

Guidelines for Developing Sequence Diagram Specifications, Exemplified for IPTV

RAGNHILD KOBRO RUNDE, JAN ØYVIND AAGEDAL



Ragnhild Kobra Runde is Associate Professor at the Department of Informatics, University of Oslo



Jan Øyvind Aagedal is a Solution Architect at Telenor Norway

IPTV (Internet Protocol Television), integrating internet and TV technologies, is an important part of Telenor's strategy to increase product and service innovations. In this article, we explain how sequence diagrams may be used for describing the functionality of the IPTV platform. Sequence diagrams are well suited for this purpose, as they may describe the communication, both between the customers and the IPTV platform, and between the different parts of the platform. In version 2.x of UML (Unified Modeling Language), sequence diagrams were extended with additional constructs giving better expressiveness, but at the same time making them more difficult to create and use in an unambiguous manner. In this article, we present guidelines for developing sequence diagram specifications, and demonstrate the use of these guidelines on the IPTV example.

1 Introduction

IPTV (Internet Protocol Television), see e.g. [1], is a rapidly evolving suite of technologies, protocols, and standards. With IPTV, a potential exists for innovative new applications and services, including time-shifted video content and TV contents delivered to any display: mobile, PC, or home. However, the IP network infrastructure will have to accommodate much larger traffic volumes and, as opposed to Web-TV, preserve the loss-free quality of the IPTV stream as it traverses the network.

IPTV is an important part of Telenor's 3Play program, intended to defend and increase the market share and drive average revenue per user for customers in the retail customer market by providing innovative and high quality TV and video-on-demand services over IP. The proposed solution reuses or extends the existing business processes, service value

chains, and information systems where applicable, and establishes new interfaces towards the service platform (SmartVision from Thomson [2], see Figure 1) and the end-to-end monitoring solution for IPTV (Agama from Agama Technologies [3]).

The various aspects of the IPTV platform may be described using a wide variety of modelling languages, informal notations and ordinary prose. In this article, we focus on how sequence diagrams may be used to specify the communication between the different parts of the IPTV system, including its customers and other end users. In version 2.x of UML (Unified Modeling Language) [4], sequence diagrams were extended with features from MSC (Message Sequence Charts) [5]. These constructs made UML sequence diagrams more expressive than in the former UML 1.x [6], paving the way for a more extended usage of sequence diagrams in the development of larger systems.

Using principles from the STAIRS method [7], [8], sequence diagrams may be used to specify the behaviour of telecom systems in a way that is both intuitive and precise. This makes the diagrams suitable for various usages such as interaction with end users, availability analysis as described in e.g. [9], [10], or even as the basis for automatic transformation into code as envisioned in MDA (Model Driven Architecture) [11].

Sequence diagrams are usually incomplete specifications, typically describing only some example runs of the system and not the complete system behaviour. Using the refinement principles of STAIRS, a sequence diagram can be made more complete by information being added to it in a consistent manner, in the sense that all systems meeting the requirements of the refined diagram, will automatically also meet the requirements of the original diagram.

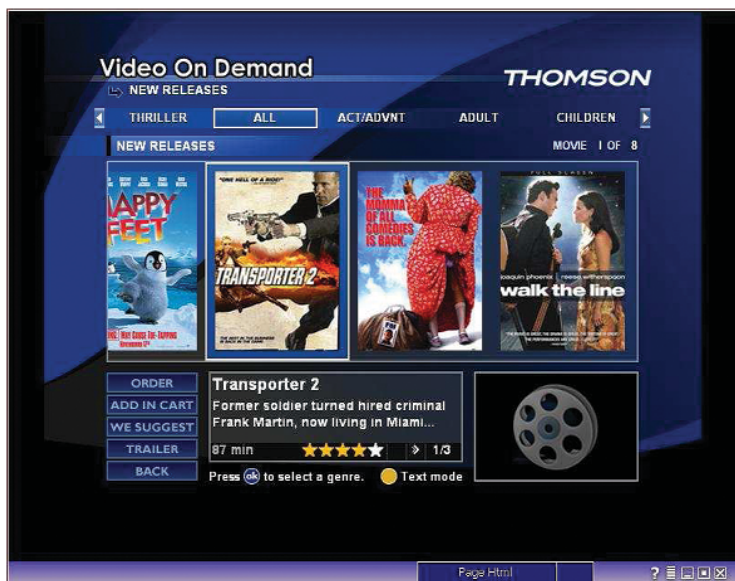


Figure 1 Screenshot from a video-on-demand service on the IPTV platform by Thomson (SmartVision)

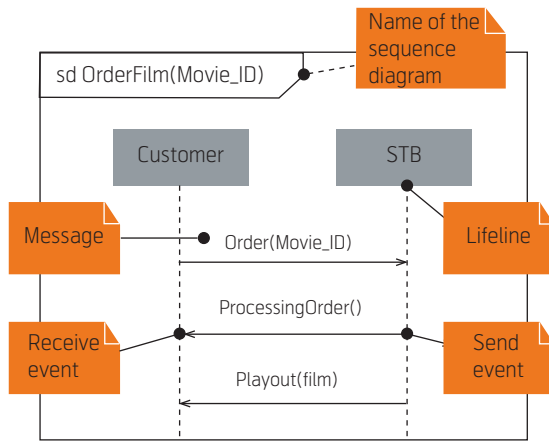


Figure 2 A simple sequence diagram describing how the customer interacts with the set-top box (STB) for ordering and watching a movie using IPTV

The remainder of this article is organized as follows: In Section 2 we give a brief introduction to sequence diagrams. Sections 3 and 4 describe how to create and refine sequence diagrams, respectively. Finally, Section 5 contains the conclusion.

2 The Nature of Sequence Diagrams

In this section we give a brief introduction to UML 2.x sequence diagrams. A sequence diagram describes system behaviour in the form of messages sent between lifelines, typically representing system components and/or users of the system. As a simple example, consider the sequence diagram in Figure 2,

describing the communication between a customer and the set-top box (STB) when the customer wants to watch a movie using IPTV. First, the customer sends the message `Order (Movie_ID)` to the STB, specifying the desired movie.¹⁾ The STB then replies with the receipt message `ProcessingOrder ()` before starting to play the requested movie.

To explain the meaning of a single sequence diagram, we use trace sets where each trace is a sequence of events representing one system run. The most typical examples of events are the sending and reception of a message. The diagram in Figure 2 includes six events, two for each message. In a system run following a given sequence diagram, the send event of each message must occur some time before the corresponding receive event. Also, events belonging to the same lifeline must occur in the topdown order given in the sequence diagram. Hence, the sequence diagram in Figure 2 describes two system runs, i.e. traces, both starting with the Customer sending the order, the STB receiving it, and the STB sending the `ProcessingOrder ()` message. Then, the two traces differ in whether the customer receiving the `ProcessingOrder ()` message or the STB sending the `Playout (film)` message happens first, before the customer receiving the `Playout (film)` message is the last event to happen in both cases.

In contrast to e.g. UML state machines and Java programs, sequence diagrams are usually incomplete specifications, describing only some of the possible system runs. Traces not described by a sequence

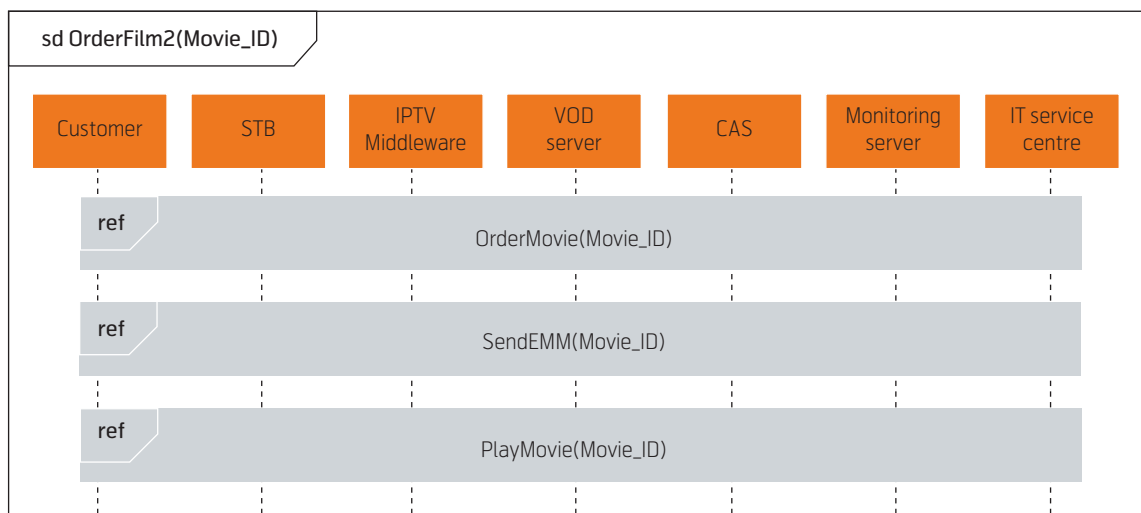


Figure 3 An overview of the example of ordering a film using IPTV, consisting of the three sub-scenarios `OrderMovie (Movie_ID)`, `SendEMM (Movie_ID)` and `PlayMovie (Movie_ID)`. The contents of these are given in Figures 4, 6 and 8, respectively

¹⁾ Typically, the customer will use the remote control to browse through the available selection of films before making his/her choice. In the diagram, we assume that this browsing has already been performed, as indicated by `Movie_ID` being a parameter to the diagram.

diagram are not necessarily undesirable, but the sequence diagram gives no information regarding their status. In such a setting, it is particularly important to be able to specify negative as well as positive traces. Positive traces represent valid system runs, while negative traces represent invalid system runs, i.e. traces that the system should *not* exhibit. If the system is able to produce a negative trace, then the system is not correct with respect to this sequence diagram specification.

In general, we say that a sequence diagram describes a set of positive and a set of negative traces. Traces not described as either positive or negative are referred to as inconclusive and may later be added as either positive or negative to the sequence diagram (see Section 4.1).

3 Creating Sequence Diagrams

In this section, we focus on explaining the main guidelines for creating sequence diagrams, i.e. how to use the different sequence diagram constructs in practical development. For each construct, we also give an example of how it may be used in developing a description of how to order and view a movie using IPTV.

An overview of the example is given in Figure 3. In addition to the set-top box (STB), the parts of the IPTV platform relevant for this example is the IPTV Middleware, a video-on-demand server (VOD server), a conditional access server (CAS), a monitoring server and the service centre. The sequence diagram in Figure 3 consists of three referenced diagrams; the contents of these will be given later. For the meaning of this sequence diagram, each reference should be taken as a syntactical shorthand for the contents of the referenced diagram.

3.1 Alternative Behaviours

As explained in Section 2, the sequence diagram in Figure 2 describes two different traces, depending on whether the Customer receives the `ProcessingOrder()` message before the STB sends the `PlayOut(film)` message or not. Which one of these we actually get when running the final system will typically depend on the amount of delay between the customer and the STB. Both traces are perfectly acceptable in the system, but it is also sufficient if the system is constructed in such a manner that only one of them may happen. Such implementation freedom, where the sequence diagram describes a set of (positive) traces that the system developers may select from when implementing the system, is an example of what is often referred to as underspecification.

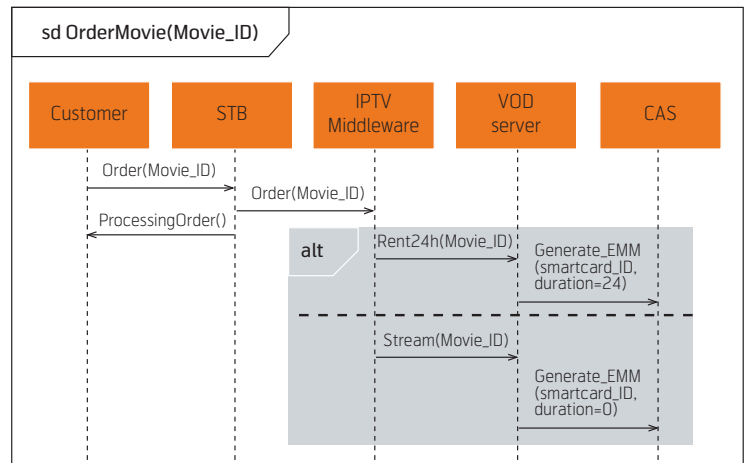


Figure 4 Description of `OrderMovie`, the first of the referenced sequence diagrams in Figure 3. The `alt` operator is used to describe alternative behaviour where the system developers may choose to implement one or more of the alternatives, but not necessarily all

Alternatives representing underspecification may also be specified using an operator called `alt`. An example sequence diagram using this operator is given in Figure 4. In this sequence diagram, the STB forwards the `Order(Movie_ID)` request to the IPTV middleware. The diagram then describes two alternatives for what may happen next. Either the IPTV middleware tells the video-on-demand (VOD) server that the movie identified by `Movie_ID` should be rented for 24 hours, or it tells the server to stream the movie. In both cases, the VOD server continues with sending a `Generate_EMM` message to the conditional access server (CAS), with the duration-parameter set to either 24 or 0 depending on the message received from the IPTV middleware. EMM stands for Entitlement Management Message, used for decrypting ECMs (Entitlement Control Messages) sent during the playout of the movie. These ECMs are in turn used for descrambling the movie signals. (The ECMs and descrambling will be described in Figure 8.)

A sequence diagram may also describe alternative traces representing different behaviour that should all be possible in the final system. Unfortunately, UML 2.x does not distinguish between the two different kinds of alternatives and uses the `alt` operator in both cases, leading to the same sequence diagram being understood differently by different people. To avoid potential confusion and misunderstanding, [7] proposed `xalt` as a new operator covering one of the traditional uses of `alt`.

As an example of the use of `xalt`, consider the sequence diagram in Figure 5. This diagram describes the monitoring that will take place during playout of the movie (the rest of playout will be described in Figure 8). The STB sends status messages to the

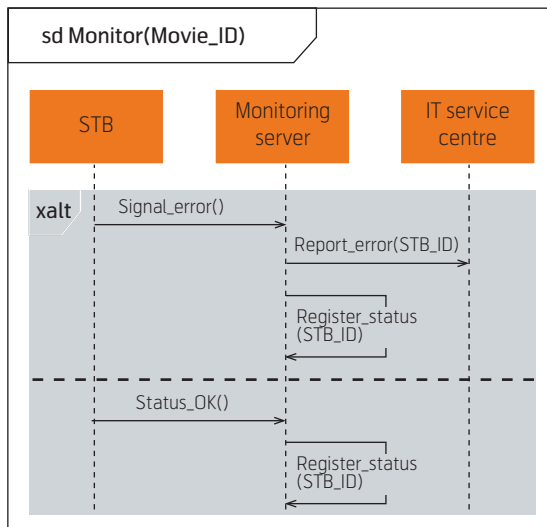


Figure 5 During playback of the movie the STB monitors the movie signals and sends report messages to the monitoring server. The **xalt** operator is used to describe alternative behaviours that should all be implemented in the system

monitoring server. If the server receives the error message `Signal_error()`, it should report the error to the IT service centre before registering the status in its database. If `Status_OK()` is received, no report is sent to the service centre, and the monitoring server simply registers the status.

As the STB may send both kinds of status messages and the monitoring server should be able to handle both of them, the alternatives are specified using the

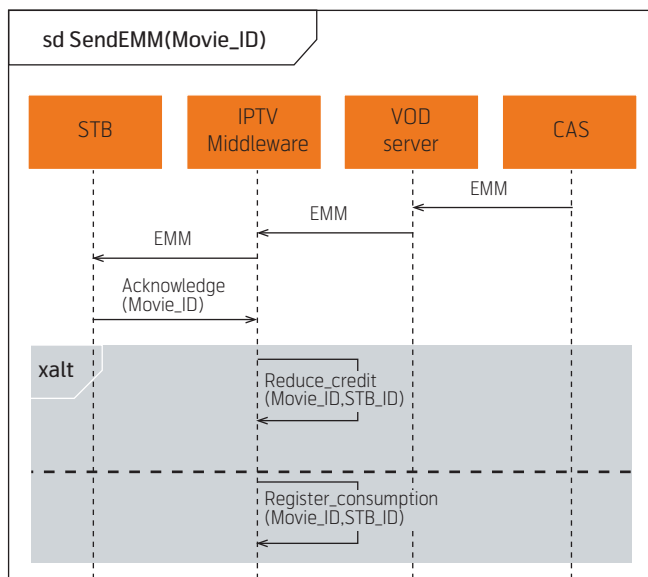


Figure 6 Description of `SendEMM`, the second of the referenced sequence diagrams in Figure 3. The **xalt** operator is used to describe alternative behaviours where there are some underlying constraints deciding when each of them may be chosen

Guidelines for using **alt** and **xalt**

- Use **alt** to specify alternatives that represent similar traces, i.e. to model
 - underspecification (implementation freedom).
- Use **xalt** to specify alternatives that must all be present in an implementation, i.e. to model
 - inherent nondeterminism, as in the specification of a coin toss;
 - alternative traces due to different inputs that the system must be able to handle (as in Figure 5);
 - alternative traces where the conditions for these being positive are abstracted away (as in Figure 6).

xalt operator. If the **alt** operator had been used instead, the alternatives had been interpreted as underspecification, and the result might have been a system only able to handle `Status_OK()` messages. Clearly, this is not desirable.

Another use of **xalt** is demonstrated in Figure 6, which describes how an EMM is sent to the STB and how the system registers the transmission of the movie. When receiving the EMM, the STB sends an acknowledgement to the IPTV middleware, which is responsible for charging the customer for the movie. This may be done either by reducing the customer credit (prepaid), or by simply registering the consumption of the movie postponing the actual charging to later. In this case, **xalt** is used to model alternatives where the exact conditions for each of them being chosen is not known (or abstracted away as being irrelevant at this level of description). In Section 3.2 we demonstrate how guards may be added in order to constrain the applicability of each of the alternatives.

A third use of **xalt** is in cases of nondeterminism, as when describing the tossing of a coin, where both heads and tails should be possible results of the toss. Such a sequence diagram description of a coin toss may be found in [12].

The important question when choosing between **alt** and **xalt** for describing alternative behaviour is: Do these alternatives represent similar traces in the sense that implementing only one is sufficient, or should all be implemented by the system? Use **alt** in the first case, **xalt** in the second.

3.2 Guarded and Constrained Behaviours

In Figure 6 **xalt** was used to specify that charging of a movie could be achieved either by immediately reducing the customer credit, or by registering the consumption preparing for later charging. With the current description the system may choose arbitrarily between these two, but as indicated in the previous

Guidelines for using guards

- Use guards in an **alt**/**xalt** construct to constrain the situations in which the different alternatives are positive.
- Always make sure that for each alternative, the guard is sufficiently general to capture all possible situations in which the described traces are positive.
- In an **alt** construct, make sure that the guards are exhaustive. If doing nothing is valid, specify this by using the empty diagram **skip**.

section, there exist some conditions for when each of them may be chosen. In Figure 7 these conditions are made explicit in the form of guards. With the guards the sequence diagram now specifies that the credit should be reduced only if the credit of the customer is at least as much as the movie charge, while consumption should be registered (and charging postponed) only if the customer credit is too low.

In this example, the guards added were mutually exclusive. However, this is not a general requirement when using guards. Another possibility could have been to add only the first guard, leaving the second alternative unguarded. Registering the consumption may then happen regardless of the credit, and if the credit is large enough, both reducing the credit and registering the consumption would be valid system behaviours.

The sequence diagram in Figure 7 also includes a constraint **EMM ok**, written in curly braces on the STB lifeline. A constraint specifies a condition that should always hold in the given scenario. In this case, the customer should be charged only if the EMM is ok (due to the fact that if the EMM is not ok, then the customer will not be able to watch the movie). What should happen if the EMM is not ok is not described in this diagram, i.e. it represents an example of inconclusive behaviour (cf. Section 2).

In general, if a guard (or constraint) holds, the behaviour specified is positive. If the guard (or constraint) does *not* hold, the given behaviour is negative and should *not* occur in the system. What the system should do instead will often be described in another part of the specification, but the behaviour may also be unspecified, i.e. inconclusive. As traces with a false guard represent negative behaviour, this has the important consequence that each guard should be carefully selected to make sure it covers *all* possible situations where the traces of the operand represent positive, i.e. valid, behaviours.

As will be demonstrated in the next section, guards may also be used in an **alt** construct. As in the case of

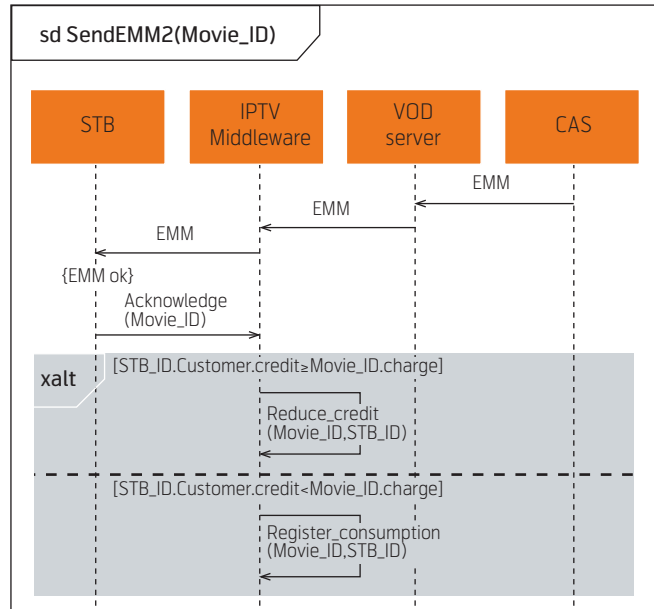


Figure 7 The sequence diagram of Figure 6 extended with guards constraining the applicability of the two alternative behaviours, and **EMM ok** as a constraint that should always hold in this scenario

xalt, the guards of an **alt** may be overlapping.

For consistency with the UML 2.x specification, we recommend ensuring that the guards are exhaustive, such that in any situation at least one of the operands may be chosen.

3.3 Repeated Behaviours

In our IPTV example it remains to describe the contents of **PlayMovie (Movie_ID)**, referenced in Figure 3. This is done by the sequence diagram in Figure 8. In IPTV the movie stream from the VOD server to the STB is scrambled in order to obfuscate the signal. Both the VOD server and the STB receive the key needed to scramble/descramble the stream as part of an ECM (Entitlement Control Message) created by the CAS. Because the obfuscation is relatively easy to break, new ECMs are created typically

Guidelines for repetition

- Use **loop** if the same scenario may happen several times in a row.
- The **loop** parameter states how many iterations that may be performed. Underspecification is achieved by letting the parameter be a set instead of a single number.
- A conditional loop may be mimicked by
 - setting the loop parameter to \mathbb{N} , the set of natural numbers;
 - placing the condition as a constraint at the top of the loop operand;
 - placing the negation of the constraint immediately after the loop.

every ten seconds. In order to ensure that each ECM reaches the STB before it is needed for descrambling, the ECMs are injected into the movie stream where appropriate.

In Figure 8 a guarded **alt** construct is used to describe that the CAS may create a new ECM if at least ten seconds have passed since the previous ECM, but it may also choose to do it less often. As a concrete CAS may choose to always send new ECMs if possible, the alternatives are here specified using the **alt** operator in accordance with the guidelines given in Section 3.1. The **loop** operator is used to describe that its operand (i.e. the possible sending of an ECM, scrambling, descrambling and playing out the movie stream) is repeated a number of times. This operator takes a parameter, a set of values where each value is either a natural number or infinity. This set indicates how many times the loop may be performed. In the case of our example, the parameter \mathbb{N} (the set of all

natural numbers) indicates that the loop happens a finite, but unknown, number of times. It is the constraints on the VOD server lifeline that specifies that the process should be repeated only as long as the movie is still playing. During playout, monitoring is also performed as described in the referenced sequence diagram, **Monitor(Movie_ID)** from Figure 5.

3.4 Negative Behaviours

As explained in the introduction, sequence diagrams are typically used to describe example runs of the system, and not for complete system descriptions. This makes it important to be able to describe not only positive, but also negative system behaviours stating what the system is *not* supposed to do. In Section 3.2, we described how false constraints and guards lead to negative behaviours. In STAIRS, negative behaviours may also be described using one of the operators **refuse**, **veto** and **assert**.

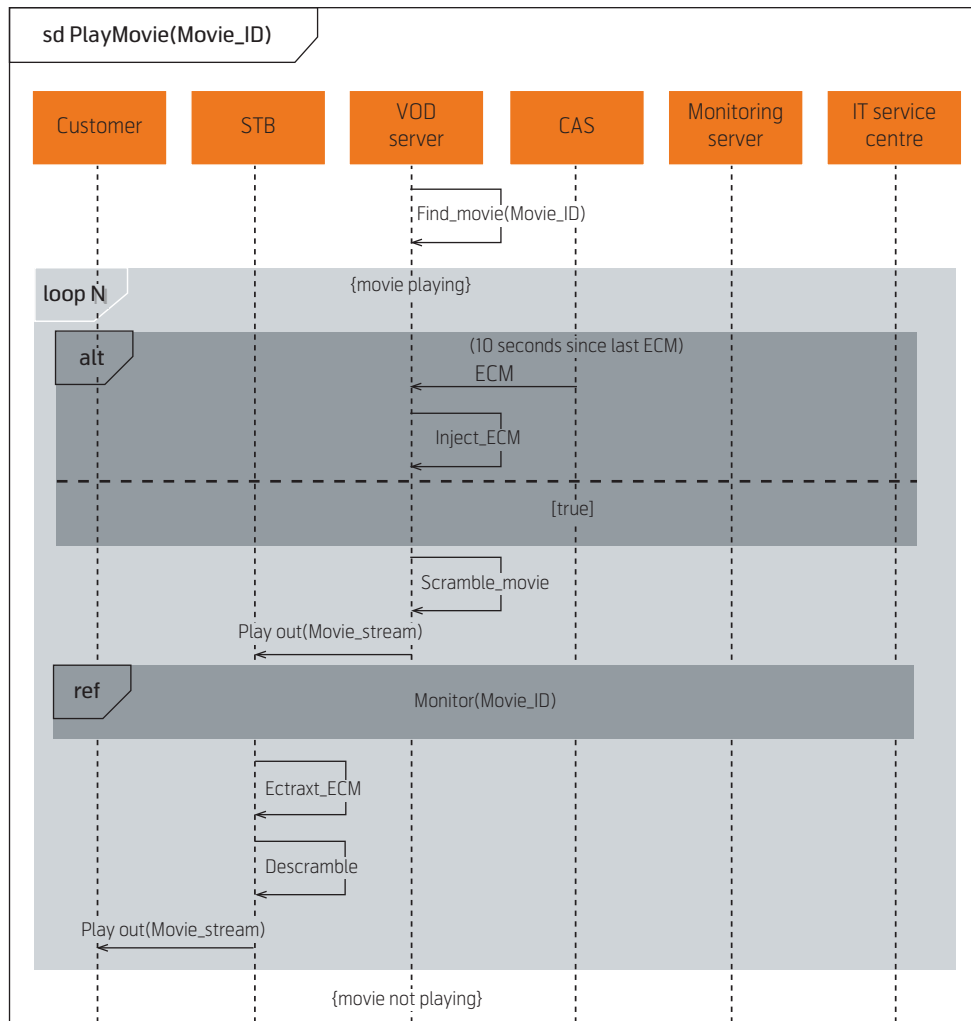


Figure 8 Description of **PlayMovie**, the third and last of the referenced sequence diagrams in Figure 3. ECMs (Entitlement Control Messages) contain the keys used for scrambling and descrambling the movie stream and are injected into the stream at appropriate places. The **loop** construct, together with the constraints on the VOD server lifeline, describes how the process is repeated a finite, but unknown number of times as long as the movie is still playing

Guidelines for negation

- To effectively constrain the implementation, the specification should include a reasonable set of negative traces.
- Use **refuse** when specifying that one of the alternatives in an **alt** construct represents negative traces.
- Use **veto** when the empty trace (i.e. doing nothing) should be positive, as when specifying a negative message in an otherwise positive scenario.
- Use **assert** on an interaction fragment when all possible positive traces for that fragment have been described.

The negation operators of UML 2.x are **assert** and **neg**. UML 2.x **assert** corresponds to **assert** as described here, while the **neg** operator of UML 2.x is in STAIRS split into **refuse** and **veto**. Similar to the case of **alt/xalt**, this is due to subtle, but important differences in the various uses of **neg**. A discussion of different interpretations of **neg** may be found in [13].

The use of **veto** and **assert** are exemplified in Figure 9, which is a modified version of the diagram in Figure 5. In the first operand, **assert** is used to describe that after the monitoring server receives the `Signal_Error()` message, it should report the error and register the status, but nothing else. In general, the **assert** operator states that the traces of its operand represent the only valid system behaviours, all other traces should be understood as negative.

In the second operand of the diagram in Figure 9, **veto** is used to describe that after the `Status_OK()` message, no error should be reported before the status is registered. In general, **veto** is used to describe that the traces of its operand represent negative behaviours, while skipping the operand and continuing with the subsequent behaviour is positive.

An example of the use of **refuse** is given in Figure 10, which is a modified version of the diagram in Figure 4. In this case, **refuse** is used to describe that the traces given in the second **alt** operand should *not* occur, i.e. streaming is not a possible system behaviour.

The operators **refuse** and **veto** are very similar in that both specify that the traces of its operand should be considered negative. The difference is that with **refuse**, continuing with the rest of the diagram is not an option. Replacing **refuse** with **veto** in Figure 10 would lead to the second **alt** operand describing that doing nothing (i.e. neither renting nor streaming the movie) should be considered positive system

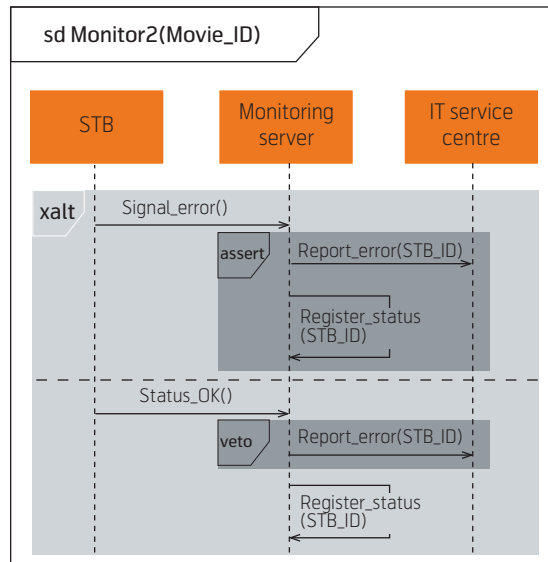


Figure 9 The sequence diagram of Figure 5 extended with negative behaviours. The **assert** operator is used to describe that the contents of its operand represent the only positive behaviours, while **veto** is used for describing what should not occur in an otherwise positive scenario

behaviour, which is not the intention. On the other hand, replacing **veto** with **refuse** in Figure 9 would lead to the second **xalt** operand having *no* positive traces. Again, this is not as desired.

4 Refining Sequence Diagrams

During a system development process, the system description, including the sequence diagrams, may be changed for several reasons. For instance, new requirements may be added, or existing requirements changed, due to an improved understanding of user needs or due to changes in the system environment (e.g. new technology becoming available). Errors in the original description may be corrected, and alternative descriptions addressing different concerns may be added (for instance, a customer and a system developer may be interested in very different aspects of the system).

In this article, we focus on refinement, which is the process of adding more information to the sequence diagram(s) describing the system and thereby getting closer to a complete system description. The goal is not necessarily a complete description, but rather a description with sufficiently many details to obtain the required understanding of the system.

STAIRS defines three basic refinement notions referred to as supplementing, narrowing and detailing, which will be described in the following subsections. These three refinement notions all ensure that

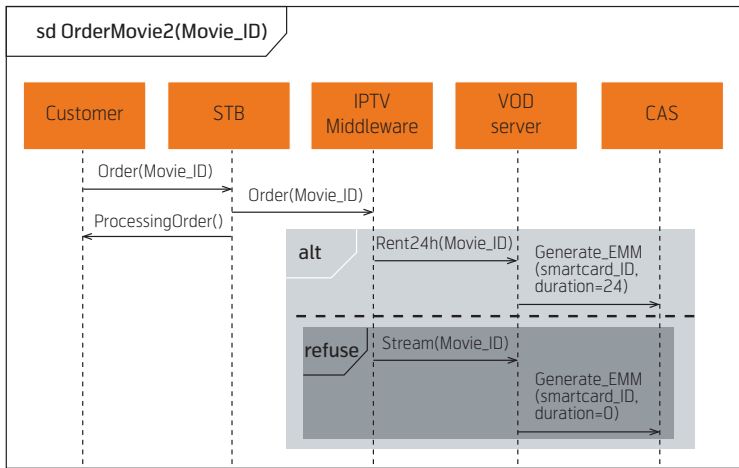
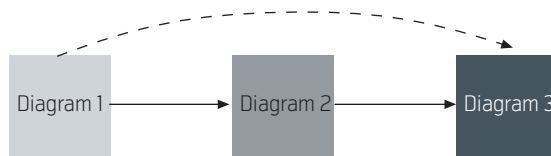


Figure 10 A modification of the sequence diagram of Figure 4 using the refuse operator to describe that one of the alt operands represents negative behaviours

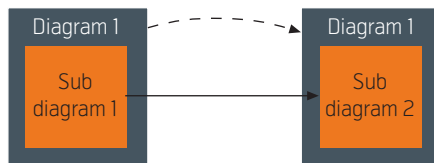
Guidelines for supplementing

- Use supplementing to add positive or negative traces to the sequence diagram.
- When supplementing, all of the original positive traces must remain positive and all of the original negative traces must remain negative.
- Do not use supplementing on the operand of an assert.

sequence diagrams may be developed in a step-wise and compositional manner as illustrated in Figure 11. Step-wise development means not only that a series of refinement steps will lead to a refinement of the original diagram, but also that the various refinement steps may be combined into one single step.



a) Step-wise development. After a series of valid refinement steps, the result will also be a refinement of the original diagram (and every other diagram in the refinement chain).



b) Compositional development. Each part of the sequence diagram may be refined separately. When composed with the rest of the sequence diagram again, the result will be a refinement of the original diagram.

Figure 11 Step-wise and compositional development. Solid arrows indicate the use of one of the refinement notions, while the dashed arrows indicate implied refinements

4.1 Supplementing

Due to the partial nature of sequence diagrams, there will usually be many traces described as neither positive nor negative in the diagram. As explained in Section 2, such traces are referred to as inconclusive. By supplementing, previously inconclusive traces are added to the diagram, either as positive or negative. This is illustrated in the left part of Figure 12. Supplementing will most typically be performed during the early phases of the system development process, for instance for capturing additional user requirements or adding fault tolerance to the system.

An example of supplementing is the diagram in Figure 9, which adds negative behaviours to the original diagram in Figure 5. All traces that were positive in the original diagram are positive also in the refinement. Figure 9 may be further supplemented by adding more positive and/or negative behaviours. However, supplementing should not be used on the assert operand, as assert assumes that all possible positive behaviour for that part of the diagram has been described already.

4.2 Narrowing

Narrowing means to reduce underspecification by redefining positive traces as negative, as illustrated in the right part of Figure 12. Using narrowing, there are thus fewer possible system behaviours, and less implementation freedom left in the specification.

One example of narrowing is the diagram in Figure 7, which adds constraints and guards to the original diagram in Figure 6. In the original diagram, both xalt alternatives could be chosen regardless of the customer credit, while adding the guards reduces the applicability of each of the alternatives. Traces with a negative guard in Figure 7 (i.e. reducing the credit even if the movie charge is higher than the current

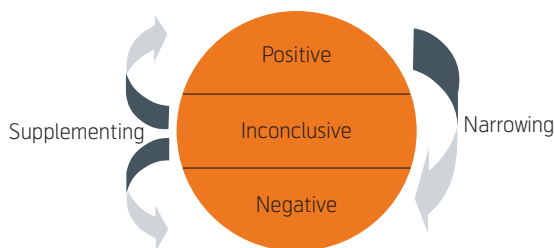


Figure 12 Supplementing refinement means adding more behaviours to the description by re-categorizing inconclusive traces as positive or negative, while narrowing refinement means reducing underspecification by redefining positive traces as negative

Guidelines for narrowing

- Use narrowing to limit underspecification (i.e. implementation freedom) by redefining positive traces as negative.
- In cases of narrowing, all of the original negative traces must remain negative.
- Guards may be added to an **alt** construct as a legal narrowing step.
- Guards may be added to an **xalt** construct as a legal narrowing step.
- Guards may be narrowed, i.e. the refined condition must imply the original one.

credit) were positive in Figure 6, making this an example of narrowing. In general, adding guards to an **alt/xalt** construct always results in a narrowing refinement. Similarly, guards may be narrowed as long as the refined guard implies the original one.

Another example of narrowing is the diagram in Figure 10, which redefines the positive traces in one of the original **alt** operands in Figure 4 as negative. This may be understood as a design/implementation choice, deciding to offer only 24 hour rental and not streaming rental.

4.3 Detailing

The last of the basic refinement notions is detailing, which means reducing the level of abstraction by decomposing one or more lifelines and/or messages. With detailing, positive behaviours remain positive, and negative behaviours remain negative, but the traces describing the behaviours may include more details.

As an example, consider the two versions of *Order-Film* given in Figures 2 and 3 (together with the diagrams describing the referenced behaviours). From the customer's point of view, the basic behaviour of these two diagrams is the same. The customer only interacts with the set-top box, and may view this as

Guidelines for detailing

- Use detailing to increase the level of granularity of the specification by decomposing lifelines and/or messages.
- When detailing, document the decomposition by explaining the relation between the concrete and the abstract lifelines, and between the concrete and the abstract messages.
- When detailing, make sure that the refined traces are equal to the original ones when abstracting away the details introduced at the concrete level.

the complete IPTV system. As long as the customer gets the desired services, the fact that the IPTV platform really consists of a set-top box, the IPTV middleware, a video-on-demand server etc, is irrelevant for him/ her. Similarly, the customer does not care whether the film is played out as one long stream (the message *Playout (film)* in Figure 2) or as a continuous sequence consisting of smaller parts of the film (the *Playout (Movie_stream)* messages in the refinement in Figure 3, as given by the diagram in Figure 8).

5 Conclusions

In this article, we have presented guidelines for creating and refining sequence diagrams in a sound manner. As a running example, we used the scenario where a customer wants to watch a movie using IPTV. With their focus on the communication and information flow between the different parts of the IPTV platform, sequence diagrams are well suited for describing this and similar scenarios. The given guidelines are particularly useful for choosing which sequence diagram construct to use in each situation, and for supporting a step-wise and compositional development of the specification.

Acknowledgements

The research on which this article reports has partly been funded by the Research Council of Norway through the SARDAS project (15295/431).

References

- 1 *IPTV*. Wikipedia, the free encyclopedia. <http://en.wikipedia.org/wiki/IPTV> (November 2008).
- 2 *IPTV – Smartvision – TV over DSL – Video on Demand Service – Thomson products*. http://www.thomson.net/GlobalEnglish/Products/smartvision_video_service_platform/smartvision-iptv/Pages/default.aspx (November 2008).
- 3 *Agama Technologies – Digital TV monitoring and quality assurance solutions*. <http://www.agama.tv/> (November 2008).
- 4 *OMG. UML 2.1.2 Superstructure Specification*. Object Management Group, 2007 (document: formal/07-11-02 edition)
- 5 *ITU. Message Sequence Chart (MSC)*. Geneva, International Telecommunication Union, 1999. (Recommendation Z.120)

- 6 OMG. *UML 1.5 Specification*, document: formal/03-03-01 edition. Object Management Group, 2003.
- 7 Haugen, Ø, Stølen, K. STAIRS – Steps to analyze interactions with refinement semantics. **In:** *Proc. UML 2003 – The Unified Modeling Language: Modeling Languages and Applications*. Lecture Notes in Computer Science, 2863, 388-402, Springer, 2003.
- 8 Haugen, Ø, Husa, K E, Runde, R K, Stølen, K. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 4 (4), 349-458, 2005.
- 9 Lund, M S. Model-based testing with the escalator tool. *Teletronikk*, 105 (1), 117-125, 2009 (this issue).
- 10 Feudjio, A G V, Schieferdecker, I. Availability testing for web services. *Teletronikk*, 105 (1), 81-89, 2009 (this issue).
- 11 OMG. *MDA Guide Version 1.0.1*. Object Management Group, 2003 (document: ormsc/03-06-01 edition)
- 12 Refsdal, A, Runde, R K, Stølen, K. Underspecification, inherent nondeterminism and probability in sequence diagrams. **In:** *Proc. Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, Lecture Notes in Computer Science, 4037, 138-155, Springer, 2006.
- 13 Runde, R K, Haugen, Ø, Stølen, K. How to transform UML neg into a useful construct. **In:** *Norsk Informatikkonferanse NIK'2005*, 55-66, Tapir, 2005.

For a presentation of Ragnhild Kobro Runde please turn to page 2.

Jan Øyvind Agedal is a Solution Architect at Telenor Norway, Product&IS. His current project assignment is as Chief Architect for the 3Play TV project, having an e2e technical responsibility for the IPTV technologies in the project. Agedal is MSc (siv.ing.) from NTNU, Norway and Dr.Scient. from UiO, Norway. Agedal was until 2007 Senior Research Scientist at SINTEF.