESPVIK, TERJE JENSEN AND MAGNUS KRAMPELL

Irena Grgic (29) is Research Scientist at Telenor R&D, Kjeller. She is mainly involved in activities related to QoS and charging for different networks and systems, and studies related to network evolution, both in international and national projects. She holds an MSc in Electrical Engineering from the University of Zagreb in 1999. She was Task Leader of Task 5 in EURESCOM P806-GI.

irena.grgic@telenor.com

Ola Espvik (57) is Senior Research Scientist and Editor of Telektronikk. He has been with Telenor R&D since 1970 doing research in traffic engineering, simulation, reliability and measurements. His present research focus is on Quality of Service and Quality Assurance mainly related to various EURESCOM projects. He holds an MSc in physics from the Norwegian University of Science and Technology 1968. He was Project Leader of EURESCOM 806-GI.

ola.espvik@telenor.com

1 Introduction

The EURESCOM QoS (EQoS) framework provides a unified understanding of Quality of Service (QoS) related to all entities engaged in multi-provision [see the "Understanding EQoS" in this article]. A procedure for how to establish a system of Service Level Agreements / QoS-agreements for any scenario has been outlined in [20]. The procedure itself explains the practical implementation of the EQoS generic principles, which results in a set of agreements and relationships relevant for a provider running it. The granularity and variants of input required to run the procedure are also discussed in [20]. The input may vary including information like (1) the description of the service to be provided, (2) selection of relevant QoS parameters (and related values) as a basis for negotiations between the user and the primary provider, (3) the knowledge of potential business/technical configurations, (4) other relevant inputs, e.g. regulatory concerns, economic issues, business strategy, etc.

An application of the procedure was exemplified for the case of Voice over IP (VoIP) provided according to the ETSI TIPHON Scenario 1 [see "TIPHON" in this article]. Several reasons made VoIP service case "win" over other IP-based services, e.g. IP Virtual Private Network (VPN), e-commerce, etc., like:

- VoIP combines the traditional telecommunications and Internet worlds;
- VoIP enables voice, video and data transfer in the same session via packet networks;
- VoIP stresses a universal presence of IP, nowadays enhanced with the introduction of initial traffic engineering mechanisms, and advanced applications;
- VoIP recognises the mature technologies involved in the provision of the voice services over packet-based networks (e.g. Digital Signal Processors – DSPs, codecs, access techniques – high-speed modems, xDSL, etc.);
- VoIP leads to multi-provision, especially when interconnecting to circuit-switched network.

In addition, the material available on VoIP (or more particular IP telephony) achieved in the ETSI Project TIPHON [21] composed a well-established basis for investigating the SLA structure. Although other solutions are feasible in the example elaborated here, Scenario 1 from the TIPHON was considered as a basis. Some other scenarios, such as more "realistic" scenario for today's IP telephony provision – where an Internet Service Provider (ISP) is identified as one of the leading actors – could be discussed. But, since an ISP can be considered as a merger of IPNPs and ICPs, comprising their functionality (see below), the more complex case was chosen for this study. The business relationships, along with the technical realisation of the service for a given scenario are described below.

Before analysing the VoIP service case itself, it would be useful to first discuss the term – what is considered under VoIP? The brief explanation would be "the technology enabling the transmission of voice traffic in packets". More details on VoIP can be found in "Understanding VoIP" in this article.

The necessary service components, the main elements of the system supporting this service, and possible scenarios are discussed in the next chapter. The viewpoints of different actors involved in the scenario studied are elaborated into more details in Sections 3.1–3.4, followed by the content of each of the relevant agreements in Sections 4.1–4.4.

2 TIPHON VoIP – Service, Business Model

The VoIP service is provided to a user, who is connected through an H.323 terminal to an IP network. The service enables the calling user to initiate and receive the telephony service provided over the IP network. The premises for the user are: a LAN supporting the TCP/IP suite realising the access to the public IP network, and an H.323 terminal attached to it. The scenario examined here is TIPHON Scenario 1 [21], where the phone call is originated by an IP-based user (H.323 user), and terminates at a user of the Switched Circuit Network (SCN). The SCN may include both public and private networks like PSTN/ISDN/GSM. As stated in [22] the services provided by a TIPHON compliant system should enable:

- the setup of calls which originate at an H.323 client on an IP–based network and terminate at terminals on PSTN/ISDN/GSM networks;

Terje Jensen (38) is Research Manager at Telenor R&D, Kjeller, responsible for co-ordinating projects in the area of QoS and network design. He earned his PhD degree in 1995 from the Norwegian University of Science and Technology. Other activities include performance modelling and analysis, dimensioning and network evolution studies. He was Task Leader in EURESCOM P806-GI.

terje.jensen1@telenor.com

Magnus Krampell (45) is Project Supervisor at EURESCOM GmbH, supervising projects in the areas of IP/Internet and QoS. He is presently on leave from Telia AB, where he has been employed since 1990. Experiences include: Development of TMN systems, TINA work at Bellcore in USA and research and development of broadband systems and services. Mr. Krampell received his BSc in Mathematics in 1984 and his LicSc in Computer Science in 1988.

krampell@eurescom.de

- backward call clearing;
- forward call clearing;
- detection of a non–recoverable failure of any of the critical resources involved in the call which initiates the clearing of the call;
- user services which make use of end–to–end bidirectional and unidirectional Dual Tone Multi Frequency (DTMF) signalling;
- inability to complete the call within the PSTN/ISDN/GSM network shall be detected and communicated to the calling party (e.g. busy tone);
- ability for inband audio tones and announcements to be received by the caller (e.g. special information tones, referral messages, etc.).

Also, some supplementary services like choosing the level of QoS per call or per subscription, the address translation between IP address and E.164 number including additional information, e.g. calling line identification presentation, malicious call tracing for calls initiated from an IP-based terminal, etc.

The network architecture and reference configuration supporting the delivery of telephone calls originated in an IP network and terminated in SCN includes the following elements[1)] [24]:

H.323 terminal[2)] is an end-user device that provides real-time two-way voice, video or data communications.

GateWay (GW) is the element that ensures the interworking between IP and SCN domains. It consists of three parts – Media Gateway (MG) which provides translation of the transmission formats between the terminals (e.g. G.729 to G.711), Media Gateway Controller (MGC) which provides call handling, controls the MG, receives Signalling System No. 7 (SS7) signalling from SG and IP signalling from GK, and Signalling Gateway (SG) which assures interoperability of signalling between IP and SCN domain (e.g. H.323/H.245 to Q.931).

GateKeeper (GK) is the element responsible for control and management of various elements within the configuration. A single GK can manage a collection of terminals, GWs, and it provides address translation and call control services, as well as resource management.

The **IP access**, **IP network**, **SC network**, and **SCN terminal** (i.e. ordinary telephony set) are also necessary elements, but since they are traditionally present on the market their description is omitted. An important element of VoIP provision is **Back End Services** (BES) (e.g. authentication, billing, address resolution), but in our example they are considered to be distributed to other elements like GW and GK, as described later.

The reference configuration as shown in Figure 1, and described in [25] indicates the reference points relevant when exchanging information between different elements i.e. entities that own them. The reference points relevant for the scenario investigated here are:

A – between the H.323 terminal and its GK;

B – between the H.323 terminal and the GW (MG);

C – between the GW (MGC) and the GK;

D – between two GKs;

Ea – between the GW (MG) and the SCN;

Eb – between the GW (SG) and the SCN.

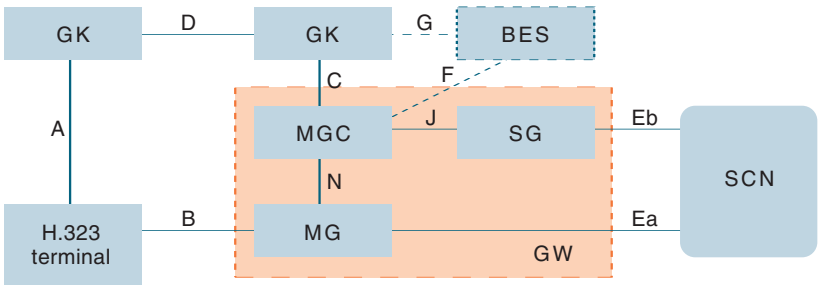


Figure 1 The reference configuration for basic call in TIPHON

1) Detailed lists of functional blocks included in a particular element for Scenario 1 are given in [24].

2) Note that a PC with implemented H.323 and TCP/IP protocol stacks can be used, but the characteristics of it must be considered when evaluating the QoS.

A EQoS Basics

The EURESCOM QoS (EQoS) framework was developed by the EURESCOM P806-GI project [http://www.eurescom.de/public/projects/p800-series/P806/P806.htm], which run from 1998 to 2000. It is a QoS framework generally applicable for multi-provider environment, and it is independent of service, technology, network configurations seen in several service provision situations. This framework may be used also for QoS control, measurement and QoS assessment purposes. Requirements stated by the Open Network Provisioning (ONP) directives issued by EU are considered by the EQoS framework, as well as principles of other existing QoS frameworks.

One of the main achievements is an unambiguous **terminology** where the terms relevant for the service provision in general and QoS are defined. The *Quality of Service* (QoS) is defined as the degree of conformance with an agreement between user and provider. QoS is described through the selection of a set of QoS parameters, specification of QoS target values and the choice of QoS measurements and evaluation mechanisms. A QoS *parameter* is a variable that is used to assess QoS. The above given QoS definition arose as a result of investigating the extensive amount of documentation published by different fora, e.g. International Telecommunication Union (ITU-T), European Standardisation Institute (ETSI), TeleManagement Forum (ex Network Management Forum, NMF), International Standards Organisation (ISO) / International Electrotechnical Commission (IEC) documents, etc. One may say that this definition is a strengthening of the QoS definition given in ITU-T E.800 Recommendation [E.800] as illustrated in Figure A.1. The ITU-T E.800 relates QoS to a user's satisfaction. This allows for a certain level of subjective evaluation taking into account that different users would likely have different expectations and understanding of the service provisioning. In principle, expectation and understanding may neither be completely covered (or covering) the area of satisfaction. Compared to this, EQoS relates QoS to an agreement in order to introduce more objective considerations. There might not be complete overlap between the area of satisfaction and the agreement. As seen from a provider's point of view, the area of satisfaction for an individual user may vary during time, e.g. depending on a user's equipment, mood, etc., making it too volatile to follow closely. Therefore an agreement is used as reference for QoS.

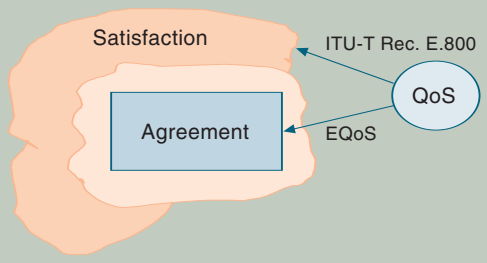


Figure A.1 QoS references (agreement and satisfaction)

The basic set of definitions developed in EQoS includes also the terms: *entity*, *service*, *user*, *provider*, and *agreement*, as explained in the following and depicted in Figure A.2.

An *entity* is a generic unit characterised by its set of states and transitions. A number of entities can be composed into a new entity. Here, the behaviour of an entity is described as seen from an outside observer. An *interface* is a logical boundary between two entities that envelops a set of interaction points. An *interaction point* is a point where two entities exchange information. Here, a boundary should be understood in a wider sense than a physical point. A *service* is the result from executing a set of functions and is provided at the interface. A *provider* is an entity that provides service to another entity. A *user* is an entity that makes use of a service provided by another entity.

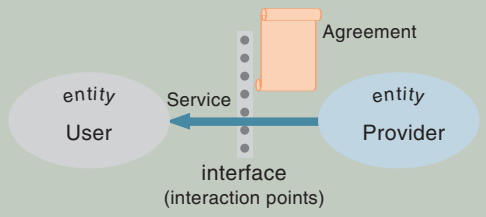


Figure A.2 Provider, user, service, interface, agreement

Apart from terminology, the EQoS achievements include its main concepts, i.e. the one-stop responsibility and its recursive applicability enabling an entity to tackle QoS issues. Simply, in a multi-provision environment, the QoS provided by an entity might depend on adequate operation of other entities. Thus, there may be a need for relating the QoS levels obtained for the different interfaces, see Figure A.3.

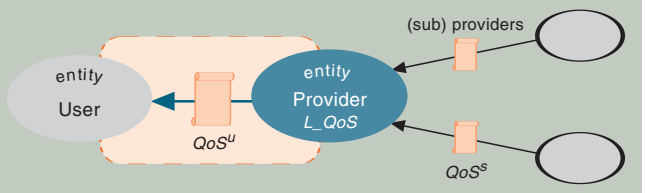


Figure A.3 QoS related to interfaces between (groups of) entities

Naturally, multiple interaction points may exist between one pair of actors in an actual configuration. However, due to the potential dependencies between interfaces, mechanisms for supporting the delivered QoS at the interface may be implemented in a number of ways. In principle, the QoS provided/delivered at the interface between entity Provider and entity User can be formulated as a function *f*:

$$QoS^u = f(QoS^s, L\_QoS) \quad (1)$$

where:

- QoS^u is the QoS experienced by User on the interface between entity User and entity Provider;

- *QoS*^{s} is the QoS experienced by Provider on the interface between entity Provider and entities Sub Providers;
- *L_QoS* are QoS mechanisms involved, which are present "locally" to entity Provider.

It should be emphasised that although dependencies are identified between QoS referring to different interfaces, a single provider is responsible for the aggregated QoS towards a certain user. That is, in Figure A.3, Provider is the main one responsible for QoS towards User, even though this depends on other Sub Providers as exemplified by (1). This is named the *one-stop responsibility* concept. Naturally, a user may see various levels of services (and service components) that may be composed.

The QoS is expressed by assigning values (e.g. target values or actual, measured values) to a number of QoS parameters. Different viewpoints and instances may be referred to, like the requested QoS, offered QoS, contracted QoS, delivered QoS, perceived QoS, and so forth (see [ETR003]). Considering the various phases of service life cycles, various relevance of parameters is assumed for the phases. Furthermore, the actors involved may change their roles as the various service life cycle phases evolve.

A generic structure for interconnection agreements can be identified by describing possible aspects that should be included. One motivation for this is to allow more rapid, accurate and automatic establishments of such agreements. In particular, this is requested because of the multitude of providers that could be involved.

The considered issues include:

- Interface description;
- Traffic patterns expected including relevant traffic parameters and values;
- QoS parameters and related target values;
- Measuring mechanisms and scheme description;
- Reaction pattern to apply in case agreed restrictions on traffic patterns or QoS parameter values are not fulfilled.

Several of the terms (like traffic patterns) could be generalised in order to be applicable for every service life cycle phase. The corresponding terms might then be adapted in order to describe better the relevant aspects.

The actors/entities must also be seen to behave according to the intentions behind the agreement. Therefore, enforcing the behaviour sought for the actors, adequate reactions should be implemented. The nature of reactions would be multiple, depending on the phenomena on which the reaction is to "regulate". That is, load control, charging schemes, legal actions, etc., which all may be covered by the reaction patterns.

More details on EQoS framework can be found in Telenor R&D Note 33/99 [TN3399].

References

E.800 ITU. *Terms and definitions related quality of service and network performance including dependability.* Geneva, 08/94. (ITU-T E.800.)

ETR003 ETSI TC-NA. *Network Aspects : general aspects of Quality of Service and Network Performance.* ETSI Technical Report ETR 003 (ref. RTR/NA-042102), October 1994. (ETR 003.)

TN3399 Espvik, O et al. *EQoS : a generic framework for Quality of Service (QoS). A QoS approach to Internet and IN multiprovison.* Kjeller, Telenor R&D, 1999. (N 33/99.)

The remaining reference points, i.e. G, F, J, N are not relevant considering the assumptions made for this case study.

There are numerous possibilities for the user to be connected to the IP network like: dial up access, LAN, leased line, etc., (ref. [24]), but the scenario studied here is limited to a LAN access via LAN owned by the user. Therefore, the role of the IP Access Provider (IPAP) is omitted here. Also, since the destination SCN may have an impact on the QoS associated to the service [26], the scenario is further restricted to the choice of a particular SCNP i.e. ISDN provider.

The entities taking part in the service provision in the case considered here are identified as actors taking certain roles, and owning certain functionality (Figure 2). Those entities are briefly described below in the sense of the functionality and role(s) they have in the service provision configuration. Note that the symbols on arrows identify QoS agreements (QoSA) to be described using the EQoS framework.

Figure 2 identifies the entities to be considered in the service delivered to the end-user A (see also Figure A.1 in [23]). Note that neither IPAP nor Back End Service Provider (BESP) are identified in the figure.

User A: an end-user, who uses an H.323 terminal for VoIP; the terminal also supports H.225 (call control) and H.235 (security). The user directly accesses the IP network provider through a private IP network, i.e. a LAN.

ITSP (IP Telephony Service Provider): A service provider who offers telephony services over IP networks [22]. The major service component directly realised by the ITSP are the Registration, Admission and Status (RAS) services offered to the user within the gatekeeper function with which the user is registered. The GK

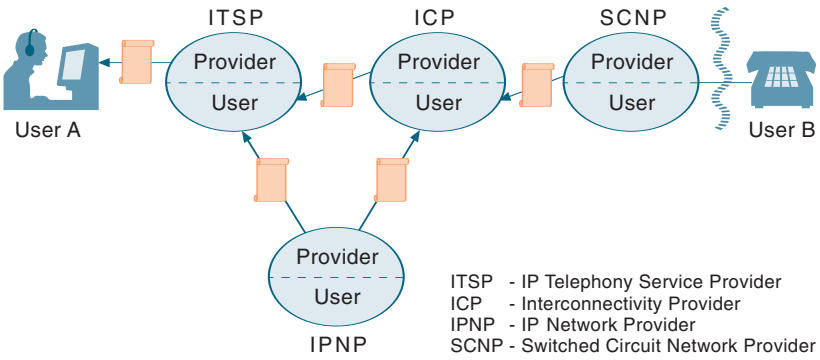


Figure 2 QoS agreements for the IP telephony service

also provides the address resolution functions between H.323 alias addresses and E.164 addresses (this may be delivered by the ITSP, which relies on services provided, e.g. by the SCNP or BESP). Call Acceptance Control (CAC), including path selection and traffic control functions are also realised by the ITSP, although part of them may also be sub-contracted to a BESP. The gatekeeper function in the ITSP may be realised through a "gatekeeper cloud" within which several gatekeepers collaborate. The ITSP also relies on service components provided by the IP Network Provider (IPNP) and by the Inter Connectivity Provider (ICP).

IPNP (IP Network Provider): A provider of IP interconnection between IP service providers and users. It relies on layer 2 network providers to physically realise the IP transfer services. The LAN on which the H.323 user is connected, routes external communications on the IP network controlled by the IPNP. The service offered by the IPNP is IP transfer, including routing and local traffic control in routers. The IPNP may optionally support advanced traffic mechanisms that allow supporting *intserv* and/or *diffserv* IP transfer. Whenever this is the case, traffic control functions are also realised in the IPNP. The ITSP should take care that enough resources are provisioned in the IPNP such that a call that is accepted by the ITSP within its CAC function can indeed be supported by the IPNP with the appropriate QoS.

ICP (InterConnectivity Provider): A service provider who offers services for access between IP and SCN [22]. The main functions realised by the ICP are a gatekeeper and a gateway functions, which provide for real-time, two-way communications between H.323 terminals on an IP-based network and terminals on SCNs. The GW realises the interconnection between SCN and IP telephony signalling functions; it also provides the coding functions that are used to translate IP packets into information streams suitable for circuits.

SCNP (Switched Circuit Network Provider): An entity that provides the PSTN and/or ISDN and/or GSM network services to the ICP. On the other hand, it is also a provider of the telephony service to a User B, i.e. called party, but since the QoS issues at this interface are well described in a number of standards, that relationship is not elaborated here.

An overview of functionality present in each of the entities identified in the chain is given in Table 1.

Table 1 Identification of functionality within entities

Entity	Functionality within entity
Calling user A	H.323 terminal
	Supports H.225 (call control) and H.235 (security)
	LAN
ITSP	GateKeeper (GK)
IPNP	IP transfer (includes IP routing)
ICP	GateKeeper (GK)
	Signalling Gateway (SG)
	Media Gateway Controller (MGC)
	Media Gateway (MG)
SCNP	ISDN signalling
	Circuit switching
Called user B	Telephony terminal
	Supports ISDN signalling

An illustration of both network configuration and business model, with the interfaces (business/technical) to be considered in a particular agreement is given in Figure 3. All roles involved in VoIP service provision/usage for Scenario 1, along with the elements/functionality they own, are depicted in that figure.

The business relationships are indicated with the double-lined arrows with the agreements attached. Technical relationships show the actual flow of traffic/information exchange, both for control (Tc) and speech (Tu) traffic. Note that the illustration of one element, e.g. a router (R), may indicate the actual presence of a number of similar elements.

3 Viewpoints of Different Actors

In this chapter each of the providers involved in the VoIP service provision/usage, their viewpoints on agreements and issues they should consider when agreeing on QoS are focused on. The viewpoints of providers, i.e. ITSP, ICP, IPNP, and SCNP are tackled in the following sections, while the end-user viewpoint is omitted. Each of the sections describes the examples of input (service description, QoS parameters and objectives, business model), procedure, and the output as pointers to the various agreements relevant for a particular provider the section is devoted to.

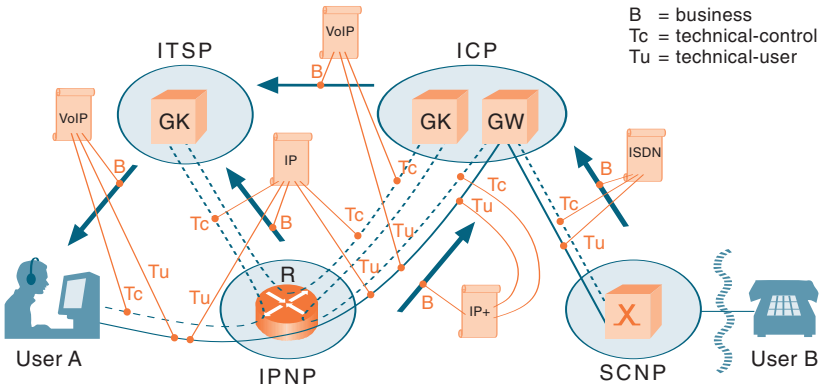


Figure 3 Network configuration and business model for the scenario chosen

3.1 ITSP

Considering the elements of **input** for this provider, the following is needed:

Service description – the service provided by the ITSP to its user is a telephony service over IP network. In the scenario studied here, user A initiates the call and realises voice communication with any user being connected to the SCN. The constraints on the user are: ownership of LAN and an H.323 terminal connected to the LAN. The user will be enabled to phone any user attached to the SCN[3)].

In order to provide this service, the ITSP has to rely on the following services:

- The transport of IP packets provided by IPNP, including the management, monitoring, metering for accounting purposes, etc.;
- Call connection and call management provided by the ICP, which realises the IP-SS7 interworking (e.g. H.245-Q.931) and voice decoding/transcoding (e.g. by applying G.711). The address translation from IP to E.164 number is also included into the service.

QoS parameters and objectives – The QoS parameters and objectives to be considered must be derived from the end-user requirements. The QoS parameters of main concern are delay, jitter and loss, presented to and agreed with the user. The objectives may be achieved after performing subjective tests, e.g. acceptability tests. As a result, e.g. Mean Opinion Score (MOS), reliability and availability depending on Network Performance (NP) parameters would be achieved. Those objectives can be used as a basis when the ITSP is making its list of requirements for the sub-providers – IPNP and ICP. Some examples of the relevant QoS parameters, their objectives, and related measurements can be found in the ITU-T Recommendations for PSTN voice quality, e.g. for:

- Connection quality:
  - Intrusive measurements: [E.434] for connectivity, [E.431] for call establishment/ clearing delay, and [E.428] for connection retention
  - Non-intrusive measurement: [E.425] for connectivity
- Call clarity:
  - Intrusive measurement: [P.861] Intrusive Perception Model, [G.107], [G.108], [G.109] for E-model
  - Non-intrusive measurement: [P.561] for objective measurement.

The constraints on the user domain should be included as a part of its profile within the agreement with the end-user – on the type of access used (in this example it is a LAN), type of terminal used for realising the connection (in this example it is an H.323 terminal) and bit-rate for the traffic expected to be inserted by the end-user to the ITSPs domain (this may vary if user

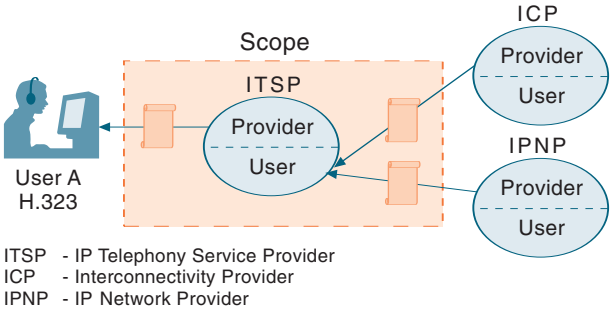


Figure 4 The ITSP's scope of relevant agreements

[3)] *Note: no value added services e.g. IN services have been considered here, only basic service.*

B VoIP Basics

Voice over packet-based networks has been discussed but never realised on a large scale until the Internet Protocol (IP), on which the Internet is built, inspired enthusiasts to try to save money avoiding long-distance phone-call charges.

Lately, the development of the IP technology evolved so it can support voice applications to an extent close to the traditional PSTN speech quality. Still numerous issues are open to be solved.

What is VoIP, IP Telephony, Internet Telephony, etc.?

A brief explanation of VoIP would be "the technology enabling the transmission of voice traffic in packets". Simply, VoIP technology is a technology that allows voice traffic to be transported across any IP-based network (e.g. LAN, WAN, MAN, etc.). On the other hand, IP telephony is a practical application of VoIP technology for building a complete telephony infrastructure that provides features and capabilities comparable to (and interacting with) today's PSTN over an IP infrastructure. Usually, IP telephony refers to the voice communication realised in the controlled environment of e.g. managed intranet. In addition, Internet telephony is an application that allows general transport of voice calls over the public Internet.

IP Telephony

As mentioned above, Internet telephony is the transport of telephone calls over the Internet, no matter whether traditional telephony devices, multimedia PCs or dedicated terminals take part in the calls and no matter whether the calls are entirely or only partially transmitted over the Internet.

How does IP Telephony Work?

The analogue voice signal is first converted into a Pulse Code Modulation (PCM) digital stream. The PCM stream is analysed, where echo, silence are removed, and tone detection is performed. Remaining PCM samples are forwarded to CODEC, where voice frames are created. Different CODECs may compress PCM steam, e.g. ITU-T G.711 (traditionally used for PSTN telephony) generates 64 kb/s, G.723.1 generates between 5.3 and 6.4 kb/s, etc. Each frame has its length and contains certain amount of "speech". These frames are further packetised, and the Real Time Protocol (RTP) header is added. In addition 8 bytes of the User Datagram Protocol (UDP) header (information on source, destination and port) are attached and the IP header containing the IP address of both source and destination is added. Recall that the UDP does not implement guaranteed message delivery, meaning that the recipient does not automatically acknowledge the sender when a message is received, as it is mandatory in the TCP. Hence, a UDP datagram can get "lost" on the way from sender to receiver, and the protocol itself does nothing to detect or report this condition. Another way in which UDP works unreliably is in the receipt of a burst of multiple datagrams. Unlike TCP, UDP provides no guarantees that the order of delivery is preserved. Since the original was created in real-time, a missing packet cannot be re-sent, and thus creates a gap in the data stream, which the recipient hears as clipped words or run-together speech (Figure B.1). In addition, the packet sequence may not be kept and the reconstituted bit-stream does not conform with the original.

Some VoIP applications use algorithms to overcome the effect of lost and delay of packets. Forward Error Correction (FEC) and advanced encoding schemes are used to alleviate some of these problems.

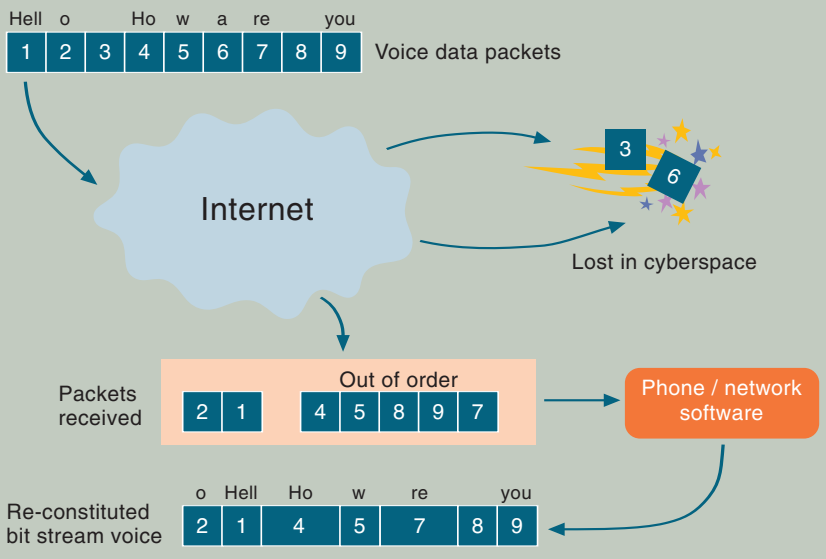


Figure B.1 Possible effect of packet lost in VoIP

Why VoIP?

Well, the advantages could be significant. In traditional circuit switched networks, when a connection is established, a channel is dedicated end-to-end for the duration of the communication. In a packet switched world, the resources (e.g. bandwidth) can be shared, which implies more effective usage of e.g. link capacity. Speech in its nature has many so-called "silent" periods, which can be compressed by different compression techniques, and also save some bandwidth. The charging and the price of a VoIP call versus traditional PSTN call is still an open issue, but it certainly attracts attention with the ability to reduce toll charges.

Another reason making VoIP attractive is the fact of having digital last mile, which opens for more services easily implemented. Merging of data and voice, in addition, would eliminate different systems supporting different services. Therefore, a single link would be suitable for all services.

However, there remain significant technical issues (e.g. quality, access portion capacity) that must be addressed before VoIP will be widely accepted, both in residential and business market.

VoIP and H.323

Now, in order to make an IP telephony service using VoIP technology, a number of issues need to be considered. The ITU-T Recommendation H.323 provides a framework for this. The H.323 covers technical requirements for audio and video communications services in Local Area Networks (LANs). H.323 references the T.120 specification to enable conferences, which include a data capability. The scope of H.323 does not include the LAN or the transport layer used to connect various LANs. Elements needed for interaction with the Switched Circuit Network (SCN) are within the scope of H.323. More details on H.323 are given in the "Understanding H.323" in this article.

Having H.323 does not solve all the issues. Some of the issues to be resolved before an IP telephony service, capable of competing (or interacting) with PSTN, can be provided commercially are:

- **Service interoperability**, e.g. call on demand, failure detection, appropriate tones/signalling, call tracing, caller id, etc.
- **Call control** procedures, information flows and protocols for e.g. call setup and release, gatekeeper discovery, endpoint registration, user authentication, Dual Tone Multi Frequency (DTMF) signalling.
- **Address translation** between E.164 and IPv4/v6 addresses – since IP networks are dynamic addressing environments, it is not possible to look up users by IP address
- Technical aspects of **charging/billing** – charges must be based on particular methods such as collect call, credit card call or basic call and on particular parameters such as time of day, type of service and duration of call.
- Technical aspects of **security** – in the first place a protection of the network against accidental or malicious failures, including congestion and signalling problems. And authentication, authorisation, encryption and privacy of calls, as well.
- End-to-end **QoS** aspects, including transcoding and echo-cancellation – both for connection/session setup and speech quality. In telecommunications a very detailed definition of the QoS has been applied and requested by the regulators. The end-user expects VoIP to deliver a speech quality and reliability comparable to today's PSTN telephony.
- **Mobility** aspects – a roaming user can access the VoIP service by using different techniques like mobile IP and application roaming.

There are a lot of activities world-wide e.g. in standardisation bodies, industrial fora, research projects, etc. trying to solve the open issues. One of them – the ETSI project TIPHON [http://www.etsi.org/tiphon] – is working closely with ITU-T towards standardising solutions to these issues. The work is going on, and there are still many details to attend to.

would make only a call, or have an additional option of sending data, video).

Business model – The ITSP is the primary provider for the end-user. On the other hand, the ITSP, as a user, depends on the services provided by IPNP and ICP. Therefore, from the ITSP viewpoint, three agreements are relevant (Figure 4):

- End-user – ITSP, where ITSP provides the VoIP service;
- ITSP-IPNP, where ITSP uses the service of transport of IP packets;
- ITSP-ICP, where ITSP uses the call connection and management towards SCN.

The **procedure** for considering the QoS by ITSP should take into account the objectives agreed with the end-user, while having in mind the expenses and characteristics of the services provided by ICP and IPNP. Additionally, the mechanisms and functionality implemented in the ITSP domain, e.g. the performance characteristics for the GK, should be taken into account.

For example, if ITSP agreed to offer the end-user the speech quality of MOS = 4, then the end-to-end delay of max 300 ms should be assured (one-way delay is 150 ms [3]). That implies the delay portions introduced by a GW should be no greater than 20 ms (total serving time and queuing time of ca. 10 ms), and the service provided by IPNP has to consider the propagation delay of 40 ms per 5000 miles. Serv-

ing/queuing delays for high-speed routers could be neglected.

After running the procedure by means of e.g. techno-economic analysis guiding the decision the output is given as the resulting set of QoS agreements described in Sections 4.1, 4.2, and 4.3 for the end-user, ICP and IPNP, respectively.

3.2 ICP

Considering the elements of **input** for this provider, the following is needed:

Service description – Call management in the ICP domain and Call connection to the SCNP network including decoding/coding are the main functions of the ICP's services. The service provided by the ICP for the ITSP consists of:

Call management and call set-up:

- Authentication;
- Registration;
- Status;
- Security;
- Number/address translation;
- Signalling to SCN (ISDN in the example here).

The billing information exchange might also be considered and may depend on BESP.

- Call connection towards non-IP network, e.g. SCN (i.e. ISDN) network;
- Decoding and transcoding;
- Digital audio data transfer.

QoS parameters and objectives – The QoS parameters and objectives to be considered could be derived from the end-user requirements i.e. the User-ITSP agreement. The parameters for e.g. delay and speech quality have to be translated to the portions and dependencies coming from the ICP domain.

Considering for example the delay, only the portion in the ICP network is to be considered. The objective for total delay for call setup should belong to the range of 3–5 s [1].

Speech quality on the other hand can be affected by delay, jitter and loss at IP-level in ICP-network (if the ICP uses the IPNP as a provider of this network the requirements has to be included in the ICP-IPNP agreement). Also the codec behaviour will affect speech quality. Subjective measurement of speech quality can be made using MOS. Such target values will then have to be mapped to the ITSP-ICP interface.

Overall reliability and availability may also be a requirement.

Business model – The ICP acts as a provider towards the ITSP, having an agreement on the service to be delivered by the ICP. In order to provide the service described above, the ICP as a user depends on the services provided by the SCNP and the IPNP. Agreements have to be made with them for the services, as follows:

- SCNP – call setup and call connection/switching in the SCN network;
- IPNP – access, transport and routing of IP-packets in the IP-network.

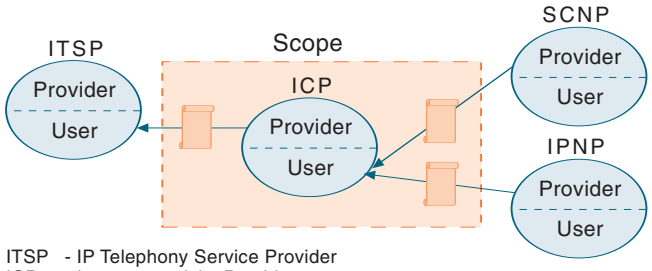
The QoS parts of the agreements to be considered by ICP when making decisions on QoS are shown in Figure 5.

After having the input as defined above the **procedure** for agreeing on the QoS should be applied as follows:

The ICP has to make a decision on which (range of) target values for QoS parameters of e.g. delay, jitter and loss it can assure. The decision would be made based on three types of information – the QoS of services delivered by SCNP, delivered by IPNP, and the mechanisms/functionality implemented in ICP's domain. The techno-economic analysis relating those three factors and other criteria, e.g. economic aspects, would enable the ICP to find the better solution.

The **output** is given as the resulting set of QoS agreements relevant for the ICP-service, and are outlined in Section 4.2, 4.3, and 4.4 for ITSP, IPNP, and SCNP, respectively.

Figure 5 The ICP's scope of relevant agreements



3.3 IPNP

Considering the elements of input for this provider, the following is needed:

Service description – the service provided by the IPNP to ITSP and ICP includes IP-based transport network services. Other functions may also be needed in order to support this service,

e.g. DNS. For the ITSP the IPNP provides a service originating calls within the IP network domain. This covers the transfer of signalling from the users to the ITSP and between the ITSP and ICP (in both directions), and the transfer of the encoded speech flows over the IP network. The transferred payload is in the form of IP packets. In the scenario examined here, only one IPNP is considered to be present. More than one entity may play the IPNP role, but in that case they could be looked upon as a merger where internal agreements between two IPNPs are not considered here. Such agreements, e.g. peer-to-peer agreements were studied in [10]. When identifying relevant interfaces, it is important to notice that in this case study there is no business relationship between the IPNP and the end-user, but only between ITSP and IPNP. On the other hand there is a technical relationship between the end-user and IPNP domain, and not between the end-user and ITSP.

The service specification may differ depending on the implementation, and in that case additional features like virtual Point of Presence (POP) service and/or web hosting for ITSP by IPNP could be included.

QoS parameters and objectives – The relevant QoS parameters and objectives should be given for two separate service components – one for network access towards the end-user, and the other on the IP packets transport for the ITSP and ICP. For the latter, a permanent IP connection might be the solution, but it is also possible to use virtual carrier channels i.e. IP PVCs. The parameters relevant for IP transfer could be found in [8] and IETF's RFCs [16], [17], [18], [19]. Also, some suggestions on the objectives for the performance of IP transport could be found in the first draft of ITU-T I.381 Recommendation [9]. Overall reliability and availability may also be a requirement.

The traffic pattern to be considered should include both the traffic generated by end-users, and the one generated by ITSP and ICP itself.

Business model – As shown on Figure 6 in the scenario studied here, the IPNP plays only a provider role. Note that IPNP may depend on the network service(s) provided by Layer 2, Layer 1 network operators, ATM, FR retailers, etc. Since the scenario elaborated here focuses on IP, such relationships are out of scope. In our scenario, the IPNP has business relationships with the ITSP and ICP. It has, as mentioned above, technical relationships with the end-user, ITSP and ICP. The agreements IPNP should take care of are depicted in Figure 6 (i.e. ITSP-IPNP, ICP-IPNP).

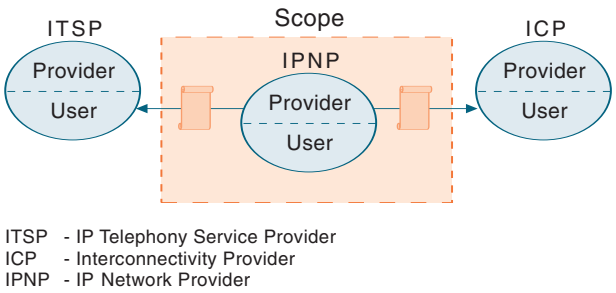


Figure 6 An IPNP's scope of relevant agreements

Considering the **procedure**, when agreeing the QoS with ITSP an IPNP has to consider two segments i.e. the access provision towards end-users and the IP network service towards ITSP. Apart from the mechanisms implemented in IPNP domain, the QoS delivered by sub-providers from L2, and L1 (if any) should be taken into account when deciding on the value/range of QoS parameters. When considering the service towards ICP, the support of both signalling and user generated traffic (i.e. voice) should be taken care of. The means of achieving the decision may include different types and tools of techno-economic analysis, regulatory issues, as well as other criteria.

The **output** of running the procedure is the set of agreements done with ITSP and ICP, as given in Section 4.3.

3.4 SCNP

Having a viewpoint of SCNP, the following **input** is needed:

Service description –The service provided by SCNP to ICP covers the "termination of calls" originated in an IP-based network. When the ICP is presenting the signalling and the encoded speech conversation type traffic in the SCNP domain applicable format, the case is the same as for any incoming call - thus the service is also the same.

The service includes both the call connection (speech samples transfer) and call management (signalling and control) i.e. both the user payload and signalling traffic will be taken care of in order to realise the connection with any ISDN based. The ISDN service provided by SCNP to user B is assumed to be traditionally dealt with, so although it should be considered by SCNP, it is not considered in this article.

QoS parameters and objectives – The QoS parameters and objectives to be considered as seen from the SCNP do not differ too much from those traditionally used when agreeing on QoS with any other network provider. A set of ITU-T recommendations relevant for SS7, different

C TIPHON Basics

ETSI VoIP activity is centred on the TIPHON (Telecommunications and Internet Protocol Harmonisation Over Networks) project. The mission of TIPHON is to combine IP with other telecommunication technologies to enable voice communication IP networks to interwork with Switched Circuit Networks (SCN). TIPHON is developing service-oriented solutions that a variety of operators can use. Wherever possible, TIPHON makes use of available standards, of which the most important one is H.323 (version 2). Even though ETSI is working in Europe, TIPHON deliverables are aimed at gaining world-wide acceptance. Companies supporting TIPHON are, e.g., AT&T, Cisco, Ericsson, Lucent, Intel, Microsoft, Motorola, Nokia, Nortel, Siemens, Philips.

Main work items of TIPHON are:

- Requirements for service interoperability.
- Global TIPHON architecture, interfaces and functions.
- Call control procedures, information flows and protocols.
- End-to-end QoS parameters.
- Address translation between E.164 and IP.
- Technical aspects of billing and accounting.
- Security profiles and procedures.

The objective of this project is to support a market that combines telecommunications and Internet technologies to enable communication over IP–based networks to work with existing Switched Circuit Networks (SCNs) and vice versa. The interoperability between those two networks is of interest, but not actual individual network itself.

Following scenarios have been within the scope of the TIPHON:

Scenario 1: communication between IP-based users and SCN-based users, where the initiator is the IP network user (Figure C.1).

Scenario 2: communication between IP-based users and SCN-based users, where the initiator the SCN based user (Figure C.1).

Scenario 3: communication between SCN-based users, using IP-based networks for the connection/transport between the involved users (Figure C.2).

Scenario 4: communication between IP-based users, using SCNs for the connection/transport between the involved users (Figure C.3).

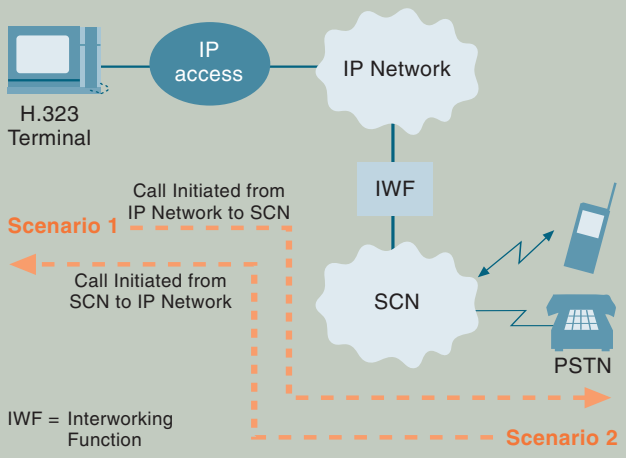


Figure C.1 Scenario 1 and Scenario 2

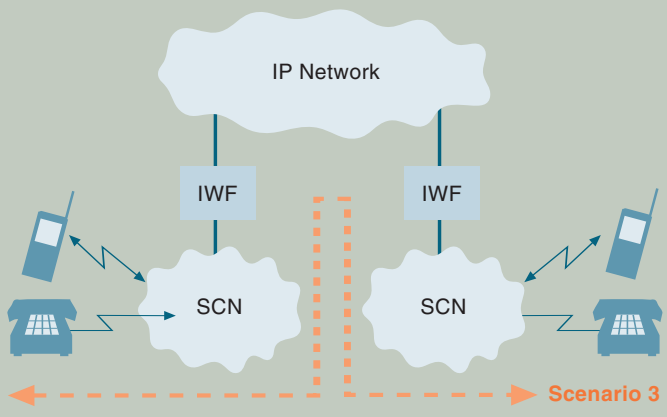


Figure C.2 Scenario 3

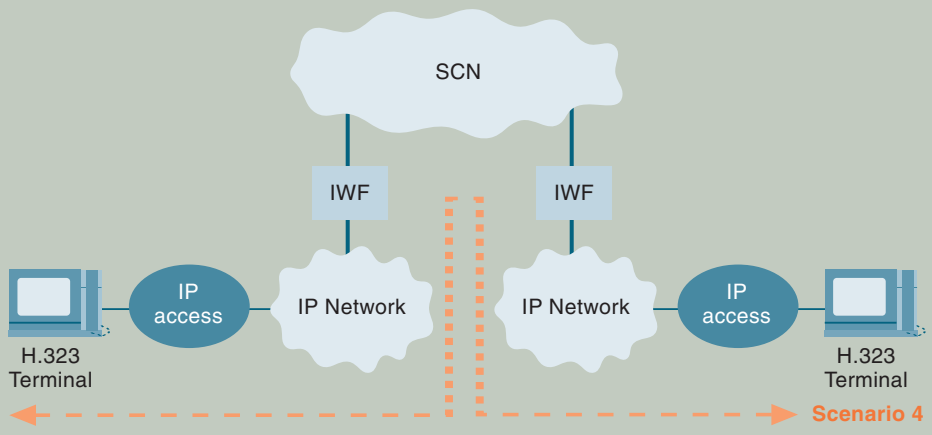


Figure C.3 Scenario 4

Note that the IP network does not have to be the Internet, but any IP network, e.g. intranet. IP-based user has H.323 terminal. Also, SCN can include both private and public networks, like PSTN, ISDN, GSM, etc. Interworking functions (IWF) can be implemented separately from or integrated into the existing SCN or IP–based network in order to provide the required interoperability.

In order to ensure an acceptable service provision, a number of issues has to be solved. Some of these include:

1 Requirements for service interoperability – enabling e.g. call on demand, detection of failures, appropriate tones/signalling, QoS selection, call tracing, caller id.

2 Global architecture, i.e. reference configurations, functional models, location of functionality, e.g. gateway functions between IP networks and SCNs and interfaces at these gateways.

3 Call control procedures, information flows and protocols, navigating the call setup and teardown, gatekeeper discovery, endpoint registration, user authentication, Dual Tone Multi Frequency (DTMF) signalling.

4 Address translation between E.164 and IPv4/v6 addresses.

5 Technical aspects of accounting, charging, billing.

6 Security - technical aspects like protection of the network against accidental or malicious failures, including congestion and signalling problems. Also, authentication, authorization, encryption and privacy of calls.

7 End–to–end QoS aspects – both voice quality and call setup quality are investigated. In addition effects of transcoding and echo–cancellation are considered.

8 Mobility aspects.

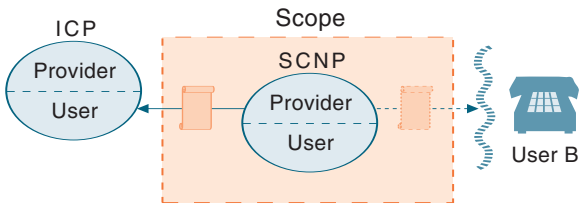
codecs, traditional telephony service, etc. can be considered as appropriate for this case. The parameters (e.g. delay and speech quality) have to be translated to the portions and dependencies within the SCNP domain.

Business model – In the scenario presented here, the SCNP acts as an interconnect service provider, offering "termination of calls" and SS7 based connection control for ICP. On the other hand, the SCNP is a provider of traditional telephony service to user B, but that relationship is not analysed here. The SCNP should support the ICP's requests for call connection in the switched network domain.

The QoS agreements to be considered by SCNP are shown in Figure 7.

After having the input as defined above, the **procedure** for agreeing on the QoS should include making the decision on which (range of) target values for QoS parameters of e.g. delay, jitter and loss the SCNP can assure to the ICP. The decision may depend on the other SCNPs involved, but basically all the relevant QoS objec-

Figure 7 The SCNP's scope of relevant agreements



ICP - Interconnectivity Provider
SCNP - Switched Circuit Network Provider

SCN impairments	VoIP impairments
Loss (also known as loudness loss) Reduction in signal strength that results in a received speech energy level that is too low	Loss This can be due to the type of head set used on H.323 terminal or echo cancel software
Noise Circuit noise and other noise-like artefacts introduced by the transmission system	Noise This can be due to the IP network performance parameters such as jitter or loss.
Echo Either talker echo, where the talker has experience of his/her voice returned after transmission delay, or listener echo, where the listener has the experience of hearing an echo of the talker	Echo This can be due to the type of transmission between user and H.323 terminal or errors on echo canceller software due to packet delay.

Table 2 Matching SCN voice communication and VoIP impairments

tives can be found in the existing ITU-T standards. Also, the expenses introduced by implementing different QoS mechanisms should be considered compared to the price of similar services offered by other co-operating companies. A techno-economic tool could perform such a calculation.

The output is the agreement made with the ICP as described in Section 4.4.

4 Output – QoS Agreements' Contents

The following sections cover particular agreements made between actors involved in the VoIP provision/usage.

4.1 End User – ITSP

In this section the content of the agreement made between an end-user and an ITSP is described. As shown in Figure 2 for the example scenario, the ITSP is a primary provider responsible for delivery of VoIP service to the end-user. The service provided by the ITSP to the end-user, as well as the functionality it owns are described in Section 3.1. One important issue when making an agreement with the human end-user is the "understandability" of the language the statements in QoSA are described with. Generally speaking a goal may be to offer the voice service of same quality to the IP-based user A as it is provided to the SCN-based user B. However, the content of the agreements for those two services (IP telephony and traditional telephony) would be different. Using the experience gained for providing traditional telephony, one may notice that transmission quality of speech communications has been dominated by three classes of impairments, as given in Table 2. Table 2 is an approach to match SCN and VoIP impairments.

Other types of subjective impairments are the availability and security that is related to the route possible for the ICP GK/GW and user validation and/or data encryption respectively.

The possible content of the end-user and ITSP agreement is given in the following.

Interface Description

Referring to the reference model (Figure 1), the information exchange between user A and ITSP is supported via A-interface for call-management functions and between user A and ICP via the B-interface for the digital speech traffic channels. The physical location of the service delivery point is actually realised via IPNP.

At the A-interface the RAS, H.225 and H.245 protocols together with UDP/IP are involved in the call setup process. At the B-interface the RTP/UDP and IP are involved. The IPv4 is used, as well as H.323 version 2.

Traffic Patterns

The traffic pattern can be described in terms of the average and maximum data rate and the intensity of signalling. Also, maximum and average number of calls/hour (or time interval chosen) should be specified. Maximum and average number of simultaneous calls from the end-user could be specified together with maximum and average bandwidth. Variation during day and week may also be specified.

In addition, as the service covers speech conversation type end-to-end user communication, the average call duration may also be considered for the real time, interactive speech conversation payload, as well as the payload structure, i.e. the protocol stack, codec dependent data units, possible header decompression, etc.

QoS Parameters and Objectives

The constraints on the equipment located in user's domain could be given as minimum requirements on the H.323 terminal type (or PC/audio input output device is used), type of application and codec used, effects of tandeming codecs etc. Also, LAN configuration/performance must be conformant to the requested minimum specified by the ITSP. Factors affecting QoS in case of LAN access are transmission delays through Network Interface Card (NIC) and jitter in data buffers related to the NIC.

Considering call set-up quality, the relevant parameters are mainly related to the A-reference point. The QoS parameters to be considered include:

D H.323 Basics

The H.323 is an ITU-T Recommendation that encompasses audio, video, and data communications across packet-switched networks, including the Internet. This standard enables the interoperability between multimedia applications produced by different vendors. Originally, it was a superset of recommendations setting standard for multimedia communications over Local Area Networks (LANs), which provide no guaranteed Quality of Service (QoS). In 1996 the scope was broaden to include not only LAN, but all packet-based networks, e.g. IP-based networks. The standard envelops many issues like stand-alone devices, embedded personal computer technology as well as point-to-point and multipoint conferences. The H.323 also addresses call control, multimedia management, bandwidth management, interfaces between LANs and other networks, security and supplementary services.

The H.323 is part of a larger series of communications standards. It is a part of the H.32X series that enables videoconferencing across a range of networks, e.g. H.320 addressing ISDN and H.324 addressing PSTN communication. Two versions of H.323 are published – version 1 in January 1996, version 2 in September 1998, while version 3 is under development in ITU-T SG 16.

Why H.323?

First of all, the environment is getting more mature for the H.323 applications – PCs are enhanced for multimedia (MMX CPUs), IP LANs are faster (10 Mb/s Ethernet to GigaEthernet). Second, important players have recognised this standard and its advantages – companies like Microsoft, Cisco, VocalTec, IBM, Intel have joined efforts in different fora dealing with H.323, e.g. iNOW, TIPIA, pulver.com.

Some of the H.323 advantages are:

Platform and Application Independence – H.323 is not tied to any hardware or operating system; it is applicable in wide spectra of equipment like voice-only handsets, full multimedia video-conferencing stations, multimedia PCs;

Network Independence – It is designed to run on top of common network architectures, can provide additional benefits when more network services are available, e.g. RSVP-aware domains, ATM QoS, etc.;

Standard Codecs – H.323 envelops different standards for handling audio/video streams, so the compatibility and interoperability between the devices/applications from different vendors are assured;

Bandwidth Management – The number of simultaneous calls can be managed ensuring sufficient bandwidth available for the traffic classes. An example is the Automatic Bandwidth Management mechanism that may increase and decrease bitrate according to the network behaviour – changes in delay, jitter, and packet loss.

Security – This is a version 2 feature. It refers to the H.235 that addresses: Authentication (the identity of conference participants is checked), Integrity (data received is indeed the data sent – the representation of data is not changed), Privacy (protection of data from eavesdropping by encryption), and non-Repudiation (assuring that participants in the conference can not deny participation later on).

Supplementary Services – These features are included in version 2. Fast Call Setup reduces the delay between the control and media streams, Call Transfer and Call Diversion, as defined by the H.450 series. H.450.1 defines the signalling protocol between H.323 endpoints for the control of supplementary services. H.450.2 defines Call Transfer and H.450.3 Call Diversion. Call Transfer allows a call established between endpoint A and endpoint B to be transformed into a new call between endpoint B and a third endpoint, endpoint C. Call Diversion provides the supplementary services Call Forwarding Unconditional, Call Forwarding Busy, Call Forwarding No Reply and Call Deflection.

Multipoint Support – H.323 supports conferencing between multiple end-points, by introducing the Multipoint Control Unit (MCU) it can support conferencing of three or more end-points;

Multicast Support – H.323 may support multipoint multicasting[1)] if any group management protocol is implemented, e.g. IGMP;

Internetworking – It assures the communication not only within a pure H.323 environment, but also between packet-switched with circuit-switched networks (applying H.320, H.324);

Flexibility – Terminals with different capabilities can participate in the same conference, although information may include different type of media, e.g. only audio, multimedia - video, data and audio, only data-terminals, etc.

Interoperability – H.323 establishes methods to exchange information between end-points for setting common capabilities for the conference, establishes also call setup and control protocols. Hence, users do not have to worry about the compatibility at the receiving point. Version 2 enhances the T.120/H.323 integration.

Having such features, H.323 enables multimedia applications usage on the existing IP-based infrastructure with no QoS guarantees. Some of the H.323 applications (e.g. videoconferencing, Internet telephony, video telephony, whiteboard, business conferencing, distance education, support and help desk applications, interactive shopping, audio/video mail, video on demand (VoD), telemedicine) are expected to be drivers of future communications market.

---

1) Multicast sends a single packet to a subset of destinations on the network without replication. On the other hand, unicast sends multiple point-to-point transmissions, while broadcast sends to all destinations.

- Delays in call processing and number translation;
- Delays in access, authorisation, registration, etc.;
- Availability of GK;
- Call rejection probability.

Considering call connection/speech quality, the parameters are mainly related to the B-reference point. The QoS parameters to be considered there are, e.g.:

- Delay, jitter and loss at IP-level;
- Codec delay (could be given as constraint on user's domain);
- Availability of GW;
- Speech quality.

Measurements

Measurements have to be established in the GK in order to monitor the call management related parameters. This may be done by logging the call set-up attempts and other call management activities. The GW log can contain the information on the established and completed calls. Some parameters do not have to be measured, but are matter of design, e.g. codec delay. Also, measurements related to the IP-level QoS parameters needs to be devised.

Reaction Patterns

Considering non-technical reactions, the customer support interface should be defined, e.g. phone number, web-site, mail address. Such an interface may be at the ITSP premises, or sub-provided by the IPNP, as well. The trouble ticketing process is to be specified, and the failure definitions (network connectivity/accessibility outage criteria, ITSP availability related outage criteria, call set up success ratio etc.) is to be considered for fault reporting, for escalation in case of service restoration time objectives not fulfilled, etc.

From the ITSP (or IPNP) the traffic related problems may be reported, and even a connectivity or service outage purpose related "alarm signalling", feedback signalling may be introduced, using the web interface of the user A.

Compensation schemes may also be specified, e.g. making the ITSP assuring discount for the users involved.

4.2 ITSP – ICP

In this section the content of the agreement made between an ITSP and an ICP is described. As shown for the example scenario (Figure 2), the ITSP has to set up a QoS agreement with the ICP, in order to ensure the delivery of the parameters agreed with the end-user.

Overview of Functionality Involved

The functions involved in this QoS agreement (i.e. ICP and ITSP) are described in Chapter 2. The ICP has GW in its domain; thus it provides the following services:

- Call management and set-up to SCN users (authentication, registration, status, security, number/address translation, signalling to SCN, billing information exchange;
- Call connection to SCN (decoding and transcoding, digital audio data transfer).

QoS Affecting Factors

Considering the elements involved (i.e. GK and GW) several factors can affect the QoS, e.g. in the GK:

- Call processing delays;
- Processing and look-up delays associated with security issues;
- Delays in accessing back-end services, and in the GW;
- The choice of speech codec;
- Transcoding(s) or Tandem Free Operation with the SCN;
- The performance of the speech codec to various types of network degradation (including effects of any error concealment mechanisms present in the coder);
- Signal processing delays;
- Call processing delays;
- The packetisation method used;
- Processing delays associated with security issues;
- The design of jitter buffers;
- Delays through the audio or digital media paths;
- The performance of network echo-cancelling devices;
- DTMF tone handling.

Interface Description

Referring to the reference model (Figure 1), the information exchange between ITSP and ICP is supported via D-interface for call-management functions and the B-interface for the digital audio traffic channels.

At the D interface the RAS, H.225 and H.245 protocols together with TCP/IP are involved in the call setup process. At the B interface the codec protocols, e.g. G.723, G.729 or G.711 together with RTP and UDP/IP are involved.

A choice necessary to be made is which level of the Open Systems Interconnection Reference Model (OSIRM) [27] to chose for specifying QoS parameters. For a full description of e.g. the setup time at the border of the D interface, the actual data element signals in the protocol will have to be used to specify the QoS parameters.

Traffic Patterns

The traffic pattern can be described in terms of the average and maximum data rate and the intensity of signalling.

At the D interface maximum and average number of calls/hour (or chosen time interval) should be specified. Variation during day and week should also be specified.

At the B interface maximum and average number of simultaneous calls should be specified together with maximum and average bandwidth. Variation during day and week should also be specified.

QoS-Parameters and Objectives

QoS parameters to be considered in this QoS agreement are mainly those related to the GK and GW functionality. The ICP internal network has to be taken into account as well. Considering call set-up quality, it is mainly related to the D-reference point. The QoS parameters to be considered are:

- Delays in call processing and number translation;
- Portion of call set-up time relevant for the ITSP domain;
- Availability of GK (ICP);
- Call rejection probability.

Considering speech quality, it is mainly related to the B-reference point. The QoS parameters to be considered there are:

- Delay, Jitter, Loss at IP-level in ICP-network;
- Codec delay;
- End-to-end delay;
- Availability of GW;
- Speech quality;
- Transmission Quality (E-model).

This interface will carry the main traffic; therefore throughput, speed and delay are important QoS parameters. Organising QoS parameters by applying the adapted 3x3 matrix from I.350 results in Table 4.

Some examples of objectives and measuring points related to the QoS parameters are given in Table 5.

For a very strict approach when defining the QoS parameters, the signals in each protocol at D and B interfaces could be used. For example, it may be required to measure when a specific request signal is sent and when the acknowledge signal is received. The result of measurements should be mapped further to the “higher level” QoS parameter like setup time.

Measurements

Measurements have to be established in the GK in order to monitor the call management related parameters. This may be done by logging the call set-up attempts and other call management activities. According to that, in the GW, the established and completed calls can be logged.

Not all parameters have to be measured if they are mainly fixed by design. This could for example be the codec delay. Also, some of the parameters may not be possible to measure at the D and B interaction points. For example, speech quality must be measured at the user interface. But, since the codec and GW design affect this parameter a relationship between these parameters has to be determined. Some indications on the measurements and measuring points are given in Table 5.

Reaction Patterns

An escalation scheme for fault and degradation management can be used to trigger actions according to specified thresholds of each parameter. The severity of the degradation/fault will determine the actions. It can be linked to a trouble ticketing system to ensure a structured way of managing upcoming deviations. The reactions may be both ways. For the provider, if the QoS threshold is exceeded, a reaction is triggered. For the user, if allowed traffic patterns are exceeded a reaction is triggered. Reaction patterns may not only be manually executed, but also implemented in the network and automatically performed.

QoS parameter	Objectives	Measurements	Traffic pattern
Call set-up time (end-to-end)	3–5 s [1]	GK log at D	Low traffic
GK Availability	≥ 99.7 %	GK log	
Call rejection probability	< 0.3 %	GK log	
Delay of IP-packets One-way	< 100 ms [8]	At B	High traffic
Jitter of IP-packets	< 20 ms [8]	At B	High traffic
Loss of IP-packets	< 2 % [8]	At B	High traffic
End-to-end speech delay	< 150 ms	Portion of total delay at D	
Codec delay Note: codec type dependent	< 67.5 ms	No	
Speech quality	MOS > 3.5	MOS can be measured only at user interface.	
GW Availability	> 99.97 %	GW log	

Table 3 An example of QoS objectives for the end-user-ITSP agreement

4.3 ITSP – IPNP, IPNP – ICP

Since IPNP is providing similar service to ITSP and ICP, both of the related agreements are presented in this section. The main difference is related to the interface description, since the description of the interface towards ITSP has to include also the interconnection points for end-user's access to the IP network. The transport of both signalling traffic and speech traffic is supported by routers and links in the IPNP's infrastructure.

Therefore both of the agreements made by IPNP (i.e. ITSP-IPNP, and ICP-IPNP) are described in the same section, and the possible content is as given in the following.

Interfaces

The network access and transport carrier service delivery interface is implemented by customer premises networks edge routers or gateway router units. IPv4 is used.

The measurement interface for the link layer traffic can be the path end points, or the gatekeepers and call control signalling access point in the user's domain.

QoS measurement interfaces: at the path end-points for the IP layer reference event based (ITU-T I.380 or IETF RFCs [16], [17], [18], [19] measurements.

QoS data conveyance, reporting, alarm monitoring, trouble ticketing and problem handling interfaces could be negotiated between the parties.

Function / Performance Criterion	Access	Information transfer	Disengagement
Speed	Call setup time (D) Delay from request to acknowledge in H225/ /RAS /Q.931 protocols (D)	Delay at IP-level (B) Jitter at IP-level (B)	Disengagement delay (D) Mean disengagement time (D) Maximum disengagement time (D)
Accuracy	Misrouted call probability (D)	Severely errored period ratio (B) Missequenced packet delivery ratio (B)	Disengagement denial ratio (D)
Dependability	Call rejection probability (D) GK availability GW availability	Loss at IP-level (B) Call loss probability (B,D) GK availability GW availability	Premature disconnect probability (D) GK availability GW availability

Table 4 The 3x3 matrix for the ITSP-ICP QoS agreement; (B) and (D) indicate related interfaces

Traffic Patterns

For all the possible IP transport links, the traffic is to be characterised by:

- The purpose of use, type of traffic payload, e.g. signalling link, link for encoded and packetised speech-flows;
- The maximum available bandwidth (committed bit-rate);
- Average packet length (size);
- In case of handling multiple communication sessions, maximum and average number of simultaneous calls.

Variation during day and week can be specified.

QoS Parameters

The relevant QoS parameters can be organised in 3x3 matrix, as illustrated in Table 6.

The relevant QoS parameters and their objectives are given in Table 7.

Also non-technical/administrative parameters (e.g. availability of maintenance and QoS support related human resources) could be considered between the entities.

Measurements

Measurements have to be identified for the IP network access provisioning (e.g. interface availability/established network access point accumulated down time, average and worst case time to restore), as well as for the IP transfer service provisioning (e.g. bit-rate, number of completed simultaneous calls).

Call management related parameters may be measured by logging the call set-up attempts and other call management activities. In the GW the established and completed calls can be logged.

Reaction Pattern

In case of observation of relevant problems in QoS delivery, or the risk of non-compliance, for a period of time (e.g. 5, 10, 30 minutes), depending on the failure/problem event definition, the parties should agree to initiate appropriate reactions, i.e. processes with specified procedure (chain of activities), using defined tools/techniques and presenting outcomes of the following type:

- Apply "alarm monitoring based alarm signalling" in case of increased blocking probability or implement a traffic control process;

QoS parameter	Objectives	Measurements	Traffic pattern
Call set-up time	A portion of 3–5 s [1]	GK log at D	Low traffic
GK Availability	≥ 99.95 %	GK log	
Call rejection probability	< 0.05 %	GK log	
Delay IP-packets	A portion of 100 ms [8]	At B	High traffic
Jitter IP-packets	A portion of 20 ms [8]	At B	High traffic
Loss IP-packets	≤ 2 % [8]	At B	High traffic
GW Availability	≥ 99.95 %	GW log	

Table 5 An example of QoS objectives

- Specify the acceptable trouble ticketing process and fault administration, create the documentation;
- Specify troubleshooting for major problems;
- Specify an escalation procedure;
- Define service restoration process and fault clearing time related information exchange;
- Initiate "helpdesk" activation.

4.4 ICP – SCNP

This section elaborates the content of the QoS agreement to be found at the interface between ICP and SCNP. Such a configuration can be considered similar to the "traditional" interconnection of two SCNPs, implying that relevant QoS parameters as well as related measurements, traffic patterns and even reaction patterns are specified as in existing recommendations and standards for a particular SCN.

Overview of Roles (Functionality) Involved

According to the scenario and reference model presented in Chapter 2, there is a relationship between SCNP and ICP. In general, the SCN refers to either public networks (e.g. PSTN, ISDN, PLMN) or private networks. Here, the assumption is that SCNP provides ISDN, consisting of Customer premises Equipment (CPE), Access Network (AN), and the Core Network (CN) [23]. The CPE is connected via the AN to the CN via User-Network-Interface (UNI), but this problem is considered to be out of scope, since standards are already widely available and operable. Simply, SCNP envelops SCN network provider and SCN access provider. The functionality relevant for this example is ISDN sig-

E H.323 Architecture

An illustration of the architecture of IP telephony network based on H.323 is given in Figure E.1. The network architecture consists of four types of network elements: terminals, Gatekeepers (GKs), Gateways (GWs), and Multipoint Control Units (MCUs). Theoretically, communication can be realised between two terminals connected to a LAN. However, practically, an efficient communication system capable of connecting to the outside world (e.g. SCN) can be built only by introducing some of the other elements. The functionality of each network element is introduced below.

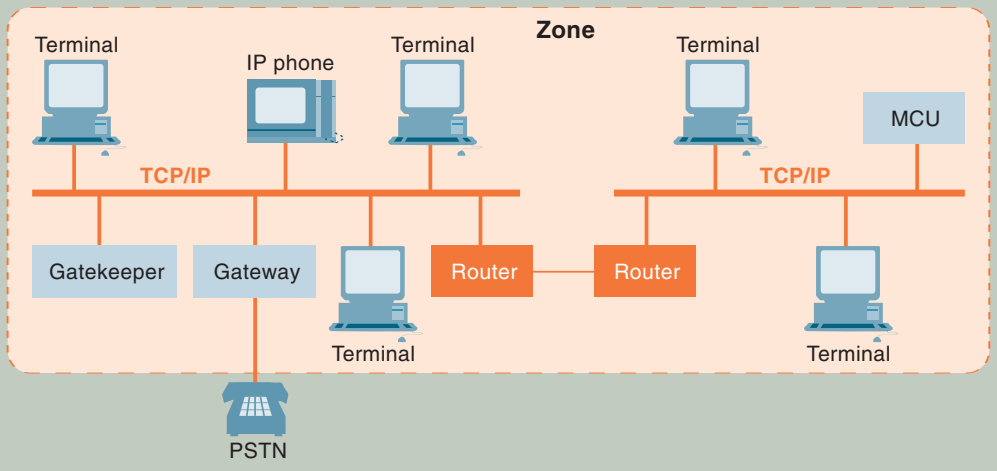


Figure E.1 H.323 network architecture

Terminals are end-points capable of receiving/initiating calls. They generate and receive bi-directional real-time information streams. A terminal can be either software running in a computer or dedicated equipment. Support of voice communication is mandatory, while video and data are optional. In addition, all H.323 terminals must include, so-called System Control part, which includes H.245 control used to negotiate channel usage and capabilities, "stripped" version of Q.931 for call signalling and setup, as well as Registration/Admission/Status (RAS) interface.

Gatekeeper manages a so-called zone (see Figure E.1) which is a collection of terminals, GWs, and MCUs. A number of zones build an H.323 network. GK act as a central points managing all the calls within its zone. It performs numerous functions, like:

1. Address resolution and call routing, where address translation means translation of alias addresses to transport addresses using translation table.

2. Admissions control used to determine whether an endpoint is allowed to terminate or originate a call. It may be based on authorisation, bandwidth or some other criteria.

3. Bandwidth control assures a certain amount of bandwidth to be reserved for H.323 traffic and distributed between the connections. When the limit is reached no more connections can be opened, so other traffic has enough capacity.

4. Zone management - Terminals within a zone register to their GK, which adds the corresponding address to the registration table.

5. Call control signalling, call authorisation, bandwidth management, and call management. Call control signalling means that the gatekeeper may process the control signals (Q.931) in point to point conferences instead of passing them directly between terminals. Call authorisation allows the GK to reject a call depending on its properties. The reasons for rejection can be user defined, e.g., restricted access from/to particular terminals or GWs, or restricted access during certain periods. Bandwidth management is closely related to bandwidth control allowing the gatekeeper to reject calls from a terminals if the available bandwidth is low. In call management the gatekeeper keeps a list of on-going calls to indicate that a terminal is busy or to provide information for the bandwidth management function.

Optionally, GK may route H.323 calls, which allows more effective call control and service providers can bill for calls in their network. The routing service may also be used to redirect calls to other terminals if a called terminal is unavailable. Additionally, gatekeepers can balance the load among multiple gatekeepers based on some routing logic. The GK acts like an interface to other H.323 networks. Gatekeepers are optional elements but if they are present terminals have to use them.

Gateway is responsible for connecting IP telephone network to other type of networks, e.g., PSTN, ISDN. The gateway performs translation between different transmission formats and communication procedures. Also, it is responsible to set up and clear calls on both sides. Terminals communicate with gateways using the H.245 and Q.931 protocols.

MCU is needed only if centralised and hybrid multipoint multimedia conferences are used. An MCU consists of Multipoint Con-

troller (MC) and a number of Multipoint Processors (MP). The MC handles control information and the MPs handles the streams.

Often it is possible to combine several different network elements into the same physical unit. For example, GK functionality might be incorporated into the GW and MCU, or MCU could be implemented into the terminals in order to allow multipoint conferences without any separate MCU unit.

The H.323 traffic can be considered as a mixture of audio, video, data, and control signals. **Control** communication includes signalling for call setup, capability exchange, signalling of commands and indications, and messages to open and describe the content of logical channels. All audio, video, and control signals pass through a control layer that formats the data streams into messages for output to the network interface. The reverse process takes place for incoming streams. This layer also performs logical framing, sequence numbering, error detection, and error correction as appropriate to each media type. The Q.931, RAS, and RTP/RTCP protocols perform these functions. **Audio** signals contain digitised and compressed speech. The mandatory algorithm is ITU-T G.711 (64 kb/s PCM codec), while the rest are optional, e.g. G.729a, G.723.1, G.728, etc. Choice of a codec affects quality and should be a trade-off between speech quality, bit rate, computer power, and signal delay. **Video** signals are optional in general, but if implemented then H.261 is a default, while support for H.263 is optional. Video information is transmitted at a rate no greater than that selected during the capability exchange.

Data conferencing is optional, but when supported enables applications like shared whiteboards, application sharing, and file transfer. H.323 supports data conferencing through the T.120 specification, which addresses point-to-point and multipoint data conferences. It provides interoperability at the application, network, and transport level. A recommendation for multicast support in T.120 is known as T.125 Annex A or the Multicast Adaptation Protocol.

H.323 protocol architecture is illustrated in Figure E.2.

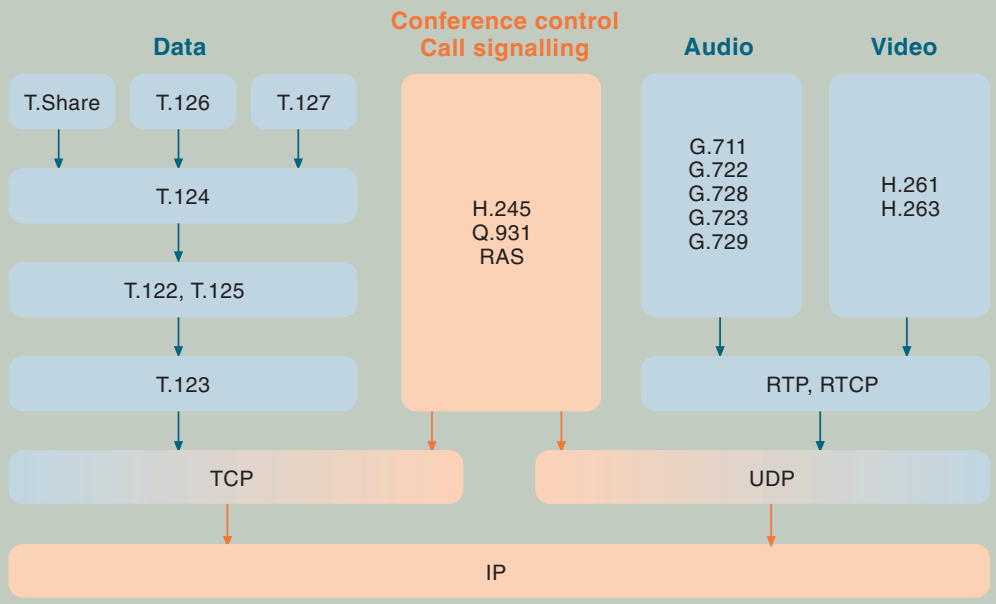


Figure E.2 H.323 protocol architecture

Additional information on H.323 can be found in [H.323], as well as on [http://www.databeam.com/h323].

Reference

H.323 ITU. *Packet-based multimedia communications systems.* Geneva, 09/99. (ITU-T H.323.)

nalling. The interworking between IP and SCN is realised via ICP functionality, i.e. GK, GW.

Interface Description

Relating reference points and the functionality as described above, the Ea, and Eb reference points are used for exchanging the information between ICP and SCNP. The Ea is used for exchanging speech traffic, while Eb is used for exchange of the signalling information. The signalling protocol used is SS7 [15], while the codec used for speech is G.711 [4].

Traffic Pattern

Considering the fact that the call is originated in H.323 terminal, the traffic generated (i.e. coming from ICP domain into SCNP domain) can be expressed like, e.g.:

Table 6 The 3x3 matrix for ITSP-IPNP and ICP-IPNP agreements

Function / Performance Criterion	Access	Information transfer	Disengagement
Speed	Call setup time	IP packet transfer delay [8], [16] IP packet transfer jitter [8], [16]	Disengagement delay Mean disengagement time Maximum disengagement time
Accuracy	Misrouted call ratio Call failure probability	Severely errored period ratio Missequenced packet delivery ratio	Disengagement denial ratio Call premature disconnect ratio
Dependability	Call rejection probability GK availability GW availability Edge-router availability	Loss at IP-level Call loss probability GK availability GW availability	Call clearing failure ratio GK availability GW availability

- Max number of calls and traffic volume during a reference period;
- Average number of calls and traffic volume during a reference period;
- Variations in number of calls and traffic volume during agreed time period, e.g. day, week;
- Bit-rate for both signalling and speech traffic/payload type;
- Prediction on the geographic distribution of the traffic.

Table 7 An example of QoS parameters/objectives/measurement points

QoS parameter	Objectives	Measurement points
Call set-up time	A portion of 3-5 s [1]	GK at D
GK Availability	≥ 99.95 %	GK
Call rejection probability	< 0.05 %	GK
Delay IP-packets One-way RTT	 A portion of 100 ms [8] ≤ 200 ms	At B
Jitter IP-packets	A portion of 20 ms [8]	At B
Loss IP-packets	A portion of 2 % [8]	At B
GW Availability	≥ 99.95 %	GW
Mean service interruption duration	< 1 h	

QoS-Parameters

The 3x3 generalised matrix from EQoS can be used to categorise the QoS parameters investigated in [26]. The parameters relevant for this interface are:

- Call set up time in the SCN affecting call set up quality;
- Network transmission delays in SCN affecting end-to-end delay; and
- Speech quality.

The QoS parameters to be considered in this agreement are those related to the service provided by SCN, i.e. ISDN.

Some of the QoS considerations associated to the SCN are related to echo cancellation[4)] (which affects call quality), since it is assumed that the echo canceller is located in the PSTN exchange.

The QoS parameters relevant for this agreement are given in Table 8.

[4)] *If SCN is considered to be GSM or ISDN, no echo cancellation is needed.*

Function / Criterion	Access	Information transfer	Disengagement
Speed	Delay (call set up, number translation, authorisation, etc.) Mean access time Maximum access time	Delay Jitter	Delay Mean disengagement time Maximum disengagement time
Accuracy	Incorrect access ratio	Severely errored period ratio	Premature disengagement
Dependability	Access denial ratio (call rejection probability, malicious calls ratio, etc.)	Call loss probability Misrouted call ratio Missequenced packet delivery ratio	Disengagement denial ratio

Table 8 The 3x3 matrix for the ICP-SCNP QoS agreement

The recommendations and standards related to the call set up delay are mainly referred to the ITU-T standards for SS7, and those contributing to the call speech quality are mainly taken from ITU-T G.711.

Considering the performance of the signalling in ISDN, the application call control part is described in [14], while [1] and [7] bring some more details on the ISDN performance. In [12] the Message Transfer Part (MTP) performance is described. In [14] the following parameters and their objectives are defined:

Availability

- Of a signalling route set [13] should not be less than 0.99998. This corresponds to a total downtime for a user signalling relation of ten minutes per year maximum

- Of the signalling network should be sufficiently high as to meet the signalling route set downtime objectives stated above.

Dependability

- False operation will be avoided if no more than one in 108 of all signal units transmitted is accepted (error detection [11], transmission fault indication [5], [6])

- Signalling malfunction should cause no more than 2 in 105 (provisional value) of all ISDN calls to be unsuccessful.

Delay

- Signalling delay with the components (see Figure 1 in [14]).

Considering the objectives for the traffic pattern, the bit-rate might be observed, as well as the availability and reliability.

Measurement Schemes

The points of observation are as depicted in Figure 1, i.e. Ea and Eb, where relevant QoS parameters values and traffic characteristics should be measured. The location of points would be decided when exact implementation is dealt with. There are various ways to perform measures, both on the ICP and SCNP side. Since this configuration can be considered similar to the interconnection of two SCNPs, the relevant measurement schemes can be adopted from the standards/recommendations, e.g. [2].

Reaction Patterns

Technical reactions should be identified and agreed upon, both for cases when the ICP injected non-conformant traffic, and the SCNP did not provide agreed QoS. Examples of such reactions are load control, call logging, resource management, warnings, error messages and alarm signalling/reporting. Problem indication purpose interactions and escalation procedure based on the implemented fault handling/management process may be specified.

Other Issues

The non-technical parameters related to e.g. help-desk availability (365/24), reparation time, etc. are not treated within the QoS agreement, but are very important elements of each interconnection agreement.

5 Concluding Remarks

Multiprovision configuration for supporting VoIP case based on the ETSI TIPHON Scenario 1 (communication between H.323-based user and SNC-based user) is analysed, giving the picture of responsibilities of all users and providers involved end-to-end. The viewpoints and considerations of various providers have been investigated for each of the providers recognised in the VoIP service provisioning. At the interfaces identified between entities related QoS agreements have been elaborated into details, i.e. end-user-ITSP, ITSP-ICP, ITSP-IPNP, IPNP-ICP and ICP-SCNP QoS parts of SLAs are presented. As part of establishing the QoS parts of SLAs at each interface, it is necessary to describe:

- Roles/functionality;
- Interface description;
- Traffic pattern;
- QoS parameters and their objectives;
- Measurements; and
- Reaction patterns.

This study has demonstrated the practical value of applying the EQoS, which enables a harmonised understanding of QoS for any provider involved in the multi-provision of a service. Thus, the procedure proposed in [20] outlines to service providers an approach used when establishing agreements with both users and sub-providers.

Acknowledgements

Although the text is the sole resonsibility of the listed authors, it has been composed of results from the EURESCOM P806-GI project. The fruitful discussions and contributions from all parties involved in that project are greatly appreciated.

References

1 ITU-T. *Network grade of service parameters and target values for circuit-switched services in the evolving ISDN.* Geneva, ITU, 05/99. (ITU-T Recommendation E.721.)

2 ITU-T. *Framework for service quality agreement.* Geneva, ITU, 10/96. (ITU-T Recommendation E.801.)

3 ITU-T. *One-way transmission time.* Geneva, ITU, 02/96. (ITU-T Recommendation G.114.)

4 ITU-T. *Pulse code modulation (PCM) of voice frequencies.* Geneva, ITU, 10/88. (ITU-T Recommendation G.711.)

5 ITU-T. *Characteristics of primary PCM multiplex equipment operating at 2048 kbit/s*. Geneva, ITU, 11/88. (ITU-T Recommendation G.732.)

6 ITU-T. *Characteristics of primary PCM multiplex equipment operating at 1544 kbit/s.* Geneva, ITU, 11/88. (ITU-T Recommendation G.733.)

7 ITU-T. *Network performance objectives for connection processing delays in an ISDN*. Geneva, ITU, 03/93. (ITU-T Recommendation I.352.)

8 ITU-T. *Internet Protocol Data Communication Service – IP Packet Transfer and Availability Performance Parameters.* Geneva, ITU, 1998. (ITU-T Recommendation I.380.)

9 ITU-T. *Internet Protocol Communication Service – IP Performance Objectives and Allocations, Draft 10/99.* Geneva, ITU. (Draft ITU-T Recommendation I.381.)

10 *IN/Internet interconnect scenarios and harmonised agreements.* Heidelberg, EURESCOM, 1999. (EURESCOM P806-GI Deliverable 2.)

11 ITU-T. *Signalling link*. Geneva, ITU, 07/96. (ITU-T Recommendation Q.703.)

12 ITU-T. *Message transfer part signalling performance.* Geneva, ITU, 03/93. (ITU-T Recommendation Q.706.)

13 ITU-T. *Hypothetical signalling reference connection.* Geneva, ITU, 03/93. (ITU-T Recommendation Q.709.)

14 ITU-T. *Performance objectives in the integrated services digital network application.* Geneva, ITU, 03/93. (ITU-T Recommendation Q.766.)

15 ITU-T. *ISDN user-network interface layer 3 specification for basic call control*. Geneva, ITU, 05/98. (ITU-T Recommendation Q.931.)

16 IETF. Paxon, V et al. *Framework for IP Performance Metrics*. 1998. (IETF RFC 2330.)

17 IETF. Mahdavi, J, Paxson, V. *IPPM Metrics for Measuring Connectivity*. 1999. (IETF RFC 2679.)

18 IETF. Almes G et al. *A One-way Packet Loss Metric for IPPM*. 1999. (IETF RFC 2680.)

19 IETF. Almes G et al. *A Round-trip Delay Metric for IPPM*. 1999. (IETF RFC 2681.)

20 Jensen T et al. Managing QoS in Multi-Provider Environment – a Framework and Further Challenges. *Telektronikk*, 96 (2), 71–79, 2000

21 ETSI. *Telecommunications and Internet Protocol Harmonisation Over Network (TIPHON); Description of Technical Issues*. Valbonne, 1998. (TR 101 300 V1.1.5 (1998-12).)

22 ETSI. *Telecommunications and Internet Protocol Harmonisation Over Network (TIPHON); Service Requirements for service interoperability, Scenario 1*. Valbonne, 1998. (TR 101 306 V1.2.3 (1998-02).)

23 ETSI. *Telecommunications and Internet Protocol Harmonisation Over Network (TIPHON); Service Requirements for service interoperability, Phase II*. Valbonne, 1998. (TR 101 307 V1.2.3 (1998-02).)

24 ETSI. *Telecommunications and Internet Protocol Harmonisation Over Network (TIPHON); Network architecture and reference configurations, Scenario 1*. Valbonne, 1998. (TR 101 312 V1.3.2 (1998-06).)

25 ETSI. *Telecommunications and Internet Protocol Harmonisation Over Network (TIPHON); Network architecture and reference configurations, Phase II: Scenario 1 + Scenario 2.* Valbonne, 1999. (TR 101 313 V0.4.2 (1999-02).)

26 ETSI. *Telecommunications and Internet Protocol Harmonisation Over Network (TIPHON); General Aspects of Quality of Service (QoS).* Valbonne, 1998. (TR 101 329 V2.2.2 (1998-10).)

27 ITU-T. *Information technology – Open Systems Interconnection – Basic reference model: The basic model*. Geneva, ITU, 07/94 . (ITU-T Recommendation X.200.)

Some Physical Considerations Concerning Radiation of Electromagnetic Waves

KNUT N. STOKKE

After graduating from the Norwegian Technical University (Trondheim) in 1958, Knut N. Stokke (71) worked from 1959 to 1969 in the Planning Division of the Broadcasting Office of the Norwegian Telecom Administration, and thereafter with the Transmission Section where his activities included specifications and regulations for broadcasting transmitters and transposers. In 1987 he joined the new regulatory organisation, the Norwegian Telecommunications Regulatory Authority, where he was head of the Section for Broadcasting. Knut Stokke has been a member of the Norwegian delegation to the major broadcasting conferences, and has also participated in various ex-CCIR Study Groups and more specifically Study Groups 5 and 6 (now St.Gr. 3). Knut Stokke retired 1 March 1999.

Introduction

Propagation of electromagnetic waves has in many ways been mathematically described since the first successful experiments by Hertz in 1887. However, what physically happens when electromagnetic waves are propagated into space seems to have been of less interest.

However, there is one figure which has been used in some textbooks, and this figure is shown in Figure 1. Even this figure does not explain what happens when the waves start to radiate from an antenna element.

In this paper a few physical examples are discussed. A half-wave antenna is used as reference for the considerations, but other antenna elements may also be used.

When we want to look at the radiation from an antenna, it is necessary to observe the direction for the field power P_f (power per area) in an electromagnetic field. The direction is perpendicular to both the electric field component E and the magnetic field component H in an electromagnetic field, and in such a way that a right-handed corkscrew rotated the minor angle from E to H moves in the direction of P_f. This law of nature coincides with the directions in vector calculations (Poynting vector).

Figure 3 of this paper is intended to explain how we get radiation from a standing wave in a half-wave antenna. However, we also have radiation from an antenna wire (travelling wave antenna) which is terminated with its characteristic impedance, and where we consequently have no

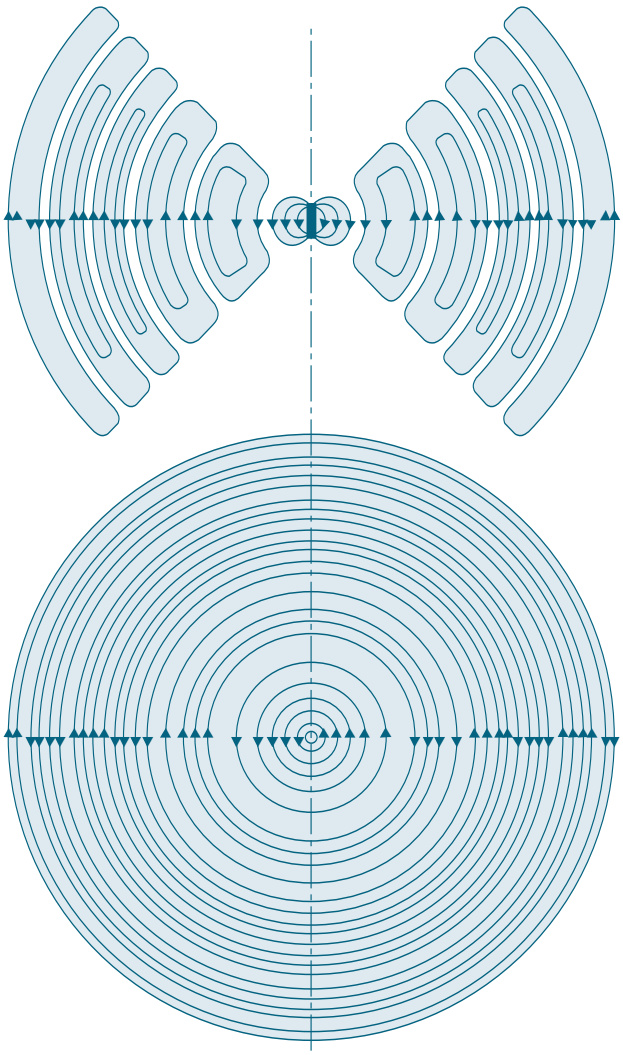


Figure 1 Propagation from an antenna element

standing wave. Figure 5 indicates how the radiation may be initiated.

Examples of combination of radiation from a standing wave and a travelling wave are also described (Figures 6 and 7).

Electron Charge, Field Lines, and Radiation from a Standing Wave

If we want to study what happens when electric current is sent into a conductor, we have to consider an immense flow of electrons. However, it is often more convenient to consider one or a few electrons and assume that the other electrons behave in almost the same manner.

An electron has a mass, $9.108 \cdot 10^{-31}$ kg, and a charge, $1.602 \cdot 10^{-19}$ coulomb, and it is *the charge* that is most important concerning electromagnetic wave propagation.

As a basis for the considerations Figure 2 may be used. In Figure 2 A we have an electron at a certain distance from a conductor, in this case a $\lambda/2$ conductor. The $\lambda/2$ element is neutral, that is, at zero (0) voltage level, this element is on a positive level referred to the negative charge of the electron. We then have an electric field with direction from the conductor to the electron.

In the left part of Figure 2 B is indicated just the moment when the electron (and thereby the charge of the electron) is put into the $\lambda/2$ conductor causing a voltage u at this end of the conductor. The direct current component accelerates the electron along the conductor, and the static energy of this component is transformed into velocity energy of the electron (= current). At the right end of the conductor the direct current component is 0, and thereafter we get conditions as shown in Figure 2 D. We then have *a standing wave* in the element, and a standing wave have high values for voltage and current when we operate at resonance lengths.

In practice the conditions are not ideal, and the direct current component will most probably disappear after several periods.

We could also have used Figure 2 C to see that the direct current component has to disappear in order to get symmetrical conditions for the dynamic voltage and current in the element.

Looking at Figure 2 D, the voltages at the ends of the $\lambda/2$ element will swing between $+u/2$ and $-u/2$. These voltages are caused by the charges $+e/2$ and $-e/2$. *Then we may look at the whole phenomenon as if we have two charges $+e/2$ and $-e/2$ always moving in opposite directions*, as indicated in Figure 2 E. The currents caused by the opposite moving charges $+e/2$ and $-e/2$ go in the same direction.

It is in principle not so important if we here use $e/2$ or only e. We may for instance have started with two electrons instead of one, and consid-

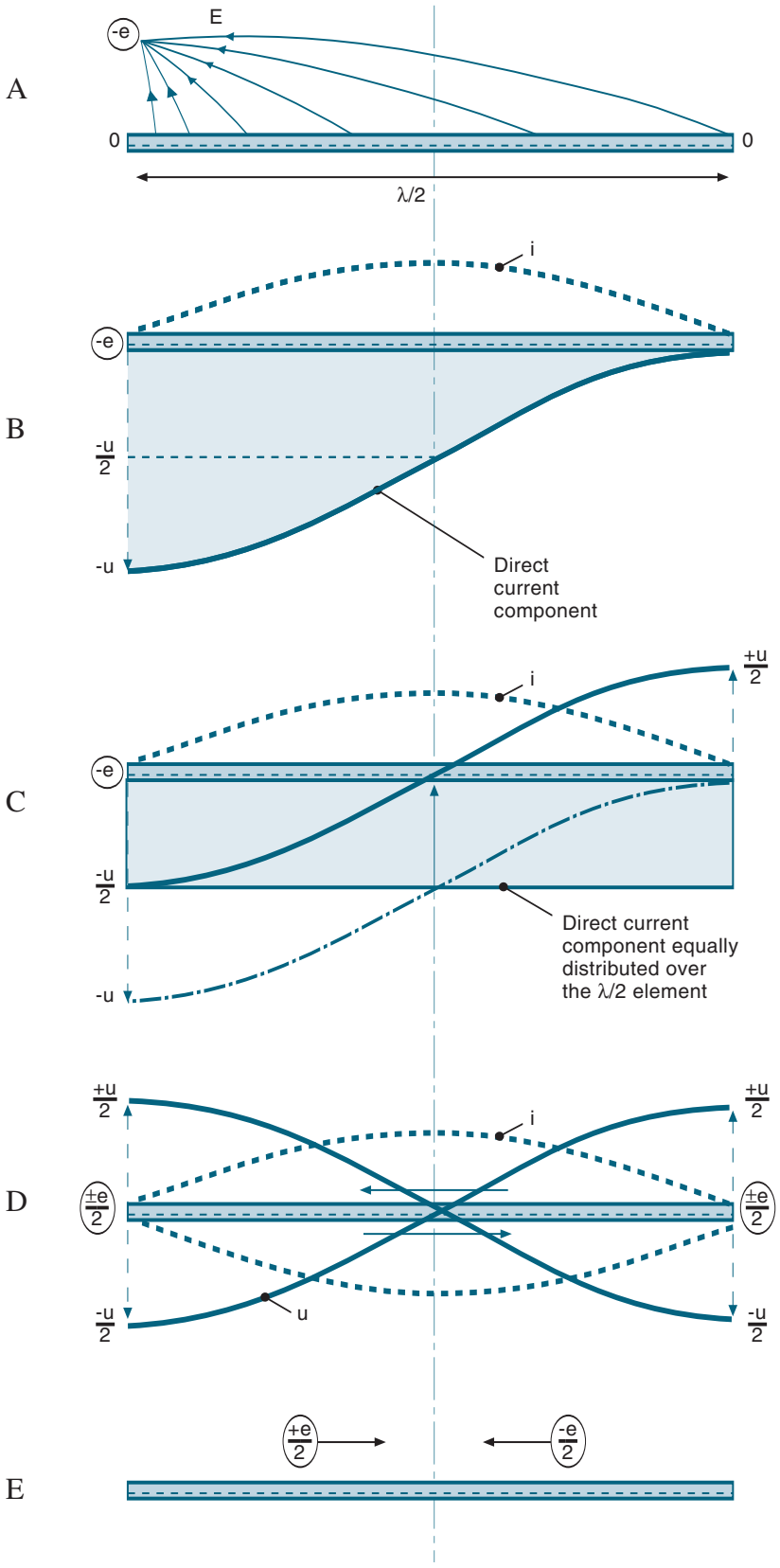


Figure 2 Voltages and currents in a resonant conductor

ered one positive and one negative electron charge.

And in the next figure, Figure 3, we assume to have one positive and one negative charge moving in opposite directions in a $\lambda/2$ conductor. The movements of the charges are symmetrical relative to the equilibrium point, that is, relative to the centre of the conductor.

It is important to choose the right starting point for our considerations. *Here we choose the moment when the two charges pass each other at the centre of the element. In addition we choose the start of a charging or retardation part of the period*, that is, when the current moves in opposite direction of the voltage. This may be compared with a pendulum just leaving the lowest point and being retarded by the gravitation. We will later on look at what happens if we start at any other point.

In Figure 3 A the two oscillating charges are at the equilibrium point of the element. At this moment we have no electric field *E* near the element because the two charges neutralise each other.

In Figure 3 B where the charges have moved away from the equilibrium point, we have electric field lines *E* from the positive to the negative charge. And as the charges move away from each other, they are in *a charging period*.

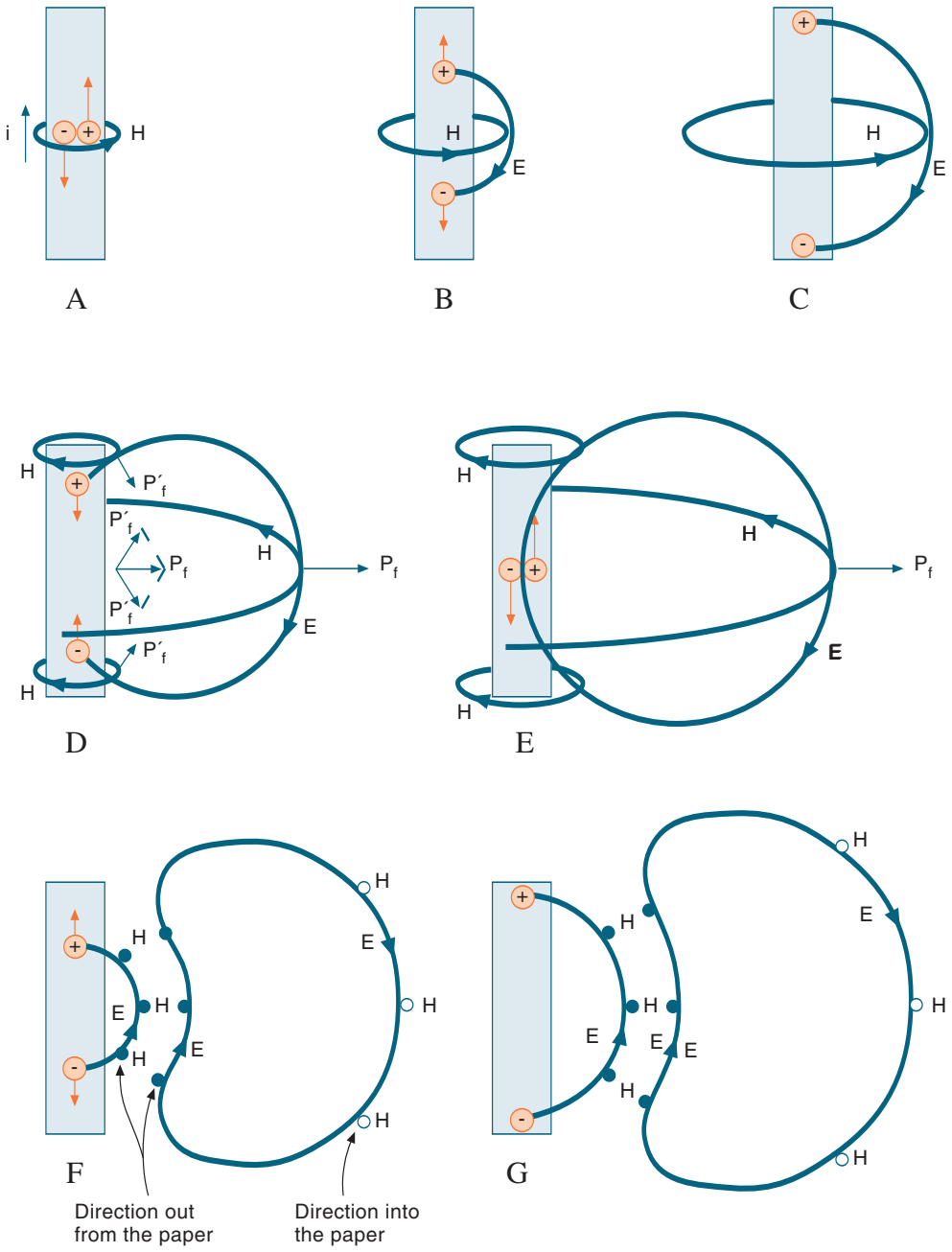


Figure 3 Fields around an antenna element

A moving charge is a current. Around this charge we have a magnetic field H with the same direction as the rotation of a right-handed corkscrew screwed along the positive current direction, or as the rotation of a left-handed corkscrew screwed along the negative current direction. We therefore already have a magnetic curl field.

Field lines are often used to describe a field. However, if we then should describe a field completely, we have to take account of an infinite number of field lines, and this is of course impossible. Here we will look only at a few field lines in order to get an impression of what happens to the fields between and around charges.

If we then further on follow the movement of charges in Figure 3 [1], the charges move towards the ends of the $\lambda/2$ conductor where they stop. At that moment we have only potential energy and no current in the conductor, and therefore no magnetic field near the conductor (Figure 3 C). Thereafter the charges will move towards each other because of the potential difference, that is, we are in *the discharging part of the period*. We now get a new magnetic field near the conductor, but this magnetic field has an opposite direction compared to the first magnetic field.

How the power moves out *perpendicular to the conductor* is indicated by the Poynting vectors P_f shown in Figure 3 D. At the same time the field lines move outwards in space, as also shown in Figure 3 D. We also see that the rotating magnetic field may move away from the conductor into space.

However, this magnetic field H is dependent on the electric field E (or the electric displacement $D = \varepsilon \cdot E$) to move away from the conductor. And when the charges come to the centre of the conductor as indicated in Figure 3 E, we have a very interesting case. Then the charges are neutralised, and there is therefore *no potential difference between the ends of the electric field line*. The ends may be tied together, and we have an *electric curl field* that may, together with the magnetic field, move away from the conductor.

There are other examples where objects may be tied together because of small or no potential differences. We may for instance mention what happens when a soap bubble is leaving a thin tube.

In Figure 3 F we have new electric and magnetic fields. Because they have the same direction as the nearest part of the old fields, they repulse each other, and the conditions may be as indicated in Figures 3 F and 3 G. We have now got a radiating element or an antenna element because of the standing wave in the conductor, and we may then say that *a standing wave is an antenna*.

What happens further on may be seen in Figure 1. We observe that the wavelength near the antenna element varies. It is important to be aware of this phenomenon when we want to measure the wavelength of a frequency.

We may now look at what happens when we do not start at the beginning of the charging part of the period.

If we start somewhere else in the charging part of the period, we get a reduced first period. But if we start in the discharging part of the period, *the Poynting vector points towards the conductor*. We then have no radiation, and *nothing happens before the beginning of the charging part of the next period*.

We also have to look into the problems concerning the length of an antenna. If we have a $\lambda/2$ element as indicated in Figure 4, the charges may move unobstructed. But if the antenna element is shorter, the charges will move to the ends of the element and remain there until the next $\lambda/2$ change (between + and –, or between – and +). The concentration of charges at the ends of the short element dl will increase until the middle of the $\lambda/2$ period, and will thereafter decrease until the next $\lambda/2$ change. This is because the two elements are fed with the same frequency f or wavelength λ.

An especially interesting situation we have for a Hertzian dipole where there is only an infinitesimal difference dl in the +/– or –/+ changes. We

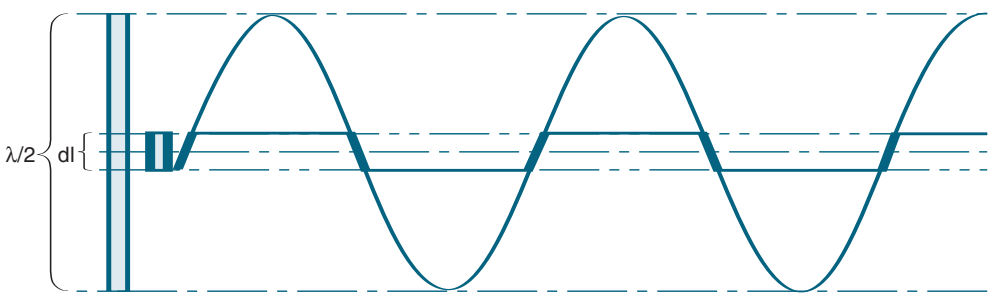


Figure 4 Movement of charges in antenna elements

then have a sudden change of polarity at the changes, and we may consider the changes to follow a square pulse series.

Radiation from a Travelling Wave

The radiation from the antenna elements we have considered until now, *is caused by the standing waves on the elements*. However, there is another phenomenon which causes radiation, and that is *the travelling wave effect* [2].

If we look at Figure 5 A, a forward travelling wave is initiated because the antenna is terminated by its characteristic impedance. Consequently there is no standing wave on the antenna.

In Figure 5 A is also indicated the conditions when a positive and a negative charge move along the conductor. As the charges move along the conductor, we get an electric field as indicated by only two field lines. But at the same

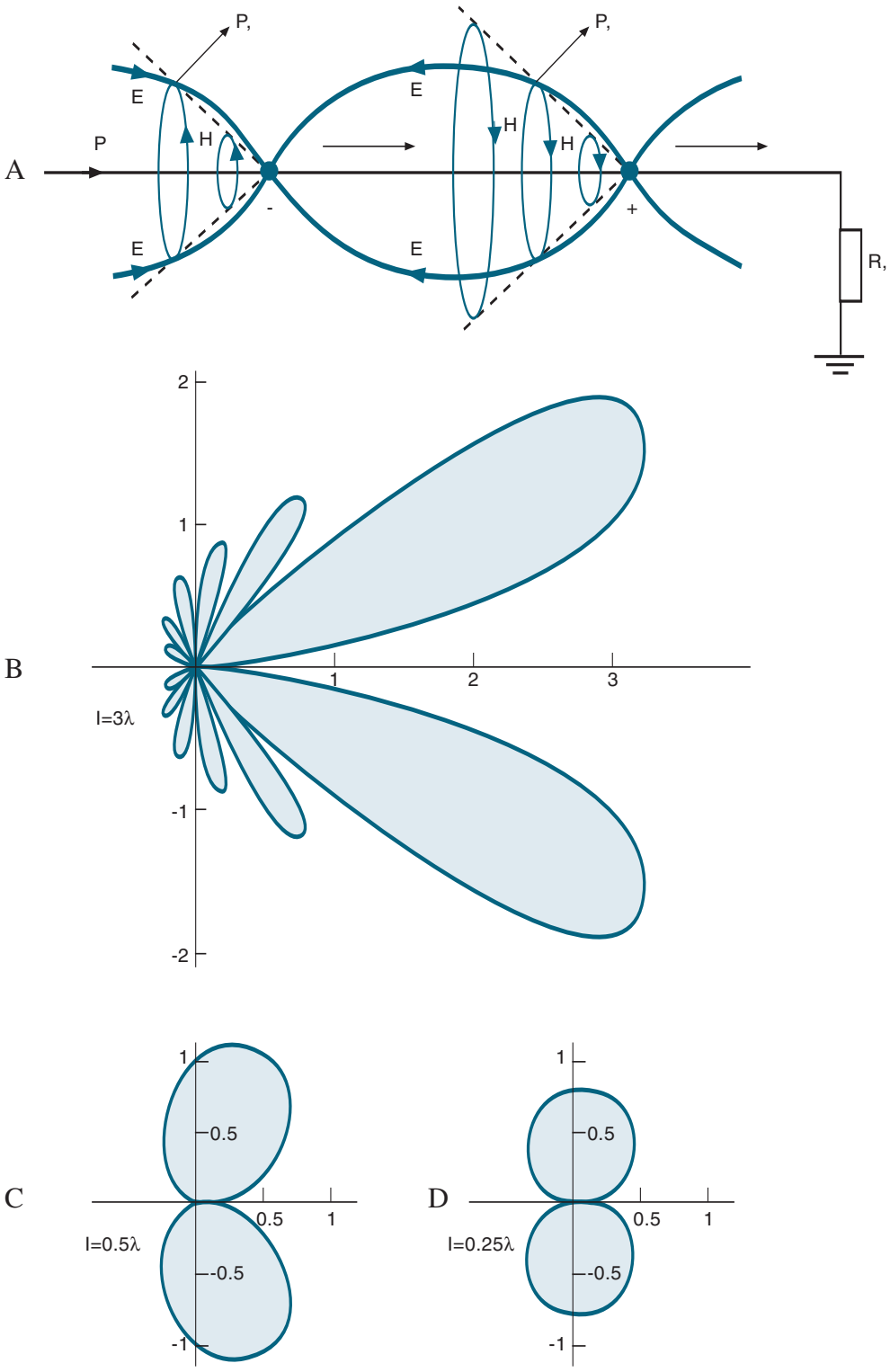


Figure 5 Travelling wave antenna

time a charge creates a magnetic field which moves out from the conductor with about the same velocity as the velocity of the charge along the conductor (dependent on the electric and the magnetic conditions in the surroundings). In free space or homogeneous conditions the resulting movement of the magnetic field is 45° out from the conductor, and a part of this magnetic field may be combined with a part of the electric field and cause radiation. The Poynting vector is at right angle to both the electric and the magnetic field where the field lines cross each other.

For short conductor lines the travelling wave effect is rather weak, but when the antenna wire is several wavelengths, we may have some gain relative to a short monopole over ideal conducting plane. In fact, the travelling wave antenna is one of the few antenna types which may have some gain at very low frequencies.

The antenna pattern for a travelling wave antenna may be calculated by using the assumption that the antenna is an end-fire array of collinear Hertzian dipoles coupled in series. The equation for the antenna pattern will then be [3]:

$$|E| = \frac{60 \cdot I_0}{r} \cdot \frac{\sin\theta}{1-\cos\theta} \cdot \sin\left(\frac{\pi l}{\lambda}(1-\cos\theta)\right) \quad (1)$$

where I_0 is the r.m.s. value of the current along the antenna wire, r is the distance to the measuring site, θ is the angle to the antenna wire, l is the length of the antenna wire, and λ is the wavelength.

In free space the antenna pattern is a rotation diagram. When the antenna is near the ground, the antenna pattern will because of reflections and influence of the ground constants get another form.

The antenna pattern is measured at constant distance, and the first part of Equation (1) may therefore also be considered constant. The equation for the relative antenna pattern may therefore be written as:

$$E_r = \left| \frac{\sin\theta}{1-\cos\theta} \cdot \sin\left(\frac{\pi l}{\lambda}(1-\cos\theta)\right) \right| \quad (2)$$

In Figure 5 B is shown the pattern for a travelling wave antenna 3λ long. However, we should observe that there is a certain travelling effect for short antennas, as for instance indicated in Figures 5 C and 5 D.

Combination of Radiation from a Standing Wave and a Travelling Wave

We have looked especially at what happens when we put some electrons or charges into a half-wave radiator. If the radiator does not receive more energy, the radiation will be attenuated oscillations. In order to keep the radiation going, we have to feed the radiating wire with power from a generator or transmitter.

The best place to feed a $\lambda/2$ radiator is in the middle of the radiator. There we have current maximum and voltage minimum, and therefore a well defined and relatively low impedance (73.2 ohm). And because we now have divided the half-wave radiator into two parts, we have *a half-wave dipole*.

If we feed a λ radiator in the middle, we get very high impedance because of minimum current and maximum voltage.

It is assumed that the current distribution in the half-wave dipole is a cosine function (referred to the centre of the dipole). It is therefore also assumed that the feed current distribution is a cosine function, as indicated in Figure 5. These assumptions are in good agreement with practical measurements.

If the feeder and the dipole have the same impedance (73.2 Ω), both the current and the voltage in the dipole are Q times higher than in the feeder. It is important to be aware of this fact especially when constructing insulators for nar-

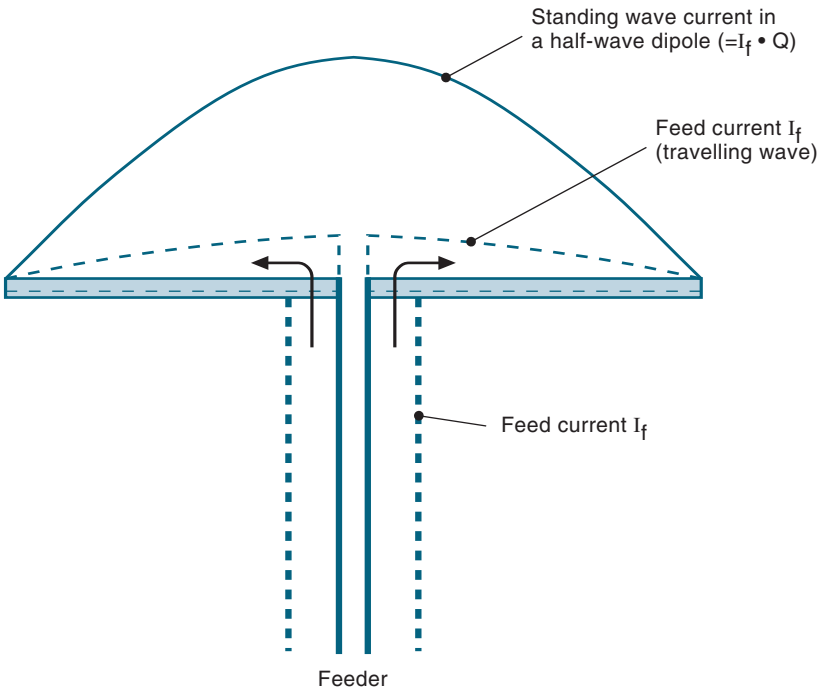


Figure 6 Current distribution in feeder and half-wave dipole

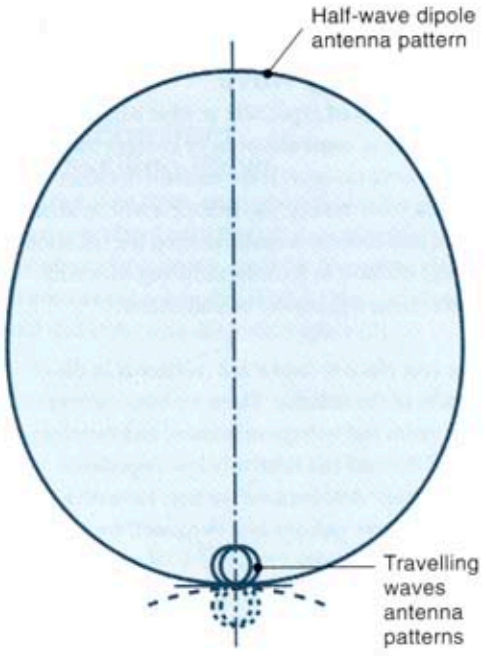


Figure 7 Antenna patterns for a λ/2 dipole and for the travelling wave in a λ/2 dipole with $Q \approx 10$

row-band LF, MF and HF antennas where the Q values may be very high.

The maximum radiation from a half-wave dipole due to the standing wave is transverse to the dipole. The relative antenna pattern (relative diagram) for a half-wave dipole is [1]:

$$E_r = \left| \frac{\cos\left(\frac{\pi}{2}\cos\theta\right)}{\sin\theta} \right| \qquad (3)$$

where θ is the angle to the antenna wire.

However, we also have radiation because of the travelling waves in the dipole (see Figure 6), and the maximum values of this radiation is at a certain angle to the dipole (see Figure 5). This phenomenon may give some radiation along the dipole, and this radiation is circularly polarised. For high Q values this radiation does not influence the dipole characteristics to any substantial degree, but for low Q values the travelling wave radiation may act both on the form of the antenna pattern and on the impedance of the dipole.

In Figure 7 is indicated the conditions when we have a Q value of about 10. Due to the 0.8 (2 dB) reference value (see Figure 4 D), the cosine current distribution and the difference 4.8 dB – 2.15 dB in antenna reference value [4], (see Figure 4 D), the difference between the maximum radiation from the standing wave and the travelling wave should be about 23 dB.

There are several ways to feed an antenna wire with power from the transmitter or generator. Figure 8 is an example of the conditions we may have for a dipole and an end-fire antenna when the lengths of the antennas are in resonance.

Figure 8 A is a dipole where we have a multiple of resonant lengths. However, we have here symmetrical conditions and the maximum radiation is therefore transverse to the antenna wire.

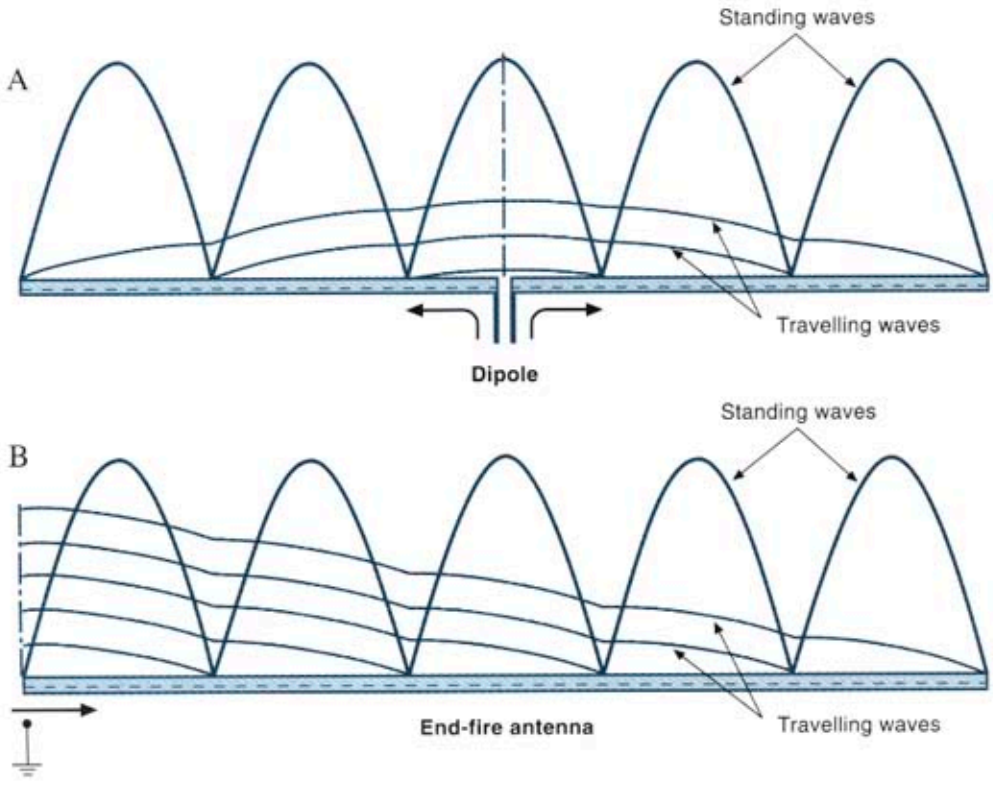


Figure 8 Multi-resonant dipole and end-fire antennas

In Figure 8 B is shown a single-wire antenna fed at one end of the wire. When we have a wire of several wavelengths, we get conditions like we have for a travelling wave antenna of the same length.

When we have non-resonant antenna wires, we get a similar effect, but then at a much lower intensity.

It may be mentioned that long telephone cables or long power lines may act as travelling wave antennas for low frequencies. This may for instance cause problems for telephone carrier systems.

If for instance we have a mismatched cable, we may have a situation as indicated in Figure 9. The forward travelling wave and the reflected wave interfere with each other, giving a standing wave in the cable.

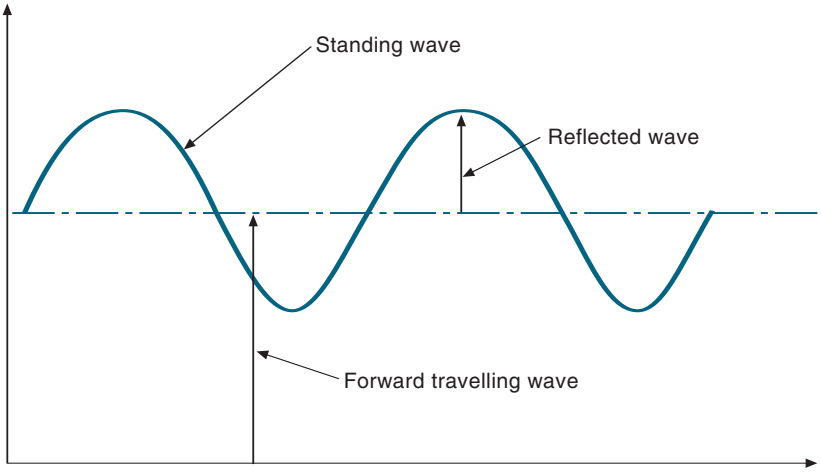


Figure 9 Standing wave

The outer conductor of a coaxial cable is relatively very thin compared to the wavelength. A standing wave in the cable will therefore penetrate the outer conductor, and we therefore also get a standing wave on the outer side of the outer conductor. It has already been said that a standing wave is an antenna. *A mismatched cable is therefore capable of both receiving and radiating electromagnetic energy.*

Another interesting phenomenon takes place when we bury a mismatched coaxial cable into lossy ground. The losses will then reduce the standing wave on the outside of the outer conductor, and therefore also reduce the standing wave in the cable. The result is lower standing wave ratio in the cable, and consequently better signal quality.

If we put some energy into an antenna and thereafter disconnect the antenna from the energy source, we get damped oscillations as indicated in Figure 10. If we want to have continuous radiation of for instance a carrier frequency, the transmitter has to feed the antenna with a certain amount of energy each period. The ratio between this amount of energy and the energy already in the antenna, is dependent on the Q value of the antenna.

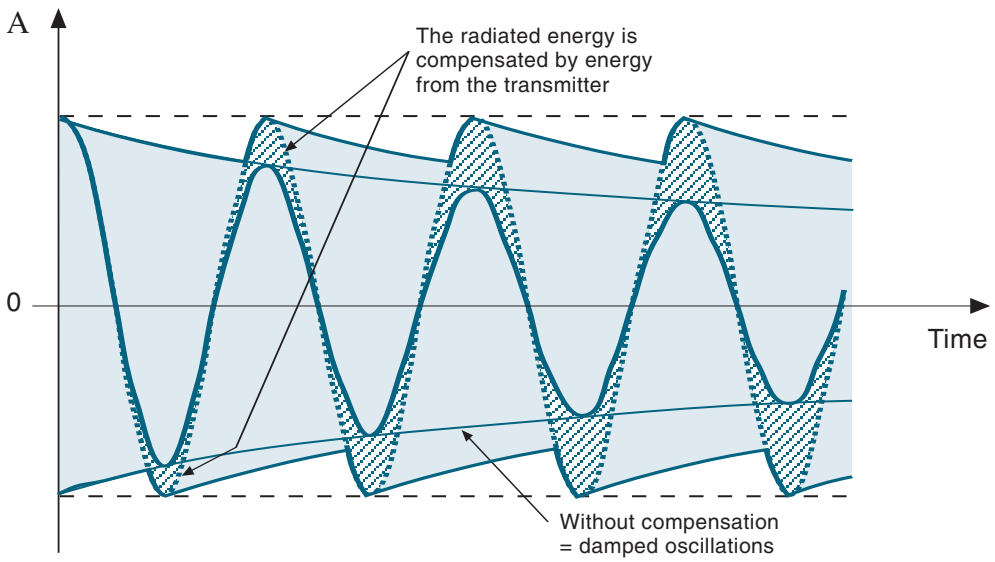


Figure 10 Radiation and compensation

Radiation from Noise Sources

The maximum available thermal noise power at a site is:

$$P_m = kTB \tag{4}$$

where k is Boltsmann's constant ($1.38 \cdot 10^{-23}$ J/K), T is the absolute temperature of the objects in the surroundings, and B is the bandwidth in Hz.

Increased thermal influence causes an increase in electron movement. An increase in electron movement also gives more free moving or "travelling" electrons. And travelling electrons may give radiation in the same manner as in a travelling wave. However, dependent on the surroundings we may have reflections and standing waves, and consequently noise radiation from these standing waves.

Because of high travelling electron densities in lightening and other atmospheric discharges, we may have high electromagnetic noise radiation from such sources. The background electromagnetic noise at low frequencies is dominated by these discharges. Due to high thunderstorm activities in continental areas, the noise at low frequencies is higher inland than in coastal areas.

If we exclude man-made noise (industrial noise, sparks, etc.), the background noise above about 20 MHz in a communication system is normally dominated by thermal noise.

Conclusion

In trying to understand radiation of electromagnetic waves, it may be of some interest to have a physical explanation of what happens. However, in order to describe the phenomenon, it may often be necessary to simplify the explanations.

References

1 Stokke, K N. *Radiotransmisjon*. Oslo, Universitetsforlaget, 1971. (In Norwegian.)

2 Stokke, K N. Travelling wave antennas. *Telektronikk*, 90 (4), 63–66, 94.

3 Glazier, E V D, Lamont, H R L. *The Services Textbook of Radio, Transmission and Propagation.* London, Her Majesty's Stationary Office, 1958.

4 ITU. *Propagation in non-ionized media*. ITU 1994 Series Volume. Geneva, ITU, 1995. (Rec. ITU-R PN.341-3.)

Telektronikk Index 2000

Telektronikk (96) 1, 2000 – Broadband Radio Access

Telektronikk (96) 2, 2000 – The Economy of Internet Services

Telektronikk (96) 3, 2000 – Security

Telektronikk (96) 4, 2000 – Languages for Telecommunications Applications