Deriving Tests from UML 2.0 Sequence Diagrams with neg and assert

Mass Soldal Lund and Ketil Stølen
Department of Informatics, University of Oslo, Norway
SINTEF ICT, Norway
{msl,kst}@sintef.no

ABSTRACT

In this paper we define an algorithm for deriving tests from UML 2.0 sequence diagrams based on the operational semantics for sequence diagrams defined in [13]. The algorithm is a modified and adapted version of the algorithm presented in [19, 20]. This modified algorithm is based on the standard semantic model of sequence diagrams and allows diagrams to contain the operators neg and assert. The derived tests are themselves sequence diagrams.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Algorithms, verification

Keywords

UML, sequence diagrams, test derivation

1. INTRODUCTION

Unified Modeling Language (UML) sequence diagrams [16] and their predecessor Message Sequence Charts (MSC) [9] are specification languages that have proved themselves to be of great practical value in system development. As high level specifications of communication scenarios they should provide an excellent starting point for testing of the system under development.

In this paper we define an algorithm for deriving tests from sequence diagram specifications that takes into consideration the partial nature of sequence diagrams as well as the notion of invalid and universal behavior introduced in UML version 2.0 with the operators neg and assert. The algorithm is an adaption from [19, 20] and stays within the same framework for conformance testing ("ioco-testing").

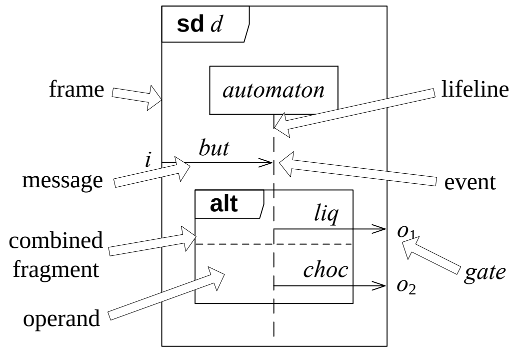

Figure 1: Sequence diagram (example borrowed from [20])

The algorithm generates tests in the form of sequence diagrams. We believe that these tests are as well suited to test other specifications and designs as to test implementations. Specifically, we may apply the same operational semantics to execute these tests as we use to generate them. Further we believe that by using sequence diagrams as the format of the tests, we should get tests that are easily readable by the people who write UML specifications.

In sum the contribution of this paper is: A test generation scheme based on a well-known algorithm for conformance testing and a formal operational semantics for sequence diagrams, which takes as input sequence diagrams that may contain the operators assert and neg and that produces tests in the form of sequence diagrams. We are not aware of any other approach that does this.

The structure of the paper is as follows: In the reminder of this section we give a brief introduction to UML sequence diagrams. In section 2 we present an abstract syntax for sequence diagrams and in section 3 we present the operational semantics. Section 4 provides the format of the tests generated and section 5 the test generation algorithm. In section 6 we present related work, and finally, in section 7 we draw the main conclusions.

1.1 Sequence diagrams

Sequence diagrams is a graphical specification language defined in the UML 2.0 standard [16]. The standard defines the graphical notation, but also an abstract syntax for the diagrams. Hence the language has a well-defined syntax.

Figure 1 shows a sequence diagram d in the graphical notation. A sequence diagram consists of a *frame*, representing the environment of the specified system, and one or more *lifelines*, representing components of the system. Arrows represent *messages* sent between lifelines or between a lifeline and the environment. If the beginning or the end of an arrow is at a lifeline this represents an *event*. Where an arrow meets the frame, there is a *gate*. *Combined fragments* are operators, like the choice operator alt, and each combined fragment has one or more *operands*.

1.2 Semantic model

The UML standard defines the semantics of sequence diagrams informally. Most notably, this is a trace based semantics. In [7,8,17] a trace based denotational semantics for sequence diagrams is formalized, called the STAIRS semantics. The STAIRS semantics is the underlying foundation of this work.

Both the UML standard and the STAIRS semantics divide the trace universe into valid, invalid and inconclusive traces for a given diagram, and hence regard sequence diagrams as partial specifications. This differ from the LTS based approach of [19,20] which only has valid and invalid traces. Hence, one of the modifications to the test derivation algorithm in [19,20] is an adaption to a semantic model with inconclusive traces.

More formally: Let $\mathcal{H}$ denote the trace universe. The denotation of a sequence diagram d is a pair of sets of traces $[\![d]\!] = (p, n)$ such that $p \subseteq \mathcal{H}$ is interpreted as the valid traces and $n \subseteq \mathcal{H}$ as the invalid traces. $p \cup n$ needs not exhaust the trace universe, so $\mathcal{H} \setminus (p \cup n)$ may differ from $\emptyset$ and is referred to as inconclusive traces. Further we allow specifications to be inconsistent, i.e., we may have that $p \cap n \neq \emptyset$.

Universal behavior is specified by the UML operator assert. To specify invalid behavior we introduce a new operator refuse. The reason for this is that the UML operator neg is ambiguous and has (at least) two distinct interpretations [17]. Following [17] we define refuse as one of these interpretations and let neg represent the other as a derived operator. In the denotational semantics assert and refuse are defined as:

$$\begin{aligned}
[\![\mathsf{assert}\ d]\!] &\overset{\text{def}}{=} (p, n \cup (\mathcal{H} \setminus p)) \\
[\![\mathsf{refuse}\ d]\!] &\overset{\text{def}}{=} (\emptyset, p \cup n)
\end{aligned}$$

Note that refuse defines all valid and invalid behavior as invalid, so nested refuse does not yield valid behavior, and assert defines all behavior that is not valid as invalid and hence empties the set of inconclusive traces.

2. SYNTAX

We apply an abstract syntax for sequence diagrams defined in [7,8].

2.1 Messages, events and traces

The atom of a sequence diagram is the event. An event consists of a message and a kind where the kind decides whether it is the transmit event or the receive event of the message. A message is a signal, which represents the contents of the message, with the addresses of the transmitter and the receiver. Formally a signal is a label, and we let $\mathcal{S}$ denote the set of all signals. The transmitters and receivers are lifelines. Let $\mathcal{L}$ denote the set of all lifelines. Further let $\mathcal{G}$ denote the set of all gates. We treat gates as special kinds of lifelines, hence we have $\mathcal{G} \subset \mathcal{L}$. Gates are conceptually different from lifelines, but in this way definitions involving lifelines will carry over to gates, and this has some practical advantages. A message m is defined as a triple:

$$(s, tr, re) \in \mathcal{S} \times \mathcal{L} \times \mathcal{L}$$

with signal s, transmitter tr and receiver re. $\mathcal{M}$ denotes the set of all messages.

We let $\mathcal{K} = \{!, ?\}$ be the set of kinds, where ! represents transmit and ? represents receive. An event e is then a pair of a kind and a message:

$$(k, m) \in \mathcal{K} \times \mathcal{M}$$

$\mathcal{E}$ denotes the set of all events. We sometimes write $!s$ and $?s$ as shorthand for the events $(!, (s, tr, re))$ and $(?, (s, tr, re))$.

On messages we define a transmitter function $tr.\_ \in \mathcal{M} \to \mathcal{L}$ and a receiver function $re.\_ \in \mathcal{M} \to \mathcal{L}$:

$$tr.(s, tr, re) \overset{\text{def}}{=} tr \qquad re.(s, tr, re) \overset{\text{def}}{=} re$$

On events we define a kind function $k.\_ \in \mathcal{E} \to \mathcal{K}$ and a message function $m.\_ \in \mathcal{E} \to \mathcal{M}$:

$$k.(k, m) \overset{\text{def}}{=} k \qquad m.(k, m) \overset{\text{def}}{=} m$$

We let the transmitter and receiver functions also range over events, $tr.\_, re.\_ \in \mathcal{E} \to \mathcal{L}$:

$$tr.(k, m) \overset{\text{def}}{=} tr.m \qquad re.(k, m) \overset{\text{def}}{=} re.m$$

Further we define a lifeline function $l.\_ \in \mathcal{E} \to \mathcal{L}$ that returns the lifeline of an event and a function $l^{-1}.\_ \in \mathcal{E} \to \mathcal{L}$ that returns the opposite lifeline of an event (i.e. the receiver of its message if its kind is transmit and the transmitter of its message if its kind is receive), which may be a gate:

$$l.e \overset{\text{def}}{=} \begin{cases} tr.e & \textbf{if } k.e = ! \\ re.e & \textbf{if } k.e = ? \end{cases} \qquad l^{-1}.e \overset{\text{def}}{=} \begin{cases} tr.e & \textbf{if } k.e = ? \\ re.e & \textbf{if } k.e = ! \end{cases}$$

A trace is a (finite or infinite) sequence of events:

$$\langle e_1, e_2, \ldots, e_i, \ldots \rangle$$

We let $t_1 \frown t_2$ denote concatenation of the traces t_1 and t_2 and $\langle\rangle$ denote the empty trace.

2.2 Diagrams

A sequence diagram is built from events, the binary operators seq, par, strict and alt, and the unary operators loop, neg, refuse and assert. In addition we let skip represent the empty sequence diagram. Let $\mathcal{D}$ be the set of all syntactically correct sequence diagrams. $\mathcal{D}$ is defined recursively as follows:

$$\begin{array}{rcl}
 & & \mathsf{skip} \in \mathcal{D} \\
e \in \mathcal{E} \wedge l.e \notin \mathcal{G} & \Rightarrow & e \in \mathcal{D} \\
d_1, d_2 \in \mathcal{D} & \Rightarrow & d_1\ \mathsf{seq}\ d_2 \in \mathcal{D} \wedge d_1\ \mathsf{par}\ d_2 \in \mathcal{D} \wedge \\
 & & d_1\ \mathsf{strict}\ d_2 \in \mathcal{D} \wedge d_1\ \mathsf{alt}\ d_2 \in \mathcal{D} \\
d \in \mathcal{D} & \Rightarrow & \mathsf{neg}\ d \in \mathcal{D} \wedge \mathsf{refuse}\ d \in \mathcal{D} \wedge \\
 & & \mathsf{assert}\ d \in \mathcal{D} \\
d \in \mathcal{D} \wedge n \in \mathbb{N} & \Rightarrow & \mathsf{loop}\langle n \rangle\ d \in \mathcal{D}
\end{array}$$

where $\mathbb{N}$ is the set of natural numbers including zero.

Composition of diagrams by matching and elimination of gates is outside the scope of this paper. Note therefore that this is a grammar for sequence diagrams and not composition as such.

As can be expected, we have associativity of alt, par, seq and strict. We also have commutativity of alt and par. We have that the empty sequence diagram skip is the identity element of seq, par and strict, and define:

$$\begin{aligned}\mathsf{neg}\ d &\overset{\mathsf{def}}{=} \mathsf{skip\ alt\ refuse}\ d\\ \mathsf{loop}\langle 0\rangle\ d &\overset{\mathsf{def}}{=} \mathsf{skip}\end{aligned}$$

We assume the following precedence of the operators: refuse, assert, loop, neg, seq, strict, par, alt, such that refuse binds stronger than assert, etc.

Note that an event at a gate cannot be part of a diagram. Hence, the representation of diagram d in figure 1 is:

$$d = ?but\ \mathsf{seq}\ (!liq\ \mathsf{alt}\ !choc)$$

The function $ll.\_ \in \mathcal{D} \rightarrow \mathbb{P}(\mathcal{L})$ returns the set of lifelines present in a diagram. For $e \in \mathcal{E}$ and $d, d_1, d_2 \in \mathcal{D}$ the function is defined recursively:

$$\begin{aligned}ll.e &\overset{\mathsf{def}}{=} \{l.e\}\\ ll.\mathsf{skip} &\overset{\mathsf{def}}{=} \emptyset\\ ll.(\mathsf{op}\ d) &\overset{\mathsf{def}}{=} ll.d,\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}, \mathsf{loop}\}\\ ll.(d_1\ \mathsf{op}\ d_2) &\overset{\mathsf{def}}{=} ll.d_1 \cup ll.d_2,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}\}\end{aligned}$$

The function $gates.\_ \in \mathcal{D} \rightarrow \mathbb{P}(\mathcal{G})$ returns the gates of a diagram:

$$\begin{aligned}gates.e &\overset{\mathsf{def}}{=} \begin{cases}\{l^{-1}.e\} & \textbf{if } l^{-1}.e \in \mathcal{G}\\ \emptyset & \textbf{if } l^{-1}.e \notin \mathcal{G}\end{cases}\\ gates.\mathsf{skip} &\overset{\mathsf{def}}{=} \emptyset\\ gates.(\mathsf{op}\ d) &\overset{\mathsf{def}}{=} gates.d,\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}, \mathsf{loop}\}\\ gates.(d_1\ \mathsf{op}\ d_2) &\overset{\mathsf{def}}{=} gates.d_1 \cup gates.d_2,\\ &\quad\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}\}\end{aligned}$$

The function $ev.\_ \in \mathcal{D} \rightarrow \mathbb{P}(\mathcal{E})$ returns the events of a diagram:

$$\begin{aligned}ev.e &\overset{\mathsf{def}}{=} \{e\}\\ ev.\mathsf{skip} &\overset{\mathsf{def}}{=} \emptyset\\ ev.(\mathsf{op}\ d) &\overset{\mathsf{def}}{=} ev.d,\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}, \mathsf{loop}\}\\ ev.(d_1\ \mathsf{op}\ d_2) &\overset{\mathsf{def}}{=} ev.d_1 \cup ev.d_2,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}\}\end{aligned}$$

We assume there are no syntactical repetition of gates in a diagram:

$$d_1\ \mathsf{op}\ d_2 \in \mathcal{D} \Rightarrow gates.d_1 \cap gates.d_2 = \emptyset$$

3. OPERATIONAL SEMANTICS

The operational semantics is defined by the combination of two transition systems, which we refer to as the *execution system* and the *projection system*. The execution system is a transition system:

$$[\_, \_] \in \mathcal{B} \times \mathcal{D}$$

where $\mathcal{B}$ represents the set of all states of the communication medium and $\mathcal{D}$ the set of all syntactically correct sequence diagrams. The projection system is a transition system:

$$\Pi(\_, \_, \_) \in \mathbb{P}(\mathcal{L}) \times \mathcal{B} \times \mathcal{D}$$

The projection system is used for finding enabled events at each stage of the execution and is defined recursively in a way allowing fairness between the lifelines in the sequence diagram.

These two systems work together in such a way that for each step in the execution, the execution system updates the projection system by passing on the current state of the communication medium, and the projection system updates the execution system by returning the state of the diagram after the execution of the event.

We also formalize a meta level that encloses the execution system.

3.1 Communication model

The communication medium is defined as a set of fifo-buffers. Let $\mathcal{F} = \mathcal{L} \times \mathcal{L} \times \mathcal{M}^*$ be the set of all fifo-buffers, and $\mathcal{B} = \mathbb{P}(\mathcal{F})$. We write $(tr, re) : b$ for $(tr, re, b) \in \mathcal{F}$. For $\beta_1, \beta_2 \in \mathcal{B}$ we define:

$$\begin{aligned}&\beta_1 \ll \beta_2 \overset{\mathsf{def}}{=}\\ &\quad \{(tr, re) : b_1 {}^\frown b_2 \mid (tr, re) : b_1 \in \beta_1 \wedge (tr, re) : b_2 \in \beta_2\}\\ &\quad \cup \{(tr, re) : b_1 \mid (tr, re) : b_1 \in \beta_1 \wedge \neg\exists b_2 : (tr, re) : b_2 \in \beta_2\}\\ &\quad \cup \{(tr, re) : b_2 \mid (tr, re) : b_2 \in \beta_2 \wedge \neg\exists b_1 : (tr, re) : b_1 \in \beta_1\}\end{aligned}$$

Over $\mathcal{B}$ we define the functions $add \in \mathcal{B} \times \mathcal{M} \rightarrow \mathcal{B}$ (adds a message), $rm \in \mathcal{B} \times \mathcal{M} \rightarrow \mathcal{B}$ (removes a message) and $ready \in \mathcal{B} \times \mathcal{M} \rightarrow \mathbb{B}$ (returns **true** iff the communication medium is in a state where it can deliver the message):

$$\begin{aligned}add(\beta, m) &\overset{\mathsf{def}}{=} \beta \ll \{(tr.m, re.m) : \langle m\rangle\}\\ rm(\beta, m) &\overset{\mathsf{def}}{=} (\beta \setminus \{(tr.m, re.m) : \langle m\rangle {}^\frown b \mid\\ &\qquad (tr.m, re.m) : \langle m\rangle {}^\frown b \in \beta\}) \cup\\ &\quad\ \{(tr.m, re.m) : b \mid\\ &\qquad (tr.m, re.m) : \langle m\rangle {}^\frown b \in \beta\}\\ ready(\beta, m) &\overset{\mathsf{def}}{=} \exists b : (tr.m, re.m) : \langle m\rangle {}^\frown b \in \beta\end{aligned}$$

A function $update \in \mathcal{B} \times \mathcal{E} \rightarrow \mathcal{B}$ is defined as:

$$update(\beta, e) \overset{\mathsf{def}}{=} \begin{cases}add(\beta, m.e) & \textbf{if } k.e = !\\ rm(\beta, m.e) & \textbf{if } k.e = ?\end{cases}$$

Because events at gates are not present in the syntactical representation of a sequence diagram, we must assume that messages sent from gates are available in the execution of the diagram. We do this by defining an initial state of the communication medium in the following way:

$$\begin{aligned}initC.e &\overset{\mathsf{def}}{=} \{(tr.e, re.e) : m.e\}\\ &\qquad \textbf{if } k.e = ? \wedge tr.e \in \mathcal{G}\\ initC.e &\overset{\mathsf{def}}{=} \emptyset\ \textbf{if } k.e = ! \vee tr.e \notin \mathcal{G}\\ initC.\mathsf{skip} &\overset{\mathsf{def}}{=} \emptyset\\ initC.(d_1\ \mathsf{op}\ d_2) &\overset{\mathsf{def}}{=} initC.d_1 \cup initC.d_2,\\ &\quad\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}\}\\ initC.(\mathsf{op}\ d) &\overset{\mathsf{def}}{=} initC.d,\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}\}\\ initC.(\mathsf{loop}\langle n\rangle\ d) &\overset{\mathsf{def}}{=} initC.d \ll initC.(\mathsf{loop}\langle n-1\rangle\ d)\end{aligned}$$

Because all gate names are unique, this means the initial state of the communication medium will contain a separate fifo for each message sent from a gate to a lifeline in the diagram. The exception is messages from gates inside loops, where the fifo will be a sequence of (identical) messages which length corresponds to the number of iterations of the loop.

3.2 The execution system

The execution system has two rules. The first rule represents the execution of a single event and uses the projection system to find an enabled event to execute. It is defined as:

$$\begin{array}{l}[\beta, d] \xrightarrow{e} [update(\beta, e), d'] \\ \quad \textbf{if } \Pi(ll.d, \beta, d) \xrightarrow{e} \Pi(ll.d, \beta, d') \wedge e \in \mathcal{E}\end{array}$$

The second rule of the execution system executes silent events. The rules of the projection system produce silent events when resolving the sequence diagram operators refuse, assert, and loop. We define the set of silent events to be:

$$\mathcal{T} = \{\tau_{refuse}, \tau_{assert}, \tau_{loop}\}$$

with $\mathcal{E} \cap \mathcal{T} = \emptyset$. In the following we let τ range over the elements of $\mathcal{T}$.

The reason for introducing all these different silent events will be evident when we define the meta execution. The execution rule itself is obvious:

$$[\beta, d] \xrightarrow{\tau} [\beta, d'] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{\tau} \Pi(ll.d, \beta, d') \wedge \tau \in \mathcal{T}$$

The empty diagram skip cannot be rewritten, but we assert that it produces the empty trace, i.e.:

$$[\beta, \mathsf{skip}] \xrightarrow{\langle\rangle} [\beta, \mathsf{skip}]$$

This also means that execution terminates if skip is reached.

3.3 The projection system

The simplest case is the diagram consisting of only one event e. In this case the system delivers the event if the event is enabled with respect to a set of lifelines and the state of the communication medium. The event must belong to one of the lifelines in the set L; this argument of the system is introduced to make the rules for weak sequencing work properly. Secondly, the event must either be a transmit event or its message must be available in the communication medium:

$$\begin{array}{l}\Pi(L, \beta, e) \xrightarrow{e} \Pi(L, \beta, \mathsf{skip}) \\ \quad \textbf{if } l.e \in L \wedge (k.e = ! \vee ready(\beta, m.e))\end{array}$$

The weak sequencing operator seq defines a partial order on the events in a diagram; the ordering of events on each lifeline and between the transmit and receive of a message is preserved, but all other ordering of events is arbitrary. Because of this, there may be enabled events in both the left and the right argument of a seq if there are lifelines represented in the right argument of the operator that are not represented in the left argument. This leads to two rules for the seq operator:

$$\begin{array}{l}\Pi(L, \beta, d_1 \textsf{ seq } d_2) \xrightarrow{e} \Pi(L, \beta, d_1' \textsf{ seq } d_2) \\ \quad \textbf{if } ll.d_1 \cap L \neq \emptyset \wedge \\ \qquad \Pi(ll.d_1 \cap L, \beta, d_1) \xrightarrow{e} \Pi(ll.d_1 \cap L, \beta, d_1')\end{array}$$

$$\begin{array}{l}\Pi(L, \beta, d_1 \textsf{ seq } d_2) \xrightarrow{e} \Pi(L, \beta, d_1 \textsf{ seq } d_2') \\ \quad \textbf{if } L \setminus ll.d_1 \neq \emptyset \wedge \\ \qquad \Pi(L \setminus ll.d_1, \beta, d_2) \xrightarrow{e} \Pi(L \setminus ll.d_1, \beta, d_2')\end{array}$$

The strict operator defines a strict ordering of events, such that all events on the left side of the operator must be executed before the event on the right side:

$$\begin{array}{l}\Pi(L, \beta, d_1 \textsf{ strict } d_2) \xrightarrow{e} \Pi(L, \beta, d_1' \textsf{ strict } d_2) \\ \quad \textbf{if } ll.d_1 \cap L \neq \emptyset \wedge \\ \qquad \Pi(ll.d_1 \cap L, \beta, d_1) \xrightarrow{e} \Pi(ll.d_1 \cap L, \beta, d_1')\end{array}$$

The parallel operator par specifies interleaving of the events from each of its arguments; in other words parallel merge of the executions of each of the arguments. The rules of par are similar to the rules of seq, but simpler since we do not have to preserve any order between the two operands. One of the operands is chosen arbitrarily:

$$\begin{array}{l}\Pi(L, \beta, d_1 \textsf{ par } d_2) \xrightarrow{e} \Pi(L, \beta, d_1' \textsf{ par } d_2) \\ \quad \textbf{if } \Pi(ll.d_1 \cap L, \beta, d_1) \xrightarrow{e} \Pi(ll.d_1 \cap L, \beta, d_1')\end{array}$$

$$\begin{array}{l}\Pi(L, \beta, d_1 \textsf{ par } d_2) \xrightarrow{e} \Pi(L, \beta, d_1 \textsf{ par } d_2') \\ \quad \textbf{if } \Pi(ll.d_2 \cap L, \beta, d_2) \xrightarrow{e} \Pi(ll.d_2 \cap L, \beta, d_2')\end{array}$$

The rules for alt choose which of the operands should be executed, but will only choose an operand if it has enabled events:

$$\begin{array}{l}\Pi(L, \beta, d_1 \textsf{ alt } d_2) \xrightarrow{e} \Pi(L, \beta, d_k') \\ \quad \textbf{if } \Pi(L, \beta, d_k) \xrightarrow{e} \Pi(L, \beta, d_k'),\ k \in \{1, 2\}\end{array}$$

The rules for refuse, assert and loop simply resolve the operators and produce silent events:

$$\Pi(L, \beta, \textsf{refuse } d) \xrightarrow{\tau_{refuse}} \Pi(L, \beta, d)$$

$$\Pi(L, \beta, \textsf{assert } d) \xrightarrow{\tau_{assert}} \Pi(L, \beta, d)$$

$$\Pi(L, \beta, \textsf{loop}\langle n \rangle\ d) \xrightarrow{\tau_{loop}} \Pi(L, \beta, d \textsf{ seq loop}\langle n-1 \rangle\ d)$$

3.4 Meta execution

The execution system defined above may be interpreted as a labeled transition system (LTS). This makes defining test derivation in the fashion of [19, 20] rather easy since this approach is based on test derivation from LTS's. What this approach lacks however is the distinction between invalid and inconclusive behavior and the distinction between potential and universal behavior we get from the sequence diagram operators neg and assert. From an operational point of view this holds true also for our execution system, in the sense that the properties of a trace being invalid or universal are properties at a higher level than the execution of the system (for a discussion on this, see [13]). In order to capture these properties we define a meta system (for meta execution):

$$\langle\!\langle \_, \_, \_, \_ \rangle\!\rangle \in \mathcal{H} \times \mathcal{EX} \times \mathbb{P}(\mathcal{L}) \times \mathcal{MO}$$

where $\mathcal{H}$ is the set of all traces, $\mathcal{EX} = \mathcal{B} \times \mathcal{D}$ the set of all (states of) execution systems, $\mathbb{P}(\mathcal{L})$ the powerset of all lifelines, and $\mathcal{MO} = \{\textsf{norm}, \textsf{ass}, \textsf{ref}, \textsf{assref}\}$ a set of modes.

For this system we define the following execution rules:

$$\begin{array}{l}\langle\!\langle t, [\beta, d], L, mo \rangle\!\rangle \longrightarrow \langle\!\langle t ^\frown \langle e \rangle, [\beta', d'], L, mo \rangle\!\rangle \\ \quad \textbf{if } [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge (l.e \in L \vee l^{-1}.e \in L)\end{array}$$

$$\begin{array}{l}\langle\!\langle t, [\beta, d], L, mo \rangle\!\rangle \longrightarrow \langle\!\langle t, [\beta', d'], L, mo \rangle\!\rangle \\ \quad \textbf{if } [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge l.e \notin L \wedge l^{-1}.e \notin L\end{array}$$

The idea with these rules is that we collect the trace from the execution, but only events with messages related with the lifelines (and gates) in L. By applying:

$$\langle\!\langle \langle\rangle, [initC.d, d], gates.d, \textsf{norm} \rangle\!\rangle$$

as the initial state of meta execution, we get a trace of which all events represent communication with the environment of the diagram.

The silent events are not visible in the trace collected, but may change the mode of the execution:

$$\langle t, [\beta, d], L, mo\rangle \longrightarrow \langle t, [\beta', d'], L, mo'\rangle$$
$$\textbf{if } [\beta, d] \xrightarrow{\tau} [\beta', d'] \wedge \tau \in \mathcal{T}$$

where

$$mo' = \begin{cases} \textsf{ass} & \textbf{if } \tau = \tau_{assert} \wedge mo = \textsf{norm} \\ \textsf{ref} & \textbf{if } \tau = \tau_{refuse} \wedge mo = \textsf{norm} \\ \textsf{assref} & \textbf{if } (\tau = \tau_{refuse} \wedge mo = \textsf{ass}) \vee \\ & \quad (\tau = \tau_{assert} \wedge mo = \textsf{ref}) \\ mo & \textbf{otherwise} \end{cases}$$

The intuition here is that norm represents execution of partial behavior, ass universal behavior, ref invalid behavior and assref behavior that is both universal and invalid. τ_{assert} and τ_{refuse} mean a region of the diagram specifying universal or invalid behavior has been entered, and hence change the mode of the execution.

We let $\longrightarrow^*$ denote the reflexive, transitive closure of $\longrightarrow$ above.

4. TESTS

A test is a sequence diagram consisting of only one lifeline. It operates by sending messages to the system under test and observing the messages received. A test is also able to observe the absence of messages from the system under test, intuitively timeouts. Based on the observations a test will terminate with one out of three possible verdicts: **pass**, **fail** or **inconclusive**.

The timeout/absence of reception is represented by a special event $\theta(l)$ where l is a lifeline (in practice the lifeline of the test). Let $\Theta = \{\theta(l) \,|\, l \in \mathcal{L}\}$ and $\Theta \cap (\mathcal{E} \cup \mathcal{T}) = \emptyset$. We let θ range over the elements of Θ and use θ as shorthand notation for $\theta(l)$ when l is implicitly given. We let $l.\_$ also range over Θ such that $l.\theta(l) = l$.

The verdicts are represented by three terminating diagrams (similar to skip), which extend the definition of sequence diagrams:

$$\textsf{pass}, \textsf{fail}, \textsf{inconc} \in \mathcal{D}$$

A test derived from the diagram in figure 1 would typically look like:

$$T = !but \ \textsf{seq}\ (?liq \ \textsf{seq pass alt}\ ?choc \ \textsf{seq pass alt}\ \theta \ \textsf{seq inconc})$$

A graphical representation of this test is shown in figure 2.

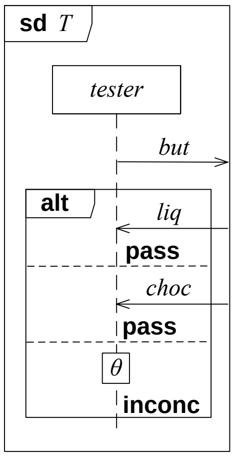

Figure 2: Example test

5. TEST DERIVATION

In this section we present the test generation algorithm. We do however need a number of auxiliary definitions to do this.

5.1 Definitions

First of all we need to distinguish between internal and external events. The external events of a diagram d are those events that send messages to gates or receive messages from gates. We define E_O and E_I to be the external transmit events and external receive events of d, respectively:

$$E_O \stackrel{\mathsf{def}}{=} \{e \in ev.d \,|\, k.e = ! \wedge l^{-1}.e \in \mathcal{G}\}$$
$$E_I \stackrel{\mathsf{def}}{=} \{e \in ev.d \,|\, k.e = ? \wedge l^{-1}.e \in \mathcal{G}\}$$

The events in $ev.d \setminus (E_O \cup E_I)$ are then the internal events. A function $enabled \in \mathcal{EX} \to \mathbb{P}(\mathcal{E})$ returns the set of enabled events of an execution state:

$$enabled([\beta, d]) \stackrel{\mathsf{def}}{=}$$
$$\{e \in \mathcal{E} \,|\, \exists t \in \mathcal{T}^*, \beta' \in \mathcal{B}, d' \in \mathcal{D} : [\beta, d] \xrightarrow{t \frown \langle e\rangle} [\beta', d']\}$$

An execution state is *quiescent* if no external transmit events or internal events are enabled. This is the same as saying that only external receive events are enabled. We define a quiescence function $\delta \in \mathcal{EX} \to \mathbb{B}$ that returns **true** iff an execution state is quiescent:

$$\delta([\beta, d]) \stackrel{\mathsf{def}}{=} enabled([\beta, d]) \subseteq E_I$$

Quiescence may be treated as an observable event. We let $\delta \notin (\mathcal{E} \cup \mathcal{T} \cup \Theta)$ denote the observation of quiescence. A function $out \in \mathcal{EX} \to \mathbb{P}(\mathcal{E} \cup \{\delta\})$ returns the set of observable transmit events of an execution state:

$$out([\beta, d]) \stackrel{\mathsf{def}}{=} \begin{cases} \{\delta\} & \textbf{if } \delta([\beta, d]) \\ E_O \cap enabled([\beta, d]) & \textbf{otherwise} \end{cases}$$

We need a way of keeping track of the modes in the test generation algorithm. We define a *state* as an execution state and a mode. Let $\mathcal{ST}$ denote the set of all states:

$$\mathcal{ST} \stackrel{\mathsf{def}}{=} \mathcal{EX} \times \mathcal{MO}$$

In the following we let S range over sets of states, i.e., $S \subseteq \mathcal{ST}$. We overload the function out to also range over sets of states, and define:

$$out(S) \stackrel{\mathsf{def}}{=} \bigcup_{([\beta,d],mo) \in S} out([\beta, d])$$

Finally we define an overloaded function $\_\textbf{after}\_ \in \mathcal{ST} \times \mathcal{H} \to \mathbb{P}(\mathcal{ST})$, $\_\textbf{after}\_ \in \mathbb{P}(\mathcal{ST}) \times \mathcal{H} \to \mathbb{P}(\mathcal{ST})$ that returns the new states after execution of a trace of external events:

$$([\beta, d], mo) \textbf{ after } t \stackrel{\mathsf{def}}{=} \{([\beta', d'], mo') \,|$$
$$\langle \langle\rangle, [\beta, d], gates.d, mo\rangle \longrightarrow^* \langle t, [\beta', d'], gates.d, mo'\rangle\}$$

$$S \textbf{ after } t \stackrel{\mathsf{def}}{=} \bigcup_{([\beta,d],mo) \in S} ([\beta, d], mo) \textbf{ after } t$$

5.2 Algorithm

Test generation is defined by a (non-deterministic) function $testGen \in \mathbb{P}(\mathcal{ST}) \rightarrow \mathcal{D}$. Let $d \in \mathcal{D}$ be the diagram from which we generate the test, $T \in \mathcal{D}$ be the generated test and $tester \in \mathcal{L}$ the name of the lifeline of the test.

We start generation by applying the function on the set of states reachable from the initial state of the diagram by execution of only silent and internal events:

$$T = testGen(([initC.d, d], \mathsf{norm})\ \textbf{after}\ \langle\rangle)$$

The algorithm is based on providing stimulus to and making observations of the diagram, and is defined by the three equations (or rules) shown below. Verdicts are determined by a combination of observations and modes.

Equation (1) observes the diagram. A branch is made for every possible observation. In case of observation of an event the diagram may provide, the test generation continues. In the other case, observation of an event the diagram cannot provide, a verdict is given and the test generation terminates:

$$\begin{array}{l} testGen(S) = \\ \quad \theta(tester)\ \mathsf{seq}\ \textbf{if}\ \delta \in out(S)\ \textbf{then} \\ \qquad\qquad testGen(S\ \textbf{after}\ \langle\rangle) \\ \qquad\quad \textbf{else}\ verdict\ \textbf{fi}\ \mathsf{alt} \\ \quad \mathsf{alt}_{e \in E_O}\ e^p\ \mathsf{seq}\ \textbf{if}\ e \in out(S)\ \textbf{then} \\ \qquad\qquad testGen(S\ \textbf{after}\ \langle e\rangle) \\ \qquad\quad \textbf{else}\ verdict\ \textbf{fi} \end{array} \tag{1}$$

where

$$verdict = \begin{cases} \mathsf{fail} & \textbf{if}\ \forall([\beta, d], mo) \in S : mo \in \{\mathsf{ass}, \mathsf{assref}\} \\ \mathsf{inconc} & \textbf{otherwise} \end{cases}$$

The intuition behind the verdicts is as follows: If all states have mode either ass or assref, we only have universal behavior and observing something not specified by the diagram should be bad. In all resulting cases we just observe something unspecified, and hence the behavior is inconclusive.

Equation (2) provides the diagram with a stimulus and continues the test generation with all states that are reachable after execution of this event:

$$\begin{array}{l} e \in \{e' \in E_I \mid S\ \textbf{after}\ \langle e'\rangle \neq \emptyset\} \Rightarrow \\ \quad testGen(S) = e^p\ \mathsf{seq}\ testGen(S\ \textbf{after}\ \langle e\rangle) \end{array} \tag{2}$$

Events e^p in the above equations are events prepared for being part of the test, and are defined as follows:

$$\begin{array}{rcl} (!, (s, l, g))^p & \overset{\mathsf{def}}{=} & (?, (s, l, tester)) \\ (?, (s, g, l))^p & \overset{\mathsf{def}}{=} & (!, (s, tester, l)) \end{array}$$

where we have that $l \notin \mathcal{G}$ and $g \in \mathcal{G}$.

Equation (3) is termination of the algorithm. If the diagram accepts no stimulus we provide a verdict. If all states reached have modes ref or assref this means we have execution of only invalid behavior and return the verdict fail. Else, the verdict is pass:

$$\begin{array}{l} \{e \in E_I \mid S\ \textbf{after}\ \langle e\rangle \neq \emptyset\} = \emptyset \Rightarrow \\ \quad testGen(S) = \begin{cases} \mathsf{pass} & \textbf{if}\ \exists([\beta, d], mo) \in S : \\ & \quad mo \neq \mathsf{ref} \wedge mo \neq \mathsf{assref} \\ \mathsf{fail} & \textbf{if}\ \forall([\beta, d], mo) \in S : \\ & \quad mo = \mathsf{ref} \vee mo = \mathsf{assref} \end{cases} \end{array} \tag{3}$$

The way the semantics of sequence diagrams is defined, all prefixes of specified behavior is inconclusive unless assert is used. For this reason we want maximal tests and enforce the following restrictions to the application of equations (1)-(3):

1. We always observe (equation (1)) after providing stimulus (equation (2)).

2. We always provide stimulus (equation (2)) after observing θ.

3. We only terminate (equation (3)) after observing θ.[1]

We may do this because we only have finite loop and hence are only able to specify finite behavior.

6. RELATED WORK

The UML Testing Profile [15] is an extension of UML which defines a method for specifying tests as sequence diagrams. The profile contains an informal notion of test execution as well as mappings from tests specified with this profile to tests in JUnit and TTCN-3. It does, however, not specify or define test generation, and in this respect our work may be seen as complementary to the profile.

Several approaches to generating tests from UML statecharts have been made, e.g., [6,11,14]. Because statecharts are complete specifications and have no neg construct, our approach is complementary to these and not a competing approach.

In [18] a scheme for generating tests from UML 2.0 sequence diagrams and statecharts is sketched, which allow sequence diagrams to contain neg. It is not clear, however, how formal the approach is and what is the role of the statecharts.

In [4] a method for executing sequence diagrams as tests is described, and [2] describes a method for extracting tests from sequence diagram specifications. Also in these approaches the level of formality is not clear. Further, they are based on UML 1.3 and hence do not deal with high level operators such as alt, assert and neg.

In [5] test generation from SDL [10] with MSCs as test purposes is described, and [1] defines test generation from MSCs. These approaches are more formal than the above mentioned UML sequence diagram approaches, but still they are less general than our approach because MSCs do not contain the assert and neg operators.

7. CONCLUSIONS AND FURTHER WORK

In this paper we have presented an algorithm for generating tests from sequence diagram specifications. The algorithm is based on a formal operational semantics for sequence diagrams and is an adaption of a well-known algorithm for generation of tests for conformance testing from LTS specifications [19,20]. Our algorithm takes as input diagrams that may contain the operators neg and assert and the tests generated are themselves represented as sequence diagrams. We are not aware of any other approach to test generation from UML that does this.

The operational semantics and the test generation algorithm are implemented in the term rewriting language Maude [3]. These implementations will become part of a tool for analysis of sequence diagrams currently under development.

[1]In the example shown in section 4 we have relaxed this requirement to get a test that is not unnecessary complicated

In the future we will look into the use of this test generation algorithm, together with a suitable means for executing the tests against specifications, for verification of refinement steps in system development along the lines of what we did with SDL specifications in [12]. We also wish to look into possible extension to the operational semantics and the test generation scheme, such as timed testing and probabilistic testing.

Other possibilities for future work would be to establish compliance with the UML Testing Profile and integration with testing of statecharts.

8. ACKNOWLEDGEMENTS

The work of this paper was conducted within and funded by the Basic ICT Research project SARDAS (15295/431) under the Research Council of Norway. It was conceived and carried out during a visit to the Formal Methods and Tools Group at the University of Twente. Thanks to Judith Rossebø for giving comments on the paper.

9. REFERENCES

[1] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell. Automatic generation of conformance tests from Message Sequence Charts. In *SDL and MSC workshop (SAM'02)*, number 2599 in LNCS, pages 170–198. Springer-Verlag, 2003.

[2] L. Briand and Y. Labiche. A UML-based approach to system testing. *Softw. Syst. Model.*, 1(1):10–42, 2002.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1.1)*. SRI International, Menlo Park, Apr. 2005.

[4] F. Fraikin and T. Leonhardt. SeDiTeC - Testing based on sequence diagrams. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 261–266. IEEE Computer Society, 2002.

[5] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC based test generation for distributed test architectures. In *9th International SDL Forum, The Next Millennium (SDL'99)*, pages 389–404. Elsevier, 1999.

[6] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *2000 International Symposium on Software Testing and Analysis (ISSTA'00), Proceedings of the ACM SIGSOFT*, volume 25 of *Software Engineering Notes*, pages 60–70, Sept. 2000.

[7] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Softw. Syst. Model.*, 4(4):355–367, 2005.

[8] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, transformations and tools. International Workshop, Dagstuhl Castle, Germany, September 2003. Revised selected papers*, number 3466 in LNCS, pages 1–25. Springer-Verlag, 2005.

[9] ITU-T. *Message Sequence Chart (MSC), ITU-T Recommendation Z.120 (11/1999)*, 1999.

[10] ITU-T. *Specification and description language (SDL), ITU-T Recommendation Z.100 (11/1999)*, 2000.

[11] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proc.-Softw.*, 146(4):187–192, Aug. 1999.

[12] M. S. Lund. Testing decomposition of component specifications based on a rule for formal verification. In *Third International Conference on Quality Software (QSIC 2003)*, pages 154–160. IEEE Computer Society, 2003.

[13] M. S. Lund and K. Stølen. *A fully general operational semantics for UML sequence diagrams with potential and mandatory choice*. Research report 330, Department of Informatics, University of Oslo, 2006.

[14] J. Offnutt and A. Abdurazik. Generating tests from UML specifications. In *UML'99 – The Unified Modeling Language Beyond the Standard, Second International Conference*, number 1723 in LNCS, pages 416–429. Springer-Verlag, 1999.

[15] OMG. *UML Testing Profile, version 1.0*. Object Management Group, 2005. OMG Document: formal/2005-07-07.

[16] OMG. *Unified Modeling Language: Superstructure, version 2.0*. Object Management Group, 2005. OMG Document: formal/2005-07-04.

[17] R. K. Runde, Ø. Haugen, and K. Stølen. How to transform UML neg into a useful construct. In *Norsk Informatikkonferanse (NIK'05)*, pages 55–66. Tapir, 2005.

[18] D. Sokenou. Ein UML-basierter Testansatz zum Klassen- und Integrationstest objektorientierter Systeme. In *Software Engineering 2005. GI-Edition: Lecture Notes in Informatics (LNI) P-64*, pages 91–102. Bonner Köllen Verlag, 2005.

[19] J. Tretmans. Testing concurrent systems: A formal approach. In *10th International Conference on Concurrency Theory (CONCUR'99)*, number 1664 in LNCS, pages 46–65. Springer-Verlag, 1999.

[20] J. Tretmans. Testing techniques. Reader, Univeriteit Twente, 2002.