Department of Informatics

How to transform UML neg into a useful construct

Ragnhild Kobro Runde, Øystein Haugen, Ketil Stølen

Research Report 326
ISBN 82-7368-280-3
ISSN 0806-3036

November 2005

UNIVERSITAS OSLOENSIS • MDCCCXI •

How to transform UML neg into a useful construct

Ragnhild Kobro Runde[1] Øystein Haugen[1] Ketil Stølen[1,2]
[1]Department of Informatics, University of Oslo [2]SINTEF ICT
{ragnhilk,oysteinh}@ifi.uio.no, ketil.stolen@sintef.no

Abstract

In UML, the operator neg is used to specify negative, or unwanted, system behaviour. We agree that being able to specify negative behaviour is important. However, the UML neg is currently not well-suited for this purpose, the main problem being that a single operator is used with several different meanings depending on the context. In this paper we investigate some alternative definitions of neg. We also propose a solution in which neg is replaced by two new operators for specifying negative behaviour.

1 Introduction

In the interactions of UML 2.0 [OMG04], the unary operator neg is used to describe negative or invalid behaviour, i.e. scenarios that must not occur. By using structuring mechanisms like neg and the other interaction operators of UML 2.0, it is possible to "describe a number of traces in a compact and concise manner" [OMG04].

Interactions are typically used only to describe possible executions of the system. This means that nothing can be deduced about the validity of system behaviour *not* described by the interaction. In such a setting, it is particularly important to be able to specify negative as well as positive behaviours. Hence, we say that an interaction describes a set of positive behaviours, and a set of negative behaviours. Behaviours that are neither categorized as positive nor as negative are called inconclusive [HS03].

The problem with the UML neg operator is that people tend to interpret it differently depending on the context in which the operator appears. This makes it difficult to define a precise semantics for neg. The advantages of having a precise semantics include the following:

- it gives a clear answer in cases where intuition is weak.
- it facilitates formal reasoning.
- it forms a basis for tool-vendors and methodology builders.

2 Background

To set the stage for detailed discussion, in this section we give a brief introduction to UML 2.0 interactions and their semantics as defined in STAIRS [HHRS05a, HHRS05b].

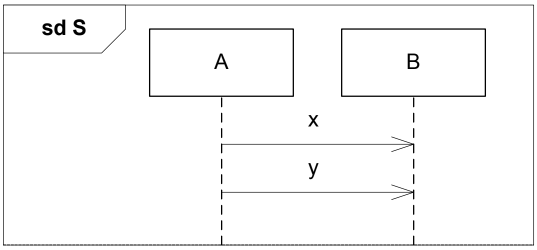


Figure 1: A simple interaction

Interactions and trace semantics

We define the semantics of interactions by sets of traces. A trace is a sequence of events, used to represent some system run. We distinguish between two kinds of events, the sending and reception of a message m, denoted $!m$ and $?m$, respectively.

A simple example of an interaction in the form of a sequence diagram is given in Figure 1. This diagram specifies that A sends the messages x and y to B. A and B are called lifelines, each representing an object or a component in the system or its environment. The traces described by a diagram like this, are all possible sequences of events in the diagram such that both the send event is ordered before the corresponding receive event, and events on the same lifeline are ordered from the top and downwards. For the example in Figure 1, it follows that the first event to happen must be the sending of x. After that, either B may receive x or A may send y. The diagram thus specifies the two traces $\langle !x, ?x, !y, ?y \rangle$ and $\langle !x, !y, ?x, ?y \rangle$.

Formally, we define the semantics of interaction diagrams by a function $[\![\ ]\!]$ that for any diagram d yields a pair (p, n) of trace-sets. The traces in p are called the positive traces of d, representing traces that may be the result of running the final system. The traces in n are called the negative traces of d, and must not appear in the final implementation. Traces that are neither defined as positive nor as negative are called *inconclusive*. We let $\mathcal{H}$ denote the set of all traces where for each message, the send event is ordered before the corresponding receive event.

To illustrate the semantics of an interaction, we use a circle that is divided into three regions as shown in Figure 2. The circle as a whole corresponds to $\mathcal{H}$, the set of all possible traces. The topmost region represents the positive trace-set p, the lowest region represents the negative trace-set n, while the middle region contains the inconclusive traces.

Definition 1 *The semantics of a diagram consisting of a single event e, is given by:*

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

Weak sequencing (seq) is the implicit composition mechanism construct used to create diagrams like the one in Figure 1, one of its possible textual representations being $(!x \ \mathsf{seq}\ ?x)\ \mathsf{seq}\ (!y\ \mathsf{seq}\ ?y)$.

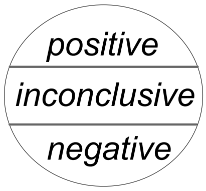


Figure 2: Illustrating the traces of an interaction

Definition 2 *Assume* $[\![\ d_1\ ]\!] = (p_1, n_1)$ *and* $[\![\ d_2\ ]\!] = (p_2, n_2)$*. We then define the semantics of* seq *by:*

$$[\![\ d_1 \ \mathsf{seq}\ d_2\ ]\!] \stackrel{\mathsf{def}}{=} (p_1 \succsim p_2, (n_1 \succsim p_2) \cup (n_1 \succsim n_2) \cup (p_1 \succsim n_2))$$

where weak sequencing of trace-sets, $\succsim$*, is defined as:*

$$s_1 \succsim s_2 \stackrel{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in \mathcal{L} : h \restriction l = h_1 \restriction l \frown h_2 \restriction l\}$$

where $\mathcal{L}$ *is the set of all lifelines,* $\frown$ *is the concatenation operator on sequences, and* $h \restriction l$ *is the trace* h *with all events not taking place on the lifeline* l *removed.*

More complex interactions are constructed through the application of various operators. In this paper we only need one more operator, namely alt for specifying alternative behaviour. Definitions of other central operators may be found in e.g. [HHRS05a].

Definition 3 *Assume* $[\![\ d_1\ ]\!] = (p_1, n_1)$ *and* $[\![\ d_2\ ]\!] = (p_2, n_2)$*. We then define the semantics of* alt *by:*

$$[\![\ d_1 \ \mathsf{alt}\ d_2\ ]\!] \stackrel{\mathsf{def}}{=} (p_1 \cup p_2, n_1 \cup n_2)$$

Refinement

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.

Definition 4 *Assume* $[\![\ d\ ]\!] = (p, n)$ *and* $[\![\ d'\ ]\!] = (p', n')$*. Refinement,* $\rightsquigarrow$*, is defined by:*

$$d \rightsquigarrow d' \stackrel{\mathsf{def}}{=} n \subseteq n' \ \wedge\ p \subseteq p' \cup n'$$

If $d \rightsquigarrow d'$, we say that d is refined by d', or that d' is a refinement of d.

More details with motivation and examples using this definition may be found in [HHRS05a] and [HHRS05b]. In [HHRS05c] we have proved that refinement as defined above is transitive, meaning that the result of several successive refinement steps will be a valid refinement of the original specification.

3 Alternative definitions of neg

Obviously, the intuition behind $\mathsf{neg}\ d$ is that the positive traces described by d should be taken as negative. But as a definition, it is not complete, because it fails to answer questions such as:

1. Does $\mathsf{neg}\ \mathsf{neg}\ d$ mean the same as d?
2. What happens to the negative traces of d?
3. What should be the positive traces of $\mathsf{neg}\ d$?

In this section we focus on the last question, by investigating the following possible answers:

a. The negative traces of d.

b. All traces except the negative traces of $\mathsf{neg}\ d$.

c. No trace at all.

d. The empty trace.

All of these are feasible answers, and to some extent they are already used in practice. We will give formal definitions for each of the given alternatives, and discuss them from both a practical and a formal point of view. During our treatment of alternative a, we will also answer questions 1 and 2 above.

A basic premise in the following discussion is that we want compositionality (in formal theory often called monotonicity), meaning that by refining the different interaction operands separately, we obtain a refinement of the complete interaction. In the context of neg this means that for a (sub-)specification $\mathsf{neg}\ d$, if d' is a refinement of the operand d, then $\mathsf{neg}\ d'$ should be a refinement of $\mathsf{neg}\ d$.

Alternative a: The positive traces of $\mathsf{neg}\ d$ are the negative traces of d

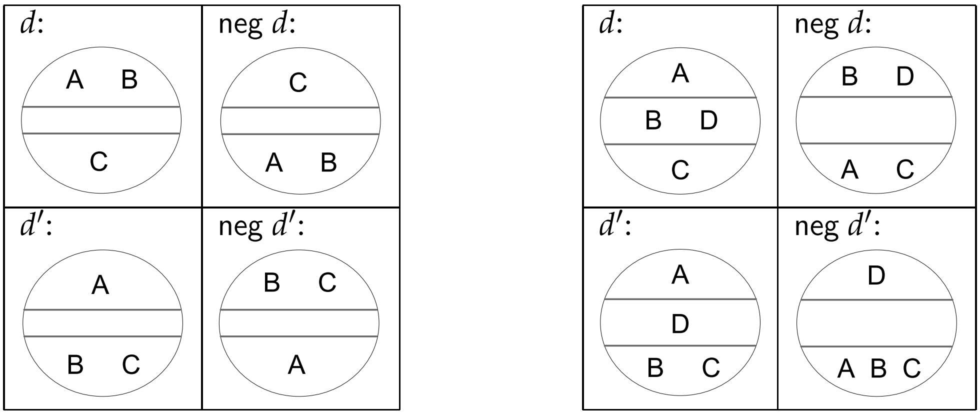
Intuitively, given the interpretation of negation in classical logic, it is natural to view $\mathsf{neg}\ d$ as the opposite of d, taking the negative traces of d as the positive traces of $\mathsf{neg}\ d$.

Definition 5 *Assume $[\![\ d\ ]\!] = (p, n)$. Then the semantics of* neg *may be defined by:*

$$[\![\ \mathsf{neg}\ d\ ]\!] \overset{\mathsf{def}}{=} (n, p)$$

With this definition we get $\mathsf{neg}\ \mathsf{neg}\ d = d$, as is the case of negation in classical logic. However, it is not obvious that logical negation is a good parallel to negation in the context of interactions. In fact, the following example gives a formal argument for why this is not a good interpretation of neg.

Consider the four specifications in Figure 3(a), where we only have the three traces A, B and C. The specification d' is a valid refinement of d, $d \rightsquigarrow d'$, where the originally positive trace B has been redefined as negative in d'. The figure also illustrates the negated specifications $\mathsf{neg}\ d$ and $\mathsf{neg}\ d'$. Since we have $d \rightsquigarrow d'$, by compositionality we should also have $\mathsf{neg}\ d \rightsquigarrow \mathsf{neg}\ d'$. However, this is not



(a) Using definition 5

(b) Using definition 6

Figure 3: Example specifications

the case as the trace B has been moved from negative to positive, violating the refinement requirement $n \subseteq n'$.

To sum up, if we want compositionality with respect to neg (and we do), then definition 5 is not a good definition for neg. Also, we may conclude that the answer to question 1 above is that neg neg d should *not* be the same as d.

Negative traces remain negative

As indicated by the above example, the negative traces in d cannot be defined as positive for neg d. By a similar argument, it may be demonstrated that they cannot be defined as inconclusive either. The only remaining alternative is to include the negative traces of d in the negative trace-set of neg d. Hence, the negative traces of neg d must be the positive and negative traces of d, combined.

The answer to question 2 is then that the negative traces of d remain negative for neg d. Hence, for the rest of this paper we only consider definitions where this is indeed the case. We now return to the discussion of the different alternatives for the positive traces of neg d.

Alternative b: The positive traces of neg d are all traces except the negative traces of neg d

Another possibility is to let the positive traces of neg d be all traces except the negative traces of neg d (which, as we have argued, should consist of the positive and negative traces of d). The intuition here is that by using neg, the specification focuses on what is not allowed, and therefore everything else should be considered legal (i.e. positive).

Definition 6 *Assume* $[\![\text{ d }]\!] = (p, n)$*. Then the semantics of* neg *may be defined by:*

$$[\![\text{ neg } d \;]\!] \overset{\text{def}}{=} (\mathcal{H} \setminus (p \cup n), p \cup n)$$

Figure 3(b) gives an example using this definition, with $\mathcal{H}$ (the universe of possible traces) being the set $\{A, B, C, D\}$. In the example we have both $d \rightsquigarrow d'$ (by categorizing the trace B as negative) and neg $d \rightsquigarrow$ neg d' (by redefining the

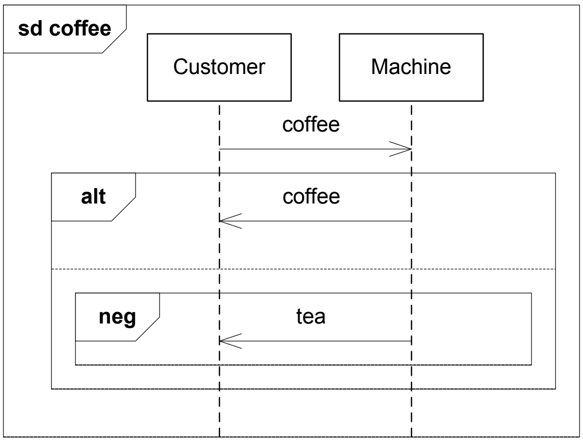


Figure 4: Ordering coffee

trace B as negative) as desired. Compositionality is not simply a feature of this particular example, in Appendix A we prove that it is in fact a general property of definition 6.

As the example demonstrates, all inconclusive traces for d are by definition made positive for neg d. This makes it impossible to write a specification defining some positive and negative traces, while wanting the rest to be inconclusive (as they will be considered positive by definition). Hence, using neg with this definition results in a less powerful specification language.

Alternative c: neg d has *no* positive traces

The two previous sections have discussed the two alternatives where the positive traces of neg d should be the negative or inconclusive traces of d, concluding that none of these is an ideal solution. A third alternative, then, is to say that neg d has no positive traces at all. The operator neg should be used to specify negative behaviour, and nothing else.

Definition 7 *Assume* $[\![d]\!] = (p, n)$. *Then the semantics of* neg *may be defined by:*

$$[\![\mathsf{neg}\, d]\!] \overset{\mathsf{def}}{=} (\emptyset, p \cup n)$$

To understand the effect of this definition, it is useful to look at example interactions using neg in combination with other operators such as seq and alt. For the examples, we will use a simple vending machine selling tea and coffee.

The interaction in Figure 4 specifies two traces, both starting with the customer ordering coffee. The first alt-operand specifies that the machine giving coffee to the customer is a positive trace. This operand defines no negative traces. The second alt-operand specifies no positive traces, but states that giving tea is negative. Using the definition of seq to combine the first message of ordering coffee with the alt-fragment, we get that ordering and getting coffee is positive, while ordering coffee and getting tea is negative.

The interaction in Figure 5 is meant to specify that if the customer orders tea with no sugar, then the machine should *not* add sugar before giving the tea to the customer. However, with the given definitions of neg and seq, the actual meaning

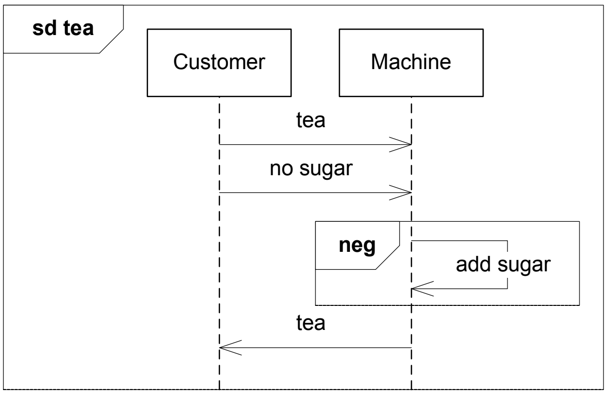


Figure 5: Ordering tea

of this interaction is not what intuition believes it to be. The neg-fragment in itself defines adding sugar as negative. Together with the remaining traces of the interaction, we then get that the trace of ordering tea, ordering no sugar, the machine adding sugar and then giving tea is a negative trace. This is in accordance with the intuition.

But, using the definitions of seq and neg, we get that the interaction in Figure 5 has *no* positive traces! This results from fact that the neg-fragment has no positive traces, and that a set of traces sequenced with the empty set gives the empty set. This is not in accordance with the intuition that the interaction should have the same positive traces as the same interaction only with the neg-fragment simply removed.

A possible solution to this problem could be to change the definition of weak sequencing of trace-sets to include special cases for the empty set.

Definition 8 *Weak sequencing of trace-sets may instead be defined as:*

$$s_1 \succsim s_2 \overset{\mathsf{def}}{=} \begin{cases} s_2 & \textit{if } s_1 = \emptyset \\ s_1 & \textit{if } s_2 = \emptyset \\ \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in \mathcal{L} : h \restriction l = h_1 \restriction l \frown h_2 \restriction l\} & \textit{otherwise} \end{cases}$$

However, it turns out that this definition does not give compositionality for seq. With seq being the basic composition operator, it is essential that it is compositional in order to let the different parts of a basic interaction be refined separately. Hence, we have kept definition 2, for which we have proved compositionality in [HHRS05c].

Alternative d: The only positive trace for neg d is the empty trace

Another alternative, motivated by the above discussion on Figure 5, is to let the empty trace be positive for neg d.

Definition 9 *Assume* $[\![\, d \,]\!] = (p, n)$. *Then the semantics of* neg *may be defined by:*

$$[\![\, \mathsf{neg}\ d \,]\!] \overset{\mathsf{def}}{=} (\{\langle\rangle\}, p \cup n)$$

As opposed to the empty trace-set $\emptyset$ having no behaviours at all, the empty trace $\langle\rangle$ represents the possible behaviour of doing nothing. In the context of Figure 5, this is a highly relevant distinction.

Using definition 9, we get the same negative trace(s) as we did with definition 7. The difference is that Figure 5 with definition 9 also specifies the intended result that ordering tea, ordering no sugar and then receiving tea (without sugar) is positive. This is because a trace sequenced with the empty trace equals the original trace, meaning that we may simply omit the negative fragments from an interaction when calculating its positive traces.

From this example, it is tempting to believe that definition 9 is a better definition of neg than definition 7. However, returning to the example in Figure 4, we will demonstrate that this is not the case. With our new definition of neg, the second alt-operand in Figure 4 gives that the empty trace $\langle\rangle$ is positive. Combining this with the first message in the diagram, we get that ordering coffee but not getting anything is a positive trace. This result is however not intended when using neg in this context.

Hence, neither definition 7 nor definition 9 are perfect definitions for neg. In the next section we propose a solution that is based on multiple operators for negation.

Suggested solution

As we have demonstrated, it is not possible to give one single context-free definition of neg capturing all its intuitive meanings. We do not regard introducing context-sensitive definitions to be a good solution, as they tend to be very complicated and difficult to use. Instead, we strongly believe that most of the problems come from having only one operator for specifying negative behaviour. Hence, our solution is to define two different operators for negation, capturing the most important uses of the neg operator.

First we introduce skip, the empty diagram corresponding to a program doing nothing.

Definition 10 *The semantics of* skip*, the empty diagram, is defined as*

$$[\![\ \mathsf{skip}\ ]\!] \stackrel{\mathsf{def}}{=} (\{\langle\rangle\}, \emptyset)$$

Of the alternative definitions given for neg, we believe that definition 7 is the most fundamental one. We therefore define a new basic operator refuse using this definition.

Definition 11 *Assume* $[\![\ d\ ]\!] = (p, n)$*. We then define the semantics of* refuse *by:*

$$[\![\ \mathsf{refuse}\ d\ ]\!] \stackrel{\mathsf{def}}{=} (\emptyset, p \cup n)$$

The operator refuse may be used to define other operators for specifying negative behaviour, and is also meant to be used when specifying that one of the operands of an alt-fragment represents negative behaviour (as in Figure 4).

We may now define another operator veto as a high-level operator.

Definition 12 *Assume* $[\![d]\!] = (p, n)$*. We then define the semantics of* veto *by:*

$$\mathsf{veto}\ d \overset{\mathsf{def}}{=} \mathsf{skip\ alt}\ (\mathsf{refuse}\ d)$$

This definition is equal to the alternative definition 9 above, and is meant to be used when specifying additional messages that make otherwise positive behaviours negative (as in Figure 5).

If desirable, it is also possible to define another operator corresponding to definition 6, in order to specify that everything apart from the given traces should be positive. Additional negation operators may also be defined. However, this should be done with great care, because even though more operators may result in a more expressive language, it might lead to reduced readability.

A major challenge is to find unambiguous and intuitive names for the different negation operators proposed here. We do not claim to have succeeded in this task. Our main contribution is that it is necessary to have more than one operator for specifying negative behaviour, and that some of these might be high-level operators derived from a well-chosen set of basic operators.

4 Related work

Störrle [Stö03] discusses three alternative definitions for neg. His first alternative, referred to as "not the traces of d", defines that neg d has no positive traces, and that the negative traces of neg d are the positive traces of d. Since the negative traces of d are lost with this definition, this alternative is rejected. His second alternative, "anything but d", is similar to our alternative definition 6, except that the negative traces of d are considered positive for neg d. The third alternative he proposes, "flip valid and invalid", equals our definition 5. Of the three alternatives, Störrle chooses this third interpretation, calling it "the only consistent approach", as it gives neg neg $d = d$. But, as we have argued in this paper, logical negation is not the most obvious interpretation of negation in the context of interactions.

In [CK04], Cengarle and Knapp define the semantics of UML 2.0 interactions by the notions of positive and negative satisfaction. Based on a similar argument as the one we presented in favour of definition 9, they regard the empty trace as positive for neg d. However, the negative traces of d are inconclusive for neg d (as in the first of Störrle's definitions). For alternatives, specified by alt, they define that a trace is negative only if it is negative in both operands. This means that with their semantics, the interaction in Figure 4 has *no* negative traces even though it uses the operator neg.

Cengarle and Knapp pose an interesting question related to the use of negation in specifications: Should a trace be negative if a prefix of it is specified as negative? The answer in [CK04] is principally yes, proposing an even stronger approach where a trace is taken as negative as soon as it has completed a negative sub-diagram. An advantage of this is that it allows for earlier identification (or even prevention) of negative traces.

As an example of this approach, consider again the specification in Figure 4 (interpreting neg as refuse). The specification states that ordering coffee and then receiving tea is a negative trace. But what about the trace which then continues with the customer receiving coffee as well — should this be positive or negative?

In STAIRS 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.

The alternative, as proposed in [CK04], would be to say that the trace of ordering coffee, first receiving tea and then receiving coffee is negative. In general, once a negative (sub-)scenario has occurred, the total trace will be negative independently of what happens next. Formally, this may be achieved by the following definitions:

Definition 13 *Assume* $[\![\ d_1\ ]\!] = (p_1, n_1)$ *and* $[\![\ d_2\ ]\!] = (p_2, n_2)$*. Then the semantics of* refuse *may be defined by:*

$$[\![\ \mathsf{refuse}\ d_1\ ]\!] \overset{\mathsf{def}}{=} (\emptyset, (p_1 \cup n_1) \succsim \mathcal{H})$$

The semantics of seq *may be defined by:*

$$[\![\ d_1\ \mathsf{seq}\ d_2\ ]\!] \overset{\mathsf{def}}{=} (p_1 \succsim p_2, n_1 \cup (p_1 \succsim n_2))$$

In appendix A, we prove compositionality with respect to these alternative definitions.

These definitions are somewhat different from those in [CK04]. Translated to our formalism, they append $\mathcal{H}$ only to the negative traces of the first operand of seq, and not to the negative traces of the second operand or to the negative traces of neg. We believe that our definition corresponds more closely to the intuition behind this alternative approach. In contrast to the definitions in [CK04], definition 13 also ensures that skip is an identity element for seq, i.e. $[\![\ d\ \mathsf{seq}\ \mathsf{skip}\ ]\!] = [\![\ d\ ]\!]$.

Even without considering the possible formal definitions, there is at least one large disadvantage of this approach. If a trace is negative as soon as it has traversed a negative region, it follows that for a trace to be positive, all of its prefixes would have to be positive or at least inconclusive. In our vending machine example, this means that it would not be possible to make a consistent specification where ordering and getting coffee is positive, while ordering coffee and not getting anything (i.e. nothing more happens) is negative.

5 Conclusions

We have discussed alternative definitions of neg, considering both formal and practical issues. None of the proposed alternatives are able to capture all the different uses of the neg operator. As a result, we find it necessary to have more than one operator for specifying negative behaviour. In this paper we have proposed to replace neg with two new operators, for which the given definitions ensures compositionality.

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 the Research Council of Norway. We thank the other SARDAS members for fruitful discussions, and in particular Mass Soldal Lund for coming up with the first vending machine example. We thank Iselin Engan for helpful comments on the final draft of this paper.

References

[CK04] María 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.

[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*, Online First:1–13, 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.

[HHRS05c] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. Technical Report 309, Department of Informatics, University of Oslo, 2005.

[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.

[OMG04] Object Management Group. *UML 2.0 Superstructure Specification*, document: ptc/04-10-02 edition, 2004.

[Stö03] Harald Störrle. Assert, negate and refinement in UML-2 interactions. In *Proc. Wsh. Critical Systems Development with UML (CSDUML'03)*, Technical report TUM-I0317, pages 79–93. Institut für Informatik, Technische Universität München, 2003.

A Proofs

For all proofs in this section, we use $[\![\ d\ ]\!] = (p, n)$, $[\![\ d'\ ]\!] = (p', n')$, $[\![\ d_1\ ]\!] = (p_1, n_1)$, $[\![\ d'_1\ ]\!] = (p'_1, n'_1)$, $[\![\ d_2\ ]\!] = (p_2, n_2)$ and $[\![\ d'_2\ ]\!] = (p'_2, n'_2)$. We also use $(s_1 \cup s_2) \precsim s_3 = (s_1 \precsim s_3) \cup (s_2 \precsim s_3)$, as proved in [HHRS05c]. Also, recall that a definition of neg is compositional if for any two specifications d and d' such that $d \rightsquigarrow d'$, we also have $\mathsf{neg}\ d \rightsquigarrow \mathsf{neg}\ d'$. Similarly for compositionality of other operators such as seq and refuse.

Theorem 1 *Definition 6 of* neg *is compositional with respect to refinement as defined in definition 4.*

Proof: Using definition 6, we get $[\![\ \mathsf{neg}\ d\ ]\!] = (\mathcal{H} \setminus (p \cup n), p \cup n)$ and $[\![\ \mathsf{neg}\ d'\ ]\!] = (\mathcal{H} \setminus (p' \cup n'), p' \cup n')$. By definition 4, we need to prove $p \cup n \subseteq p' \cup n'$ and $\mathcal{H} \setminus (p \cup n) \subseteq (\mathcal{H} \setminus (p' \cup n')) \cup (p' \cup n')$. The last condition is trivial, as the right side equals $\mathcal{H}$. As d is refined by d' (by assumption), we have $n \subseteq n'$ and $p \subseteq p' \cup n'$, giving $p \cup n \subseteq p' \cup n'$ as required.

Theorem 2 *Definition 7 of* neg *is compositional with respect to refinement as defined in definition 4.*

Proof: Using definition 7, we get $[\![$ neg d $]\!] = (\emptyset, p \cup n)$ and $[\![$ neg d' $]\!] = (\emptyset, p' \cup n')$. By definition 4, we need to prove $p \cup n \subseteq p' \cup n'$ and $\emptyset \subseteq \emptyset \cup (p \cup n)$. The last condition is trivial. As d is refined by d', we have $n \subseteq n'$ and $p \subseteq p' \cup n'$, giving $p \cup n \subseteq p' \cup n'$ as required.

Theorem 3 *Definition 9 of* neg *is compositional with respect to refinement as defined in definition 4.*

Proof: Similar to the proof of Theorem 2, replacing every occurence of $\emptyset$ with $\{\langle\rangle\}$.

Theorem 4 *Definition 13 of* refuse *and* seq *is compositional with respect to refinement as defined in definition 4.*

Proof for refuse**:** Using definition 13, we get $[\![$ refuse d $]\!] = (\emptyset, (p \cup n) \succsim \mathcal{H})$ and $[\![$ refuse d' $]\!] = (\emptyset, (p' \cup n') \succsim \mathcal{H})$. By definition 4, we need to prove $(p \cup n) \succsim \mathcal{H} \subseteq (p' \cup n') \succsim \mathcal{H}$ and $\emptyset \subseteq \emptyset \cup ((p' \cup n') \succsim \mathcal{H})$. The last condition is trivial. As d is refined by d', we have $n \subseteq n'$ and $p \subseteq p' \cup n'$, which together with definition 2 of $\succsim$ gives $n \succsim \mathcal{H} \ \subseteq n' \succsim \mathcal{H}$ and $p \succsim \mathcal{H} \subseteq (p' \cup n') \succsim \mathcal{H}$, i.e. $(p \succsim \mathcal{H}) \cup (n \succsim \mathcal{H}) \subseteq (p' \succsim \mathcal{H}) \cup (n' \succsim \mathcal{H})$ as required.

Proof for seq**:** Using definition 13, we get $[\![$ d_1 seq d_2 $]\!] = (p_1 \succsim p_2, n_1 \cup (p_1 \succsim n_2))$ and $[\![$ d_1' seq d_2' $]\!] = (p_1' \succsim p_2', n_1' \cup (p_1' \succsim n_2'))$ By definition 4, we need to prove $n_1 \cup (p_1 \succsim n_2) \subseteq n_1' \cup (p_1' \succsim n_2')$ and $p_1 \succsim p_2 \subseteq (p_1' \succsim p_2') \cup (n_1' \cup (p_1' \succsim n_2'))$. As d_1 is refined by d_1' and d_2 is refined by d_2', we have $n_1 \subseteq n_1'$, $p_1 \subseteq p_1' \cup n_1'$ and $n_2 \subseteq n_2'$, which together with definition 2 of $\succsim$ gives $n_1 \cup (p_1 \succsim n_2) \subseteq n_1' \cup ((p_1' \cup n_1') \succsim n_2') = n_1' \cup (p_1' \succsim n_2') \cup (n_1' \succsim n_2')$. By lemma 1, this is a subset of $n_1' \cup (p_1' \succsim n_2')$ which was to be proved. Similarly, we get $p_1 \succsim p_2 \subseteq (p_1' \cup n_1') \succsim (p_2' \cup n_2') = (p_1' \succsim p_2') \cup (p_1' \succsim n_2') \cup (n_1' \succsim (p_2' \cup n_2'))$. By lemma 1, this is a subset of $(p_1' \succsim p_2') \cup (p_1' \succsim n_2') \cup n_1'$ which was to be proved.

Lemma 1 *Given an arbitrary interaction d with negative trace-set n, i.e. $[\![\, d \,]\!] = (p, n)$, we have $n \succsim s \subseteq n$ for any trace-set $s \subseteq \mathcal{H}$.*

Proof: By induction over the structure of d.
Base case: d is a single event e, i.e. $n = \emptyset$ by definition 1. As $\emptyset \succsim s = \emptyset$ by definition 2, we get $\emptyset \succsim s \subseteq \emptyset$ as required.
Case 1: $d =$ neg d_1, i.e. $n = (p_1 \cup n_1) \succsim \mathcal{H}$ by definition 13. As $\mathcal{H} \succsim s \subseteq \mathcal{H}$ for arbitrary s, we get $(p_1 \cup n_1) \succsim \mathcal{H} \succsim s \subseteq (p_1 \cup n_1) \succsim \mathcal{H}$ as required.
Case 2: $d = d_1$ alt d_2, i.e. $n = n_1 \cup n_2$ by definition 3. By induction hypothesis, we have $n_1 \succsim s \subseteq n_1$ and $n_2 \succsim s \subseteq n_2$, giving $(n_1 \cup n_2) \succsim s = (n_1 \succsim s) \cup (n_2 \succsim s) \subseteq n_1 \cup n_2$ as required.
Case 3: $d = d_1$ seq d_2, i.e. $n = n_1 \cup (p_1 \succsim n_2)$ by definition 13. By induction hypothesis, we have $n_1 \succsim s \subseteq n_1$ and $n_2 \succsim s \subseteq n_2$, giving $((n_1 \cup (p_1 \succsim n_2)) \succsim s = (n_1 \succsim s) \cup (p_1 \succsim n_2 \succsim s) \subseteq n_1 \cup (p_1 \succsim n_2)$ as required.