The Pragmatics of STAIRS

Ragnhild Kobro Runde, Øystein Haugen, Ketil Stølen

The Pragmatics of STAIRS

Ragnhild Kobro Runde[1], Øystein Haugen[1], and Ketil Stølen[1,2]

[1] Department of Informatics, University of Oslo, Norway
[2] SINTEF ICT, Norway

Abstract. STAIRS is a method for the compositional development of interactions in the setting of UML 2.0. In addition to defining denotational trace semantics for the main aspects of interactions, STAIRS focuses on how interactions may be developed through successive refinement steps. In this tutorial paper, we concentrate on explaining the practical relevance of STAIRS. Guidelines are given on how to create interactions using the different STAIRS operators, and how these may be refined. The pragmatics is illustrated by a running example.

1 Introduction

STAIRS [HHRS05a] is a method for the compositional development of interactions in the setting of UML 2.0 [OMG05]. In contrast to e.g. UML state machines and Java programs, interactions are usually incomplete specifications, typically describing example runs of the system. STAIRS is designed to deal with this incompleteness. Another important feature of STAIRS is the possibility to distinguish between alternative behaviours representing underspecification and alternative behaviours that must all be present in an implementation, for instance due to inherent nondeterminism.

STAIRS is not intended to be a complete methodology for system development, but should rather be seen as a supplement to methodologies like e.g. RUP [Kru04]. In particular, STAIRS focuses on refinement, which is a development step where the specification is made more complete by information being added to it in such a way that any valid implementation of the refined specification will also be a valid implementation of the original specification.

In this paper we focus on refinement relations. We define general refinement, which in turn has four special cases referred to as narrowing, supplementing, detailing and limited refinement. Narrowing means to reduce the set of possible system behaviours, thus reducing underspecification. Supplementing, on the other hand, means to add new behaviours to the specification, taking into account the incomplete nature of interactions, while detailing means to add more details to the specification by decomposing lifelines. By general refinement, the nondeterminism required of an implementation may be increased freely, while limited refinement is a special case restricting this possibility.

Previous work on STAIRS has focused on its basic ideas, explaining the various concepts such as the distinction between underspecification and inherent nondeterminism [HHRS05a,RHS05b], time [HHRS05b], and negation [RHS05a],

1

as well as how these are formalized. In this paper, we take the theory of STAIRS one step further, focusing on its practical consequences by giving practical guidelines on how to use STAIRS. In particular, we explain how to use the various STAIRS operators when creating specifications in the form of interactions, and how these specifications may be further developed through valid refinement steps.

The paper is organized as follows: In Sect. 2 we give a brief introduction to interactions and their semantic model as we have defined it in STAIRS. Section 3 is an exampleguided walkthrough of the main STAIRS operators for creating interactions, particularly focusing on alternatives and negation. For each operator we give both its formal definition and guidelines for its practical usage. Section 4 gives the pragmatics of refining interactions. In Sect. 5 we explain how STAIRS relates to other similar approaches, in particular UML 2.0, while we give some concluding remarks in Sect. 6.

2   The Semantic Model of STAIRS

In this section we give a brief introduction to interactions and their trace semantics as defined in STAIRS. The focus here is on the semantic model. Definitions of concrete syntactical operators will mainly be presented together with the discussion of these operators later in this paper. For a thorough account of the STAIRS semantics, see [HHRS05b] and the extension with data in [RHS05b].

An interaction describes one or more positive (i.e. valid) and/or negative (i.e. invalid) behaviours. As a very simple example, the sequence diagram in Fig. 1 specifies a scenario in which a client sends the message cancel(appointment) to an appointment system, which subsequently sends the message appointmentCancelled back to the client, together with a suggestion for a new appointment to which the client answers with the message yes. The client finally receives the message appointmentMade.
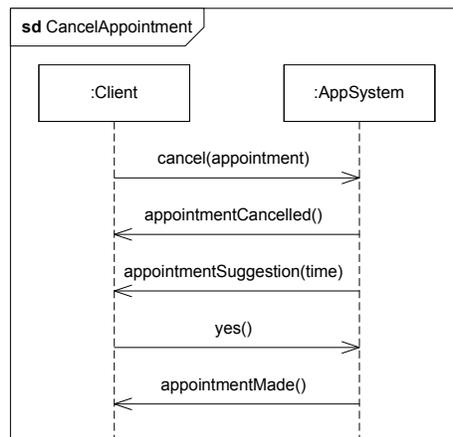

Fig. 1. Example interaction: CancelAppointment

2

Formally, we use denotational trace semantics to explain the meaning of a single interaction. A trace is a sequence of events, representing a system run. The most typical examples of events are the sending and the reception of a message, where a message is a triple (s, tr, re) consisting of a signal s, a transmitter lifeline tr and a receiver lifeline re. For a message m, we let $!m$ and $?m$ denote the sending and the reception of m, respectively. As will be explained in Sect. 3.2, we also have some special events representing the use of data in e.g. constraints and guards.

The diagram in Fig. 1 includes ten events, two for each message. These are combined by the implicit weak sequencing operator seq, which will be formally defined at the end of this section. Informally, the set of traces described by such a diagram is the set of all possible sequences consisting of its events such that the send event is ordered before the corresponding receive event, and events on the same lifeline are ordered from top down. Shortening each message to the first and the capitalized letter of its signal, we thus get that Fig. 1 specifies two positive traces $\langle !c, ?c, !aC, ?aC, !aS, ?aS, !y, ?y, !aM, ?aM \rangle$ and $\langle !c, ?c, !aC, !aS, ?aC, ?aS, !y, ?y, !aM, ?aM \rangle$, where the only difference is the relative ordering of the two events $?aC$ and $!aS$. Figure 1 gives no negative traces.

Formally, we let $\mathcal{H}$ denote the set of all well-formed traces. A trace is well-formed if, for each message, the send event is ordered before the corresponding receive event. An *interaction obligation* (p, n) is a pair of trace-sets which gives a classification of all of the traces in $\mathcal{H}$ into three categories: the positive traces p, the negative traces n and the inconclusive traces $\mathcal{H} \setminus (p \cup n)$. The inconclusive traces result from the incompleteness of interactions, representing traces that are not described as positive or negative by the current interaction. We say that the interaction obligation is contradictory if the same trace is both positive and negative, i.e. if $p \cap n \neq \emptyset$. To give a visual presentation of an interaction obligation, we use an oval divided into three regions as shown in Fig. 2.
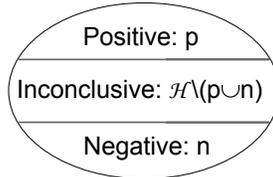

Fig. 2. Illustrating an interaction obligation

As explained in the introduction, one of the main advantages of STAIRS is its ability to distinguish between traces that an implementation *may* exhibit (e.g. due to underspecification), and traces that it *must* exhibit (e.g. due to inherent nondeterminism). Semantically, this distinction is captured by stating that the semantics of an interaction d is a *set* of m interaction obligations, $[\![d]\!] = \{(p_1, n_1), \ldots, (p_m, n_m)\}$. Intuitively, the traces allowed by an interaction obligation (i.e. its positive and inconclusive traces) represent potential alterna-

tives, where being able to produce only one of these traces is sufficient for an implementation. On the other hand, the different interaction obligations represent mandatory alternatives, in the sense that each obligation specifies traces of which at least one must be possible for a correct implementation of the specification.

We are now ready to give the formal definition of seq. First, weak sequencing of trace sets is defined by:

$$s_1 \succsim s_2 \overset{\text{def}}{=} \{ h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in \mathcal{L} : h \lceil l = h_1 \lceil l \frown h_2 \lceil l \} \quad (1)$$

where $\mathcal{L}$ is the set of all lifelines, $\frown$ is the concatenation operator on sequences, and $h \lceil l$ is the trace h with all events not taking place on the lifeline l removed. Basically, this definition gives all traces that may be constructed by selecting one trace from each operand and combining them in such a way that the events from the first operand must come before the events from the second operand (i.e. the events are ordered from top down) for all lifelines. Events from different operands may come in any order, as long as sending comes before reception for each message (as required by $h \in \mathcal{H}$). Notice that weak sequencing with an empty set as one of the operands yields the empty set.

Weak sequencing of interaction obligations is defined by:

$$(p_1, n_1) \succsim (p_2, n_2) \overset{\text{def}}{=} (p_1 \succsim p_2, (n_1 \succsim p_2) \cup (n_1 \succsim n_2) \cup (p_1 \succsim n_2)) \quad (2)$$

Finally, seq is defined by

$$
\begin{aligned}
[\![\ \mathsf{seq}\ [d]\ ]\!] &\overset{\text{def}}{=} [\![\ d\ ]\!] \\
[\![\ \mathsf{seq}\ [D, d]\ ]\!] &\overset{\text{def}}{=} \{ o_1 \succsim o_2 \mid o_1 \in [\![\ \mathsf{seq}\ [D]\ ]\!] \wedge o_2 \in [\![\ d\ ]\!] \}
\end{aligned}
\quad (3)
$$

where d is a single interaction and D a list of one or more interactions. For a further discussion of seq, see Sect. 3.4.

3 The Pragmatics of Creating Interactions

In this section, we focus on the different syntactical constructs of interactions in order to explain the main theory of STAIRS and how these constructs should be used in practical development. For each construct, we demonstrate its usage in a practical example, to motivate its formal semantics and the pragmatic rules and guidelines that conclude each subsection.

As our example, we will use a system for appointment booking to be used by e.g. doctors and dentists. The appointment system should have the following functionality:

- MakeAppointment: The client may ask for an appointment.
- CancelAppointment: The client may cancel an appointment.
- Payment: The system may send an invoice message asking the client to pay for the previous or an unused appointment.

The interactions specifying this system will be developed in a stepwise manner. In Sect. 4 we will justify that all of these development steps are valid refinement steps in STAIRS.

3.1 The Use of alt Versus xalt

Consider again the interaction in Fig. 1. As explained in Sect. 2, this interaction specifies two different traces, depending on whether the client receives the message appointmentCancelled before or after the system sends the message appointmentSuggestion. Which one of these we actually get when running the final system, will typically depend on the kind of communication used between the client and our system. If the communication is performed via SMS or over the internet, we may have little or no delay, meaning that the first of these messages may be received before the second is sent. On the other hand, if communication is performed by sending letters via ordinary mail, both messages (i.e. letters) will probably be sent before the first one arrives at the client.

Seeing that the means of communication is not specified in the interaction, all of these are acceptable implementations. Also, it is sufficient for an implementation to have only one of these available. Hence, the two traces of Fig. 1 exemplify underspecification. In the semantics, this is represented by the two traces being grouped together in the same interaction obligation.

The underspecification in Fig. 1 is an implicit consequence of weak sequencing. Alternatives representing underspecification may also be specified explicitly by using the operator alt, as in the specification of MakeAppointment in Fig. 3. In this interaction, when requesting a new appointment the client may ask for either a specific date or a specific hour of the day (for instance if the client prefers his appointments to be after work or during the lunch break). As we do not require the system to offer both of these alternatives, they are specified using alt. After the client has asked for an appointment, the appointment is set up according to the referenced interaction DecideAppTime.
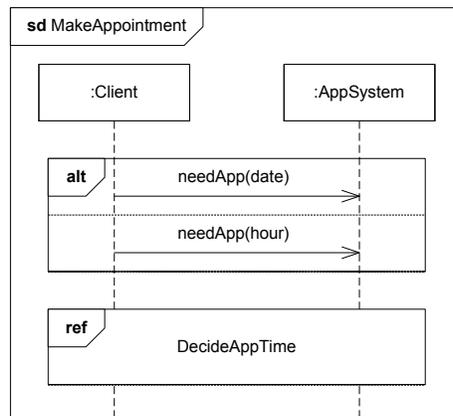

Fig. 3. MakeAppointment

The specification of DecideAppTime is given in Fig. 4. Here, the system starts with suggesting an appointment, and the client then answers either yes or no. Finally, the client gets a receipt according to his answer. As the system must

be able to handle both yes and no as reply messages, these alternatives are *not* instances of underspecification. Specifying these by using alt would therefore be insufficient. Instead, they are specified by the xalt operator (first introduced in [HS03]) in order to capture alternative traces where an implementation must be able to perform all alternatives.
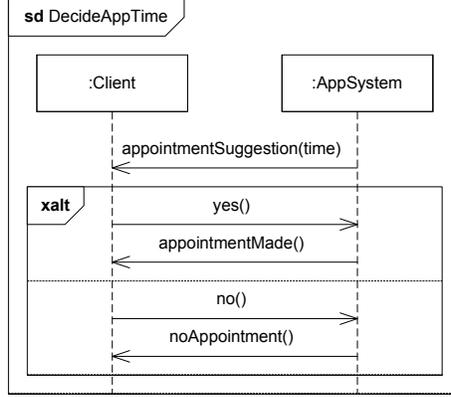

Fig. 4. DecideAppTime

In Fig. 5, which gives a revised version of CancelAppointment from Fig. 1, another use of xalt is demonstrated. In this case, xalt is used to model alternatives where the conditions under which each of them may be chosen is not known. This interaction specifies that if a client tries to cancel an appointment, he may either get an error message or he may get a confirmation of the cancellation, after which the system tries to schedule another appointment (in DecideAppTime). In Sect. 3.2 we demonstrate how guards may be added as a means to constrain the applicability of the two alternatives in this example.

A third use of xalt is to specify inherent nondeterminism, as in a coin toss where both heads and tails should be possible outcomes. More examples, and a discussion of the relationship between alt and xalt, may be found in [RHS05b] and [RRS06].

The crucial question when specifying alternatives is: Do these alternatives represent similar traces in the sense that implementing only one is sufficient? If yes, use alt. Otherwise, use xalt.

Formally, the operands of an xalt result in distinct interaction obligations in order to model the situation that they must all be possible for an implementation. On the other hand, alt combines interaction obligations in order to model underspecification:

$$[\![\text{ xalt } [d_1, \ldots d_m]]\!] \stackrel{\text{def}}{=} \bigcup_{i \in [1,m]} [\![d_i]\!] \tag{4}$$

$$[\![\text{ alt } [d_1, \ldots, d_m]]\!] \stackrel{\text{def}}{=} \{ \biguplus \{o_1, \ldots, o_m\} \mid \forall i \in [1, m] : o_i \in [\![d_i]\!] \} \tag{5}$$
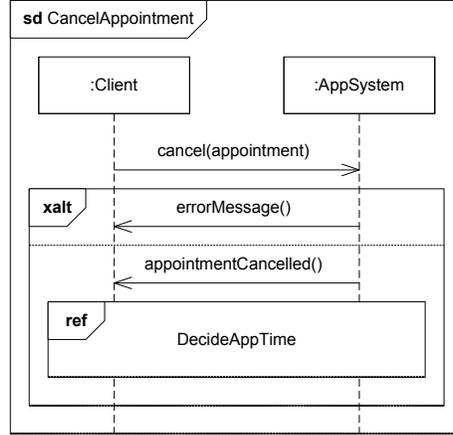
6

Fig. 5. CancelAppointment revisited

where m is the number of interaction operands and the inner union of interaction obligations, $\biguplus$, is defined as:

$$\biguplus_{i\in[1,m]} (p_i, n_i) \stackrel{\text{def}}{=} (\bigcup_{i\in[1,m]} p_i, \bigcup_{i\in[1,m]} n_i) \qquad (6)$$

The difference between alt and xalt is also illustrated in Fig. 6, which is an informal illustration of the semantics of Fig. 3. The dotted lines should be interpreted as parentheses grouping the semantics of sub-interactions, and the second seq-operand is the semantics of the referenced interaction DecideApp-Time. In each interaction obligation of Fig. 6, every trace that is not positive is inconclusive, as Fig. 3 gives no negative traces.

Every interaction using the STAIRS-operators except for infinite loop is equivalent to an interaction having xalt as the top-level operator. This is because xalt describes mandatory alternatives. If there are only finitely many alternatives (which is the case if there is no infinite loop) they may be listed one by one. In particular, we have that all of these operators distribute over xalt. For instance, we have that the interaction alt [xalt $[d_1, d_2], d_3$] is equivalent to the interaction xalt [alt $[d_1, d_3]$, alt $[d_2, d_3]$], and similarly for interactions with more than two operands and for the other operators.

The pragmatics of alt vs xalt
- Use alt to specify alternatives that represent similar traces, i.e. to model
  - underspecification.
- Use xalt to specify alternatives that must all be present in an implementation, i.e. to model
  - inherent nondeterminism, as in the specification of a coin toss.
  - alternative traces due to different inputs that the system must be able to handle (as in Fig. 4);
  - alternative traces where the conditions for these being positive are abstracted away (as in Fig. 5).
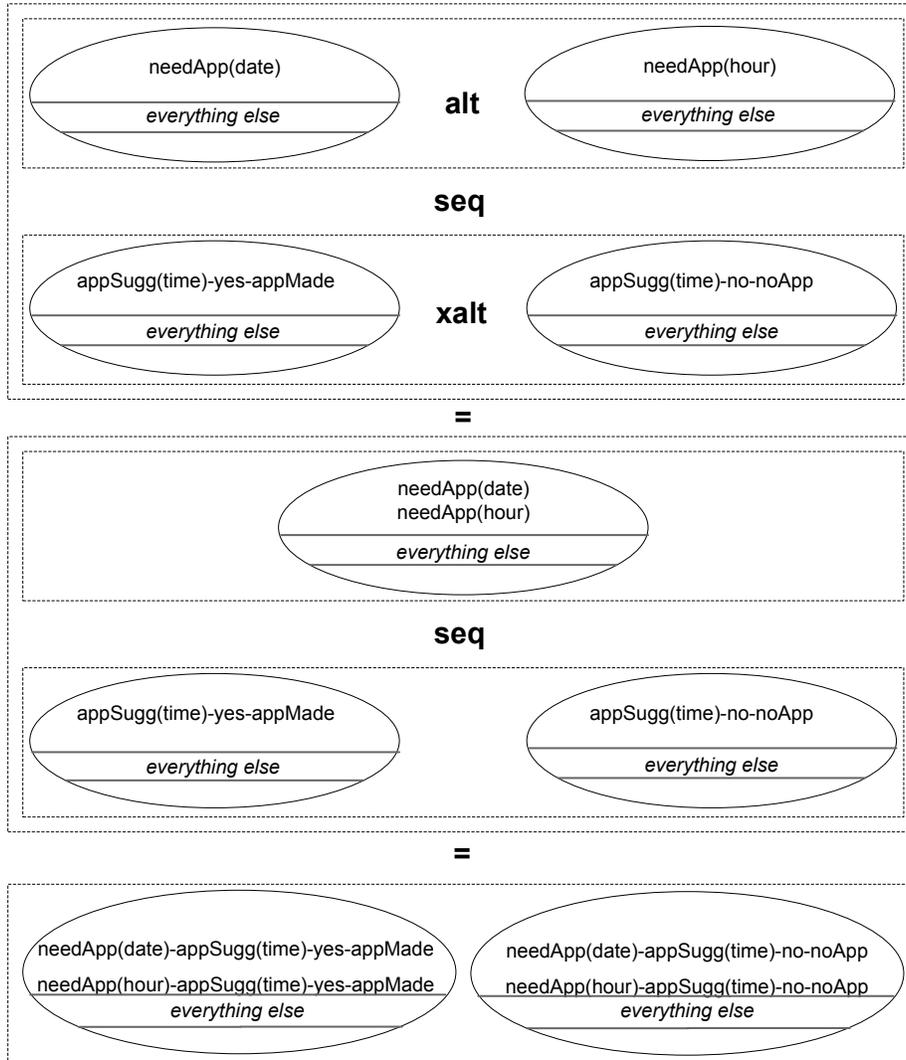

7

Fig. 6. Illustrating MakeAppointment

3.2 The Use of Guards

In Fig. 5, xalt was used in order to specify that the system should be able to respond with either an error message or with the receipt message appointment-Cancelled (followed by DecideAppTime), if a client wants to cancel an appointment. With the current specification, the choice between these alternatives may be performed nondeterministically, but as suggested in the previous section, it is more likely that there exist some conditions for when each of the alternatives

may be chosen. In Fig. 7 these conditions are made explicit by adding them to the specification in the form of guards as a step in the development process.
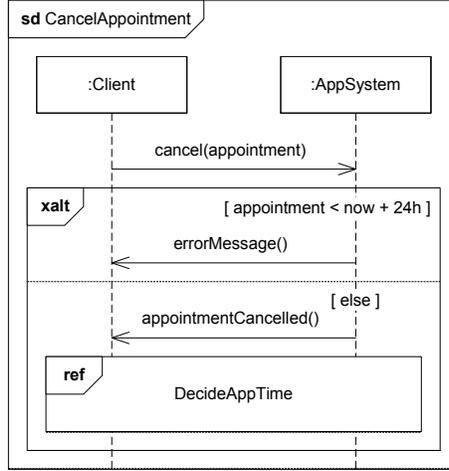

Fig. 7. CancelAppointment revisited

For the first alternative, the guard is used to specify that if the client wants to cancel an appointment less than 24 hours in advance, he will get an error message. In general, the guard else may be used as a short-hand for the conjunction of the negation of all the other guards. This means that for the second alternative of Fig. 7, the appointment will be cancelled and the system will try to schedule a new appointment only if the appointment is at least 24 hours away.

Similarly, in Fig. 8, guards are added to the alt-construct of Fig. 3 in order to constrain the situations in which each of the alternatives needApp(date) and needApp(hour) is positive. The guards specify that the client may only ask for an appointment at today or at a later date, or between the hours of 7 A.M. and 5 P.M. We recommend that one always makes sure that the guards of an alt-construct are exhaustive. Therefore, Fig. 8 also adds an alternative where the client asks for an appointment without specifying either date or hour. This alternative has the guard true, and may always be chosen. As this example demonstrates, the guards of an alt-construct may be overlapping. This is also the case for xalt.

In order to capture guards and more general constraints in the semantics, the semantics is extended with the notion of a state. A state σ is a total function assigning a value (in the set Val) to each variable (in the set Var). Formally, $\sigma \in Var \rightarrow Val$. Semantically, a constraint is represented by the special event $check(\sigma)$, where σ is the state in which the constraint is evaluated:

$$[\![\, \mathsf{constr}(c) \,]\!] \overset{\text{def}}{=} \{ \, (\{\langle check(\sigma) \rangle \mid c(\sigma)\}, \{\langle check(\sigma) \rangle \mid \neg c(\sigma)\}) \, \} \qquad (7)$$
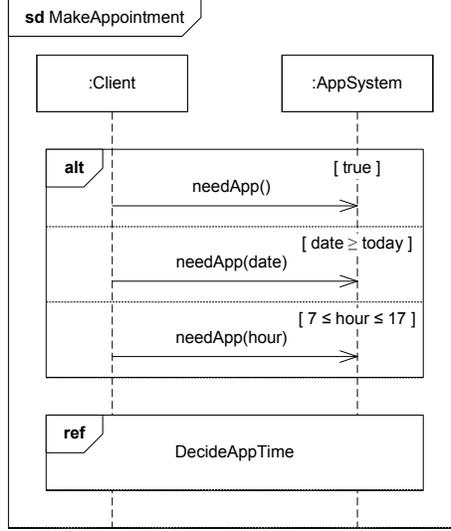
9

Fig. 8. MakeAppointment revisited

The semantics of guarded xalt is defined by:

$$[\![\; \mathsf{xalt} \; [c_1 \rightarrow d_1, \ldots, c_m \rightarrow d_m] \;]\!] \;\; \stackrel{\mathrm{def}}{=} \bigcup_{i \in [1,m]} [\![\; \mathsf{seq} \; [\mathsf{constr}(c_i), d_i] \;]\!] \qquad (8)$$

Notice that for all states, a constraint is either true and the trace $\langle check(\sigma) \rangle$ is positive, or the constraint is false and the trace $\langle check(\sigma) \rangle$ is negative. For guarded xalt (and similarly for alt defined below), this has the consequence that a guard must cover *all* possible situations in which the specified traces are positive, since a false guard means that the traces described by this alternative are negative. When relating specifications with and without guards, an alt/xalt-operand without a guard is interpreted as having the guard true. This interpretation, together with the use of constr in the definition of guards, ensures that adding guards to a specification (as in the examples above) is a valid refinement step as will be explained in Sect. 4.

The semantics of guarded alt is defined by:

$$[\![\; \mathsf{alt} \; [c_1 \rightarrow d_1, \ldots, c_m \rightarrow d_m] \;]\!] \;\; \stackrel{\mathrm{def}}{=} \qquad\qquad (9)$$
$$\{ \; \uplus \{o_1, \ldots, o_m\} \mid \forall i \in [1,m] : o_i \in [\![\; \mathsf{seq} \; [\mathsf{constr}(c_i), d_i] \;]\!] \; \}$$

The UML 2.0 standard ([OMG05]) states that if all guards in an alt-construct are false then the empty trace $\langle \rangle$ (i.e. doing nothing) should be positive. In [RHS05b], we gave a definition of guarded alt which was consistent with the standard. However, implicitly adding the empty trace as positive implies that alt is no longer associative. For this reason, we have omitted this implicit trace from definition (9).

Definition (9) is consistent with our general belief that everything which is not explicitly described in an interaction should be regarded as inconclusive for

10

that diagram. If all guards are false, all of the described traces are negative and the interaction has an empty set of positive traces. To avoid confusion between our definition and that of UML, we recommend to always make sure that the guards of an alt-construct are exhaustive. If desired, one of the alternatives may be the empty diagram, skip, defining the empty trace as positive:

$$\llbracket \text{ skip } \rrbracket \overset{\text{def}}{=} \{(\{\langle\rangle\}, \emptyset)\} \tag{10}$$

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

3.3 The Use of refuse, veto and assert

As explained in the introduction, interactions are incomplete specifications, specifying only example runs as opposed to the complete behaviour of the system. In this setting, it is particularly important to specify not only positive, but also negative traces, stating what the system is *not* allowed to do. In STAIRS, negative traces are defined by using one of the operators refuse, veto, or assert. The operators refuse and veto are both used to specify that the traces of its operand should be considered negative. They differ only in that veto has the empty trace as positive, while refuse does not have any positive traces at all. The importance of this difference will be explained later in this section, after the formal definitions. The assert operator specifies that only the traces in its operand are positive and that all other traces are negative.

In the revised version of DecideAppTime given in Fig. 9, these three operators are used in order to add negative traces to the specification in Fig. 4. Figure 9 also adds some positive traces via the loop-construct, which may be interpreted as an alt between performing the contents of the loop 0, 1, 2, 3, or 4 times. For a formal definition of loop, see [RHS05b].

Inside the loop, veto is used to specify that after the client has answered no to the suggested appointment, the system should not send the message AppointmentMade before suggesting another appointment. In the first xalt-operand, alt in combination with refuse is used to specify that the client should get the receipt message AppointmentMade when he accepts the suggested appointment, and that it would be negative if he instead got the message noAppointment. In the second xalt-operand, assert is used to specify that the system should send the message noAppointment after the client has answered with the final no, and that no other traces are allowed. This is in contrast to the first xalt-operand, which defines one positive and one negative trace, but leaves all other traces inconclusive.
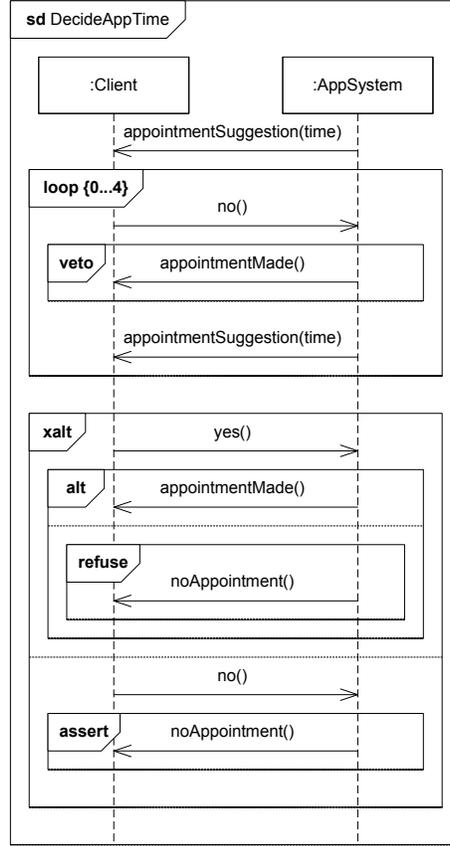
11

Fig. 9. DecideAppTime revisited

Formally, refuse and veto are defined by:

$$\llbracket \text{ refuse } [d] \rrbracket \stackrel{\text{def}}{=} \{(\emptyset, p \cup n) \mid (p, n) \in \llbracket \, d \, \rrbracket\} \tag{11}$$

$$\text{veto} \, [d] \stackrel{\text{def}}{=} \text{alt} \, [\, \text{skip}, \text{refuse} \, [d] \,] \tag{12}$$

Both operators define that all traces described by its operand should be considered negative. The difference between refuse and veto is that while refuse has *no* positive traces, veto has the empty trace as positive, meaning that doing nothing is positive for veto. To understand the importance of this difference, it is useful to imagine that for each lifeline, each interaction fragment is implemented as a subroutine. Entering a new interaction fragment will then correspond to calling the subroutine that implements this fragment. For an interaction fragment with refuse as its main operator, no such subroutine may exist, as there are no positive traces. Hence, the program fails to continue in such a case. However, an interaction fragment with veto as its main operator, corresponds to an empty routine that immediately returns and the program may continue with the interaction fragment that follows.

12

The choice of operator for a concrete situation, will then depend on the question: Should doing nothing be possible in this otherwise negative situation? If yes, use veto. If no, use refuse.

Consider again Fig. 9. Here, veto is used inside the loop construct as sending the message no (then doing nothing), and then sending AppointmentSuggestion(time) should be positive. On the other hand, refuse is used in the first xalt operand, as we did not want to specify the message yes followed by doing nothing as positive.

Using assert ensures that for each interaction obligation of its operand, at least one of the described positive traces will be implemented by the final system, as all inconclusive traces are redefined as negative. Formally:

$$\llbracket \text{ assert } [d] \rrbracket \stackrel{\text{def}}{=} \{(p, n \cup (\mathcal{H} \setminus p)) \mid (p,n) \in \llbracket d \rrbracket\} \tag{13}$$

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

3.4 The Use of seq

As explained in Sect. 2, the weak sequencing operator seq is the implicit composition operator for interactions, 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.

Consider again the revised specification of CancelAppointment in Fig. 7. In the second xalt-operand, the system sends the message appointmentCancelled to the client, and subsequently the referenced interaction DecideAppTime is performed. Here, the first thing to happen is that the system sends the message AppointmentSuggestion to the client (as specified in Fig. 9).

As seq is the operator used for sequencing interaction fragments, this means that in general *no* synchronization takes place at the beginning of an interaction fragment, i.e. that different lifelines may enter the fragment at different points in time. In the context of Fig. 7, this means that there is no ordering between the reception of the message appointmentCancelled and the sending of the message

AppointmentSuggestion, in exactly the same way as there would have been no ordering between these if the specification had been written in one single diagram instead of using a referenced interaction.

In Fig. 10, traces are added to the specification of CancelAppointment in the case where the client wants to cancel an appointment less than 24 hours before it is supposed to take place.
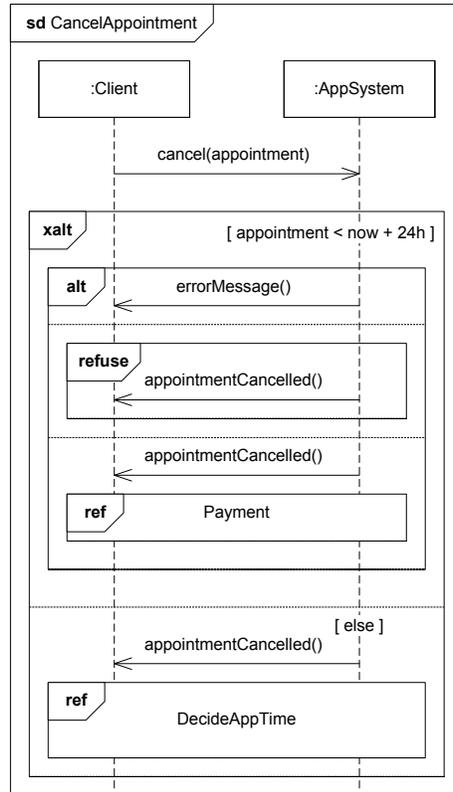

Fig. 10. CancelAppointment revisited

The first alt-operand specifies that the system may give an error message (as before). The second operand specifies that the sending of the message appointmentCancelled alone is negative, while the third operand specifies that sending the message appointmentCancelled and then performing (the positive traces of) Payment (specified in Fig. 11) is positive.

This example demonstrates that a trace (e.g. appointmentCancelled followed by Payment) is not necessarily negative even if a prefix of it (e.g. appointmentCancelled) is. This means that the total trace must be considered when categorizing it as positive, negative or inconclusive. Another consequence is that every trace which is not explicitly shown in the interaction should be inconclusive. For in-
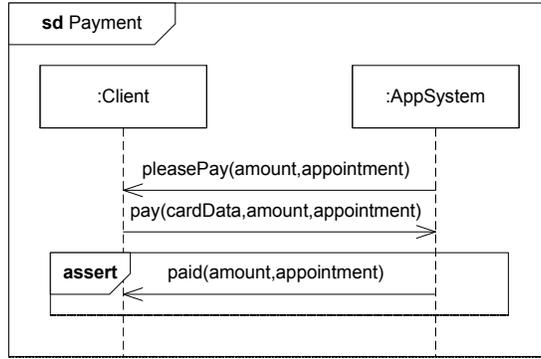
14

Fig. 11. Payment

stance, in Fig. 10 all traces where the message appointmentCancelled is followed by something other than Payment, are still inconclusive.

The formal definition of seq was given in Sect. 2. As no synchronization takes place at the beginning of each seq-operand, it follows from the definitions that i.e. seq $[d_1, \mathsf{alt}\ [d_2, d_3]] = \mathsf{alt}\ [\mathsf{seq}\ [d_1, d_2], \mathsf{seq}\ [d_1, d_3]]$ and that loop $\{2\}$ $[d] =$ seq $[d, d]$ as could be expected.

The pragmatics of weak sequencing

– Be aware that by weak sequencing,
  - a positive sub-trace followed by a positive sub-trace is positive.
  - a positive sub-trace followed by a negative sub-trace is negative.
  - a negative sub-trace followed by a positive sub-trace is negative.
  - a negative sub-trace followed by a negative sub-trace is negative.
  - the remaining trace combinations are inconclusive.

---

4 The Pragmatics of Refining Interactions

In a development process, specifications may be changed for several reasons, including capturing new user requirements, giving a more detailed design, or correcting errors. STAIRS focuses on those changes which may be defined as refinements. In this section, we explain some main kinds of refinement in STAIRS, and demonstrate how each of the development steps taken in the example in Sect. 3 are valid refinement steps.

Figure 12 illustrates how the different refinement notions presented in this paper are related. Supplementing, narrowing and detailing are all special cases of the general refinement notion. Limited refinement is a restricted version of general refinement, which limits the possibility to increase the nondeterminism required of an implementation. In Fig. 12, we have also illustrated what refinement relation is used for each of the development steps in our running example. For instance, the placement of $1 \rightarrow 5$ means that Fig. 5 is a supplementing and general refinement of Fig. 1, but not a limited refinement.
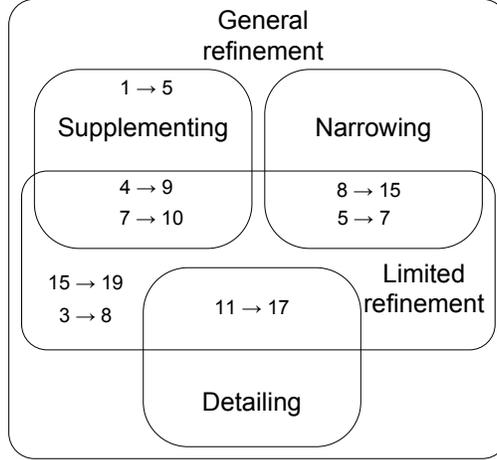
15

Fig. 12. The refinement relations of STAIRS

In the discussion of each of the five refinement notions, we will refer to the following definition of compositionality:

Definition 1 (Compositionality). *A refinement operator $\rightsquigarrow$ is compositional if it is*

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

$$d \rightsquigarrow d' \Rightarrow \text{refuse } [d] \rightsquigarrow \text{refuse } [d']$$
$$d \rightsquigarrow d' \Rightarrow \text{veto } [d] \rightsquigarrow \text{veto } [d']$$
$$d_1 \rightsquigarrow d'_1, \ldots, d_m \rightsquigarrow d'_m \Rightarrow \text{alt } [d_1, \ldots, d_m] \rightsquigarrow \text{alt } [d'_1, \ldots, d'_m]$$
$$d_1 \rightsquigarrow d'_1, \ldots, d_m \rightsquigarrow d'_m \Rightarrow \text{xalt } [d_1, \ldots, d_m] \rightsquigarrow \text{xalt } [d'_1, \ldots, d'_m]$$
$$d_1 \rightsquigarrow d'_1, \ldots, d_m \rightsquigarrow d'_m \Rightarrow \text{seq } [d_1, \ldots, d_m] \rightsquigarrow \text{seq } [d'_1, \ldots, d'_m]$$

Transitivity enables the stepwise development of interactions, while monotonicity is important as it means that the different parts of an interaction may be refined separately.

4.1 The Use of Supplementing

As interactions are incomplete specifications typically describing only example runs, we may usually find many possible traces that are inconclusive in a given interaction obligation. By supplementing, inconclusive traces are re-categorized as either positive or negative as illustrated for a single interaction obligation in Fig. 13. Supplementing is an activity where new situations are considered, and will most typically be used during the early phases of system development. Examples of supplementing includes capturing new user requirements and adding fault tolerance to the system.
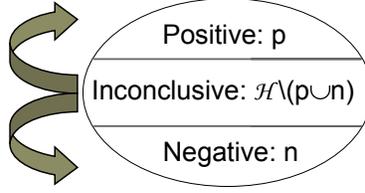
Fig. 13. Supplementing of interaction obligations

DecideAppTime in Fig. 9 is an example of supplementing, as it adds both positive and negative traces to the specification in Fig. 4. All traces that were positive in the original specification, are still positive in the refinement. Another example of supplementing is CancelAppointment in Fig. 10, which adds traces to the specification in Fig. 7. Again, all traces that were positive in the original specification remain positive in the refinement, and the negative traces remain negative.

Formally, supplementing of interaction obligations is defined by:

$$(p, n) \rightsquigarrow_s (p', n') \ \stackrel{\text{def}}{=} \ p \subseteq p' \ \wedge \ n \subseteq n' \tag{14}$$

For an interaction with a set of interaction obligations as its semantics, we require that each obligation for the original interaction must have a refining obligation in the semantics of the refined interaction. This ensures that the alternative traces (e.g. the inherent nondeterminism) required by an interaction are also required by the refinement. Formally:

$$d \rightsquigarrow_s d' \ \stackrel{\text{def}}{=} \ \forall o \in [\![\, d \,]\!] : \exists o' \in [\![\, d' \,]\!] : o \rightsquigarrow_s o' \tag{15}$$

Supplementing is compositional as defined by Definition 1.

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

4.2 The Use of Narrowing

Narrowing means to reduce underspecification by redefining positive traces as negative, as illustrated in Fig. 14. As for supplementing, negative traces must remain negative in the refinement.

One example of narrowing, is adding guards to CancelAppointment in Fig. 7. In the original specification in Fig. 5, we had for instance no constraint on the alternative with the message appointmentCancelled, while in the refinement this alternative is negative if it occurs less than 24 hours prior the appointment.

In general, adding guards to an alt/xalt-construct is a valid refinement through narrowing. Seeing that an operand without a guard is interpreted as having true
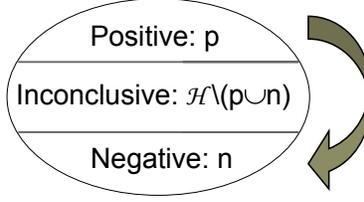
17

Fig. 14. Narrowing of interaction obligations

as guard, this is a special case of a more general rule, stating that a valid refinement may limit a guard as long as the refined condition implies the original one. This ensures that all of the positive traces of the refinement were also positive (and not negative) in the original specification.

Another example of narrowing is given in MakeAppointment in Fig. 15. Here, the refuse-operator is used to specify that the client may *not* ask for an appointment at a specific hour. This means that even though these traces were positive in the specification in Fig. 8, they are now considered negative in the sense that asking for a specific hour is not an option in the final implementation.
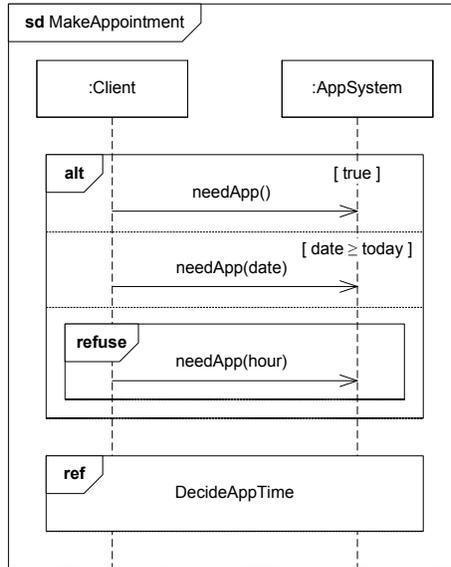

Fig. 15. MakeAppointment revisited

Formally, narrowing of interaction obligations is defined by:

$$(p, n) \rightsquigarrow_n (p', n') \stackrel{\text{def}}{=} p' \subseteq p \ \wedge \ n' = n \cup p \setminus p' \tag{16}$$

and narrowing of interactions by:

$$d \rightsquigarrow_n d' \stackrel{\text{def}}{=} \forall o \in [\![\, d \,]\!] : \exists o' \in [\![\, d' \,]\!] : o \rightsquigarrow_n o' \tag{17}$$

18

Narrowing is compositional as defined by Definition 1. In addition, the narrowing operator $\leadsto_n$ is monotonic with respect to assert.

The pragmatics of narrowing

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

4.3 The Use of Detailing

Detailing means reducing the level of abstraction by decomposing one or more lifelines, i.e. by structural decomposition. As illustrated in Fig. 16, positive traces remain positive and negative traces remain negative in relation to detailing. The only change is that the traces of the refinement may include more details, for instance internal messages that are not visible in the more abstract specification.
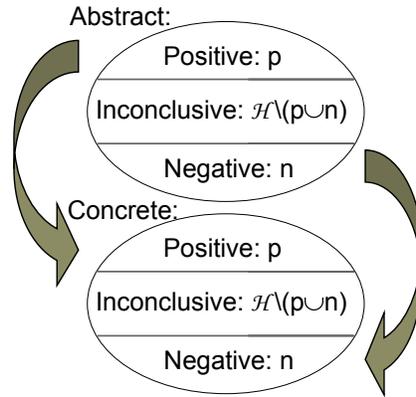

Fig. 16. Detailing of interaction obligations

Figure 17 is a detailing refinement of Payment in Fig. 11. In this case, the lifeline AppSystem is decomposed into the two lifelines Calendar, taking care of appointments, and Billing, handling payments. This decomposition has two effects with respect to the traces of the original specification. First of all, internal communication between Billing and Calendar is revealed (i.e. the messages needPay and paymentReceived), and secondly, Billing has replaced AppSystem as the sender/receiver of messages to and from the client. In general, some of the client's messages could also have been sent to/from Calendar.

We say that an interaction is a detailing refinement if we get the same positive and negative traces as in the original specification when both hiding the internal
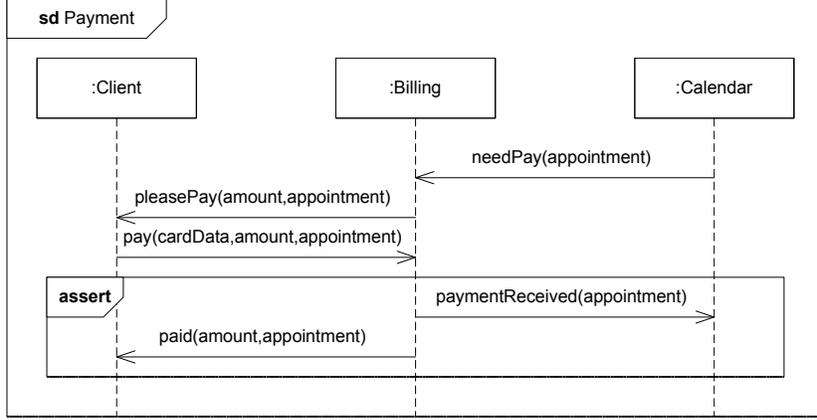
Fig. 17. Payment with decomposition

communication in the decomposition and allowing for a possible change in the sender/receiver of a message. Formally, the lifeline decomposition will in each case be described by a mapping L from concrete to abstract lifelines. For the above example, we get

$$L = ID[\textsf{Billing} \mapsto \textsf{AppSystem}][\textsf{Calendar} \mapsto \textsf{AppSystem}]$$

where ID is the identity mapping, and $L[\textsf{A} \mapsto \textsf{B}]$ is the mapping L updated so that $\textsf{A}$ maps to $\textsf{B}$.

Formally, we need to define a substitution function $subst(t, L)$, which substitutes lifelines in the trace t according to the mapping L. First, we define substitution on single events:

$$subst(e, L) \stackrel{\text{def}}{=} \begin{cases} k(s, L(tr), L(re)) & \text{if } e = k(s, tr, re), k \in \{!, ?\} \\ e & \text{otherwise} \end{cases} \qquad (18)$$

In general, a trace t may be seen as a function from indices to events. This trace function may be represented as a mapping where each element $i \mapsto e$ indicates that e is the i'th element in the trace, and we define the substitution function on traces by:

$$subst(t, L) \stackrel{\text{def}}{=} \{i \mapsto subst(t[i], L) \mid i \in [1 \ldots \#t]\} \qquad (19)$$

where $\#t$ and $t[i]$ denotes the length and the i'th element of the trace t, respectively.

We then define an abstraction function $abstr(t, L, E)$, which transforms a concrete trace into an abstract trace by removing all internal events (with respect to L) that are not present in the set of abstract events E:

$$abstr(t, L, E) \stackrel{\text{def}}{=} \{e \in \mathcal{E} \mid tr.e \neq re.e \vee e \in E\} \, \textcircled{S} \, (subst(t, L)) \qquad (20)$$

where $\mathcal{E}$ denotes the set of all events, $tr.e$ and $re.e$ denote the transmitter and the receiver of the event e, and $A \, \textcircled{S} \, t$ is the trace t with all events not in the set

A removed. We also overload *abstr* to trace sets in standard pointwise manner:

$$abstr(s, L, E) \overset{\text{def}}{=} \{abstr(t, L, E) \mid t \in s\} \tag{21}$$

Formally, detailing of interaction obligations is then defined by:

$$(p, n) \rightsquigarrow_c^{L,E} (p', n') \overset{\text{def}}{=} abstr(p, L, E) = abstr(p', L, E) \ \wedge \tag{22}$$
$$abstr(n, L, E) = abstr(n', L, E) \tag{23}$$

where L is a lifeline mapping as described above, and E is a set of abstract events.

Finally, detailing of interactions is defined by:

$$d \rightsquigarrow_c^{L,E} d' \overset{\text{def}}{=} \forall o \in [\![\, d \,]\!] : \exists o' \in [\![\, d' \,]\!] : o \rightsquigarrow_c^{L,E} o' \tag{24}$$

Detailing is compositional as defined by Definition 1. In addition, the detailing operator $\rightsquigarrow_c^{L,E}$ is monotonic with respect to assert.

The pragmatics of detailing
- Use detailing to increase the level of granularity of the specification by decomposing lifelines.
- When detailing, document the decomposition by creating a mapping L from the concrete to the abstract lifelines.
- When detailing, make sure that the refined traces are equal to the original ones when abstracting away internal communication and taking the lifeline mapping into account.

4.4 The Use of General Refinement

Supplementing, narrowing and detailing are all important refinement steps when developing interactions. Often, it is useful to combine two or three of these activites into a single refinement step. We therefore define a general refinement notion, of which supplementing, narrowing and detailing are all special cases. This general notion is illustrated for one interaction obligation in Fig. 18.

As an example of general refinement, MakeAppointment in Fig. 8 combines supplementing and narrowing in order to be a refinement of the interaction in Fig. 3. Adding an operand to the alt-construct is an example of supplementing, and is not covered by the definition of narrowing. On the other hand, adding guards is an example of narrowing, and is not covered by the definition of supplementing. For this to be a valid refinement step, we therefore need the general refinement notion, formally defined by:

$$d \rightsquigarrow_r^{L,E} d' \overset{\text{def}}{=} \forall o \in [\![\, d \,]\!] : \exists o' \in [\![\, d' \,]\!] : o \rightsquigarrow_r^{L,E} o' \tag{25}$$

where general refinement of interaction obligations is defined by:

$$(p, n) \rightsquigarrow_r^{L,E} (p', n') \overset{\text{def}}{=} abstr(n, L, E) \subseteq abstr(n', L, E) \ \wedge \tag{26}$$
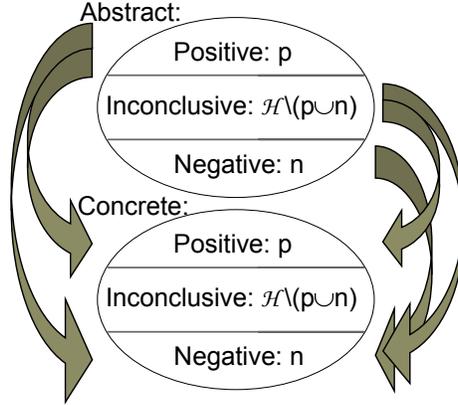$$abstr(p, L, E) \subseteq abstr(p', L, E) \cup abstr(n', L, E)$$

21

Fig. 18. General refinement of interaction obligations

Note that L may be the identity mapping, in which case the refinement does not include any lifeline decompositions (as in the case of MakeAppointment described above). Also, E may be the set of all events, $\mathcal{E}$, meaning that all events are considered when relating the traces of the refinement to the original traces. General refinement is compositional as defined by Definition 1.

Combining narrowing and supplementing may in general result in previously inconclusive traces being supplemented as positive, and the original positive traces made negative by narrowing. In order to specify that a trace *must* be present in the final implementation, and not removed by narrowing, we need to specify an obligation with this trace as the only positive, and all other traces as negative. The only legal refinement of this operand will then be redefining the trace as negative (by narrowing), leaving an empty set of positive traces and a specification that is not implementable.

The pragmatics of general refinement
– Use general refinement to perform a combination of supplementing, narrowing and detailing in a single step.
– To define that a particular trace *must* be present in an implementation use xalt and assert to characterize an obligation with this trace as the only positive one and all other traces as negative.

4.5 The Use of Limited Refinement

Limited refinement is a special case of general refinement, with less possibilities for adding new interactions obligations. By definition (25) of general refinement, new interaction obligations may be added freely, for instance in order to increase the nondeterminism required of an interaction. One example of this is CancelAppointment in Fig. 5, which is a refinement of the interaction given in Fig. 1. While the original specification only gave one interaction obligation with two positive traces, the refinement gives both this interaction obligation and also two new interaction obligations that are not refinements of the original one.

At some point during the development process, it is natural to limit the possibilities for creating new interaction obligations with fundamentally new traces. This is achieved by limited refinement, which has the additional requirement that each obligation of the refined interaction must have a corresponding obligation in the original interaction.

In STAIRS, stepwise development of interactions will be performed by first using general refinement to specify the main traces of the system, before switching to limited refinement which will then be used for the rest of the development process. Typically, but not necessarily, assert on the complete specification will be used at the same time as switching to limited refinement. This ensures that new traces may neither be added to the existing obligations, nor be added to the specification in the form of new interaction obligations. Note that using assert on the complete specification is not the same as restricting further refinements to be limited, as assert considers each interaction obligation separately.

Note also that limited refinement allows a refinement to have more interaction obligations than the original specification, as long as each obligation is a refinement of one of the original ones. One example is given in Fig. 19, which is a limited refinement of MakeAppointment in Fig. 15. In Fig. 19, alt has been replaced by xalt in order to specify that the client *must* be offered the choice of specifying a preferred date when asking for an appointment, while assert has been added to specify that there should be no other alternatives. In this particular case, we have not included the referenced interaction DecideAppTime in the scope of the assert-construct, as we want the possibility of supplementing more traces here. Transforming alt to xalt means in this example that each of the interaction obligations for Fig. 15 (there are two due to the xalt in DecideApp-Time) has two refining obligations in the semantics of Fig. 19. As all obligations in Fig. 19 have a corresponding obligation in Fig. 15, this is a valid instance of limited refinement.

Formally, limited refinement is defined by:

$$d \rightsquigarrow_l^{L,E} d' \overset{\text{def}}{=} d \rightsquigarrow_r^{L,E} d' \ \land \ \forall o' \in [\![\, d' \,]\!] : \exists o \in [\![\, d \,]\!] : o \rightsquigarrow_r^{L,E} o' \qquad (27)$$

Limited refinement is compositional as defined by Definition 1.

The pragmatics of limited refinement
– Use assert and switch to limited refinement in order to avoid fundamentally new traces being added to the specification.
– To specify globally negative traces, define these as negative in all operands of xalt, and switch to limited refinement.

---

5   Related Work

The basis of STAIRS is interactions and sequence diagrams as defined in UML 2.0 [OMG05]. Not all of the UML 2.0 operators are defined by STAIRS, but we believe that those covered are the most useful ones in practical system development. The STAIRS operator xalt is added to this set as UML 2.0 does not
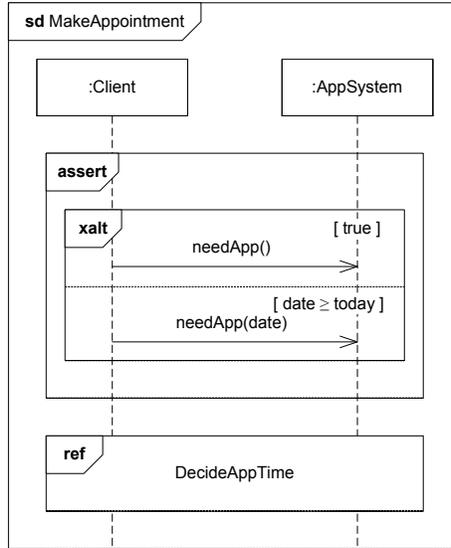
Fig. 19. MakeAppointment revisited

distinguish between alternatives that represent underspecification and alternatives that must all be present in an implementation, but uses the alt operator in both cases.

For guarded alt, we have in our semantics chosen not to follow UML 2.0 in that the empty trace is positive if no guard is true. Instead, we recommend to make all specifications with guarded alt so that the guards are exhaustive, ensuring that this will never be a problem in practice. The UML 2.0 standard [OMG05] is vague with respect to whether the traces with a false guard should be negative or not. As we have argued, classifying these as negative is fruitful as adding guards to a specification will then be a valid refinement step.

For defining negative behaviour, UML 2.0 uses the operators neg and assert. In [RHS05a], we investigated several possible formal definitions of neg, trying to capture how it was being used on the basis of experience. However, we concluded that one operator for negation is not sufficient, which is why STAIRS defines the two operators refuse and veto to be used instead of neg.

Decomposition in UML 2.0 is the same as detailing in STAIRS, but with a more involved syntax using the concepts of interaction use and gates. With the UML 2.0 syntax, the mapping from concrete to abstract lifelines is given explicitly in the diagram.

In [GS05], Grosu and Smolka give semantics for UML sequence diagrams in the form of two Büchi automata, one for the positive and one for the negative behaviours. Refinement then corresponds to language inclusion. Their refinement notion is compositional and covers supplementing and narrowing, but not detailing. All alternatives are interpreted as underspecification, and there is no means to capture inherent nondeterminism as with xalt in STAIRS.

24

In [CK04], the semantics of UML interactions are defined by the notions of positive and negative satisfaction. This approach is in many ways similar to STAIRS, but does not distinguish between underspecification and inherent nondeterminism. Their definition of the UML operator neg corresponds to the STAIRS operator veto, where the empty trace is taken as positive. [CK04] defines that for alternatives specified by alt, a trace is negative only if it is negative in both operands. Also, a trace is regarded as negative if a prefix of it is described as negative, while we in STAIRS define it as inconclusive as long as the complete trace is not described by the diagram.

Another variant of sequence diagrams is Message Sequence Charts (MSCs) [ITU99]. The important distinction between different kinds of alternatives is not made for MSCs either. As in our approach, a trace is negative if its guard is false in an MSC. Refinement of MSCs is considered by Krüger in [Krü00]. Narrowing in STAIRS corresponds closely to property refinement, while detailing corresponds to structural refinement. As there is no notion of inconclusive traces in [Krü00], refinement in the form of supplementing is not considered.

Live Sequence Charts (LSCs) [DH99,HM03] is an extension of MSCs, where charts, messages, locations and conditions are specified as either universal (mandatory) or existential (optional). An existential chart specifies a behaviour (one or more traces) that must be satisfied by at least one system run, while a universal chart is a specification that must be satisfied at all times. As a universal chart specifies all allowed traces, this is not the same as inherent nondeterminism in STAIRS, which only specifies some of the traces that must be present in an implementation. In contrast to STAIRS and UML 2.0, LSC synchronizes the lifelines at the beginning of each interaction fragment. This reduces the set of possible traces, and makes it easier to implement their operational semantics.

6   Conclusions and Future Work

In this paper we have focused on giving practical guidelines for the use of STAIRS in the development of interactions. For each concept, these guidelines have been summarized in paragraphs entitled "The Pragmatics of. . . ". We have focused on situations in which STAIRS extends or expands interactions as defined in UML 2.0 [OMG05]. This includes how to define negative behaviours and how to distinguish between alternatives that represent the same behaviour and alternatives that must all be present in an implementation. STAIRS is particularly concerned with refinement, and we have given guidelines on how to refine interactions by adding behaviours (supplementing), removing underspecification (narrowing) or by decomposition (detailing).

In [RHS05b], we gave a brief explanation of what it means for an implementation to be correct with respect to a STAIRS specification. We are currently working on extending this work, leading to "the pragmatics of implementations".

thank Iselin Engan for helpful comments on the final draft of this paper. We also thank the anonymous reviewers for constructive feedback.

References

[CK04]   Mara Victoria Cengarle and Alexander Knapp. UML 2.0 interactions: Semantics and refinement. In *Proc. 3rd Int. Wsh. Critical Systems Development with UML (CSDUML'04)*, Technical report TUM-I0415, pages 85–99. Institut für Informatik, Technische Universität München, 2004.

[DH99]   Werner Damm and David Harel. LSC's: Breathing life into message sequence charts. In *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, 1999.

[GS05]   Radu Grosu and Scott A. Smolka. Safety-liveness semantics for UML sequence diagrams. In *Proc. 5th Int. Conf. on Applications of Concurrency to System Design (ACSD'05)*, pages 6–14, 2005.

[HHRS05a] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 22(4):349–458, 2005.

[HHRS05b] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 1–25. Springer, 2005.

[HM03]   David Harel and Rami Marelly. *Come, Let's Play.: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[HS03]   Øystein Haugen and Ketil Stølen. STAIRS — Steps to analyze interactions with refinement semantics. In *Proc. International Conference on UML (UML'2003)*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003.

[ITU99]   International Telecommunication Union. *Recommendation Z.120 — Message Sequence Chart (MSC)*, 1999.

[Krü00]   Ingolf Heiko Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[Kru04]   Philippe Kruchten. *The Rational Unified Process*. Addison-Wesley, third edition, 2004.

[OMG05]   Object Management Group. *UML Superstructure Specification, v. 2.0*, document: formal/05-07-04 edition, 2005.

[RHS05a]   Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. How to transform UML neg into a useful construct. In *Norsk Informatikkonferanse NIK'2005*, pages 55–66. Tapir, 2005.

[RHS05b]   Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 12(2):157–188, 2005.

[RRS06]   Atle Refsdal, Ragnhild Kobro Runde, and Ketil Stølen. Underspecification, inherent nondeterminism and probability in sequence diagrams. In *Proc. 8th IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *LNCS*, pages 138–155. Springer, 2006.