Nordic Journal of Computing 12(2005), 157–188.

REFINING UML INTERACTIONS WITH UNDERSPECIFICATION AND NONDETERMINISM

RAGNHILD KOBRO RUNDE[1] ØYSTEIN HAUGEN[1]

KETIL STØLEN[1,2]

[1]*Department of Informatics, University of Oslo*
PO Box 1080, Blindern, NO-0316 Oslo, Norway
{ragnhilk|oysteinh}@ifi.uio.no

[2]*SINTEF ICT, NO-0373 Oslo, Norway*
ketil.stolen@sintef.no

Abstract. STAIRS is an approach to the compositional development of UML interactions, such as sequence diagrams and interaction overview diagrams. An important aspect of STAIRS is the ability to distinguish between underspecification and inherent nondeterminism through the use of potential and mandatory alternatives. This paper investigates this distinction in more detail. Refinement notions explain when (and how) both kinds of nondeterminism may be reduced during the development process. In particular, in this paper we extend STAIRS with guards, which may be used to specify the choice between alternatives. Finally, we introduce the notion of an implementation and define what it means for an implementation to be correct with respect to a specification.

ACM CCS Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications; D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Key words: UML interactions, formal semantics, nondeterminism, underspecification, refinement

1. Introduction

STAIRS [9, 8] is an approach to the compositional development of UML interactions, such as sequence diagrams and interaction overview diagrams. Interactions in UML 2.0 [13] are behavioural definitions that describe some, but not necessarily all, of the behaviour that a given system performs. Most often the interactions will describe positive behaviours, i.e. behaviours that the system is allowed to perform. There may also be behaviours that the interactions define as negative, meaning that they are unacceptable, and there may even be behaviours of the system that are not at all covered by any of the interactions defined.

This partiality of interactions is motivated by several factors. First of all, the description of a real system requires far too many interaction diagrams to define all the behaviours. To manage such a volume of diagrams would be impractical. Also, the goal of interactions is to visualize important interaction patterns.

Received February 2005; accepted September 2005.

Thus the emphasis is on importance, rather than completeness. This is in contrast to most other kinds of behavioural specifications, including UML state machines. The definition of a state machine is complete in the sense that it may be seen to define all the possible behaviours of that entity.

A methodology may initially use interactions to capture user requirements, and use these as stepping stones for the next development stages where emphasis is placed more on completeness and realizability. STAIRS supports this through the notion of refinement. In particular, the refinement definitions take into account that initial specifications in the form of interactions typically describe only a few example scenarios. A scenario not described by an initial specification is not necessarily unwanted, but it has not been thought of yet. Thus, a refinement step may be to include new scenarios in the specification, as well as to reduce the amount of underspecification and nondeterminism in the specification. Refinement may also be to describe some of the aspects of a scenario in more detail.

In this paper we focus on defining and refining specifications with nondeterminism. In the introductory chapter of the UNITY-book [3] Chandy and Misra observe:

> Nondeterminism is useful in two ways. First, it is employed to derive simple programs, where simplicity is achieved by avoiding unnecessary determinism; such programs can be optimized by limiting the nondeterminism, i.e., by disallowing executions unsuitable for a given architecture. Second, some systems (e.g., operating systems and delay-insensitive circuits) are inherently nondeterministic; programs that represent such systems have to employ some nondeterministic constructs.

STAIRS is based on this overall observation. However, contrary to Chandy and Misra we take the position that the two useful ways of using nondeterminism should be described differently.

Avoiding unnecessary determinism may for instance be achieved through underspecification. By underspecification we mean that the specification gives several alternative behaviours that are equivalent in the sense that they all serve the same purpose. For an implementation to be correct, it is sufficient to fulfil only one of the alternative behaviours. Underspecification may also be used as an abstraction mechanism, for instance by giving several alternative behaviours but not stating how to select between them. This will typically later be refined into an if-then-else construct in the implementation.

On the other hand, inherent nondeterminism is used to capture alternative behaviours that must all be possible for the implementation. A typical example is the tossing of a coin, where both heads and tails should be possible outcomes, and no legal refinement should remove one of these two alternatives. A system may also need to exhibit nondeterministic behaviour due to differences in its environment.

Inherent nondeterminism is very different from underspecification, and should be described differently. One important reason for this is that unless we do not distinguish these two, there will be no way to ensure that inherently nondeterministic behaviour is implemented as such. This may not seem like a major problem at first.

If the development team knows that a given specification should be implemented as delay-insensitive circuits you will probably get the inherently nondeterministic implementation that you expect. However, in the domain of information security, the inherently nondeterministic behaviour is fundamental for the validity of the specification. As pointed out in e.g. [10] and [12], security properties are in general not preserved by standard refinement. If nondeterminism is used as a means to hide the internal workings of a system, it is essential that it is not treated as underspecification, which allows elimination of all uncertainty (nondeterminism) in a refinement.

In [14] Roscoe points out that using inherent nondeterminism ensures security as it prevents the making of any inference about the possible outcomes, while for nondeterminism based on underspecification there are three possible conclusions about the security of a system: secure, insecure, or don't know. Hence, it makes things a lot easier if the specification language provides a way to distinguish between these two ways of using nondeterminism.

In the setting of UML interactions, the operator alt is used to specify alternative behaviours. As the UML standard [13] is rather vague on whether these alternatives represent underspecification or inherent nondeterminism, people interpret the same interaction differently, leading to confusion. This could be avoided by having two different operators for specifying alternative behaviours, as we have in STAIRS. This is particularly important as the partiality of interactions makes it important to know which of the described scenarios represent significantly different behaviours and which scenarios only serve as examples of how to achieve the same purpose.

The remainder of this paper is structured into six sections. Section 2 introduces the basic STAIRS formalism, while Section 3 uses this in an example specification illustrating nondeterminism. In Section 4 we extend the formalism with guards, and in Section 5 we discuss refinement in STAIRS with emphasis on nondeterminism. Section 6 defines what it means for a system to be a correct implementation of a STAIRS specification. Section 7 provides a brief summary and relates STAIRS to approaches known from the literature.

2. Background: UML interactions with denotational trace semantics

In this section, we present the basic STAIRS formalism. In Section 2.1 we present our textual syntax for interactions. Section 2.2 gives the fundamental trace mechanisms, while Section 2.3 formally defines denotational trace semantics for UML interactions.

2.1 *Syntax of interactions*

The set of syntactically correct interactions, denoted by $\mathcal{D}$, is defined by the BNF-grammar in Fig. 1. `Signal` represents the actual content of a message, `Lifeline` is the name of a lifeline (representing a component) in the diagram and `Set` should be an expression that evaluates to a subset of $\mathbb{N}_0$ (the natural numbers including 0).

As can be seen from the definition, a message is a triple (s, tr, re) of a signal s, a transmitter tr, and a receiver re. As a shorthand, we will often use the name of the

⟨Interaction⟩	→	⟨Empty⟩ | ⟨Event⟩ |
		⟨Weak sequencing⟩ | ⟨Refuse⟩ |
		⟨Assert⟩ | ⟨Potential alternatives⟩ |
		⟨Mandatory alternatives⟩ | ⟨Loop⟩
⟨Empty⟩	→	skip
⟨Event⟩	→	⟨Kind⟩ ⟨Message⟩
⟨Kind⟩	→	⟨Transmission⟩ | ⟨Reception⟩
⟨Transmission⟩	→	!
⟨Reception⟩	→	?
⟨Message⟩	→	(`Signal` , ⟨Transmitter⟩ , ⟨Receiver⟩)
⟨Transmitter⟩	→	`Lifeline`
⟨Receiver⟩	→	`Lifeline`
⟨Refuse⟩	→	refuse [⟨Interaction⟩]
⟨Assert⟩	→	assert [⟨Interaction⟩]
⟨Potential alternatives⟩	→	alt [⟨Interaction list⟩]
⟨Mandatory alternatives⟩	→	xalt [⟨Interaction list⟩]
⟨Loop⟩	→	loop `Set` [⟨Interaction⟩]
⟨Weak sequencing⟩	→	seq [⟨Interaction list⟩]
⟨Interaction list⟩	→	⟨Interaction⟩ |
		⟨Interaction list⟩ , ⟨Interaction⟩

Fig. 1: Syntax of interactions.

signal to stand for the whole message in cases where the transmitter and receiver are clear from the context. We let $\mathcal{L}$ denote the set of all lifelines, and $\mathcal{M}$ denote the set of all messages. We distinguish between two kinds of events; a transmission event tagged by an exclamation mark "!" represents the transmission of a message, while a reception event tagged by a question mark "?" represents the reception of a message. $\mathcal{E}$ denotes the set of all events, while $\mathcal{K}$ denotes $\{!, ?\}$.

We define the functions

$$k.\_ \in \mathcal{E} \to \mathcal{K}, \quad m.\_ \in \mathcal{E} \to \mathcal{M}, \quad tr.\_, re.\_ \in \mathcal{E} \to \mathcal{L}$$

to yield the kind, message, transmitter and receiver of an event, respectively. We also overload *tr* and *re* to yield the transmitter and receiver of a message.

We also define the functions

$$ll.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{L}), \quad ev.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{E}), \quad msg.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{M})$$

to yield the set of lifelines, events and messages of an interaction, respectively.

Interactions are built from events through the application of various operators as defined by the grammar in Fig. 1. We do not cover the complete set of operators in UML 2.0 [13], but rather focus on a few essential operators. These fundamental operators may be used to define other useful, high-level operators as demonstrated in Section 5.2. See [6] for STAIRS definitions of additional operators like parallel execution and gates.

The operators assert, alt, seq and loop are UML 2.0 operators. The operator xalt is new, proposed in [9] to model mandatory alternatives, i.e. alternatives that must all be present in the final implementation. For negation, UML 2.0 uses the operator neg. However, this operator is used in several contexts, with slightly different meanings as we explain in [16]. Therefore, we have in this paper chosen to introduce a new operator refuse that covers one of these traditional uses of neg.

We only consider interactions that are well-formed in the sense that if both the transmitter and the receiver lifelines of a message are present in the diagram, then both the transmission and the reception event of that message must be present as well. Formally:

$$\forall m \in msg.d : (tr.m \in ll.d \wedge re.m \in ll.d) \Rightarrow ((!, m) \in ev.d \wedge (?, m) \in ev.d)$$

Also, in this paper we assume that for all operators except from seq, the operand(s) consist only of complete messages, i.e. messages with both the transmission and the reception event within the operand.

2.2 *Representing executions by traces*

In STAIRS, we define the semantics of interactions by using sequences of events. By A^{ω} we denote the set of all finite and infinite sequences over the set A. We use $\langle\rangle$ to the denote the empty sequence. Moreover, by $\langle e_1, e_2, \ldots, e_m\rangle$ we denote the sequence of m elements, whose first element is e_1, whose second element is e_2, and so on. We define the functions

$$\#\_ \in A^{\omega} \to \mathbb{N}_0 \cup \{\infty\}, \quad \_[\_] \in A^{\omega} \times \mathbb{N} \to A$$

to yield the length and the nth element of a sequence. Hence, $\#a$ yields the number of elements in a and $a[n]$ yields a's nth element if $n \leq \#a$.

We also need functions for concatenation, truncation and filtering:

$$\_ \frown \_ \in A^{\omega} \times A^{\omega} \to A^{\omega}, \quad \_|\_ \in A^{\omega} \times \mathbb{N}_0 \to A^{\omega}, \quad \_ \circledS \_ \in \mathbb{P}(A) \times A^{\omega} \to A^{\omega}$$

Concatenating two sequences implies gluing them together. Hence, $a_1 \frown a_2$ denotes a sequence of length $\#a_1 + \#a_2$ that equals a_1 if a_1 is infinite, and is prefixed by a_1 and suffixed by a_2, otherwise. For any $0 \leq i \leq \#a$, we define $a|_i$ to denote the prefix of a of length i.

The filtering function $\circledS$ is used to filter away elements. By $B \circledS a$ we denote the sequence obtained from the sequence a by removing all elements in a that are not in the set of elements B. For example, we have that

$$\{1, 3\} \circledS \langle 1, 1, 2, 1, 3, 2\rangle = \langle 1, 1, 1, 3\rangle$$

A trace h is a sequence of events, used to represent a system run. For any single message, transmission must happen before reception if both events are present.

Thus we get the following well-formedness requirement on traces, stating that if at any point in the trace we have a transmission event, up to that point we must have had at least as many transmissions as receptions of that particular message:

$$\forall i \in [1, \#h] : k.h[i] = ! \Rightarrow$$
$$\#((\{\,!\} \times \{m.h[i]\}) \circledS h|_i) > \#((\{?\} \times \{m.h[i]\}) \circledS h|_i)$$

$\mathcal{H}$ denotes the set of all well-formed traces.

2.3 Semantics of interactions

The semantics of interactions is defined by a function $[\![\ ]\!]$ that for any interaction d yields a set $[\![\ d\ ]\!]$ of interaction obligations. The term obligation is used to explicitly convey that any implementation of a specification is obliged to fulfil each specified alternative. (What it formally means to fulfil an obligation is discussed in Section 6.) An interaction obligation is a pair (p, n) of sets of traces. The first set p represents positive traces that may be the result of running the final system, while the second set n represents negative traces that must not appear in the implementation of the obligation. Traces not defined as positive or negative are called *inconclusive*. As will be formally defined in Section 5, a refinement may later redefine (some of) these inconclusive traces as positive or negative. An obligation pair (p, n) is contradictory if $p \cap n \neq \emptyset$.

The empty diagram, denoted by skip, is a specification without any events that corresponds to a program doing nothing. The empty diagram defines the empty trace as positive:

$$[\![\ \textsf{skip}\ ]\!] \overset{\text{def}}{=} \{(\{\langle\rangle\}, \emptyset)\} \tag{1}$$

For an interaction consisting of a single event e, its semantics is given by:

$$[\![\ e\ ]\!] \overset{\text{def}}{=} \{(\{\langle e\rangle\}, \emptyset)\} \tag{2}$$

The actual content of the messages is not significant for the purpose of this paper. Hence, we do not give any semantic interpretation of messages as such.

The rest of this section will define the semantics of the different composition operators described briefly in Section 2.1. Table I lists the notational conventions that will be used in the following definitions.

TABLE I: Notational conventions.

Symbol	**Stands for**
d	interaction
D	list of interactions, separated by comma
h	trace
s, p, n	trace set
o	interaction obligation
O	set of interaction obligations

2.3.1 Weak sequencing

Weak sequencing is the implicit composition mechanism combining constructs of an interaction. The operator seq is defined by the following invariants:

- The ordering of events within each of the operands is maintained in the result.
- Events on different lifelines from different operands may come in any order.
- Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand, and so on.

First, we define weak sequencing of trace sets:

$$\begin{aligned} s_1 \succsim s_2 \overset{\text{def}}{=} \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in \mathcal{L} : \\ e.l \circledS h = e.l \circledS h_1 \frown e.l \circledS h_2\} \end{aligned} \tag{3}$$

where $e.l$ denotes the set of events that may take place on the lifeline l. Formally:

$$e.l \overset{\text{def}}{=} \{e \in \mathcal{E} \mid (k.e = ! \wedge tr.e = l) \vee (k.e = ? \wedge re.e = l\} \tag{4}$$

Weak sequencing of interaction obligations is defined as:

$$(p_1, n_1) \succsim (p_2, n_2) \overset{\text{def}}{=} (p_1 \succsim p_2, (n_1 \succsim p_2) \cup (n_1 \succsim n_2) \cup (p_1 \succsim n_2)) \tag{5}$$

Notice that all traces obtained by combining a negative and a positive trace-set, will also be negative. Weak sequencing of sets of interaction obligations is defined as:

$$O_1 \succsim O_2 \overset{\text{def}}{=} \{o_1 \succsim o_2 \mid o_1 \in O_1 \wedge o_2 \in O_2\} \tag{6}$$

Finally, the seq construct is defined by:

$$\begin{aligned} [\![\, \mathsf{seq}\,[d] \,]\!] &\overset{\text{def}}{=} [\![\, d \,]\!] \\ [\![\, \mathsf{seq}\,[D, d] \,]\!] &\overset{\text{def}}{=} [\![\, \mathsf{seq}\,[D] \,]\!] \succsim [\![\, d \,]\!] \end{aligned} \tag{7}$$

As an example, the interaction in Fig. 2 shows two messages both originating from L1 and targeting L2. Its semantics is calculated as:

$$\begin{aligned} & [\![\, W \,]\!] = [\![\, \mathsf{seq}\,[!x, ?x, !y, ?y] \,]\!] \\ & = (([\![\, !x \,]\!] \succsim [\![\, ?x \,]\!]) \succsim [\![\, !y \,]\!]) \succsim [\![\, ?y \,]\!] && \text{(Def. (7))} \\ & = ((\{(\{\langle !x \rangle\}, \emptyset)\} \succsim \{(\{\langle ?x \rangle\}, \emptyset)\}) \succsim \{(\{\langle !y \rangle\}, \emptyset)\}) \\ & \quad \succsim \{(\{\langle ?y \rangle\}, \emptyset)\} && \text{(Def. (2))} \\ & = (\{(\{\langle !x, ?x \rangle\}, \emptyset)\} \succsim \{(\{\langle !y \rangle\}, \emptyset)\}) \succsim \{(\{\langle ?y \rangle\}, \emptyset)\} && \text{(Defs. (3) – (6))} \\ & = \{(\{\langle !x, ?x, !y \rangle, \langle !x, !y, ?x \rangle\}, \emptyset)\} \succsim \{(\{\langle ?y \rangle\}, \emptyset)\} && \text{(Defs. (3) – (6))} \\ & = \{(\{\langle !x, ?x, !y, ?y \rangle, \langle !x, !y, ?x, ?y \rangle\}, \emptyset)\} && \text{(Defs. (3) – (6))} \end{aligned}$$

Hence, this interaction specifies one interaction obligation with two positive traces and no negative ones. The positive traces state that the transmission of x must be the first event to happen, but after that either y may be transmitted (by L1) or x may be received (by L2).

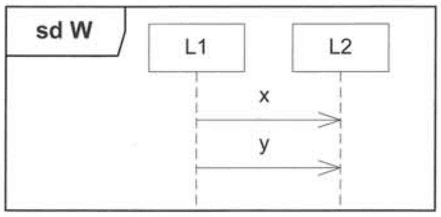


Fig. 2: Weak sequencing.

2.3.2 *Negative behaviour*

The refuse construct defines negative traces:

$$[\![\text{refuse}\,[d]]\!] \overset{\text{def}}{=} \{(\emptyset, p \cup n) \mid (p,n) \in [\![d]\!]\} \tag{8}$$

Notice that a negative trace cannot be made positive by reapplying refuse. Negative traces remain negative, since negation should be seen as an operation that characterizes traces absolutely and not relatively.

2.3.3 *Assertion*

The assert construct makes all inconclusive traces negative. Except for that the sets of positive and negative traces are left unchanged:

$$[\![\text{assert}\,[d]]\!] \overset{\text{def}}{=} \{(p, n \cup (\mathcal{H} \setminus p)) \mid (p,n) \in [\![d]\!]\} \tag{9}$$

Notice that contradictory obligation pairs remain contradictory.

2.3.4 *Potential alternatives*

The alt construct is used when specifying underspecification, i.e. to define potential traces that are equivalent in the sense that it is sufficient for an implementation to include only one of them. The semantics of alt is the inner union of each point-wise selection of interaction obligations from its operands:

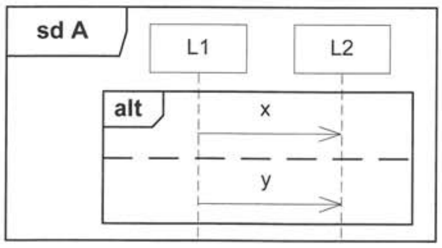
$$[\![\text{alt}\,[d_1, \ldots, d_m]]\!] \overset{\text{def}}{=} \{ \biguplus \{o_1, \ldots, o_m\} \mid \forall i \in [1,m] : o_i \in [\![d_i]\!] \} \tag{10}$$

The inner union of interaction obligations is defined as:

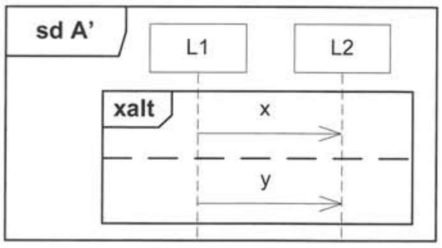
$$\biguplus_{i \in [1,m]} (p_i, n_i) \overset{\text{def}}{=} (\bigcup_{i \in [1,m]} p_i, \bigcup_{i \in [1,m]} n_i) \tag{11}$$

Fig. 3(a) gives a simple example using the alt construct. The dashed horizontal line separates the operands. We get:

$$\begin{aligned}
[\![A]\!] &= [\![\text{alt}\,[\text{seq}\,[!x, ?x], \text{seq}\,[!y, ?y]]]\!] \\
&= \{ \uplus \{ (\{\langle !x, ?x\rangle\}, \emptyset), (\{\langle !y, ?y\rangle\}, \emptyset) \} \} && \text{(Defs. (3) – (7), (10))} \\
&= \{ (\{\langle !x, ?x\rangle, \langle !y, ?y\rangle\}, \emptyset) \} && \text{(Def. (11))}
\end{aligned}$$



(a) Potential alternatives



(b) Mandatory alternatives

Fig. 3: Specifying alternatives.

2.3.5 Mandatory alternatives

The xalt construct is used to specify inherent nondeterminism, i.e. mandatory alternatives that must all be present in an implementation:

$$[\![\text{xalt } [d_1, \dots d_m]]\!] \overset{\text{def}}{=} \bigcup_{i \in [1,m]} [\![d_i]\!] \tag{12}$$

Fig. 3(b) has the same messages as Fig. 3(a), but separated by xalt instead of alt. In this case, we get two interaction obligations:

$$\begin{aligned} [\![A']\!] &= [\![\text{xalt } [\text{seq } [!x, ?x], \text{seq } [!y, ?y]]]\!] \\ &= \bigcup \{ [\![\text{seq } [!x, ?x]]\!], [\![\text{seq } [!y, ?y]]\!] \} && \text{(Def. (12))} \\ &= \bigcup \{ \{(\{\langle !x, ?x\rangle\}, \emptyset)\}, \{(\{\langle !y, ?y\rangle\}, \emptyset)\} \} && \text{(Defs. (3) – (7))} \\ &= \{ (\{\langle !x, ?x\rangle\}, \emptyset)\ ,\ (\{\langle !y, ?y\rangle\}, \emptyset) \} && \text{(Def. of } \textstyle\bigcup) \end{aligned}$$

2.3.6 Loop

For a set of interaction obligations we define a finite loop construct μ_n, where $n \in \mathbb{N}_0$ denotes the number of times the body of the loop is iterated. $\mu_n\ O$ is defined inductively as follows:

$$\mu_n\ O \overset{\text{def}}{=} \begin{cases} \{(\{\langle\rangle\}, \emptyset)\} & \text{if n} = 0 \\ O & \text{if n} = 1 \\ \mu_{n-1}\ O \succsim O & \text{otherwise} \end{cases} \tag{13}$$

For a definition of infinite loop, see [6].

In the UML 2.0 standard [13], loop is used together with limits stating the minimum and maximum number of times the content of the loop should be executed. In our definition, the set I is a generalization of this, such that the numbers in I specify the possible alternatives for how many times the loop content should be executed. Not all of these need to be actual alternatives in an implementation, meaning that the definition of loop uses the point-wise inner union between these alternatives, similar to the definition of alt:

$$[\![\ \text{loop}\ I\ [d]\ ]\!] \overset{\text{def}}{=} \{ \biguplus_{i\in I} o_i \mid \forall i \in I : o_i \in \mu_i[\![\ d\ ]\!]\ \} \tag{14}$$

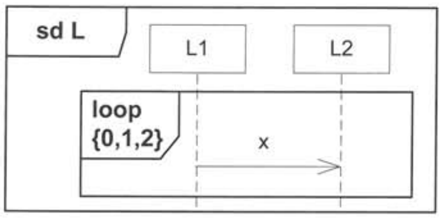


Fig. 4: Looping.

As an example, the interaction in Fig. 4 has the following semantics:

$$\begin{aligned}
&[\![\ L\ ]\!] = [\![\ \text{loop}\ \{0,1,2\}\ [\text{seq}\ [!x, ?x]]\ ]\!] \\
&= \{\ \uplus_{i\in\{0,1,2\}}\, o_i \mid \forall i \in \{0,1,2\} : \\
&\qquad\qquad o_i \in \mu_i\ [\![\ \text{seq}\ [!x, ?x]\ ]\!]\ \} && \text{(Def. (14))} \\
&= \{\ \uplus_{i\in\{0,1,2\}}\, o_i \mid \forall i \in \{0,1,2\} : \\
&\qquad\qquad o_i \in \mu_i\ \{(\{\langle !x, ?x\rangle\}, \emptyset)\}\ \} && \text{(Defs. (3) – (7))} \\
&= \{\ \uplus_{i\in\{0,1,2\}}\, o_i \mid\ o_0 \in \mu_0\ \{(\{\langle !x, ?x\rangle\}, \emptyset)\} \land \\
&\qquad\qquad o_1 \in \mu_1\ \{(\{\langle !x, ?x\rangle\}, \emptyset)\} \land \\
&\qquad\qquad o_2 \in \mu_2\ \{(\{\langle !x, ?x\rangle\}, \emptyset)\}\ \} \\
&= \{\ \uplus_{i\in\{0,1,2\}}\, o_i \mid\ o_0 \in \{(\{\langle\rangle\}, \emptyset)\} \land \\
&\qquad\qquad o_1 \in \{(\{\langle !x, ?x\rangle\}, \emptyset)\} \land \\
&\qquad\qquad o_2 \in \{(\{\langle !x, ?x\rangle\}, \emptyset)\} \\
&\qquad\qquad\qquad \gtrsim \{(\{\langle !x, ?x\rangle\}, \emptyset)\}\ \} && \text{(Def. (13))} \\
&= \{\ \uplus_{i\in\{0,1,2\}}\, o_i \mid\ o_0 \in \{(\{\langle\rangle\}, \emptyset)\} \land \\
&\qquad\qquad o_1 \in \{(\{\langle !x, ?x\rangle\}, \emptyset)\} \land \\
&\qquad\qquad o_2 \in \{(\{\langle !x, ?x, !x, ?x\rangle, \\
&\qquad\qquad\qquad \langle !x, !x, ?x, ?x\rangle\}, \emptyset)\}\ \} && \text{(Defs. (3) – (6))} \\
&= \{\ \uplus\ \{\ (\{\langle\rangle\}, \emptyset), \\
&\qquad (\{\langle !x, ?x\rangle\}, \emptyset), \\
&\qquad (\{\langle !x, ?x, !x, ?x\rangle, \langle !x, !x, ?x, ?x\rangle\}, \emptyset)\ \}\ \} \\
&= \{\ (\{\langle\rangle, \langle !x, ?x\rangle, \langle !x, ?x, !x, ?x\rangle, \langle !x, !x, ?x, ?x\rangle\}, \emptyset)\ \} && \text{(Def. (11))}
\end{aligned}$$

3. STAIRS and nondeterminism

As seen in the previous section, weak sequencing may result in several different traces with the same events in a somewhat different order. These traces are alternative means to achieve the same goal, and they are therefore grouped into the same interaction obligation as it is sufficient to keep only one of them in an implementation.

In UML 2.0, the other means to specify alternative behaviours is by using the operator alt. This is used both for specifying potential alternatives where keeping only one is sufficient, and for mandatory alternatives that must all be present in a correct implementation. In STAIRS, we have distinguished these two uses by separating between our two operators alt and xalt. Each use of UML 2.0 alt corresponds in STAIRS to either alt or xalt. In this section we present an example illustrating the use of these two operators.

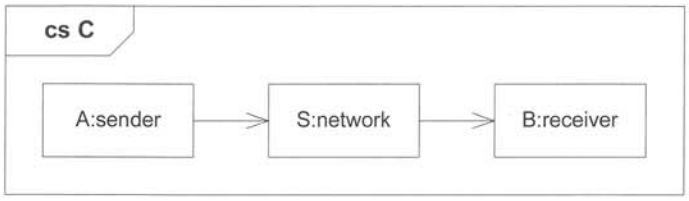


Fig. 5: Composite structure of context C.

Consider a situation where a sender communicates with a receiver through a network of type S as shown in the UML composite structure diagram in Fig. 5 (notice that this is not an interaction). A very simple communication is shown by the interaction in Fig. 6, its semantics being:

$$\{ (\{\langle !(m, A, S), ?(m, A, S), !(m, S, B), ?(m, S, B)\rangle\}, \emptyset) \}$$

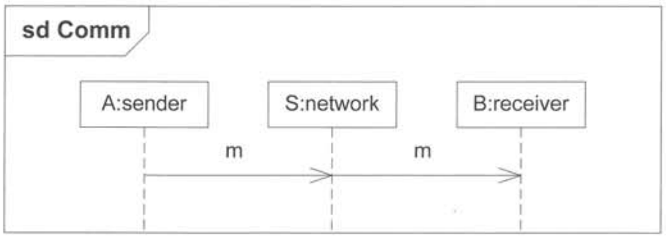


Fig. 6: Very simple communication.

Next, we would like to specify that there is a need for redundant communication through the network S. That is, the network S needs to support more than one way of bringing the message *m* from one end of the network to the other. There may be several reasons for requiring this redundancy:

- Several paths through the network will make it easier to exploit the full capacity of the network.
- Multiple paths will ensure increased internal robustness of the network and as such improve the availability of the full communication.
- Multiple paths will make it more difficult to attack the network to jeopardize the communication, and as such the communication security is improved.

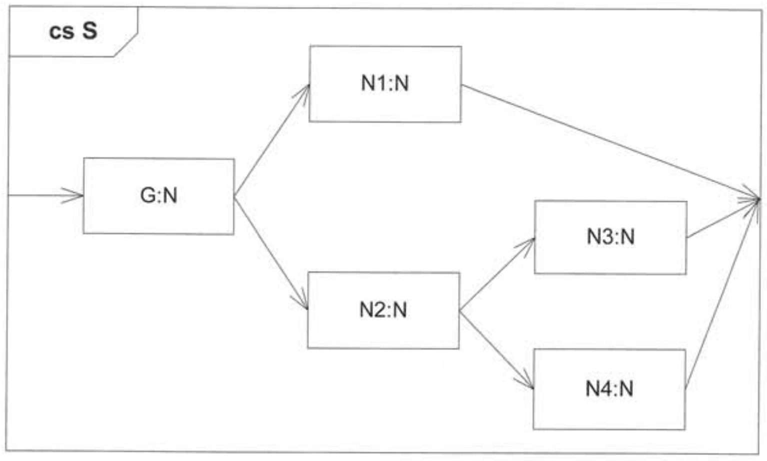


Fig. 7: Internal structure of the network S showing three communication paths.

We indicate in Fig. 7 a simple network architecture for S where there are alternative branches. A real communication network may of course have far more paths, but giving a few are sufficient for the purpose of this paper. We want to make an interaction where we require two (different) communication possibilities, and we may do this by introducing an xalt construct as shown in Fig. 8, where S is expanded according to the structure in Fig. 7.

We have used xalt here in order to express that the network must support at least two communication paths. Of course, for each concrete communication only one

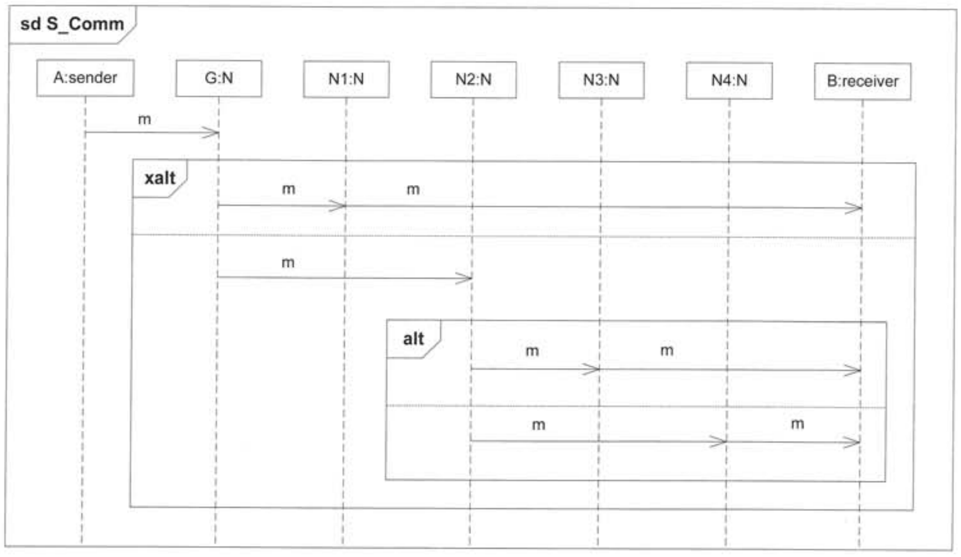


Fig. 8: Communication behaviour requiring two communication patterns.

of them will be applied. After node N2, the network S has yet another branch giving two alternative paths. For the sake of the discussion we assume that it is not important to have both of these available, and so we specify the alternatives using alt and not xalt in Fig. 8.

The semantics of S_Comm is:

$$\begin{aligned}\{\ &(\{\langle !(m,A,G), ?(m,A,G), !(m,G,N1), ?(m,G,N1),\\ &\quad !(m,N1,B), ?(m,N1,B)\rangle\}, \emptyset),\\ &(\{\langle !(m,A,G), ?(m,A,G), !(m,G,N2), ?(m,G,N2),\\ &\quad !(m,N2,N3), ?(m,N2,N3), !(m,N3,B), ?(m,N3,B)\rangle,\\ &\quad \langle !(m,A,G), ?(m,A,G), !(m,G,N2), ?(m,G,N2),\\ &\quad !(m,N2,N4), ?(m,N2,N4), !(m,N4,B), ?(m,N4,B)\rangle\}, \emptyset)\ \}\end{aligned}$$

Fig. 9 illustrates this semantics using a specialized Venn-diagram with one ellipse for each interaction obligation. Traces not shown as positive or negative in an obligation are inconclusive for this obligation.

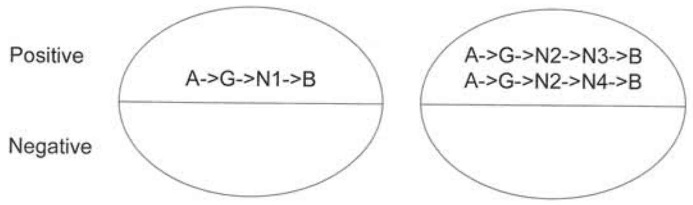


Fig. 9: Venn-diagram of the specification in Fig. 8.

Formally, S_Comm is a refinement of Comm. Refinement will be formally defined in Section 5. In Section 5 we will also develop this example further, by giving some possible refinements to illustrate the similarities and differences between the two operators alt and xalt. But first, in the next section, we formally extend STAIRS with guards, which may be used to specify the choice between alternatives.

4. Extending STAIRS with data and guards

Although the focus of interactions is on the messages, the diagrams may also be decorated with data. The most common use of data in interactions is in guards, which is a mechanism for choosing between alternatives. Data is also used in assignments and general constraints. In this section we extend our basic formalism with definitions of these concepts. The extension ensures that (sub-)interactions not including data have the same semantics as before.

4.1 Data

Since interactions mainly specify events and not data, the exact data values will most of the time be underspecified (or unspecified). Changes in the data may in general happen at any time, also when there is nothing in the diagram indicating such a change. As a consequence, in the semantic model we do not include data as such. Instead, data is represented indirectly through events representing its use in assignments, constraints, and guards.

Formally, we extend the syntax of interactions as defined by the BNF-grammar in Fig. 10. Nonterminals that are unchanged from the original syntax in Fig. 1 are not repeated. `Variable` should be either a global variable or a variable local to the lifeline on which the assignment is placed (not shown in our textual syntax), while `Expression` is a mathematical expression and `Constraint` an expression that evaluates to *true* or *false*. If an operand of guarded alt or guarded xalt does not contain an explicit guard, we interpret this as being the guard *true*.

```
⟨Interaction⟩           → ⟨Empty⟩ | ⟨Event⟩ |
                          ⟨Weak sequencing⟩ | ⟨Refuse⟩ |
                          ⟨Assert⟩ | ⟨Guarded alt⟩ |
                          ⟨Guarded xalt⟩ | ⟨Loop⟩ |
                          ⟨Assignment⟩ | ⟨Constraint⟩
⟨Assignment⟩            → assign ( Variable , Expression )
⟨Constraint⟩            → constr ( Constraint )
⟨Guarded alt⟩           → alt [ ⟨Guarded list⟩ ]
⟨Guarded xalt⟩          → xalt [ ⟨Guarded list⟩ ]
⟨Guarded list⟩          → ⟨Guarded interaction⟩ |
                          ⟨Guarded list⟩ , ⟨Guarded interaction⟩
⟨Guarded interaction⟩ → ⟨Guard⟩ → ⟨Interaction⟩
⟨Guard⟩                 → Constraint
```

Fig. 10: Syntax of interactions with data.

In the semantics, we extend the set of trace events with the two special events *write* (for assignments) and *check* (for constraints). We also need the notion of a state. Let Var be the set of all variables and Val be the set of all variable values. A state σ is then a total function assigning a value to each variable. Formally:

$$\sigma \in Var \rightarrow Val$$

For any expression *expr*, we use $expr(\sigma)$ to denote its value in σ.

4.2 Assignment

Explicit specification of variable values may be done by using assignments. In UML 2.0, assignments are written inside a rounded box on the appropriate lifeline, as illustrated in Fig. 11.

Semantically, we represent an assignment $var = expr$ by the special event $write(\sigma, \sigma')$ where σ is the state immediately before the assignment and σ' the state immediately after:

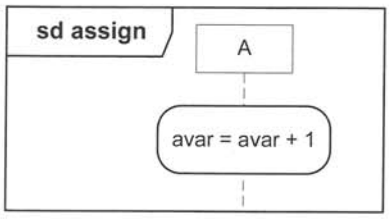


Fig. 11: Assignment.

$$\llbracket \textsf{assign}(var, expr) \rrbracket \overset{\text{def}}{=} \tag{15}$$
$$\{ (\{\langle write(\sigma,\sigma')\rangle \mid \sigma'(var) = expr(\sigma) \wedge \forall v \in Var : (v = var \vee \sigma'(v) = \sigma(v))\}, \emptyset) \}$$

4.3 Constraints (state invariants)

In UML 2.0, constraints are written within curly brackets, as illustrated in Fig. 12. A constraint is a restriction that must be fulfilled by the system, meaning that we have a negative trace if the constraint is broken.

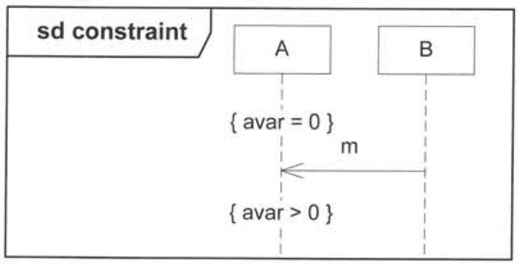


Fig. 12: Constraint.

Semantically, a constraint is represented by the special event $check(\sigma)$, where σ is the state in which the constraint is evaluated:

$$\llbracket \textsf{constr}(c) \rrbracket \overset{\text{def}}{=} \tag{16}$$
$$\{ (\{\langle check(\sigma)\rangle \mid c(\sigma)\} , \{\langle check(\sigma)\rangle \mid \neg c(\sigma)\}) \}$$

This definition ensures that if the constraint is a tautology, then the semantics of $\textsf{constr}(c)$ has no negative traces, and that a contradiction gives no positive traces:

$$\begin{aligned} &\llbracket \textsf{constr}(true) \rrbracket \\ &= \{(\{\langle check(\sigma)\rangle \mid true(\sigma)\}, \{\langle check(\sigma)\rangle \mid false(\sigma)\})\} \\ &= \{(\{\langle check(\sigma)\rangle \mid \sigma \in Var \to Val\}, \emptyset)\} \end{aligned}$$

$$\begin{aligned} &\llbracket \textsf{constr}(false) \rrbracket \\ &= \{(\{\langle check(\sigma)\rangle \mid false(\sigma)\}, \{\langle check(\sigma)\rangle \mid true(\sigma)\})\} \\ &= \{(\emptyset, \{\langle check(\sigma)\rangle \mid \sigma \in Var \to Val\})\} \end{aligned}$$

Notice that one constraint in itself gives potentially an infinite number of system traces, varying with respect to the state component only.

As an example of the use of constraints in an interaction, the complete semantics of the interaction in Fig. 12 may be calculated as:

$$\begin{aligned}
&[\![\ \textbf{constraint}\ ]\!] = \\
&[\![\ \mathsf{seq}\ [\mathsf{constr}(avar = 0), !m, ?m, \mathsf{constr}(avar > 0)]\ ]\!] \\
&= (\ (\ [\![\ \mathsf{constr}(avar = 0)\ ]\!] \succsim [\![\ !m\ ]\!]\ ) \succsim [\![\ ?m\ ]\!]\ ) \succsim [\![\ \mathsf{constr}(avar > 0)\ ]\!] \\
&\quad \text{(Def. (7))} \\
&= (\ (\ [\![\ \mathsf{constr}(avar = 0)\ ]\!] \succsim \{(\{\langle !m\rangle\}, \emptyset)\}\ ) \succsim \{(\{\langle ?m\rangle\}, \emptyset)\}\ ) \\
&\quad \succsim [\![\ \mathsf{constr}(avar > 0)\ ]\!] \\
&\quad \text{(Def. (2))} \\
&= (\ (\ \{(\{\langle check(\sigma)\rangle \mid \sigma(avar) = 0\}, \{\langle check(\sigma)\rangle \mid \sigma(avar) \neq 0\})\} \\
&\qquad \succsim \{(\{\langle !m\rangle\}, \emptyset)\}\ ) \succsim \{(\{\langle ?m\rangle\}, \emptyset)\}\ ) \\
&\quad \succsim \{(\{\langle check(\sigma')\rangle \mid \sigma'(avar) > 0\}, \{\langle check(\sigma')\rangle \mid \sigma'(avar) \leq 0\})\} \\
&\quad \text{(Def. (16))} \\
&= (\ \{\ (\{\langle check(\sigma)\rangle \mid \sigma(avar) = 0\} \succsim \{\langle !m\rangle\}, \\
&\qquad \{\langle check(\sigma)\rangle \mid \sigma(avar) \neq 0\} \succsim \{\langle !m\rangle\} \\
&\qquad \cup \{\langle check(\sigma)\rangle \mid \sigma(avar) \neq 0\} \succsim \emptyset \\
&\qquad \cup \{\langle check(\sigma)\rangle \mid \sigma(avar) = 0\} \succsim \emptyset)\ \} \\
&\qquad \succsim \{(\{\langle ?m\rangle\}, \emptyset)\}\ ) \\
&\quad \succsim \{(\{\langle check(\sigma')\rangle \mid \sigma'(avar) > 0\}, \{\langle check(\sigma')\rangle \mid \sigma'(avar) \leq 0\})\} \\
&\quad \text{(Defs. (5) – (6))} \\
&= (\ \{\ (\{\langle check(\sigma), !m\rangle \mid \sigma(avar) = 0\} \\
&\qquad \cup \{\langle !m, check(\sigma)\rangle \mid \sigma(avar) = 0\}, \\
&\qquad \{\langle check(\sigma), !m\rangle \mid \sigma(avar) \neq 0\} \\
&\qquad \cup \{\langle !m, check(\sigma)\rangle \mid \sigma(avar) \neq 0\})\ \} \\
&\qquad \succsim \{(\{\langle ?m\rangle\}, \emptyset)\}\ ) \\
&\quad \succsim \{(\{\langle check(\sigma')\rangle \mid \sigma'(avar) > 0\}, \{\langle check(\sigma')\rangle \mid \sigma'(avar) \leq 0\})\} \\
&\quad \text{(Def. (3))} \\
&= \{\ (\{\langle check(\sigma), !m, ?m\rangle \mid \sigma(avar) = 0\} \\
&\qquad \cup \{\langle !m, check(\sigma), ?m\rangle \mid \sigma(avar) = 0\}, \\
&\qquad \{\langle check(\sigma), !m, ?m\rangle \mid \sigma(avar) \neq 0\} \\
&\qquad \cup \{\langle !m, check(\sigma), ?m\rangle \mid \sigma(avar) \neq 0\})\ \} \\
&\quad \succsim \{(\{\langle check(\sigma')\rangle \mid \sigma'(avar) > 0\}, \{\langle check(\sigma')\rangle \mid \sigma'(avar) \leq 0\})\} \\
&\quad \text{(Defs. (3) – (6))} \\
&= \{\ (\{\langle check(\sigma), !m, ?m, check(\sigma')\rangle \mid \sigma(avar) = 0 \wedge \sigma'(avar) > 0)\} \\
&\qquad \cup \{\langle !m, check(\sigma), ?m, check(\sigma')\rangle \mid \sigma(avar) = 0 \wedge \sigma'(avar) > 0\}, \\
&\qquad \{\langle check(\sigma), !m, ?m, check(\sigma')\rangle \mid \sigma(avar) \neq 0 \vee \sigma'(avar) \leq 0\} \\
&\qquad \cup \{\langle !m, check(\sigma), ?m, check(\sigma')\rangle \mid \sigma(avar) \neq 0 \vee \sigma'(avar) \leq 0\})\ \} \\
&\quad \text{(Defs. (3) – (6), and formula manipulation)}
\end{aligned}$$

4.4 Guards (interaction constraints)

According to UML 2.0, alternatives (and other combined fragments) in an interaction may be guarded by an interaction constraint (also called a guard). A guard is

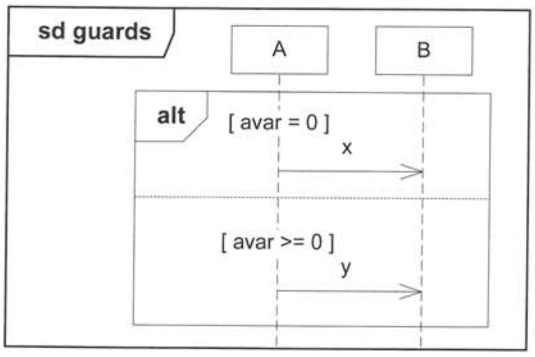


Fig. 13: Guards.

a special kind of constraint that may only occur at the beginning of the interaction operand in question. As opposed to general constraints, guards are written inside square brackets, as illustrated in Fig. 13. As the example illustrates, the guards used in an alt (or xalt) may be overlapping and need not be exhaustive.

If the guard is true, the interaction operand describes positive traces of the system. The semantics in the case of a false guard is not stated explicitly in the UML 2.0 standard [13]. However, with guards being a specialization of general constraints, it is natural to interpret traces with a false guard as negative. As will be demonstrated in Section 5, this is advantageous as it means that adding guards to an alt/xalt-construct constitutes a valid refinement step. A side effect of this is that we will be able to model guards by using the more general notion of constraints as defined in the previous section.

4.4.1 Guarded alt

UML 2.0 [13] states that if none of the operands of an alt construct has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing interaction is executed. This gives the following semantics for guarded alt:

$$\begin{aligned} &[\![\text{ alt } [c_1 \to d_1, \ldots, c_m \to d_m]]\!] \overset{\text{def}}{=} \\ &\{ \uplus\{o_1, \ldots, o_m, (\{\langle check(\sigma)\rangle \mid (\textstyle\bigwedge_{j\in[1,m]} \neg c_j)(\sigma)\}, \emptyset)\} \mid \\ &\qquad \forall i \in [1,m] : o_i \in [\![\text{ seq } [\text{constr}(c_i), d_i]]\!] \} \end{aligned} \tag{17}$$

The semantics of Fig. 13 is informally illustrated in Fig. 14. Formally, its complete semantics may be calculated as:

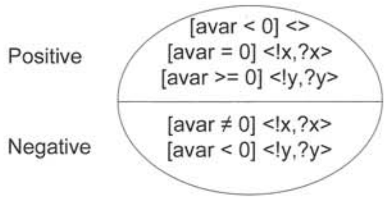


Fig. 14: Semantics of guarded alt in Fig. 13.

$$
\begin{aligned}
&[\![\ \textbf{guards}\ ]\!]\\
&= [\![\ \mathsf{alt}\ [\mathit{avar} = 0 \rightarrow \mathsf{seq}\ [!x, ?x]\ ,\\
&\qquad \mathit{avar} \geq 0 \rightarrow \mathsf{seq}\ [!y, ?y]\ ]\ ]\!]\\
&= \{\ \biguplus\ \{\ (\{\langle \mathit{check}(\sigma), !x, ?x\rangle \mid \sigma(\mathit{avar}) = 0\}\ ,\\
&\qquad \{\langle \mathit{check}(\sigma), !x, ?x\rangle \mid \sigma(\mathit{avar}) \neq 0\}),\\
&\qquad (\{\langle \mathit{check}(\sigma), !y, ?y\rangle \mid \sigma(\mathit{avar}) \geq 0\}\ ,\\
&\qquad \{\langle \mathit{check}(\sigma), !y, ?y\rangle \mid \sigma(\mathit{avar}) < 0\}), \quad \text{(Defs. (3) – (7),}\\
&\qquad (\{\langle \mathit{check}(\sigma)\rangle \mid \sigma(\mathit{avar}) < 0\}\ , \emptyset)\}\ \} \quad \text{(16), (17))}\\
&= \{\ (\{\langle \mathit{check}(\sigma), !x, ?x\rangle \mid \sigma(\mathit{avar}) = 0\}\\
&\quad \cup\{\langle \mathit{check}(\sigma), !y, ?y\rangle \mid \sigma(\mathit{avar}) \geq 0\}\\
&\quad \cup\{\langle \mathit{check}(\sigma)\rangle \mid \sigma(\mathit{avar}) < 0\}\ ,\\
&\quad \{\langle \mathit{check}(\sigma), !x, ?x\rangle \mid \sigma(\mathit{avar}) \neq 0\}\\
&\quad \cup\{\langle \mathit{check}(\sigma), !y, ?y\rangle \mid \sigma(\mathit{avar}) < 0\})\ \} \quad \text{(Def. (11))}
\end{aligned}
$$

Definition (17) giving the semantics of guarded alt is consistent with definition (10) of unguarded alt in Section 2.3. In our new setting, a specification alt $[D]$ without guards is interpreted as the specification alt $[D']$ where D' is the same list of interactions as D, each one guarded by *true*. Calculating this semantics using definition (17), gives us the same semantics as definition (10) when abstracting away all *check*-events. This is proved in [15].

4.4.2 *Guarded* xalt

We define the semantics of guarded xalt as:

$$
[\![\ \mathsf{xalt}\ [c_1 \rightarrow d_1, \ldots, c_m \rightarrow d_m]\ ]\!] \overset{\mathsf{def}}{=} \bigcup_{i\in[1,m]} [\![\ \mathsf{seq}\ [\mathsf{constr}(c_i), d_i]\ ]\!] \qquad (18)
$$

Unlike guarded alt, the semantics of guarded xalt does not implicitly include the case where all guards are false, since xalt is used to specify explicit choices that must be present in the implementation.

As an example, the semantics of Fig. 13 with alt replaced by xalt, is informally illustrated in Fig. 15. Formally, its complete semantics may be calculated as:

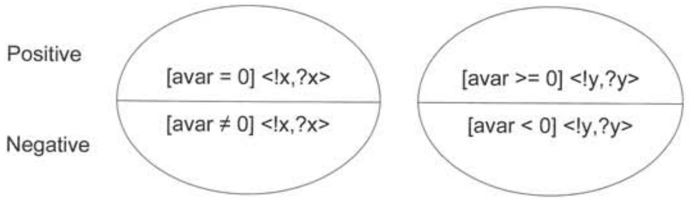


Fig. 15: Semantics of guarded xalt in Fig. 13.

$$
\begin{aligned}
&[\![\textbf{ guards }]\!] \\
&= [\![\textsf{ xalt } [avar = 0 \rightarrow \textsf{seq}\ [!x, ?x]\ , \\
&\qquad\quad avar \geq 0 \rightarrow \textsf{seq}\ [!y, ?y]\ ]\ ]\!] \\
&= \bigcup \{\ \{(\{\langle check(\sigma), !x, ?x\rangle \mid \sigma(avar) = 0\}\ , \\
&\qquad \{\langle check(\sigma), !x, ?x\rangle \mid \sigma(avar) \neq 0\})\}, \\
&\quad \{(\{\langle check(\sigma), !y, ?y\rangle \mid \sigma(avar) \geq 0\}\ , && \text{(Defs. (3) – (7),} \\
&\qquad \{\langle check(\sigma), !y, ?y\rangle \mid \sigma(avar) < 0\})\}\ \} && (16), (18)) \\
&= \{\ (\{\langle check(\sigma), !x, ?x\rangle \mid \sigma(avar) = 0\}\ , \\
&\quad \{\langle check(\sigma), !x, ?x\rangle \mid \sigma(avar) \neq 0\})\ , \\
&\quad (\{\langle check(\sigma), !y, ?y\rangle \mid \sigma(avar) \geq 0\}\ , \\
&\quad \{\langle check(\sigma), !y, ?y\rangle \mid \sigma(avar) < 0\})\ \} && \text{(Def. } \textstyle\bigcup)
\end{aligned}
$$

As for alt, removing the guards in definition (18) gives the original xalt-semantics in definition (12).

5. Refinement

In this section we discuss some important aspects of refinement in the setting of STAIRS. Section 5.1 gives the necessary background for this discussion, presenting the main refinement definitions. In the rest of Section 5 we focus on underspecification and inherent nondeterminism, and how guards may be introduced as a refinement step in both cases.

5.1 Background: Formal definitions

Refinement means to add information to a specification such that the specification becomes more complete. This may be achieved by categorizing inconclusive traces as either positive or negative, or by reducing the set of positive traces. Negative traces always remain negative. A specification may also become more complete by introducing more details.

5.1.1 Glass-box refinement

Formally, an interaction obligation (p', n') is a refinement of an interaction obligation (p, n), written $(p, n) \rightsquigarrow_r (p', n')$, iff

$$n \subseteq n' \quad \wedge \quad p \subseteq p' \cup n' \tag{19}$$

An interaction d' is a glass-box refinement of an interaction d, written $d \rightsquigarrow_g d'$, iff

$$\forall o \in [\![d]\!] : \exists o' \in [\![d']\!] : o \rightsquigarrow_r o' \tag{20}$$

THEOREM 1. *The refinement operator* $\rightsquigarrow_g$ *is*

- *reflexive:* $d \rightsquigarrow_g d$
- *transitive:* $d \rightsquigarrow_g d' \wedge d' \rightsquigarrow_g d'' \Rightarrow d \rightsquigarrow_g d''$
- *monotonic with respect to* refuse, loop, seq, *(guarded)* alt *and (guarded)* xalt*:*

$$\begin{array}{l}
d \rightsquigarrow_g d' \Rightarrow \mathsf{refuse}\ [d] \rightsquigarrow_g \mathsf{refuse}\ [d'] \\
d \rightsquigarrow_g d' \Rightarrow \mathsf{loop}\ I\ [d] \rightsquigarrow_g \mathsf{loop}\ I\ [d'] \\
d_1 \rightsquigarrow_g d'_1, \ldots, d_m \rightsquigarrow_g d'_m \Rightarrow \mathsf{seq}\ [d_1, \ldots, d_m] \rightsquigarrow_g \mathsf{seq}\ [d'_1, \ldots, d'_m] \\
d_1 \rightsquigarrow_g d'_1, \ldots, d_m \rightsquigarrow_g d'_m \Rightarrow \mathsf{alt}\ [d_1, \ldots, d_m] \rightsquigarrow_g \mathsf{alt}\ [d'_1, \ldots, d'_m] \\
d_1 \rightsquigarrow_g d'_1, \ldots, d_m \rightsquigarrow_g d'_m \Rightarrow \mathsf{xalt}\ [d_1, \ldots, d_m] \rightsquigarrow_g \mathsf{xalt}\ [d'_1, \ldots, d'_m]
\end{array}$$

PROOF. Reflexivity, transitivity and monotonicity with respect to seq, loop and unguarded alt and xalt is proved in [6]. Monotonicity with respect to refuse and guarded alt and xalt is proved in [15]. □

By definition (20), new interaction obligations may be freely added to the specification, thus increasing the mandatory nondeterminism required of an implementation. Adding new obligations is an important aspect of the STAIRS methodology. Sometimes, however, it is desirable to restrict this possibility.

A more restrictive notion of refinement is *limited glass-box refinement*, where each obligation in the new refined interaction must correspond to an obligation in the original interaction.

Formally, an interaction d' is a limited glass-box refinement of an interaction d, written $d \rightsquigarrow_l d'$, iff

$$d \rightsquigarrow_g d' \wedge \forall o' \in [\![d']\!] : \exists o \in [\![d]\!] : o \rightsquigarrow_r o' \tag{21}$$

Notice that a step of refinement may still increase the total number of obligations, but only if two different obligations in $[\![d']\!]$ refine the same obligation in $[\![d]\!]$.

Methodologically, a STAIRS specification would typically be developed by using $\rightsquigarrow_g$ initially and switching to the more restrictive $\rightsquigarrow_l$ after the desired level of nondeterminism in the specification has been reached.

5.1.2 Black-box refinement

Black-box refinement may be understood as refinement restricted to the externally visible behaviour. We define the function

$$ext \in \mathcal{H} \times \mathbb{P}(\mathcal{L}) \rightarrow \mathcal{H}$$

to yield the trace obtained from the trace given as first argument by filtering away those events that are internal with respect to the set of lifelines given as second argument:

$$ext(h, l) \overset{\mathsf{def}}{=} \{e \in \mathcal{E} \mid tr.e \notin l \vee re.e \notin l\} \circledS h \tag{22}$$

The *ext* operator is overloaded to sets of traces, to pairs of sets of traces, and sets of pairs of sets of traces in the standard pointwise manner:

$$ext(s,l) \overset{\text{def}}{=} \{ext(h,l) \mid h \in s\} \tag{23}$$

$$ext((p,n),l) \overset{\text{def}}{=} (ext(p,l), ext(n,l)) \tag{24}$$

$$ext(O,l) \overset{\text{def}}{=} \{ext((p,n),l) \mid (p,n) \in O\} \tag{25}$$

An interaction d' is a black-box refinement of an interaction d, written $d \rightsquigarrow_b d'$, iff

$$\forall o \in ext(\llbracket\, d \,\rrbracket, ll.d) : \exists o' \in ext(\llbracket\, d' \,\rrbracket, ll.d') : o \rightsquigarrow_r o' \tag{26}$$

Theorem 1 is valid also when replacing $\rightsquigarrow_g$ with $\rightsquigarrow_b$, as the properties are independent of the content of the actual traces.

Black-box refinements will often include lifeline decompositions that are not externally visible. Some lifeline decompositions may also be externally visible due to a change in the sender or receiver of a message. We have already used this in Fig. 8, where the network S was decomposed into several nodes. Formally, an interaction d' is a lifeline decomposition of an interaction d with respect to a lifeline substitution ls, written $d \rightsquigarrow^{ls} d'$, iff

$$d \rightsquigarrow_b subst(d', ls) \tag{27}$$

where $ls \in L \rightarrow L$ is a function defining the lifeline substitution and the function $subst(d, ls)$ yields the interaction d with every lifeline l in d substituted with the lifeline $ls(l)$.

5.2 *Adding positive behaviour*

We now return to our running example from Section 3. Even with two different communication paths, we have no guarantee that any of them will be available at a certain time. This is made explicit in Fig. 16, where the empty diagram (i.e. skip) is added as a third operand to the xalt-construct. When this operand is selected, we get a positive trace consisting of only two events, the transmission of m from A to G, and the reception at G. No further communication will take place, and B will never receive the message.

The semantics of N_Comm is illustrated in Fig. 17. Comparing this with Fig. 9, which illustrates the semantics of S_Comm (Fig. 8), we see that every interaction obligation given by S_Comm is also an interaction obligation by N_Comm. By definitions (19)–(20), this means that the modified specification is a valid refinement of the original one, S_Comm $\rightsquigarrow_g$ N_Comm. The last obligation in Fig. 17 illustrates that new obligations may be added freely when using standard glass-box refinement, $\rightsquigarrow_g$.

Assume now that our communication network describes the emergency network used by the police, that a police officer needs to communicate, but that the communication for some reason fails. In practice, a police officer may grab his personal mobile phone and call his colleague. This is not a mandatory choice (the police are not set up with personal mobile phones), but may be used as an alternative.

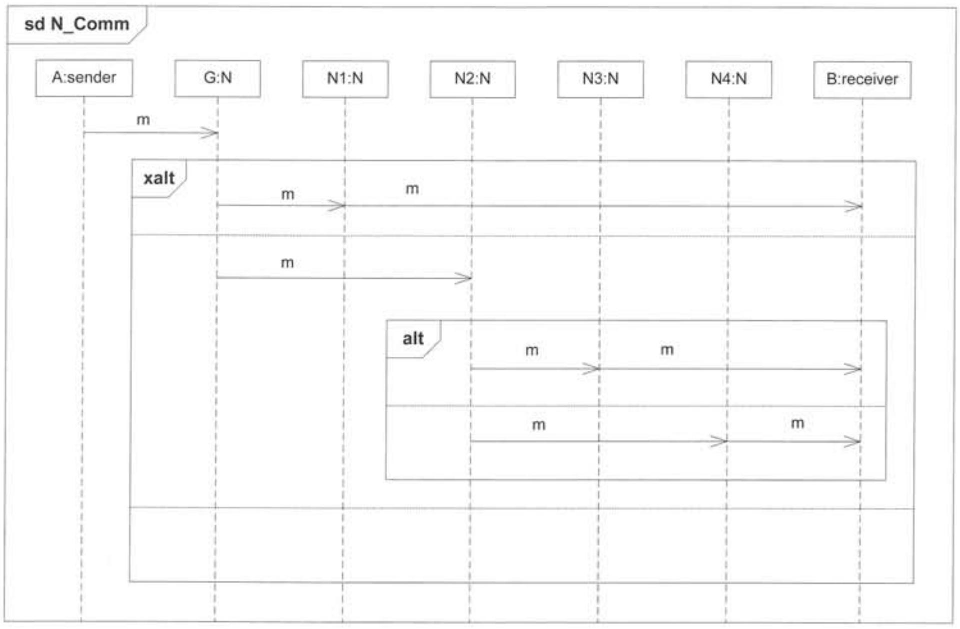


Fig. 16: Refinement by adding behaviour.

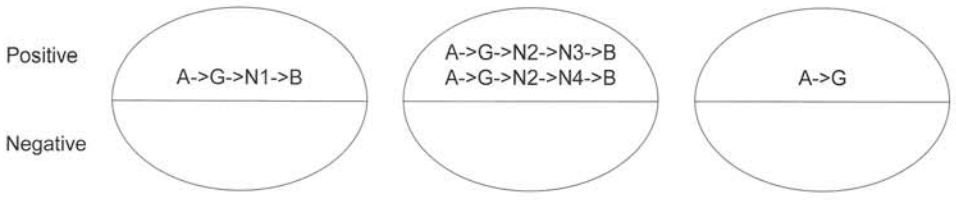


Fig. 17: Semantics of N_Comm (Fig. 16).

The resulting specification is shown in Fig. 18. The opt-construct is a high-level operator, which may be defined as

$$\mathsf{opt}\, d \overset{\mathsf{def}}{=} d\, \mathsf{alt}\, \mathsf{skip} \tag{28}$$

The modified specification affects only the last of the interaction obligations in Fig. 17, where a positive behaviour is added as illustrated in Fig. 19. By definition (19), this is a valid refinement as the negative trace-sets in both interaction obligations are empty and the positive trace-set in the N_Comm one is a subset of the new positive trace set in the M_Comm one:

$$\begin{aligned}
&\{\, \langle !(m, A, G), ?(m, A, G)\rangle \,\} \subseteq \\
&\{\, \langle !(m, A, G), ?(m, A, G)\rangle, \\
&\quad \langle !(m, A, G), ?(m, A, G), !(m, G, \mathit{Mobile}), ?(m, G, \mathit{Mobile}), \\
&\qquad !(m, \mathit{Mobile}, B), ?(m, \mathit{Mobile}, B)\rangle \,\}
\end{aligned}$$

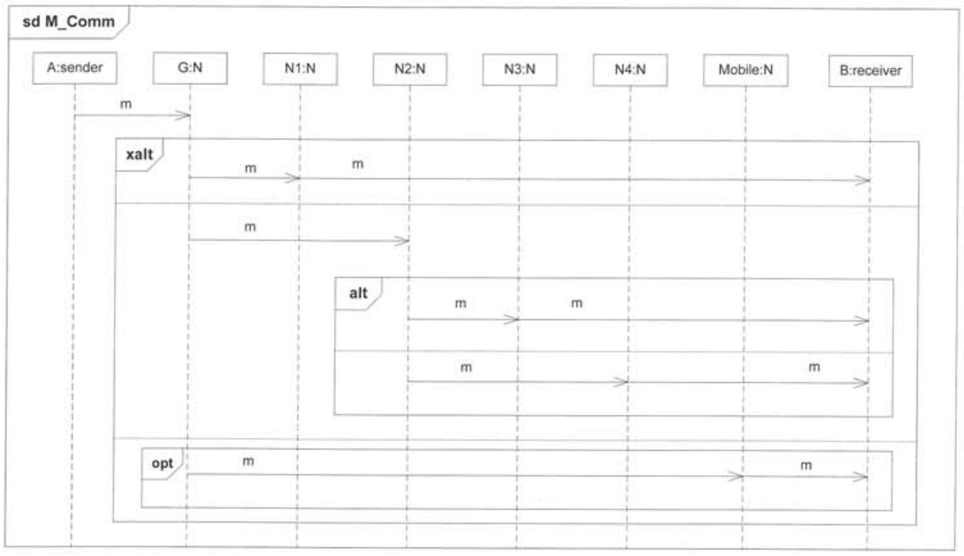


Fig. 18: Refinement by adding behaviour.

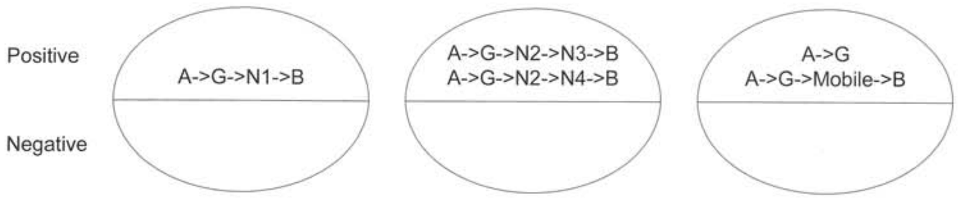


Fig. 19: Semantics of M_Comm (Fig. 18).

Notice that adding an extra lifeline (the mobile phone) to the interaction is unproblematic, as all traces involving this new lifeline were considered inconclusive in the original interaction.

5.3 Adding negative behaviour

The refinement examples in the previous section categorized earlier inconclusive traces as positive. Similarly, earlier inconclusive traces may be categorized as negative, either by specifying the negative traces explicitly through the use of refuse, or by using assert. In our network example, we decide that M_Comm is a complete description of the possible behaviours, and that everything not in the interaction should be considered negative. This gives us the interaction in Fig. 20.

The semantics of A_Comm is illustrated in Fig. 21. Comparing this with Fig. 19, which illustrates the semantics of M_Comm, we see that A_Comm is obviously a refinement of M_Comm, as we have the same positive trace-sets for both specifications and the original empty negative trace-sets are subsets of any set.

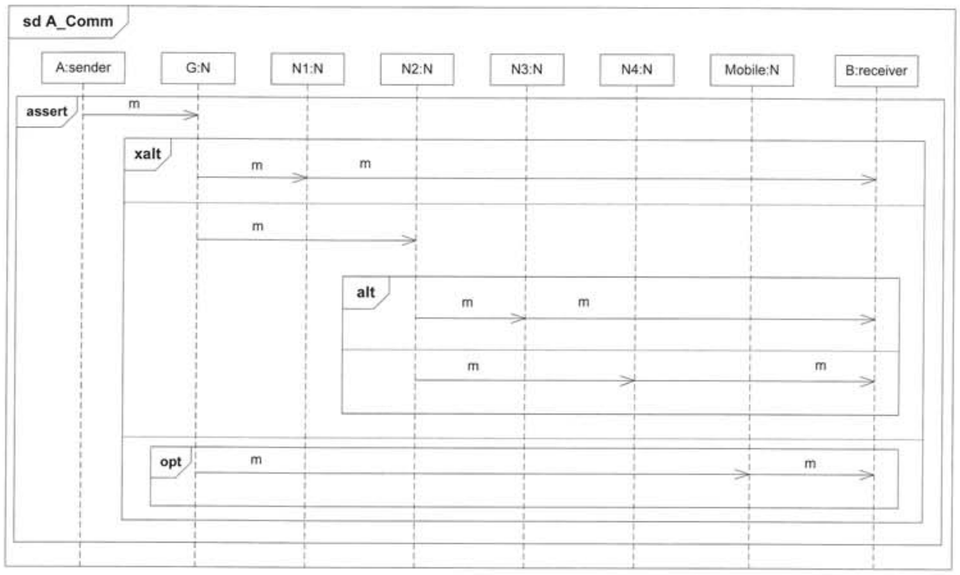


Fig. 20: Adding negative behaviour.

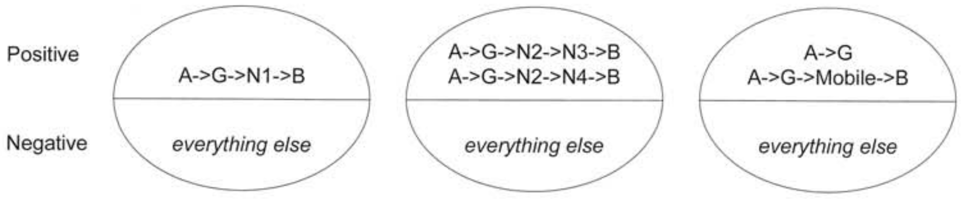


Fig. 21: Semantics of A_Comm (Fig. 20).

5.4 Redefining positive behaviour as negative

Refinement may also be used to reduce the set of positive traces by redefining them as negative. Looking at the specification in Fig. 20, we may decide that there really is no need to have both communication choices specified by the alt-construct. A refinement of this sub-specification could then be as given by Fig. 22. The complete semantics for this refinement is illustrated in Fig. 23. We see that the refined specification only affects the obligation in the middle. By definition (19), this is a valid refinement step as the negative trace-set is extended and the traces that were previously positive are now either positive or negative.

Another possible refinement of A_Comm could be to specify how the choice between the different communication paths should be made. In the case of our emergency network, using a mobile phone should only be an option if the main network fails. In the interaction in Fig. 24, the node G makes the choice between the different alternatives specified by the xalt-construct. Similarly, N2 makes the choice between the alt-operands.

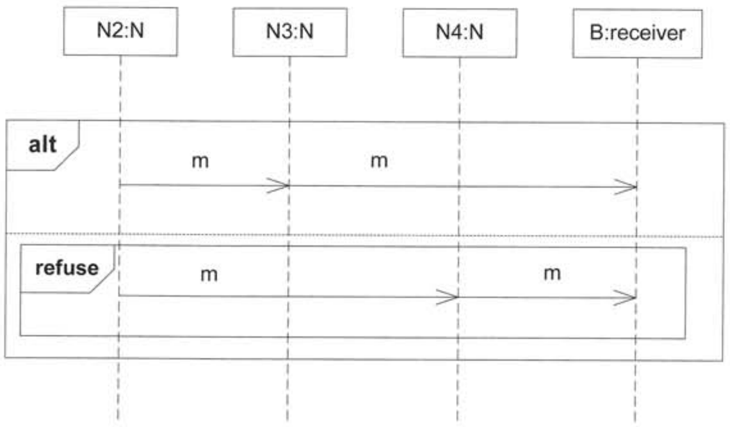


Fig. 22: Redefining positive behaviour as negative.

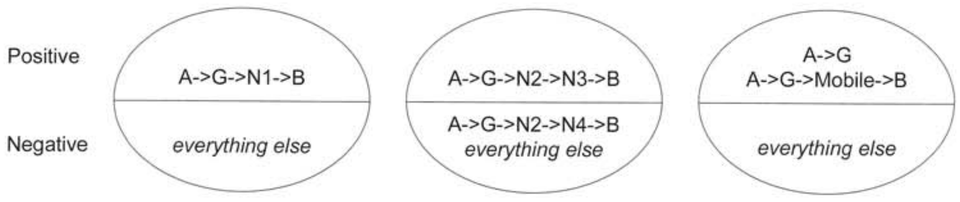


Fig. 23: Semantics of A_Comm with the refinement in Fig. 22.

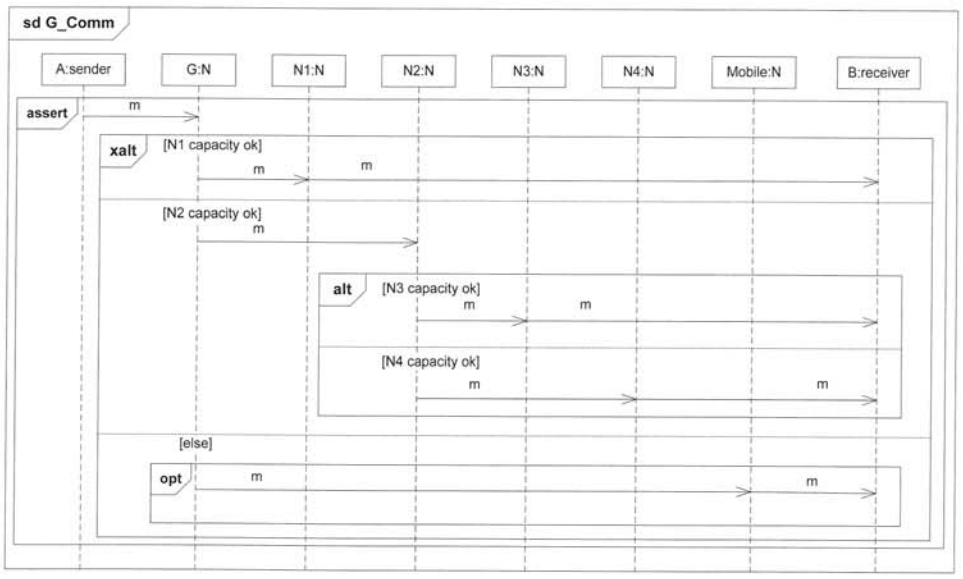


Fig. 24: Introducing guards.

We have assumed that G and N2 have information about the capacity of the different nodes. This may in practice be achieved either by continuous information back from the nodes (not shown in the described behaviour) or through evaluating the communication historically relative to known parameters of the nodes. For our purpose, it is not significant how G and N2 get their data. It is interesting, however, that for xalt the two first guards may both be true, both false, or one true and one false. All of these situations represent cases in real life. If both guards are true, the choice between the two paths may be done arbitrarily. If both guards are false, the else operand comes into effect.

We have not specified what should happen if both guards are false in the alt-fragment. However, according to definition (17) giving the semantics of guarded alt, this is equal to the empty trace, i.e. no further communication takes place.

Fig. 25 illustrates the semantics of G_Comm. All traces with a false guard are negative as specified by definitions (16)–(18). This makes G_Comm a valid refinement of A_Comm. In general, introducing guards in an alt- or xalt-construct will always be a legal refinement step as proved in [15].

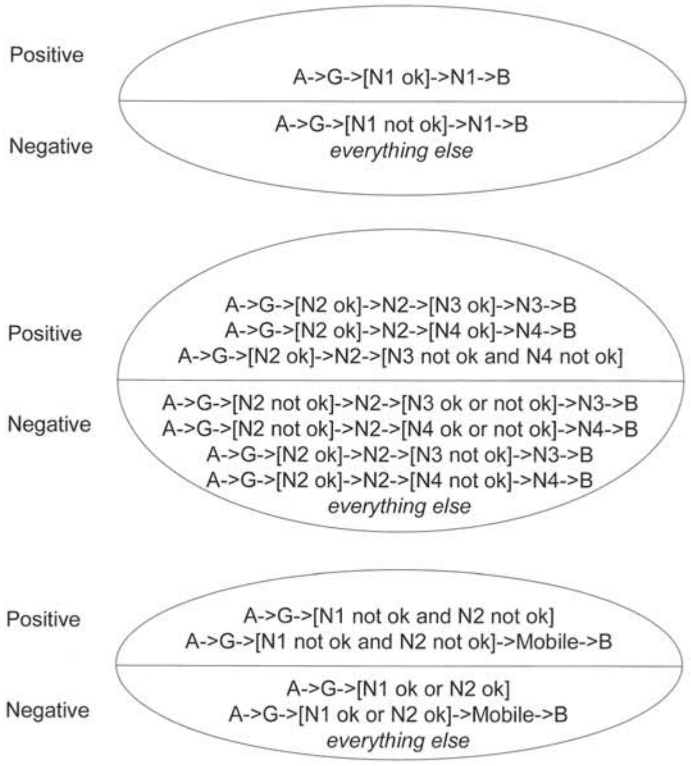

Positive
A->G->[N1 ok]->N1->B
Negative
A->G->[N1 not ok]->N1->B
everything else
Positive
A->G->[N2 ok]->N2->[N3 ok]->N3->B
A->G->[N2 ok]->N2->[N4 ok]->N4->B
A->G->[N2 ok]->N2->[N3 not ok and N4 not ok]
Negative
A->G->[N2 not ok]->N2->[N3 ok or not ok]->N3->B
A->G->[N2 not ok]->N2->[N4 ok or not ok]->N4->B
A->G->[N2 ok]->N2->[N3 not ok]->N3->B
A->G->[N2 ok]->N2->[N4 not ok]->N4->B
everything else
Positive
A->G->[N1 not ok and N2 not ok]
A->G->[N1 not ok and N2 not ok]->Mobile->B
Negative
A->G->[N1 ok or N2 ok]
A->G->[N1 ok or N2 ok]->Mobile->B
everything else

Fig. 25: Semantics of Fig. 24.

5.5 Adding more details

As another example, assume that our sender and receiver suspect that somewhere inside the network there is someone listening to and possibly manipulating their messages. They would like to encrypt their messages and agree (openly) to exchange information to set up a secret key that they shall use for subsequent encryption. Following the procedure outlined by Simon Singh in [17] on how to achieve exchanging of secret keys through insecure communication, we need to be able to describe a number of similar sequences differing basically in the value of some critical numbers.

In Fig. 26 we have shown the protocol with a generalized notation for xalt. We have supplied the xalt with an extra clause which gives one or more parameters with finite domains associated. This generalized notation is identical to replicating the operand for all values of the variable inside the domain.

The behaviour of Fig. 26 means that the sender chooses a natural number (between 0 and 255 in this example) and from that calculates another natural (here in the range 0...10), and this calculated number is transmitted over the insecure network to the receiver. The receiver does exactly the same the other way with a number that he/she chooses. From the numbers that they initially chose and the numbers that they received from each other, they are able to calculate a common key, *p*.

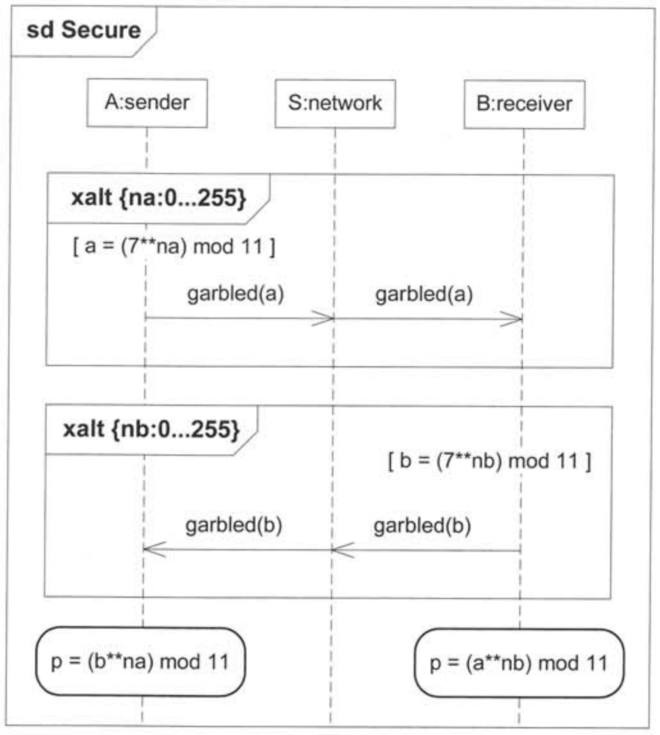


Fig. 26: Generalized xalt for the description of establishing a common secret key.

This key is secret since the network does not have sufficient information to calculate it directly. (Of course, in a real situation the one-way function will be more complicated and the numbers far larger.)

To give a couple of concrete examples, we assume in Fig. 27 that the sender has only the naturals 2 and 3 to choose from, while the receiver chooses only from 4 and 5. The specification in Fig. 27 gives rise to four interaction obligations (with $p = 1, 5, 9$ or 10), one for each possible combination of values for the two lifelines. The choice between these should be nondeterministic, giving the intruder four possible values for the key. With more alternatives for *na* and *nb*, as in the original specification, we get a lot of obligations and potential keys making it difficult for the intruder to find the correct key by plain guessing or by trial-and-error.

In Fig. 28 we indicate a possible decomposition of the sender A in the first xalt-construct in Fig. 26. A is decomposed into a random generator and a sender lifeline C. The generator loops a sufficient number of times, each time sending either 0 or 1 to the sender. Taken together, these messages will constitute the binary

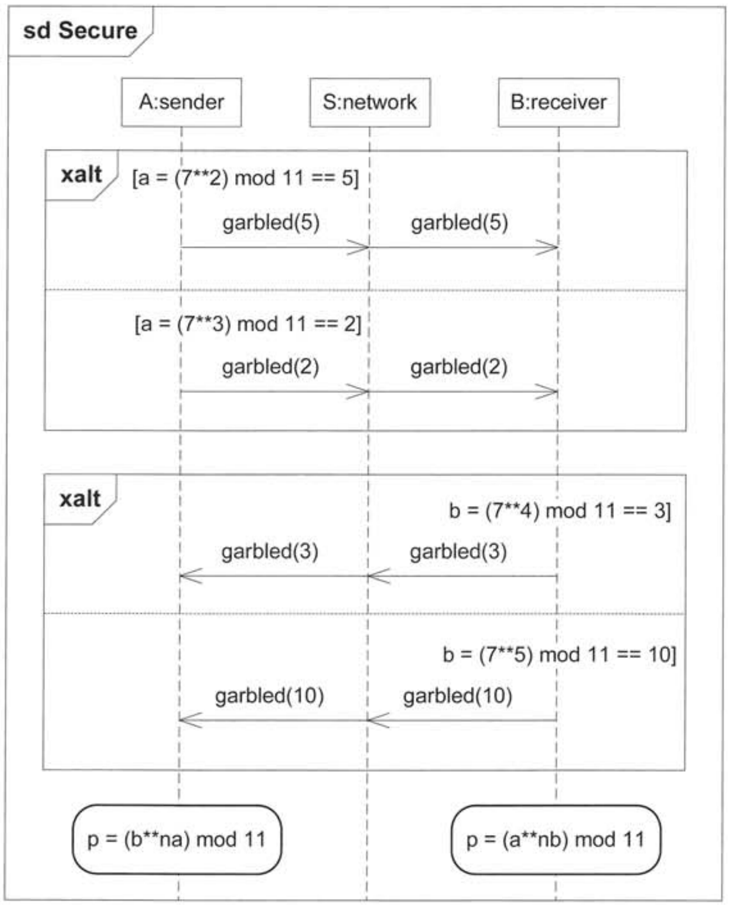


Fig. 27: A few example-values for the generalized xalt.

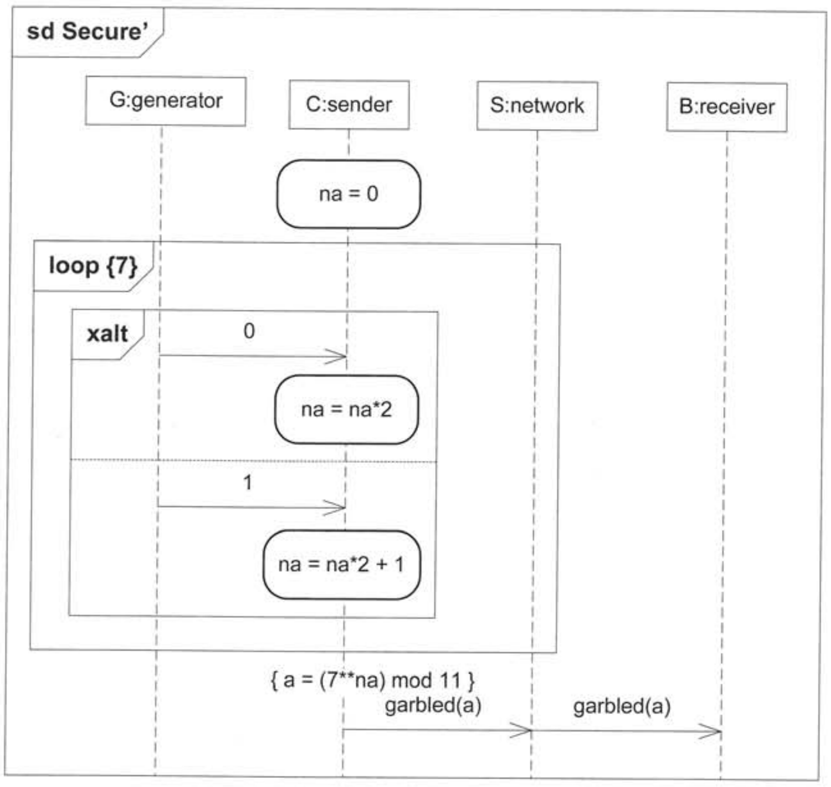


Fig. 28: Refining generalized xalt by loop (and xalt).

representation of the number *na* in Fig. 26. Using xalt here, means that both 0 and 1 must be possible in each round in the loop, giving a totally nondeterministic choice for *na*.

Simple calculations show that we will get the same possible values for *na* in both diagrams, leading to the same obligations and the same values of the parameter *a* in both cases, meaning that the decomposition is indeed a valid refinement.

6. Implementation

In this section we explain what we mean by an implementation and what it means for an implementation to be correct with respect to a STAIRS specification.

Intuitively, if the specification has only one interaction obligation, a correct implementation may only produce traces belonging to the positive and inconclusive trace sets of the obligation, i.e. no negative trace must be produced by the implementation. With more than one interaction obligation, we may in general find the same trace being positive in one obligation while negative in another.

Semantically, we represent implementations in the same way as we represent interactions, namely by sets of interaction obligations. From a semantic point of view, an implementation is a special kind of specification characterized by the following three criteria:

- Its interaction obligations contain no inconclusive traces. Hence, each interaction obligation is of the form $(p, \mathcal{H} \setminus p)$, where $p \neq \emptyset$.
- Whatever typecorrect input it receives from its environment it has at least one output (doing nothing is for example also a response). This means that for any possible environment behaviour, the implementation has at least one trace that is consistent with this behaviour. This corresponds to the notion of winning strategy in Focus [1].
- It behaves causally. Its behaviour at any point in time depends only on what has happened in its past. This is obviously a characteristic of any real-life system (but not necessarily a characteristic of a specification expressed by an interaction). This corresponds to the notion of strong causality in Focus [1].

We say that an implementation I implements a STAIRS specification S if and only if I is a limited refinement of S, i.e. $[\![S]\!] \rightsquigarrow_l [\![I]\!]$. This means that an implementation may not add interaction obligations beyond those given by the specification.

7. Conclusions

In this paper we have explored different kinds of nondeterminism and underspecification, and motivated the need for having two different operators (alt and xalt) for specifying alternative behaviours. Basically, alt defines implicit nondeterminism in the sense of underspecification or abstraction, while xalt defines inherent nondeterminism in the form of explicit choices that must all be present in a valid implementation. We claim that together, these two operators are sufficient to capture the necessary distinctions.

In this paper we have also proposed an extension to STAIRS making it possible to use guards to choose between both implicit (specified by alt) and explicit (specified by xalt) nondeterminism. In particular, the proposed semantics ensures that adding guards to a specification is a valid refinement step. It is straightforward to combine this extension with Timed STAIRS [7], which extends STAIRS with time and three-event semantics.

7.1 Related work

Most formalisms do not distinguish between nondeterminism and underspecification as we have done here. In [18], Walicki and Meldal makes a similar distinction in the setting of algebraic specifications. Their main motivation is that underspecification may some times in fact lead to overspecification, and that in these cases it would be better to use explicit nondeterminism.

In LSC (Live Sequence Charts) [4, 5], charts, locations, messages and conditions may all be characterized as either mandatory or provisional. Provisional charts are called existential and they may happen if their initial condition holds. This is comparable to potential alternatives in STAIRS. Mandatory charts in LSC are called universal. Their interpretation is that provided their initial condition holds, these charts must happen. A universal chart specifies all allowed traces, and is therefore *not* the same as mandatory alternatives in STAIRS, which only specifies some of the traces that must be present in an implementation.

In [2], Cengarle and Knapp define the semantics of UML 2.0 interactions by notions of positive and negative satisfaction. This approach has many similarities with ours, but they do not distinguish between underspecification and explicit non-determinism as we do in STAIRS. With respect to negative traces, their semantics is somewhat different from ours. For alternatives, they define that a trace is negative only if it is negative in both operands. Also, they define that for all possible traces, the trace is negative if a prefix of it is specified as negative, even though the complete trace itself is not described by the diagram. This allows for earlier identification of negative traces. In contrast, we regard such a trace as inconclusive, arguing that if a trace is not described in the diagram, then the specifier has either not thought about the situation or not wanted to classify it as either positive or negative.

In this paper we have modelled data in interactions indirectly through special events representing its use in assignments, constraints, and guards. An example of an alternative approach may be found in [11], where Jonsson and Padilla define a global semantics for an MSC (Message Sequence Chart) by using an Abstract Execution Machine. Here, data are included in the model by associating with each instance an environment consisting of its local variables together with those received as message parameters.

Acknowledgements

The research on which this paper reports has partly been carried out within the context of the IKT-2010 project SARDAS (15295/431) funded by Research Council of Norway. We thank Rolv Bræk, Birger Møller Pedersen, Knut Eilif Husa, Mass Soldal Lund, Atle Refsdal, Judith Rossebø, Manfred Broy, Ina Schieferdecker, Thomas Weigert and the anonymous reviewers for helpful feedback.

References

[1] Broy, M. and Stølen, K. 2001. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer.

[2] Cengarle, M. V. and Knapp, A. 2004. UML 2.0 interactions: semantics and refinement. In *Proc. 3rd Int. Wsh. Critical Systems Development with UML*, Technical report TUM-I0415. Institut für Informatik, Technische Universität München, 85–99.

[3] Chandy, K. M. and Misra, J. 1988. *Parallel Program Design, A Foundation*. Addison-Wesley.

[4] Damm, W. and Harel, D. 1999. LSCs: Breathing life into message sequence charts. In *Proc. Formal Methods for Open Object-Based Distributed Systems*. Kluwer, 293–311.

[5] Harel, D. and Marelly, R. 2003. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer.

[6] Haugen, Ø., Husa, K. E., Runde, R. K., and Stølen, K. 2005. Why timed sequence diagrams require three-event semantics. Tech. Report 309, Department of Informatics, University of Oslo.

[7] HAUGEN, Ø., HUSA, K. E., RUNDE, R. K., AND STØLEN, K. 2005. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, Transformations and Tools*, Volume 3466 of *LNCS*. Springer, 1–25.

[8] HAUGEN, Ø., HUSA, K. E., RUNDE, R. K., AND STØLEN, K. 2005. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling, Online First*, 1–13.

[9] HAUGEN, Ø. AND STØLEN, K. 2003. STAIRS — Steps to analyze interactions with refinement semantics. In *Proc. International Conference on UML*, Volume 2863 of *LNCS*. Springer, 388–402.

[10] JACOB, J. 1989. On the derivation of secure components. In *Proc. IEEE Symposium on Security and Privacy*. IEEE Press, 242–247.

[11] JONSSON, B. AND PADILLA, G. 2001. An execution semantics for MSC-2000. In *Proc. SDL Forum*, Volume 2078 of *LNCS*. Springer, 365–378.

[12] JOSHI, R. AND LEINO, K. R. M. 2000. A semantic approach to secure information flow. *Science of Computer Programming 37*, 113–138.

[13] OBJECT MANAGEMENT GROUP. 2004. *UML 2.0 Superstructure Specification*, document: ptc/04-10-02 edition.

[14] ROSCOE, A. W. 1995. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*. IEEE Press, 114–127.

[15] RUNDE, R. K., HAUGEN, Ø., AND STØLEN, K. 2005. Refining UML interactions with underspecification and nondeterminism. Tech. Report 325, Department of Informatics, University of Oslo.

[16] RUNDE, R. K., HAUGEN, Ø, AND STØLEN, K. 2005. How to transform UML neg into a useful construct. To appear in Proc. Norwegian Informatics Conference (Norsk Informatikkonferanse).

[17] SINGH, S. 1999. *The Code Book: the Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Fourth Estate, London.

[18] WALICKI, M. AND MELDAL, S. 2001. Nondeterminism vs. underspecification. In *Proc. World Multi-Conference on Systemics, Cybernetics and Informatics*, 551–555.