Model Checking of UML 2.0 Interactions

Alexander Knapp[1] and Jochen Wuttke[2]

[1] Ludwig-Maximilians-Universität München, Germany
knapp@pst.ifi.lmu.de

[2] Università della Svizzera Italiana, Lugano, Switzerland
wuttkej@lu.unisi.ch

Abstract. The UML 2.0 integrates a dialect of High-Level Message Sequence Charts (HMSCs) for interaction modelling. We describe a translation of UML 2.0 interactions into automata for model checking whether an interaction can be satisfied by a given set of message exchanging UML state machines. The translation supports basic interactions, state invariants, strict and weak sequencing, alternatives, ignores, and loops as well as forbidden interaction fragments. The translation is integrated into the UML model checking tool HUGO/RT.

Keywords: Scenarios, UML 2.0 interactions, model checking.

1 Introduction

Scenario-based development uses descriptions of operational sequences to define the requirements of software systems, laying down required, allowed, or forbidden behaviours. In version 2.0 [1] of the "Unified Modeling Language" (UML) a variation of High-Level Message Sequence Charts (HMSCs [2]) replaced the rather inexpressive notion of interactions in UML 1.x for describing scenarios. The scenario language of UML 2.0 not only contains the well-known HMSC notions of weak sequencing, loops, and alternative composition of scenarios, but also includes a peculiar negation operator for distinguishing between allowed and forbidden behaviour. The thus gained expressiveness would make UML 2.0 an acceptable choice to model high-quality and safety-critical systems using scenario-based techniques. However, several vaguenesses in the specification document have led to several, differing efforts for equipping UML 2.0 interactions with a formal semantics (see, e.g., [3,4]).

We propose a translation of UML 2.0 interactions into automata. This synthesised operational behaviour description can be used to verify that a proposed design meets the requirements stated in the scenarios by using model checking. On the one hand, the translation comprises basic interactions of partially ordered event occurrences, state invariants, the interaction combination operators for weak and strict sequencing, parallel and alternative composition, as well as a restricted form of loops, which can have potentially or mandatorily infinitely many iterations. On the other hand, besides these uncontroversial standard constructs, we also handle a classical negation operator [3], which avoids the introduction of three-valued logics as suggested by the UML 2.0 specification by resorting to binary logic.

The translation procedure is integrated into our freely available UML model checking tool HUGO/RT [5]: A system of message exchanging UML state machines together

with the generated automaton representing a UML interaction for observing message traces is translated into the input language of an off-the-shelf model checker, which then is called upon to check satisfiability. Currently, we support interaction model checking over state machines with SPIN [6] and, partially, with UPPAAL [7].

The remainder of this paper is structured as follows: In Sect. 2 we briefly review the features of UML 2.0 interactions. In Sect. 3 we introduce our automaton model for interactions, and in Sect. 4 we describe the translation from UML 2.0 interactions into automata. Section 5 reports on the results of applying SPIN model checking with our approach. In Sect. 6 we discuss related work, and Sect. 7 concludes with a summary of the results and an outlook on future work.

2 UML 2.0 Interactions

UML 2.0 interactions consist of *interaction fragments*. The primitive fragments are *occurrence specifications*, specifying the occurrence of events within an object that is participating in the interaction. *Combined fragments* aggregate occurrence specifications into bigger interaction fragments. A combined fragment comprises an *operator*, defining the meaning of the particular fragment, and one or more *operands*. The operands are interaction fragments themselves, and can be guarded by an optional condition, limiting the possibilities for when this operand may be executed.

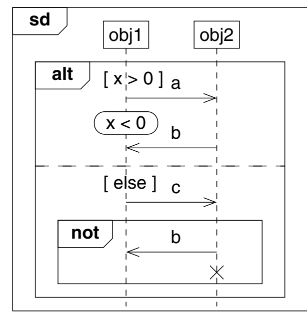


Fig. 1. Sample interaction

The example in Fig. 1 shows instances of the important aspects of a UML 2.0 interaction. The behaviour of two objects obj1 and obj2 is specified by message exchanges (sending and receiving occurrence specifications, denoted by arrow tails and heads) on their *lifelines*, object destruction (cross), state invariants (conditions in a rounded box), and combined fragments. Vertical juxtaposition of interaction fragments implies *weak sequencing*, such that in the second operand of the alternative the sending of c, *active* on obj1, comes before any event on obj1 inside the not fragment, and the receiving of c before any event on obj2. Both operands to alt are guarded by conditions, which determine which operands can be chosen at "runtime".

The primitive interaction fragments we consider are basic interactions, consisting of a set of event occurrences with a partial order [1, p. 410], and state invariants for a single or several lifelines that has to hold if the state invariant is reached. Of the interaction operators [1, p. 410–412], we consider weak (seq) and strict sequential composition, parallel composition (par), alternative (alt), weak and strict sequencing loop, ignore of messages not mentioned, and a binary negation (not).

3 Interaction Automata

We interpret a UML 2.0 interaction as an observer of the message exchanges and state changes in a system. Whenever the system under observation sends or receives a mes-

sage or one of its objects terminates successfully, the observer is notified and can act accordingly by making a move accepting the event or by producing a failure. However, it may also refrain from doing so, if it does not deem the state change relevant. Taking such an observer to be an automaton accepting words of system changes, i.e. state changes or events, the acceptance conditions for finite and infinite runs can be rendered as the corresponding ones in finite state machines and Büchi automata [8].

Interaction automata, realising such an observer from an interaction by a state-transition system, are defined over an *interaction alphabet* (L, E, Σ) of a finite set of involved lifelines L, a set E of termination, send and receive events from messages exchanged between the lifelines, and a set Σ of system states. The transitions outgoing from a state define a set of events that, when occurring, enable the transition. Moreover, transitions may be guarded by conditions arising from the conditions in the interaction. In order to reflect weak sequencing of interactions, the events η and the guard g of a transition show a set of lifelines *lifelines*(η, g), which are active when making a move by this transition. Finally, an interaction automaton may also use and manipulate a set of counters V that allow to record how often lifelines in loops have executed. For the guards we assume a propositionally closed language $\mathscr{G}_{\Sigma,V}$; it should be expressive enough to capture system state queries on Σ and to compare counters in V. For the actions, we similarly assume a language $\mathscr{A}_V$ manipulating the set of counters V.

A *run* of an interaction automaton N starts from an *initial configuration* (i, v_0) where i is N's *initial state* and v_0 the valuation of the set of counters V to zero. A run of N proceeds by *steps* $(s, v) \xrightarrow{(\sigma,\zeta)} (s', v')$ with a system state $\sigma \in \Sigma$ and a set of events $\zeta \subseteq E$, if there is a transition outgoing from s with a set of events equal to ζ such that in σ and v the transition guard is satisfied and the valuation v is updated to v' according to the transition's action. N *accepts* a finite run, if this run reaches a state in the *accepting states* A of N, and it accepts an infinite run if this run reaches one of N's *recurrence states* R infinitely often.

It may be noted that although we define interaction automata to be finitely represented, the configuration space may be infinite due to unbounded increases of counters. However, for bounded interactions, in which no lifeline in a loop is allowed to proceed arbitrarily in advance with respect to another lifeline in the loop [9], the configuration space can be kept finite, and even for unbounded interactions, the system under observation may not produce runs that exhibit unbounded differences between counters.

4 Translation of UML 2.0 Interactions

We translate UML 2.0 interactions into interaction automata following the generally agreed upon semantics of basic interactions, state invariants, and the interaction operators seq, strict, par, alt, ignore, and, in a restricted form, loop [3,4]. Furthermore, we handle a binary logic not operator; not and loop are restricted to basic interactions.[1]

In contrast to other approaches (e.g. [10,11]) we propose not to generate one automaton for every object in an interaction, but to use only a single observing interaction

[1] For a more detailed account of the translation procedure we refer the reader to the unabridged version of this article in the proceedings of the workshop "Critical Systems Development Using Modeling Languages" 2006.

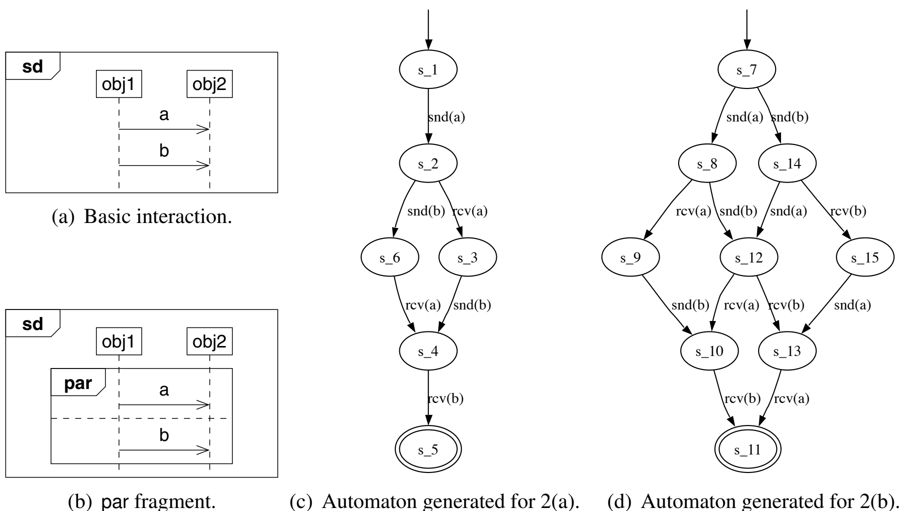
automaton for the entire interaction. This single automaton represents the property to be checked by a model checker. Our translation of basic interactions is a simplified version of the construction for Live Sequence Charts (LSCs) given by Brill et al. [12], the handling of weak sequencing cuts down techniques of Alur and Yannakakis for bounded MSCs [9].

4.1 Basic Interactions, Loops, and Negation

We first describe the translation of basic interactions, and loops and negations of basic interactions. These interaction fragments form the primitive blocks in our translation procedure and have to be represented as interaction automata directly.

Basic Interactions. The translation of basic interactions is performed by unwinding the partial order of events. For each event the partial order defines the prerequisite events, which must be unwound before that event can be unwound. The unwinding is performed in *phases* [12]. In every phase there exists a *history*, i.e., a set of events which have been unwound already. Given a phase, the function *ready* delivers the events which can be unwound in the next step, as inscriptions of transitions; we write $\mathit{isAccepting}(\mathit{phase})$ for $\mathit{ready}(\mathit{phase}) = \emptyset$. A function *nextPhase*, given a phase and a transition inscription, creates a new phase recording the additional event from the transition inscription in the history. The following algorithm *unwind* transforms phases directly into states of an interaction automaton *result*. Started for a basic interaction with the phase with empty history, the state returned by this call to *unwind* becomes the initial state of *result*.

1 $\mathit{unwind}(\mathit{phase}, \mathit{result}) \equiv$
2 $\lceil \mathit{state} \leftarrow \mathit{addState}(\mathit{result})$
3 if $\mathit{isAccepting}(\mathit{phase})$ then $\mathit{addAcceptingState}(\mathit{result}, \mathit{state})$ fi



(a) Basic interaction.

(b) par fragment.

(c) Automaton generated for 2(a).

(d) Automaton generated for 2(b).

Fig. 2. Automata for a basic interaction and par (accepting states are doubly outlined)

```
4    for label ∈ ready(phase) do
5        addTransition(result, state, label, unwind(nextPhase(phase, label), result))
6    od
7    return state⌋
```

Figure 2(a) shows an example of a basic interaction, the interaction automaton in Fig. 2(b) shows the effects of unwinding its partial order. The branching in s_2 is due to the fact that the second event can be either the reception of a or the sending of b.

Loops. The UML 2.0 defines loops which have a lower and an upper bound for the number of iterations their operand has to perform; the lower bound has to be finite, while the upper bound may be infinity. We change this and also allow the lower bound to be infinity. However, we restrict loops to contain only a basic interaction.

For finite or infinite loops of a basic interaction the basic unwinding algorithm can be reused. As weak sequencing is used for loops, the lifelines in the underlying basic interaction can make different progress. Thus the history stored in a phase for a basic interaction becomes insufficient for loops, as not all prerequisite events for a given event will be present in the history if the lifeline's event is lagging behind the lifeline of one of its prerequisites. Thus we let *loop phases* also show a history, but the computation of the next events from a loop phase is changed: We introduce counters for recording the separate progress of each lifeline. Then, an event e on lifeline l is possible in a loop phase if the following condition is met: If e has a prerequisite e' on a lifeline l', either the counter for l' is greater than the counter for l, or the counters for l and l' are equal and e' is present in the history. Upon finishing a cycle through a lifeline the counter for this lifeline has to be increased in order to make real progress.

It remains to ensure that the number of iterations of the loops indeed is between its lower and upper bound. If the lower bound is finite, a phase becomes accepting if the counters for all lifelines are equal and greater than or equal to the lower bound. If the upper bound is finite a new cycle of a lifeline may only be started, if the counter of the lifeline has not reached the upper bound. Finally, if either the lower or the upper bound of iterations is infinite, we have to introduce a recurrent state which is run through every time the lifeline counters are equal. The introduction of a recurrent phase extends the *unwind* algorithm after line 3 by

```
if isRecurrent(phase) then addRecurrentState(result, state) fi
```

Figure 3 shows an example of an automaton for a loop. The example is based on the basic interaction in Fig. 2(a), wrapped into a loop $\langle 4, \infty \rangle$.

Negation. We replace UML 2.0's notorious negation operator neg by a binary logic variant not which simply accepts all those traces that are not valid for its operand. However, an algorithm for negating general interaction automata is out of reach. Thus we restrict the application of not to basic interactions, such that the interaction automaton to be negated is deterministic and does not involve counters. The negation operation on these interaction automata basically means that all accepting states become non-accepting states and all non-accepting states become accepting states in the negated automaton. A new recurrent state is added, and from all complemented states transitions to this accepting state are added to accept all events that were not accepted in the

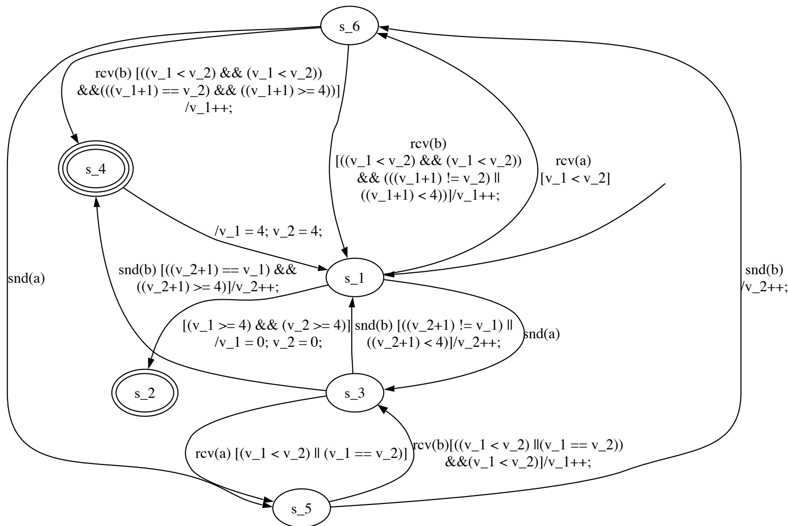


Fig. 3. The automaton for an infinite loop (recurrence states are triply outlined; transition annotations abbreviated)

corresponding state of the original automaton. This additional accepting state is also equipped with a self-loop accepting all possible events.

4.2 Interleaving, Sequencing, and Composition

We next describe the parallel (par) and weak sequential (seq) composition of two interaction automata N_1 and N_2 over a common interaction alphabet (L, E, Σ). We also give a brief account of strict sequential (strict) composition, extending this notion to introduce a general strict sequencing variant sloop of loops, and of alternatives (alt), ignore, and state invariants.

Parallel Composition and Weak Sequencing. The *parallel* composition $N_1 \| N_2$, accepting the trace $o_0 o_1 \cdots \in (\Sigma \times \wp E)^* \cup (\Sigma \times \wp E)^\infty$ by interleaving traces $o_0^{(1)} o_1^{(1)} \ldots$ and $o_0^{(2)} o_1^{(2)} \ldots$ accepted by N_1 and N_2 respectively, uses a construction very similar to the parallel composition of Büchi automata [8]. This construction only has to be adapted to cover the case that both or one of the interaction automata do not show recurrence states.

A slight modification of the construction for parallel composition can be used for obtaining the *weak sequential* composition $N_1 ;_{\curlyvee} N_2$ of N_1 and N_2: Also interleavings of traces $o_0^{(1)} o_1^{(1)} \ldots$ and $o_0^{(2)} o_1^{(2)} \ldots$ accepted by N_1 and N_2 are accepted by $N_1 ;_{\curlyvee} N_2$, but in the interleaving no $o_j^{(2)}$ is allowed to occur before an $o_k^{(1)}$ if their active lifelines overlap. Therefore, the states of $N_1 ;_{\curlyvee} N_2$ show an additional component of sets of lifelines, recording which lifelines have been covered by the interleaving of N_2.

Formally, given the interaction automata $N_1 = (S_1, V_1, T_1, i_1, A_1, R_1)$ and $N_2 = (S_2, V_2, T_2, i_2, A_2, R_2)$, their weak sequential composition $N_1 \mathbin{;_{\gtrless}} N_2$ is the interaction automaton $(S, V_1 \cup V_2, T, i, A, R)$ with

$$
\begin{aligned}
& S = S_1 \times \wp L \times S_2 \times \{0,1,2\}, \qquad i = (i_1, \emptyset, i_2, 0) \\
& (s_1, K, s_2, k) \in A \iff (s_1 \in A_1 \wedge s_2 \in A_2 \wedge k = 0) \\
& (s_1, K, s_2, k) \in R \iff k = 2 \\
& ((s_1, K, s_2, k), (\eta, g, a), (s'_1, K', s'_2, k')) \in T \iff \\
& \qquad (((s_1, (\eta, g, a), s'_1) \in T_1 \wedge s'_2 = s_2 \wedge K = K' \wedge \mathit{lifelines}(\eta, g) \cap K = \emptyset) \vee \\
& \qquad\ ((s_2, (\eta, g, a), s'_2) \in T_2 \wedge s'_1 = s_1 \wedge K' = K \cup \mathit{lifelines}(\eta, g))) \wedge \\
& \qquad k' = \begin{cases} k+1, & \text{if } (k = 0 \vee k = 1) \wedge \\ & \qquad (s'_{k+1} \in R_{k+1} \vee (s'_{k+1} \in A_{k+1} \wedge R_{1-k} \neq \emptyset)) \\ k \bmod 2, & \text{otherwise} \end{cases}
\end{aligned}
$$

The results of applying the construction for parallel composition and weak sequencing (using an optimised algorithm cutting off states that are unreachable from the initial state) to the interaction in Fig. 2(b) and Fig. 2(a) respectively, are shown in Fig. 2(d) and Fig. 2(c) and show the unrestricted interleaving in comparison with the restricted interleaving of weak sequencing.

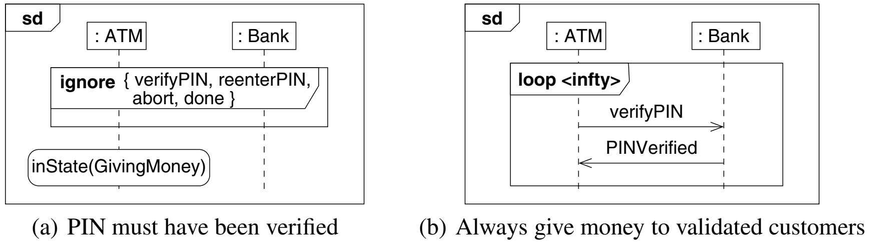
Strict Sequencing, Alternatives, and Ignores. The *strict sequential* composition $N_1 \,;\, N_2$ is achieved by building an automaton which appends N_2 at every accepting state of N_1. The simplicity of the strict sequencing construction for interaction automata also allows for the introduction of an unrestricted loop operator sloop, which enforces strict sequencing of the operand.

In alternative fragments all operands are guarded by either an explicitly given condition, or the implied condition [true]. We integrate the operand automata into a single automaton with guarded transitions from a new initial state to their respective initial states. A similar construction is employed for state invariants.

An ignore fragment specifies which messages are allowed to occur additionally in the traces generated from its operand. This is captured by adding self-loops to every state with the send and receive events from these messages, active for every possible sender or receiver, to the interaction automaton of the operand.

5 Model Checking UML 2.0 Interactions

We apply interactions as observers in model checking by translating the generated interaction automata into observing processes in the model checker SPIN. The system to be observed are message exchanging UML state machines. SPIN is called upon to check whether there is a run of the UML state machines that is accepted by the observer interaction automaton. The translation of UML state machines into SPIN, the translation from UML 2.0 interactions into interaction automata, and the translation of interaction automata into SPIN are integrated into the UML model checking tool HUGO/RT [5].



(a) PIN must have been verified (b) Always give money to validated customers

Fig. 4. Two examples for the ATM case study

Implementation. We use SPIN's accept labels to capture the acceptance conditions of interaction automata both for finite and infinite traces. For infinite traces the accept labels are generated from the recurrent states, for finite traces a special accept label with looping transitions is produced from the acceptance states. The counters of an interaction automaton are represented as variables of the observing process. For recording events the system is instrumented to communicate with the observer via rendezvous channels: Each time a message is sent or received, or a state machine terminates successfully the observer is notified.

In order to keep the size of the SPIN code produced small state sharing is used in the algorithms for basic interactions and loops. Additionally, in the automata from parallel composition and weak sequencing unreachable states are cut off. These optimisations are done on the fly without constructing the product automaton. What is more, the *unwind* algorithm produces rather large automata, even with sharing: In the worst case for n independent events the resulting number of states will be 2^n. Thus it is beneficial to encode a phase not into the states of an interaction automaton, but to employ an external bit-array which encodes the progress of the phases and to use tests on this bit-array for checking whether an event can be accepted.[2]

Verification. Some examples for an automatic teller machine (ATM) case study [14] may show how the additions to interactions in UML 2.0 add to the expressiveness, and thus to the ease of specification and verification of interesting system properties. The two examples in Fig. 4 encode two important properties of the interaction between the system's components ATM and Bank: Figure 4(a) specifies a forbidden scenario; the state invariant that money is dispensed should not be reachable if no PINVerified (all other messages are ignored) has been sent. The interaction in Fig. 4(b) is a required scenario; it must be possible to take money from the ATM infinitely often, as long as the card is valid.

Having specified the interactions and state machines for the system components (see [14]) in the input language of HUGO/RT, the verification process itself is fairly straightforward. HUGO/RT translates the model into a set of SPIN processes and calls SPIN for finding acceptance cycles. For the interaction in Fig. 4(a) no such cycle is found, verifying that the interaction is indeed not satisfiable. For the interaction in

[2] For example, the error scenario in Fig. 15-9 of the telecom case study of Baranov et al. [13] with 19 messages on 5 lifelines amounts to 207 states and 476 transitions in the phase-based translation, but only 2 states and 39 transitions using a bit-array.

Fig. 4(b) an acceptance cycle is found showing that the infinite behaviour is possible. SPIN also produces an example trail, which is retranslated into a human-readable format of UML system states. For these simple examples the translation and model checking take about three seconds on an Intel® Pentium® 4, 3.2 GHz with 2 GB of memory.

6 Related Work

Over time there have been various approaches to formalising scenario descriptions in order to facilitate the analysis of requirements or specifications. Starting from MSCs, Uchitel et al. [15,11] specified semantics for HMSCs, and then developed an approach to synthesise behavioural models in the form of labelled transition systems. Their approach aims at preserving the component structure of the system. This causes their models to allow additional behaviours, which are not explicitly specified in the scenarios, and requires refinement steps to complete the specification [11].

Damm and Harel [16] developed LSCs as a more expressive extension of MSCs. They enrich their specification language with means to express preconditions for scenarios, and facilities to explicitly specify mandatory and forbidden behaviour. Klose [17] proposes an automaton-based interpretation of LSCs and gives an algorithm to create automata out of basic LSCs [17,12]. Bontemps and Heymans [18] formalise automata constructions for strict sequencing, parallel composition, and finite iteration of LSCs. Harel and Maoz [19] propose to port the semantics of LSCs to UML 2.0.

With CHARMY, Autili et al. [20] present a tool based on an approach similar to ours. The focus of CHARMY are architectural descriptions and the verification of their consistency. The semantics of interactions, given by their translation rules, however, deviates substantially from what can be gleaned from the UML 2.0 specification. Furthermore, in the program version we tested, combined fragments are not supported.

7 Conclusions and Future Work

We have presented a translation from UML 2.0 interactions into a special class of automata showing features of finite state automata, Büchi automata and counter automata. These interaction automata have been further translated into concrete programs for model checkers. Together with matching descriptions for UML state machines the approach has been used to model check consistency between the different system descriptions. In some examples we have shown the applicability of the translation procedures to check the satisfiability of scenarios by using the model checker SPIN. The added expressiveness allows the use of our approach to specify properties which before would have required formalisms other than UML interactions.

Since in the current implementation loop and not are restricted in terms of operands, one direction of future work will be to detail to which extent these restrictions can be removed. We also intend to integrate the remaining operators specified by the UML 2.0 specification, which we have disregarded so far. Furthermore, the specification patterns for scenarios described by Autili et al. [20] should be combined with our approach. Finally, we plan to integrate timing constraints and to enhance the translation of interactions into the real-time model checker UPPAAL.

References

1. Object Management Group: Unified Modeling Language: Superstructure, version 2.0. (2005) http://www.omg.org/cgi-bin/doc?formal/05-07-04(06/07/18).
2. International Telecommunication Union: Message Sequence Chart (MSC). ITU-T Recommendation Z.120, ITU-T, Geneva (2004)
3. Cengarle, M.V., Knapp, A.: UML 2.0 Interactions: Semantics and Refinement. In Jürjens, J., Fernandez, E.B., France, R., Rumpe, B., eds.: Proc. 3rd Int. Wsh. Critical Systems Development with UML (CSDUML'04), Technical Report TUM-I0415, Institut für Informatik, Technische Universität München (2004) 85–99
4. Runde, R.K., Haugen, Ø., Stølen, K.: Refining UML Interactions with Underspecification and Nondeterminism. Nordic J. Comp. **12**(2) (2005) 157–188
5. Hugo/RT website: http://www.pst.ifi.lmu.de/projekte/hugo(06/07/18) (2000)
6. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (2003)
7. UPPAAL website: http://www.uppaal.com(06/07/18) (1995)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
9. Alur, R., Yannakakis, M.: Model Checking of Message Sequence Charts. In Baeten, J.C.M., Mauw, S., eds.: Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99). Volume 1664 of Lect. Notes Comp. Sci., Springer (1999) 114–129
10. Leue, S., Ladkin, P.B.: Implementing and Verifying MSC Specifications Using Promela/XSpin. In Gregoire, J.C., Holzmann, G.J., Peled, D., eds.: Proc. 2nd Int. Wsh. SPIN Verification System (SPIN'96). Volume 32 of Discrete Mathematics and Theoretical Computer Science., American Mathematical Society (1997) 65–89
11. Uchitel, S., Kramer, J., Magee, J.: Incremental Elaboration of Scenario-based Specifications and Behavior Models using Implied Scenarios. ACM Trans. Softw. Eng. Methodol. **13**(1) (2004) 37–85
12. Brill, M., Damm, W., Klose, J., Westphal, B., Wittke, H.: Live Sequence Charts. In Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E., eds.: Integration of Software Specification Techniques for Applications in Engineering. Volume 3147 of Lect. Notes Comp. Sci., Springer (2004) 374–399
13. Baranov, S., Jervis, C., Kotlyarov, V., Letichevsky, A., Weigert, T.: Leveraging UML to Deliver Correct Telecom Applications. In Lavagno, L., Martin, G., Selic, B., eds.: UML for Real. Kluwer (2003) 323–342
14. Schäfer, T., Knapp, A., Merz, S.: Model Checking UML State Machines and Collaborations. In Stoller, S., Visser, W., eds.: Proc. Wsh. Software Model Checking. Volume 55(3) of Elect. Notes Theo. Comp. Sci., Paris (2001) 13 pages.
15. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Trans. Softw. Eng. **29**(2) (2003) 99–115
16. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Meth. Sys. Design **19**(1) (2001) 45–80
17. Klose, J.: Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behaviour. PhD thesis, Carl von Ossietzky-Universität Oldenburg (2003)
18. Bontemps, Y., Heymans, P.: Turning High-Level Live Sequence Charts into Automata. In: Proc. ICSE Wsh. Scenarios and State-Machines: Models, Algorithms and Tools (SCESM'02), Orlando (2002)
19. Harel, D., Maoz, S.: Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. In: Proc. 5th Int. Wsh. Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06), ACM Press (2006) 13–20
20. Autili, M., Inverardi, P., Pelliccione, P.: A Scenario Based Notation for Specifying Temporal Properties. In: Proc. 5th Int. Wsh. Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06), ACM Press (2006) 21–27