Abstract. The Unified Modeling Language (UML) is the industry standard for modeling. With its recent advancement to version 2.0, there have been large amounts of changes and additions. In this paper I study some new features with a view to formal specification and verification, in particular the operators \texttt{neg} and \texttt{assert}, and notions of refinement based upon them.

1 Introduction

The Unified Modeling Language (UML) is the industry standard for modeling, it has even been dubbed “the lingua franca of software engineering” (cf. [19, p. v]). However, over the past few years, a number of rather serious shortcomings have been identified, for instance with respect to the formal semantics of UML models, most notably of the dynamic models.

Recently, however, the version 2.0 of UML has been adopted (see [15]), addressing a number of these shortcomings. In particular, Message Sequence Charts (MSC) according to the ISO standard (see [7, 6]) have been integrated. In UML, the concept underlying these notations is called \textit{interaction}.

In a companion paper and technical report (see [23, 22]), I have defined a formal semantics for most of the operators in UML interactions (including time), except those without a straightforward semantics, namely \texttt{neg} and \texttt{assert}. These are also not present in classical MSCs but have precursors in Life Sequence Charts (LSCs, see [3]). The latter, however, have been extensively used in the specification and verification of critical systems, e.g., in the automotive domain. So, the new features of UML might well be exploited for the same purpose.

The semantics of most of UML 2.0-interactions is more or less straightforward. So, I focus on the interesting parts. I discuss a number of alternative interpretations with their advantages and disadvantages with respect to notions of refinement, which would be a natural starting point both for development and verification tasks. I also generalize the interpretations to timed interactions.
2 Interactions in UML 2.0

To level the ground, I start with a brief discussion of the concrete and abstract syntax of interactions in UML 2.0. Here, I shall refer to the UML 2.0 as the “new standard” or simply “the standard” while I refer to the version 1.4 as the “old standard”.

2.1 Concrete Syntax

First of all, all diagrams now have a frame around them and a compartment displaying its type and name (see Figure 1) which makes it easier to refer to it, e.g. as a subdiagram or companion diagram.

In the old standard, there were two types of interaction diagrams, namely sequence and collaboration diagrams which both are based on the same metamodel concepts (see below). So called “metric sequence diagrams” [16] had been mentioned in UML 1.3, but neither defined nor explained, and have been abandoned in UML 1.4.

In the new standard, collaboration diagrams have been renamed to communication diagrams. A new kind of interaction diagram, timing diagrams as known in many engineering disciplines (see Figure 1) have been introduced. Timing diagrams may be considered as an elaboration of metric sequence diagrams. While communication diagrams and sequence diagrams focus on structure and message exchange, respectively, timing diagrams focus on state and state change across time. Sequence diagrams have been extended considerably, and now have approximately the same expressive power as high-level MSCs.

All these interaction diagrams may be combined ad lib by a given set of InteractionOperators. The notation is similar to that of interaction diagrams in general (see Figure 1). If there are two arguments to an InteractionOperator, they are divided by a dashed line. The notation is strongly reminiscent of MSCs. Also, in the new standard there are now interaction overview diagrams which are basically activity diagrams where the interaction diagrams are activities (see [22]). They correspond to High-Level MSCs (see [7]).

Furthermore, there are also so called overview diagrams which may be used to combine interaction diagrams into a kind of activity diagram, where the places of activity states are taken by interactions.

2.2 Basic semantics

In the new standard, “an EventOccurrence is the basic semantic unit of Interactions” (cf. [15, p. 416]), and the “sequences of EventOccurrences are the meanings of Interactions” (ibid.). Given the domain of Event Occurrences (written \(EO\)), the domain of Traces (written \(SEQ\)) is \(SEQ = EO\).

More precisely, however, “the semantics of an Interaction is given as a pair of sets of traces” (cf. [15, p. 419, emphasis added]), representing “valid traces and invalid traces” (ibid.), respectively. Thus, the semantic domain for interactions is \(SEQ \times SEQ\). “The traces that are not included [in the union of the two]
Fig. 1. A sample UML 2.0 interaction diagram, including the high-level operators `alt` and `opt`. The language of messages described by this sequence diagram may also be represented by the regular expression `a(b[c]|d)e`.

*are not described [...] and we cannot know whether they are valid or invalid*” (ibid.). That is, an interaction in UML 2.0 implicitly describes contingency, see Figure 2. It is not obvious, what valid and invalid really mean, however. For the time being, I shall interpret them as necessary vs. forbidden or must vs. must not (see Section 4.2 below).

![Fig. 2. In UML 2, interactions specify both valid and invalid traces. Unspecified traces are contingent. Probably, valid and invalid traces should be interpreted as necessary and forbidden (or must and must not).](image)

This point of view is obviously adopted from Life Sequence Charts (cf. [3]). With this definition, the mapping from the metamodel to a mathematical domain is now simply

\[
\text{(interaction)} \quad [\text{sd}(P)] = ([P], 0)
\]

for an interaction `P`, with `⟦ \_ \rceil` to denote a (denotational) semantic function. In order to distinguish between alternative interpretations of constructs, semantic brackets will be subscripted as in `⟦ \_ \rceil_{\text{interpretation}}`.

As a convention, I write the first and second component of a pair `X` as `X^+` and `X^−`, respectively. Thus, the valid and invalid traces that are the semantics of an interaction `P` may be written as `[P]^+` and `[P]^−`, respectively.
Coming back to the interpretation of \(sd\) from above, of course, \([\_\_]\) must also be defined for the other operators (see [23] for details). For example, the \(alt\) denotes alternatives in interactions, and its semantics can be defined as

\[
(alt) \quad [alt(P, Q)] = [P] \cup [Q].
\]

Here I also use the canonical extension of set operators to pairs of sets, that is, I write \(⟨A, B⟩ \cup ⟨X, Y⟩ \) to mean \(⟨A \cup X, B \cup Y⟩, \) and so on for other operators.

### 2.3 Characterization of interaction semantics

A great number of semantics and equivalences for concurrency have been defined (see e.g. [4, chapter 1] for a comparison). I shall now discuss some of the more popular semantic dimensions.

**Interleaving vs. true concurrency** This spectrum is usually associated with CSP and CCS on the interleaving end (see [2] and [13], respectively), and Petri-nets and related formalisms on the other (see [17]). Both of these semantic paradigms have been applied to MSCs in the past (cf. [5, 20] for examples and [18] for an overview). The standard avoids a clear statement in favor of interleaving semantics, merely saying that: “to explain Interactions we apply an Interleaving semantics” (cf. [15, p. 403, emphasis added]). This statement implies that the interleaving semantics provided in the standard is only an explanation, but not a definition. This would mean that other formalisms might be valid definitions (or explanations) of the semantics of interactions in UML, too.

**Linear vs. branching time** The two ends of this spectrum are often associated with CSP and CCS, respectively. Without the notion of invalid traces, the semantics of UML interactions and the notions of equivalence and refinement would be identical to traditional trace semantics. Considering also the invalid traces simply adds another set of traces, but does not change the semantic paradigm in any way. That is, the UML standard defines a linear time semantics.

**Readiness/failure traces** In readiness and failure semantics, each trace also carries a set of actions (read: EventOccurrences) which are (not) possible after the trace. For a given set \(X\) of conventional traces, it is possible to compute largest common prefixes and determine the respective possible next EventOccurrences. For a given interaction \(P\), it is possible to compute the set of ready traces \(⟨t, R⟩\) from \([P]^+\). As an example, suppose that \(Σ = a, b, c, d, e\) and \(X = \{a, b, c, a, b, d, a, e\}\). The common prefixes with their ready EventOccurrences are \(⟨a, \{b, e⟩\), \(⟨a, b, \{c, d⟩\), \(a, b, c, \emptyset⟩\), \(a, b, d, \emptyset⟩, \) and \(a, e, \emptyset⟩\). The set of failure traces of \(P\) is \(\{⟨t, \overline{R}⟩\}, \) where \(⟨t, R⟩\) is computed like the ready traces, but starting from \([P]^−\) rather than \([P]^+\). It is unclear, however, how the notion of contingency integrates with this paradigm: in failure and readiness semantics, an action is either possible or not.
3 The neg-operator

The neg-operator is probably a kind of negation. The standard does not give an example, or explain the intuition or pragmatics of this operator, but flatly declares that “the interaction operator neg designates that the combined fragment represents traces that are defined to be invalid” (cf. [15, p. 411]). This could be interpreted as

\[(N.1: \text{loose negate}) \quad \neg\!(P)_{N,1} = (\emptyset, [P]^+).\]

This interpretation could be formulated intuitively as “not the traces of \(P\)”. However, under this interpretation, all negative traces specified so far would be lost. It thus behaves strangely under double negation, for

\[\neg\!(\neg\!(P))_{N,1} = (\emptyset, \emptyset).\]

Also, the standard declares that “All InteractionFragments that are different from Negative are considered positive, meaning that they describe traces that are valid \[\ldots\]” (cf. [15, p. 370]). This suggests the following interpretation,

\[(N.2: \text{strict negate}) \quad \neg\!(P)_{N,2} = (\langle [P]^+_N, [P]^+_N \rangle)\]

where \(X = \Sigma^* - X\), that is, language complement. Observe that \([P]^- \subseteq [P]^+\) and \([P]^+ \subseteq [P]^\) for all \(P\) and for all interpretations, and therefore \([P]^+ \subseteq \neg\!(P)\) for all \(P\) and for all interpretations, and therefore \([P]^+ \subseteq \neg\!(P)\) for all \(P\).

Under interpretation \(N.2\), the traces specified by \(P\) are marked as invalid, and all other traces as valid. This could be expressed intuitively as “anything but \(P\).” However, this interpretation makes no sense for double negation. Abbreviating \(\neg\!(P)\) as \(Q\), I have \([Q] = (\langle [P]^+_N, [P]^+_N \rangle)\) and thus \([Q]^+_N = [P]^+_N\) and so

\[\neg\!(\neg\!(P))_{N,2} = \neg\!(Q)_{N,2} = \langle [P]^+_N, [Q]^+_N \rangle\]

\[= \langle [P]^+_N, [P]^+_N \rangle = \langle [P]^+_N, [P]^+_N \rangle\]

Using \(\text{flip}(\langle x, y \rangle) = \langle y, x \rangle\), this means that

\[\neg\!(\neg\!(\neg\!(P)))_{N,2} = \text{flip}(\langle \neg\!(P) \rangle_{N,2})\]

and also

\[\neg\!(\neg\!(\neg\!(\neg\!(P))))_{N,2} = \langle P \rangle_{N,2} \]

Clearly, this is odd. It leads us to a simpler, more intuitive interpretation of \(\neg\!\).
This interpretation simply reverses valid and invalid traces of $P$, but preserves the contingent traces. Intuitively, it could be formulated as “flip valid and invalid”. Note that this interpretation yields an intuitive sense for double negation:

$$[\neg(\neg(P))]_{N.3} = [P]_{N.3}$$

as $\text{flip} \circ \text{flip}$ obviously is the identity. Obviously, this interpretation is in contradiction to some of the citations from the standard as given above. Still, I shall adopt the interpretation N.3, as it is simply the only consistent approach.

4 The assert-operator

The assert-operator might be a kind of affirmation, implication, or temporal sequence. The standard unfortunately gives only a single (rather unhelpful) example (cf. [15, p. 442]), and does not provide an intuitive explanation of its meaning or usage. In this section, I discuss several possible interpretations for this operator.

4.1 assert as affirmation

Intuitively, one might interpret assert as an affirmation of the traces of its operand, in the sense of “$P$, and only $P$”. The meaning of the assert-operator is explained by the standard as “the sequences of the operand are the only valid continuations. All other continuations result in invalid traces.” (cf. [15, p. 412]) This suggests the following interpretation:

(A.1: affirm) \[ [\text{assert}(P)]_{A.1} = \langle [P]^+, [P]^+ \rangle. \]

When interpreting assert as an affirmation, one would expect that the meaning of an interaction remains constant, no matter how often it is asserted, that is, a kind of idempotency-property. One would expect that $\text{assert}(\text{assert}(P))$ is equivalent to $\text{assert}(\text{assert}(P))$, e.g., in the sense that

$$[\text{assert}(P)] = [\text{assert}(\text{assert}(P))]$$

This is obviously true for interpretation A.1. Under this interpretation, assert would completely remove contingency, but preserve valid traces and invalid traces, i.e. $[\text{assert}(P)]^+_{A.1} = [P]^+$ and $[\text{assert}(P)]^-_{A.1} \supseteq [P]^-$.

In the quotation given above, the standard seems to demand this interpretation quite imperatively. However, there is also a contradictory statement in the standard, declaring that “the invalid set of traces are associated only with the use of a Negative CombinedInteraction.” (cf. [15, p. 419]). Interpretation A.1 clearly refers to the set of negative traces in a way that cannot be achieved by understanding assert as syntactic sugar. So, there is some freedom for interpretations of assert, and I shall explore some of them now.
4.2 assert as a binary operator

Another problem with interpretation A.1 is the fact that so far it is described as being unary. It might be understood as a binary operator, too, as the UML standard declares that “the sequences of the operand (sic!) of the assertion are the only valid continuations.” (cf. [15, p. 412]). Even though the standard explicitly mentions only a single operand, it talks about it as being a continuation of a preceding trace. But it is unclear, what the scope of the preceding trace is (see example in Figure 3).

![Fig. 3. Which messages constitute the trace preceding the assert: only b or both a and b?](image)

It would be much easier, if the assert were a binary operator where the first operand declares a condition or trigger (a kind of "precharts" known from LSCs, [3]), and the second declares the consequence or result. This is also suggested by the standard when declaring that “Assertions are often combined with ignore or consider as shown in Figure 345” (cf. [15, p. 412]). The Figure mentioned is reproduced in a simplified form in Figure 4 (left). A stratified variant of the notation is proposed in Figure 4 (right). The operators ignore and consider are defined to be dual. To simplify my task, I choose ignore in the remainder, thus sparing me another auxiliary filter-function (cf. [23]).

![Fig. 4. Usage of assert/consider as suggested by the standard (left). A simple way to cleanly embed this usage into the notation (right).](image)
However, there is a fundamental problem here. Recall the intuitive interpretation of valid and invalid traces as must and must not be possible, as laid out in Section 2.2. Suppose that \( P \) is an interaction, \( \Gamma \) a set of messages to be ignored and \( \mu : EO \rightarrow MSG \) a mapping from the event occurrences of \( P \) to the messages they belong to. Then what is the meaning of \([\text{ignore}(P, \Gamma, \mu)]\)?

A first approach might yield

\[
(I.1) \quad [\text{ignore}(P, \Gamma, \mu)]] = \langle \Sigma^* \uplus \Sigma^*, \Sigma^* \uplus \Sigma^* \rangle
\]

such that \( \Sigma = \{ x \in EO | \mu(x) \in \Gamma \} \). The shuffle-operator \( \uplus \) is defined for \( v, w \in \Sigma^* \) and the empty sequence \( \epsilon \) as \( \epsilon \uplus w = w, v \uplus \epsilon = v, \) and

\[
 xv \uplus yw = \{ x(v \uplus yw), y(xv \uplus w) \}.
\]

So, for example, shuffling the two sequences \( a.b \) and \( x.y.z \) yields the following traces:

\[
 a.b.x.y.z, a.x.b.y.z, a.x.y.b.z, x.a.b.y.z, x.a.y.b.z, x.a.y.z.b, x.y.a.b.z, x.y.a.z.b, x.y.z.a.b.
\]

Observe, that the order of symbols from the original traces is respected. The shuffle operator can be extended canonically to sets of words, of course.

Interpreting \text{assert} as binary, one might alternatively define

\[
(I.2) \quad [\text{assert}(P, Q, \Gamma, \mu)] + = [P]^+ \cdot [Q]^+,
\]

with \( \Sigma, \Gamma \) and \( \mu \) as before. Recall that \( \Gamma \) denotes messages that are to be ignored, so that in the semantics, all possible event occurrences corresponding to these messages must be considered, which is just \( (\mu(\Gamma))^* = \Sigma^* \).

These last two interpretations are problematic, of course, since now, many contingent traces have become valid or invalid, and probably much more than have been intended. Also, there may now be sequences of event occurrences of arbitrary length between the trigger and the consequence, which might also not be intended.

### 4.3 assert as implication

First of all, it is somewhat reminiscent of implication in classical logic (ignoring the temporal aspect of \text{assert} for a moment). Here, I have for instance \( (\alpha \Rightarrow \beta) \Leftrightarrow (\neg \alpha \lor \beta) \). The two sides would correspond to \text{assert}(P, Q) and \text{alt}(P, \text{neg}(Q)), respectively, and one would expect them to be equivalent in some sense, e.g., as

\[
\text{[assert}(P, Q)] = \text{[alt}(\text{neg}(P), Q)]).
\]

Using \([\text{alt}(P, Q)] = [P] \cup [Q] \) as described above and interpretation N.3 from above, I would yield

\[
(A.2: \text{imply}) \quad [\text{assert}(P, Q)]_{A.2} = \langle [P]^-, [Q]^+, [P]^+ \cup [Q]^- \rangle.
\]
Note however, that under interpretation A.2, there is implication and disjunction (and even negation), but there is no operator corresponding to conjunction. Also, idempotency is not preserved, since

\[
[\text{assert}(P, P)]_{A.2} = \langle [P]^+ \cup [P]^-, [P]^+ \cup [P]^+ \rangle,
\]

that is, the specified traces are both valid and invalid, which does not make much sense in general.

### 4.4 assert as consequence

In another sense, assert might be interpreted in a more temporal way, as is also suggested by the standard, when declaring that “we expect a q message to occur once a v has occurred” (cf. [15, p. 442, with a view to Fig. 345, see Figure 4 (left) in this paper]). Intuitively, this could be expressed as “if P has occurred, then Q and only Q must follow”, i.e. formally

(A.3a) \[
[\text{assert}(P, Q)]_{A.3a} = \langle \emptyset, [P]^+, [Q]^+ \rangle.
\]

This is unintuitive in the sense that P only may occur, but does not have to occur. So, nothing at all needs to happen, and all other traces are contingent (cf. Section 2.2).

One might instead interpret assert\((P, Q)\) as “P must occur, and then Q must follow”. This would result in

(A.3b) \[
[\text{assert}(P, Q)]_{A.3b} = \langle [P]^+, [Q]^+, [P]^+ \cup [Q]^+ \rangle.
\]

Arguably, any traces forbidden by P and Q alone should still be forbidden for assert\((P, Q)\), so that one might want to refine interpretation A.3b to

(A.3c: next) \[
[\text{assert}(P, Q)]_{A.3c} = \langle [P]^+, [Q]^+, [P]^+ \cup [Q]^+ \cup [P]^+, [Q]^+ \rangle.
\]

Note that under this interpretation, assert\((P, Q)\) is not syntactic sugar for alt(seq\((P, Q)\), neg(seq\((P, neg(Q)))\)). Note also, that idempotency makes no sense under this interpretation.

### 5 Design steps

In the previous sections I have attempted to provide intuitive and formally satisfying interpretations for the operators assert and neg. These attempts have led to problems with the notions of valid and invalid traces. Therefore, in this section, I try to explore the pragmatic justification of these notions.

One might presume, that the motivation for having two separate sets of valid and invalid traces at the same time—and thus implicitly also a third set
of contingent traces—lies in an idea of consecutive refinement steps. Such a sequence of refinements could carefully carve the desired system behavior out of the totality of all traces.

As an analogy, consider the notion of loose semantics in abstract data types (cf. [24]). There, providing more and more “axioms” for a “specification” in a series of refinement steps allows less and less “implementations” for that “specification”. In this analogy, possible and impossible implementations would correspond to valid and invalid traces. Adding axioms would correspond to adding interaction fragments to a design (see Figure 5).

![Fig. 5. Refinement reduces uncertainty: valid and invalid traces must and must not be possible, respectively, while contingent only may be possible, but need not be.](image)

### 5.1 Basic relationships

All relationships between two interactions such that one of them contains more detail and less uncertainty than the other are called **elaborations**. Formally, \( P \) elaborates \( Q \) (written \( P \sim Q \)) iff \([P]^+ \supseteq [Q]^+ \land [P]^- \supseteq [Q]^-\). See Figure 5 for an illustration.

An **enrichment** is a relationship where one interaction has more valid traces than another, while nothing is said about the invalid traces. Formally, \( P \) enriches \( Q \) (written \( P \mid= Q \)) iff \([P]^+ \supseteq [Q]^+\).

Conversely, a **restriction** is a relationship where one interaction has more invalid traces than another, while nothing is said about the valid traces. Formally, \( P \) restricts \( Q \) (written \( P \mid= Q \)) iff \([P]^- \supseteq [Q]^-\).

Then, a **refinement** is a relationship where one interaction has both more valid and more invalid traces than another, that is \( P \) refines \( Q \) (written \( P \mid= Q \)) iff \(( [P]^- \supseteq [Q]^- ) \land ( [P]^+ \supseteq [Q]^+ ) \) or \( P \mid= Q \land P \mid= Q \), see Figure 5 for an illustration.

Finally, two interactions \( P \) and \( Q \) are **equivalent** (written \( P = Q \)) iff \([P] = [Q]\), or \( P \mid= Q \land Q \mid= P \). Similar to process algebras, constructions like \([\text{ignore}(Q, \Gamma, \mu)] = [P]\) can be used as a kind of refinement-relationship. Note
that equivalence is the only relationship among interactions the standard mentions ("Two Interactions are equivalent if their pair[s] of trace-sets are equal" (cf. [15, p. 420])), and that it is captured by equivalence as defined here.

Observe that only N.2 and A.1 constitute elaborations in the sense of \( \neg(P) \sim P \) and \( \text{assert}(P) \sim P \). None of the other interpretations constitute any other of the relationships defined here. However, defining an elaboration as \( [P]^+ \cup [P]^- \subseteq [Q]^+ \cup [Q]^- \) would also cover N.3 (flip negate).

These two interpretations have in common that they completely remove contingency. Thus, there can be no further (useful) elaborations afterwards, so there is always only exactly one refinement step using assert or neg. So refinement must really be achieved using other means, and so, refinement is not a justification for assert and neg under interpretations N.2 and A.1. All other interpretations, however, are not consistent with the basic relationships defined above, and those have been very basic indeed.

So either the notion of valid and invalid traces has to be abandoned in favor of a simpler "single set of traces"-semantics. This would also imply to remove the operator neg, and more or less fix interpretation A.3b for assert in the sense that \( [\text{assert}(P)]^+_A \leq [\text{assert}(P)]^+_A \). Note, that \( [\text{assert}(P)]^+_A \leq [\text{assert}(P)]^+_A \)

Or, alternatively, one might abandon the idea of refinement as a justification for assert and neg and, again, the whole idea of valid and invalid traces. But then, what justification is there?

In both cases, the standard needs clarification. One possible solution (the one I find most convincing from a practitioners point of view) is briefly discussed in Section 6 under the title "metalogical interpretation".

5.2 Applying and tracking design steps

In this section, I look at the impact of design steps. For simplicity, I assume for the time being, that a design consists of (versions of) only a single interaction diagram. This is no real restriction, since a set of interactions can be simulated using the alt-operator. So, the remarks and observations of this section can equally be applied to sets of interactions.

During development and evolution of a system, I distinguish three classes of design steps: detailing, adapting and realizing (see Figure 6).

**Detailing** means adding details so as to remove uncertainty. It subsumes all activities that leave the set of all traces unchanged, but decreases the number of contingent traces. This meaning is captured formally by the elaboration-relationship.

**Adapting** means changing a design to accommodate new ideas about a system. It can result in changed or unchanged sets of valid and invalid traces, and it can also change the overall set of traces (or leave it unchanged). There is no formal meaning for this kind of design step, but it can be

**Realizing** means that an interaction is taken as the specification of a behavioral model or a program. No implementation has contingency -
traces are either possible, or impossible, and thus valid or invalid. Realizing is also a kind of elaboration-relationship.

These design steps and the underlying relationships between interactions may be used in a number of practical scenarios.

– Suppose that a specification given as an interaction $P$ is realized by a system $S$, whose behavior is completely described by an interaction $R$. It can be automatically checked whether actually $R \vdash P$, e.g. by model-checking the valid and the invalid traces of $P$ vs. $R$.

– Seen in a different way, the valid traces of $P$ may be used as test cases for $S$. The invalid traces can not be tested of course, as this would require a complete test coverage. For this usage, thus, only enrichments are necessary.

– Imagine a maintenance project of a large system, where the people in charge of doing a certain modification do not have adequate knowledge of the system, possibly because the system is poorly documented, and the knowledge is lost. Such systems would typically have quite a large population of interactions, as compared to the modification in question. Here, it is very easy to accidentally introduce an interaction $Q$ such that $[Q] \supseteq [Q]^{-} \neq \emptyset$ (like $\text{alt}(P, \neg \text{neg}(P))$) under interpretation N.2). Obviously, this makes no sense, and thus, $Q$ should be refuted by an automated tool as being inconsistent.

– Suppose, that a series of design steps is being applied to an interaction $P_0$ resulting in $P_1, \ldots, P_n$. The impact of the design steps is easily determined by comparing $[P_i]$ with $[P_{i+k}]$. Now assume that at some point, a valid trace $t$ is identified that really should be invalid, or the other way round. When trying to eliminate this trace, one must be careful not to introduce another error instead. For this task, thus, it would be helpful to be able to automatically track the origin of the trace by identifying the design step that introduced $t$, so as to determine whether it was an accident or not. To this end, one must find $i$ such that $t \in [P_i]^+$ but $t \notin [P_{i+1}]^+$.
Clearly, these scenarios could be supported by automated tools implementing the relationships defined above. It is currently unclear, whether it is possible to have a calculus of refactoring design steps which might be formally justified.

6 Discussion

In this paper I have discussed several possible interpretations for the operators assert and negate as declared in the new UML 2.0 standard. It turns out that the explanations given in the standard are by no means adequate. It is thus currently not clear what the contribution of these operators in UML 2.0 to verification of and reasoning about software system will be. There are a number of open questions that remain to be explored.

Metalogical interpretation From a pragmatic perspective, a trace is a property, namely, that a given system does (or does not) exhibit a certain behaviour. The assert and negate are of a different kind in that they make statements about traces rather than modifying them. So, they are more like the operators in interaction overview diagrams. For this purpose however, assert and negate are not very powerful. So why not remove them from interactions and embed interactions in a traditional logic, e.g., like in Figure 7.

\[
\begin{align*}
\text{Trace} & ::= \text{UML-Interaction} \\
\text{Expr} & ::= \text{Trace} \\
& | \neg \text{Expr} \\
& | \text{Expr} \land \text{Expr} \\
& | \text{Expr} \lor \text{Expr} \\
& | \text{Expr} \implies \text{Expr} \\
& | \Box \text{Expr} \\
\end{align*}
\]

\[
\begin{align*}
[\text{Trace}] & = \text{the trace} \\
[\neg \text{Expr}] & = \neg [\text{Expr}] \\
[\text{Expr} \land \text{Expr}] & = [\text{Expr}_1] \land [\text{Expr}_2] \\
[\text{Expr} \lor \text{Expr}] & = [\text{Expr}_1] \lor [\text{Expr}_2] \\
[\text{Expr} \implies \text{Expr}] & = [\text{Expr}_1] \implies [\text{Expr}_2] \\
[\Box \text{Expr}] & = \Sigma^* [\text{Expr}] \\
\end{align*}
\]

Fig. 7. A temporal logic built out of traces: syntax (left) and semantics (right).

This is also very appealing when the underlying semantics is defined using other formalisms than traces, e.g., partial languages or a notation like TTCN, for the notion of a complement is even less trivial there.

Comparative concurrency semantics Section 2.3 very briefly sketches the relationships between the traditional notions of concurrency semantics and those defined by UML. As we have seen, the current definitions as proposed in the standard are somewhat deficient. So, it might be interesting to further study this relationship, and try to adopt notions and tools from that area. As a starter, consider the interaction between complete/partial traces and some interpretations of assert.
**Non-classical logics** In section 4.3, I attempted a logic interpretation of `assert` which failed for classical logic. But what about other logics like linear and intuitionistic logics? In intuitionistic logic, I have

\[ (1) \neg(\neg\neg\alpha \implies \alpha) \quad \text{and} \quad (2) \alpha \implies \neg\neg\alpha. \]

When translating the first axiom into `neg(assert(neg(neg(P)), P))` and using interpretations A.2 and N.2, this makes sense, in a way, since \([assert(P, P)]_{A.2}\) is contradictory. The other axiom is not true, however.

Another element of classical logics missing in intuitionistic logic is the principle of "tertium non datur", which when applied to UML 2.0-interactions would require the set of contingent traces to be empty always. This certainly removes a number of problems.

**Tools** One of the benefits of formal semantics is the possibility of building automated tools using the semantics for validation, verification and visualization purposes. I have already implemented a prototype of such a tool, formalizing the semantics and some equivalence notions. The operators `neg` and `assert`, however, need clarification before they can be added to such a tool. Or rather, in the meantime, experiments with different interpretations of these operators might shed light on their meaning and usefulness.

6.1 Related Work

There are many variants of Message Sequence Charts (MSCs), such as the 1996 and 2000 versions of the standard proper, UML 1.4 collaborations [14], Life Sequence Charts [3] and Extended Event Traces [10]. Of course, there is a body of work concerning the formal semantics of these, including [1, 5, 8, 9, 11, 12, 20, 21, 25]. See [10] and [18] for exhaustive surveys.

**Acknowledgments** Thanks go to Stephan Merz and Alexander Knapp for discussions and proof reading. Also, I’d like to thank the four anonymous referees for their helpful remarks.

**References**