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.

1 Introduction

Scenario-based development uses descriptions of operational sequences to define the requirements of software systems, laying down required, allowed, or forbidden behaviours. Sufficiently expressive variants of scenario languages, like Live Sequence Charts (LSCs [1]), aim at complete system descriptions, whereas scenarios in weaker dialects, like High-Level Message Sequence Charts (HMSCs [2]) are kept for validation purposes throughout software development [3], or for synthesising at least partial system behaviour [4].

In version 2.0 [5] of the "Unified Modeling Language" (UML), the lingua franca of software engineering, a variation of HMSCs 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, in particular concerning the new negation operator, have led to several, differing efforts for equipping UML 2.0 interactions with a formal semantics [6–9].

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. On the other hand, besides these uncontroversial standard constructs, we also handle a classical negation operator [7], which avoids the introduction of three-valued logics as suggested by the

UML 2.0 specification by resorting to binary logic. Furthermore, we also allow for potentially and mandatorily infinite loops.

In contrast to the process-centred approach by Leue and Ladkin [3] and building on the work by Klose and Wittke [10], we use a single, global automaton for representing an interaction, simplifying the handling of alternatives and loops. However, in view of the general undecidability of the model checking problem for HMSCs [11], we restrict negation and weak sequencing loops to contain only basic interactions. Hence the automata to be negated are deterministic and loop progress can be captured by counters. As a substitute for weak sequencing loops we offer general strict sequencing loops.

The translation procedure is integrated into our freely available UML model checking tool HUGO/RT [12]: 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 [13] and, partially, with UPPAAL [14].

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*. These fragments can be *occurrence specifications*, specifying the occurrence of events within an object that is participating in the interaction. Sets of occurrence specifications, which we call basic interactions, and *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.

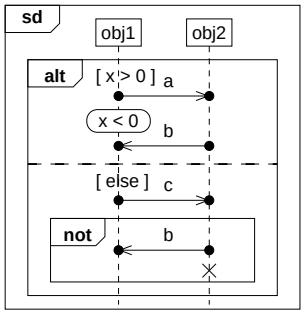


Fig. 1. Sample interaction.

The example in Fig. 1 shows instances of the important aspects of a UML 2.0 interaction. The two objects obj1 and obj2 exchange messages, specified by *message occurrence specifications* on their respective *lifelines*.[3] Weak sequencing is implicit in the second operand of the alternative, such that 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 the operand

[3] The dots at the beginning and end of each message arrow are not part of the graphical syntax of UML interactions. We included them in this example to highlight where message occurrence specifications occur.

that needs to be chosen at "runtime". Furthermore, the two operators (alt and not) occur on different levels of nesting, showing how larger, nested interactions can be composed.

The primitive interaction fragments we consider are basic interactions and state invariants. A *basic interaction* consists of a set of event occurrences with a partial order subject to the following constraints: The dispatch of a message, the *send event*, occurs before the arrival of the message, the *receive event*; all event occurrences active for the same lifeline are totally ordered; and a *termination event* (cross), if any, must be the last event on its lifeline. A *state invariant* defines a condition (in a rounded box) for a single or several lifelines that has to hold if the state invariant is reached.

Moreover, UML 2.0 puts a number of interaction-building operators at the user's disposal. In sequential composition, the behaviour of the resulting interaction is the behaviour of the first operand followed by the behaviour of the second operand. There are two kinds of sequential composition, which differ in the meaning of the word "followed". *Strict composition*, denoted by the operator strict, requires the behaviour of the first interaction to be completely performed before starting with the behaviour of the second interaction. *Weak composition* (implicit or explicitly with seq) only requires the behaviour specified for an object in the first interaction to be completely performed before starting with the behaviour for that object in the second interaction.

Other operators currently supported by our implementation are parallel composition (par), alternative (alt), loop, ignore, and negation (not). Two *parallel* interactions are to be executed by interleaving. An *alternative* means to execute one of two given interactions; this may be a non-deterministic choice if the conditions of the alt fragment overlap. A *loop* repeatedly executes its operand between bounds m and n, where it has to iterate at least m times and at most n times. Standardly only the upper bound may be infinity. We, however, also handle loops where also the lower bound is infinite. If a loop requires k iterations, this amounts to the k-fold weak sequencing of the operand. We restrict the use of weak sequencing loops to contain only a basic interaction, but we also support an unrestricted operator sloop, which enforces strict sequencing of the operand. The *negation* operator not is intended for the specification of forbidden behaviour and presents a simpler variant of the controversial neg operator of UML 2.0: Everything but the precise behaviour of its operator is valid for not. Finally, ignore allows additional messages to occur besides the ones specified in its operand.

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 message 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. Whenever the system state changes the observer may make an according move, but may also refrain from doing so, if it does not deem the state change relevant. Taking such an observer from an interaction 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 [15]: The observer accepts a finite system behaviour, if it can make a finite sequence of moves to

a final state while recording the system changes; it accepts an infinite system behaviour, if it can make an infinite sequence of moves visiting a recurrent state infinitely often.

The system behaviour as seen by an observing interaction can be captured in a finite set of involved lifelines L, a set E of termination, send, and receive events from the messages exchanged between the lifelines, and a set Σ of possible system states. We call an *interaction alphabet* a triple (L, E, Σ) of such sets.

Interaction automata are defined over such an interaction alphabet and realise a finite description of an observer by a state-transition system. 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 and the guard of a transition show a set of lifelines, which are active when making a move by this transition. Finally, an interaction automaton may also use and manipulate a set of counters that allow to record how often lifelines in loops have executed.

For the guards we therefore assume a propositionally closed language $\mathscr{G}_{\Sigma,V}$ for a set Σ of system states and a set V of counters; it should be expressive enough to capture system state queries on Σ and to compare counters in V. We write $\sigma, \upsilon \models g$ to mean that system state $\sigma \in \Sigma$ and a valuation $\upsilon : V \to \mathbb{N}$ of the counters in V satisfy $g \in \mathscr{G}_{\Sigma,V}$. The lifelines active in a set of events η and a guard g are denoted by $\mathit{lifelines}(\eta, g)$. Similarly to guards, we assume a language $\mathscr{A}_V$ of actions for manipulating a set of counters V; it should be possible to increment and reset a single or several counters. We write $(\upsilon, \upsilon') \models a$ to mean that $a \in \mathscr{A}_V$ transforms the valuation υ into the valuation υ'.

With these preliminaries, our notion of interaction automata as observers can be defined formally as follows:

Definition 1. *An* interaction automaton *over an interaction alphabet (L, E, Σ) is given by a tuple (S, V, T, i, A, R) where S is a finite set of* states*, V a finite set of* counters*, $T \subseteq S \times (\wp E \times \mathscr{G}_{\Sigma,V} \times \mathscr{A}_V) \times S$ a finite* transition relation*, $i \in S$ the* initial state*, $A \subseteq S$ a set of* accepting states*, and $R \subseteq S$ a set of* recurrence states*. For a transition $(s, (\eta, g, a), s') \in T$, η is the set of* accepted events*, g the* guard condition*, and a the* action*.*

Let $N = (S, V, T, i, A, R)$ be an interaction automaton over (L, E, Σ). A configuration *of N is a pair (s, υ) with $s \in S$ and $\upsilon \in V \to \mathbb{N}$; the* initial configuration *of N is given by (i, υ_0) where $\upsilon_0(v) = 0$ for all $v \in V$. Given a configuration $\gamma = (s, \upsilon)$ we write $s(\gamma)$ for s and $\upsilon(\gamma)$ for υ. The interaction automaton N can make a* step *from configuration γ to configuration γ' with state $\sigma \in \Sigma$ and event set $\zeta \subseteq E$, written as $\gamma \xrightarrow{(\sigma,\zeta)}_N \gamma'$, if there are $\zeta \subseteq \eta \subseteq E$, $g \in \mathscr{G}_{\Sigma,V}$, and $a \in \mathscr{A}_V$ such that $(s(\gamma), (\eta, g, a), s(\gamma')) \in T$, $\sigma, \upsilon(\gamma) \models g$, and $(\upsilon(\gamma), \upsilon(\gamma')) \models a$.*

The interaction automaton N accepts *the finite trace $o_0 \dots o_{n-1} \in (\Sigma \times \wp E)^*$, if there is a sequence $\gamma_0, \gamma_1, \dots, \gamma_n$ of configurations of N such that γ_0 is the initial configuration of N and $\gamma_j \xrightarrow{o_j}_N \gamma_{j+1}$ for all $0 \le j \le n-1$ and $s(\gamma_n) \in A$.*

The interaction automaton N accepts *the infinite trace $o_0 o_1 \dots \in (\Sigma \times \wp E)^\infty$, if there is a sequence $\gamma_0, \gamma_1, \dots$ of configurations of N such that γ_0 is the initial configuration of N and $\gamma_j \xrightarrow{o_j}_N \gamma_{j+1}$ for all $0 \le j$ and there is an $r \in R$ such that $\sharp\{j \mid s(\gamma_j) = r\}$ is infinite.*

In concrete representations of interaction automata, we show states as ellipses and transitions as arrows. Accepting and recurrence states are doubly and triply outlined, respectively. Transition inscriptions show (a description of) the set of accepted events η, the guard condition g, and the action a in the format $\eta[g]/a$; any of these parts may be omitted and the active lifelines are left implicit.

It may be noted that although we define interaction automata to be finitely represented, the configuration space of an interaction automaton may be infinite due to unbounded increases of counter values. In particular, a procedure for complementing an interaction automaton becomes hard to devise. 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 [11], 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 [6–9]. Furthermore, we handle an sloop operator for unrestricted loop composition by strict sequencing, and a not operator based on binary logic; not and loop are restricted to basic interactions.

In contrast to other approaches (e.g. [3, 16]) we propose not to generate one automaton for every object in an interaction, but to generate only a single observing interaction automaton for the entire interaction. This single automaton represents the property to be checked by a model checker. Brill et al. [17] discuss the generation of automata for basic LSCs. Our translation of basic interactions is inspired by their algorithm. However, the subset of UML 2.0 interactions that we use does not show all the features of LSCs, like liveness or simultaneous regions. Hence, we can simplify the algorithm for some cases. As it turns out, with a few extensions this basic algorithm can also be used for generating automata for loop fragments. The two sequencing operators and parallel composition can be handled by adapting algorithms for Büchi automata to interaction automata, where, in fact, weak sequencing can be seen as a restricted variant of parallel composition. Alternatives and the ignoring of messages only require the addition of some transitions to the interaction automata of the operands.

4.1 Basic Interactions, State Invariants, Loops, and Negation

We first describe the translation of basic interactions, state invariants, 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.

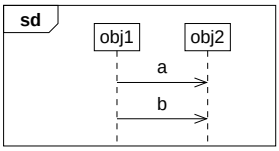
Basic Interactions. The translation of basic interactions is performed by unwinding the partial order of events. The partial order is given by the rules that for each message its sending event must occur before its receiving event, and that events on a lifeline are totally ordered with an optional termination event at the end. For each event the partial

order thus defines the prerequisite events, which must be unwound before that event can be unwound. The unwinding of basic interactions is performed in *phases* [17]. In every phase there exists a *history*, i.e., a set of events which have been unwound already. This history determines which other events can potentially be unwound in the next step. They are delivered by a function *ready* of a phase in the form of an inscription of a transition in an interaction automaton. 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.

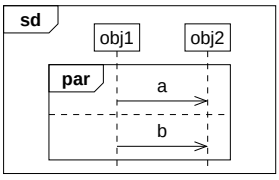
The following algorithm *unwind* transforms phases directly into states of an interaction automaton *result*. The procedure is started with the initial phase of a basic interaction with an empty history, and the state returned by this call to *unwind* becomes the initial state of the resulting interaction automaton. The algorithm terminates when there are no ready events in a phase, expressed by a predicate *isAccepting*. These phases correspond to the accepting end states of the automaton.

```
1 unwind(phase, result) ≡
2   ⌈state ← addState(result)
3    if isAccepting(phase) then addAcceptingState(result, state) fi
4    for label ∈ ready(phase) do
5        addTransition(result, state, label,
6                      unwind(nextPhase(phase, label), result))
7    od
8    return state⌋
```

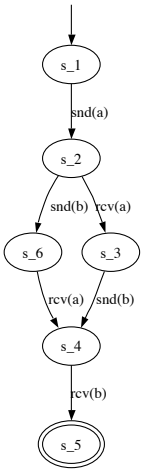
Figure 2(a) shows an example of a basic interaction, the interaction automaton in Fig. 2(c) shows the effects of unwinding its partial order. The branching in `s_2` is due to the fact that the second event can be either the reception of a or the sending of b.



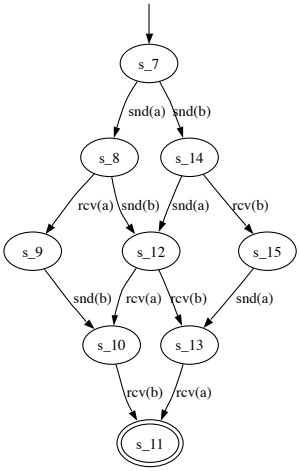
(a) Basic interaction.



(b) par fragment.



(c) Automaton generated for 2(a).



(d) Automaton generated for 2(b).

Fig. 2. A simple basic diagram and the generated automaton.

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. This way we can specify loops that must not terminate. However, we restrict loops to contain only a basic interaction.

For finite or infinite loops of a basic interaction the algorithm for unwinding a basic interaction 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. These additional conditions and actions for unwinding an event can be expressed as labels of transitions in an interaction automaton.

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 the counters are 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 also have to introduce a recurrent state which

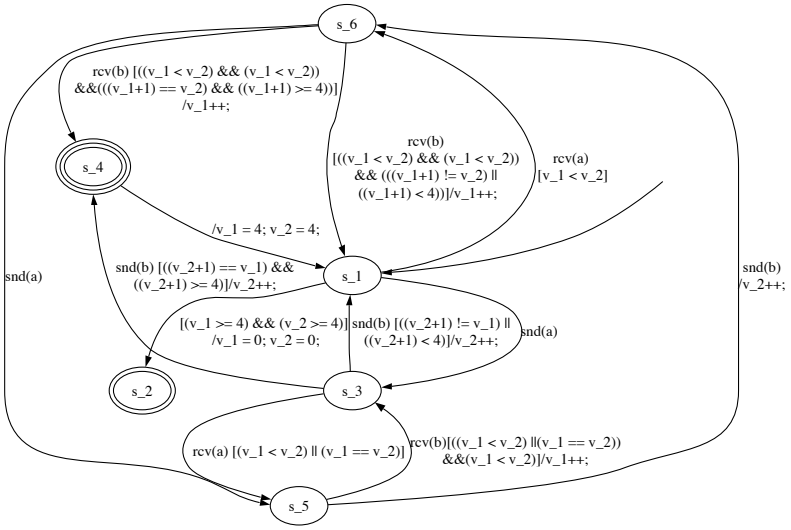
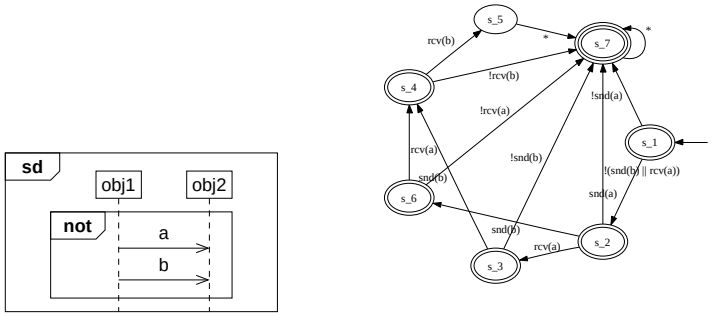


Fig. 3. The automaton for an infinite loop.



(a) UML diagram. (b) Generated automaton.

Fig. 4. A simple diagram with not and the generated automaton.

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

The particular values of the counters when running through a recurrent state become irrelevant, as it has to be reached an infinite number of times. Therefore, the counters can be reset, either to the lower bound, if this is finite, or to zero.

Figure 3 shows an example of an automaton for a loop.[4] The example is based on the basic interaction in Fig. 2(a), wrapped into a loop $\langle 4, \infty \rangle$. Following the possible message flows shows which additional states need to be created and how the guards ensure the partial order across several iterations.

It has to be noted that even for small numbers in the lower and the upper bound the explicit unfolding of a loop by weak sequencing its basic interaction with itself the required number of times (see, e.g., [18]) would become prohibitively large.

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, in particular involving counters, 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 accepting state is added, and from all complemented states transitions to this accepting state are added which accept all events that were not accepted in the corresponding state of the original automaton. This additional accepting state is also recurrent with a self-loop with all possible events.

As an example Fig. 4(b) depicts the result of applying the negation algorithm to the automaton in Fig. 2(c), that is, the result of translating the interaction in Fig. 4(a).

[4] Because of space limitations the annotations on the transitions are abbreviated, but reflect the structure and necessary guard conditions.

The expressive power of the translation of not fragments could be greatly enhanced by adopting a general negation construction for non-deterministic Büchi automata [19]; however, the size of the resulting automaton would be exponential in the size of the input automaton.

State Invariants. A state invariant defines a condition that must be met whenever a run through the interaction reaches the invariant. In UML 2.0, such a state invariant covers a single lifeline, but we treat more general invariants that may spawn several lifelines; see, e.g., [20]. The automaton construction is straightforward: A single transition going from the initial state to an accepting state shows the condition of the state invariant as its label.

4.2 Interleaving and Sequencing

For two interaction automata N_1 and N_2 over a common interaction alphabet (L, E, Σ), we describe the parallel, weak sequential, and strict sequential composition of these automata. These operations reflect the semantics of par, seq, and strict, respectively. We also introduce a general strict sequencing variant sloop of loops.

Acceptance of an interaction automaton from parallel composition means that the trace $o_0 o_1 \cdots \in (\Sigma \times \wp E)^* \cup (\Sigma \times \wp E)^\infty$ has to be representable by an interleaving of traces $o_0^{(1)} o_1^{(1)} \ldots$ and $o_0^{(2)} o_1^{(2)} \ldots$ accepted by N_1 and N_2 respectively. An interaction automaton obtained by weak sequential composition also accepts 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, but in the interleaving no $o_j^{(2)}$ is allowed to occur before an $o_k^{(1)}$ if their active lifelines overlap. Finally, an interaction automaton arising by strict sequential composition accepts a trace $o_0 o_1 \ldots$, if N_1 accepts a finite or infinite prefix of this trace and, if the prefix is finite, N_2 accepts the remainder.

Parallel Composition. The *parallel* composition $N_1 \parallel N_2$ uses a construction very similar to the parallel composition of Büchi automata [15], where a special counter k is employed to record if first N_1 and then N_2 are running through a recurrent state and only if both automata have run through a recurrent state, $N_1 \parallel N_2$ goes through a recurrent state. This construction only has to be adapted to cover that both or one of the interaction automata do not show recurrence states.

More precisely, 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 parallel, interleaving composition $N_1 \parallel N_2$ is the interaction automaton $(S, V_1 \cup V_2, T, i, A_1 \times A_2 \times \{0\}, S_1 \times S_2 \times \{2\})$ with

$$
\begin{aligned}
&S = S_1 \times S_2 \times \{0,1,2\}, \qquad i = (i_1, i_2, 0)\\
&((s_1, s_2, k), (\eta, g, a), (s_1', s_2', k')) \in T \iff\\
&\qquad (((s_1, (\eta, g, a), s_1') \in T_1 \wedge s_2' = s_2) \vee ((s_2, (\eta, g, a), s_2') \in T_2 \wedge s_1' = s_1)) \wedge\\
&\qquad k' = \begin{cases} k+1, & \text{if } (k = 0 \vee k = 1) \wedge \\ & \qquad (s_{k+1}' \in R_{k+1} \vee (s_{k+1}' \in A_{k+1} \wedge R_{1-k} \neq \emptyset)) \\ k \bmod 2, & \text{otherwise} \end{cases}
\end{aligned}
$$

The result of applying the construction for parallel composition (using an optimised algorithm cutting off states that are unreachable from the initial state) to the interaction in Fig. 2(b) is shown in Fig. 2(d) and shows the unrestricted interleaving in comparison with the automaton in Fig. 2(c).

Weak Sequencing. A slight modification of the construction for parallel composition can be used for obtaining the *weak sequential* composition $N_1 \mathbin{;_\ll} N_2$ of N_1 and N_2: The full interleaving must be restricted such that no transition from N_2 covering lifelines K overtakes a transition from N_1 covering some lifeline in K. Therefore, the states of $N_1 \mathbin{;_\ll} N_2$ show an additional component of sets of lifelines, recording which lifelines have been covered by the interleaving of N_2; in a state with lifeline component K, a transition from N_1 is only possible, if its covered lifelines are not in K.

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 $N_1 \mathbin{;_\ll} N_2$ is the interaction automaton $(S, V_1 \cup V_2, T, i, A, R)$ with

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

The automaton resulting from an application of the construction (again optimised to cut off unreachable states and states from which no accepting state is reachable) to an interaction like in Fig. 2(b), but replacing par with seq, is the same as in Fig. 2(c).

Strict Sequencing. The *strict sequential* composition $N_1 \mathbin{;} N_2$ 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 strict sequencing loops by an sloop operator: If the lower bound of sloop is infinity, the accepting state of the interaction automaton from the operand to sloop is connected to its initial state and this initial state is turned into a recurrent state. If the lower bound of sloop is not infinity, a counter, a new initial and a new accepting state are added, and all formerly accepting states become non-accepting. The new initial and the accepting state are connected with a transition increasing the counter. The new accepting state is targeted from the new initial, subject to the condition that at least the minimal number of iterations and at most the maximal number of iterations, if finite, has occurred. If the upper bound is infinite the new initial state becomes a recurrent state, and a transition resetting the counter if the minimal number of transitions has occurred is introduced.

4.3 Alternatives and Ignores

Alternatives. In alternative fragments all operands are guarded by either an explicitly given condition, or the implied condition [true]. Because several operands can have overlapping guards, in general automata with alternatives are non-deterministic. However, the semantics requires that in each run of the system at most one operand of the alternative is executed. Thus, given interaction automata for each operand of an alt fragment, we integrate them into a single automaton with guarding transitions from a new initial state. The guards of these transitions show the conditions of the operands. This implies that our implementation ignores the possibility of delayed choice as proposed in the MSC standard semantics [2].

Ignores. 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.

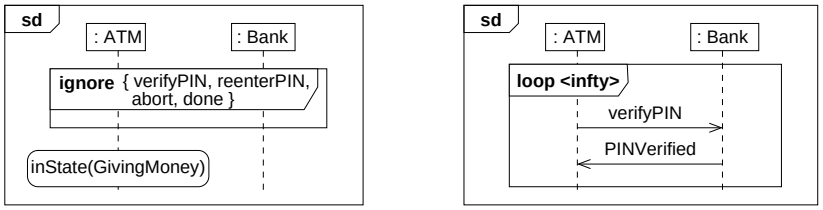
5 Model Checking UML 2.0 Interactions

We apply interactions as observers in model checking by translating the interaction automata generated from interactions by the procedure above as observing processes in the off-the-shelf model checker SPIN. The system to be observed are message exchanging UML state machines. SPIN is called upon to model 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 as well as the translation of interaction automata into SPIN are integrated into the UML model checking tool HUGO/RT [12].

5.1 Implementation

SPIN offers the free use of accept labels in a process, one of which must be visited infinitely often in a model checking run in order to produce an acceptance cycle. We use this mechanism 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 object terminations and the messages exchanged in the observed system, 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.

The implementation of the translation of UML 2.0 interactions into interaction automata employs several optimisations in order to keep the size of the SPIN code produced small. First of all, state sharing is used in the algorithms for basic interactions and loops. In the automata from parallel composition and weak sequencing several states



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

Fig. 5. Two examples for the ATM case study.

can be cut off: those unreachable from the initial state; and states from which no accepting state is reachable and no infinite loop through a recurrent state is possible. 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: For n independent events on n lifelines the resulting number of states will be 2^n; for n consecutive messages from one lifeline to another the number of states is $(n+2)(n+1)/2$. 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.[5]

5.2 Verification

Some examples for an automatic teller machine (ATM) case study [21] 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 UML state machines representing the system model (see Fig. 6 in the appendix) show the supported messages of the two system components Bank and ATM. The two examples in Fig. 5 encode two important properties of the system: Figure 5(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. 5(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 in the input language of HUGO/RT, a textual UML description language called UTE, 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. 5(a) no such cycle is found, verifying that the interaction is indeed not satisfiable. For the interaction in Fig. 5(b) an acceptance cycle is found showing that the infinite behaviour is possible. The SPIN code for the observer process in this case takes the following form:

```
proctype Observer() {
  int v_1; int v_2; bit v_3[4];
```

[5] For example, the error scenario in Fig. 15-9 of the telecom case study of Baranov et al. [20] 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.

```
bit direction; byte sender; int behavioural; byte receiver;

nc_s_1:
if
:: observer?direction,sender,receiver,behavioural ->
   if
   :: direction == SEND && behavioural == send_pinVerified &&
      sender == obj_bank && receiver == obj_atm &&
      (!v_3[0] && v_3[1] && !v_3[2] && !v_3[3]) && v_2+1 == v_1 ->
      atomic { v_2++; v_3[0] = 0; v_3[1] = 0; goto accept_nc_s_2 }
   :: direction == RECEIVE && behavioural == send_pinVerified && ...
   ...
fi;
accept_nc_s_2:
if
:: true -> atomic { v_1 = 0; v_2 = 0; goto nc_s_1 }
fi
}
```

where `bit v_3[4]` represents the bit-array for storing the history of the underlying basic interaction. SPIN produces an example trail and this is retranslated into a human-readable format of UML system states. For both 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. [4, 16] specified semantics for HMSCs, and then developed an approach to synthesise behavioural models in the form of labelled transition systems. Their approach does not focus on trace equivalence between the scenario specification and the synthesised behavioural model, but aims at preserving the component structure of the system. This causes their models to allow additional behaviours, which are not specified in the scenarios, and requires refinement steps to complete the specification [16].

Damm and Harel [1] propose 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 [22] and Klose and Wittke [10] propose an automaton-based interpretation of LSCs and give an algorithm to create automata out of basic LSCs [22, 17]. However, they do not give an algorithm for high-level LSCs, and thus lack the ability to compose verifiable models from more than basic charts. The issues surrounding the rather peculiar UML 2.0 negation operator have led to various attempts to define a formal semantics for UML interactions [6–9]. In contrast, Harel and Maoz [23] propose to port the semantics of LSCs to UML 2.0.

With CHARMY Pelliccione et al. [24, 18] present a tool based on an approach similar to ours. The focus of CHARMY are architectural descriptions and the verification of their consistency. For this UML 2.0 is extended, aiming to integrate the missing features from LSCs into UML 2.0. In CHARMY also state machines and interactions are used to define the system model and properties. The semantics of interactions, given by 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, limiting their expressiveness considerably.

The LTSA approach of Uchitel et al. [16] is based on (H)MSCs as scenario specification language. As their goal is to derive the necessary system structure from the scenarios, no additional system specification in the form of state machines is assumed. Therefore, the kind of verification we are attempting with HUGO/RT is not possible with LTSA. With the few scenarios in our example case study it is also impossible to make LTSA create an architecture automaton that resembles our state machines. Therefore, there cannot be a direct comparison of the results from LTSA and our approach.

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. Furthermore, many of the operators now also allow shorter diagrams, because the new syntax makes explicit unwinding or repetition unnecessary.

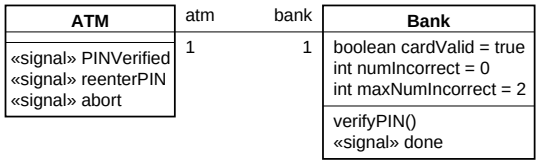
The utility of this approach generally arises from the possibility to compose and modify simple interactions by using operators such as alt, loop and not. Since in the current implementation loop and not are restricted in terms of operands, one direction of future work will be to detail to which extent these restrictions can be removed. Also, the operators consider, break, critical, and assert, specified by the UML 2.0 specification, have been disregarded and we intend to integrate them in the future. Furthermore, the specification patterns for scenarios described by Autili et al. [18] 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

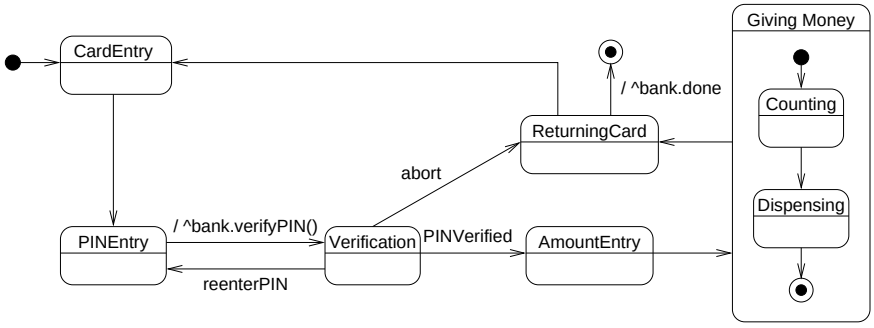
1. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Meth. Sys. Design **19** (2001) 45–80
2. International Telecommunication Union: Message Sequence Chart (MSC). ITU-T Recommendation Z.120, ITU-T, Geneva (2004)
3. Leue, S., Ladkin, P.B.: Implementing and Verifying MSC Specifications 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
4. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Trans. Softw. Eng. **29** (2003) 99–115

5. 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).
6. Störrle, H.: Semantics of Interactions in UML 2.0. In: Proc. IEEE Symp. Human Centric Computing Languages and Environments (HCC'03), IEEE Computer Society (2003) 129–136
7. 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
8. Cavarra, A., Küster-Filipe, J.: Formalizing Liveness-Enriched Sequence Diagrams Using ASMs. In Zimmermann, W., Thalheim, B., eds.: Proc. 11th Int. Wsh. Abstract State Machines (ASM'04). Volume 3052 of Lect. Notes Comp. Sci., Springer (2004) 62–77
9. Runde, R.K., Haugen, Ø., Stølen, K.: Refining UML Interactions with Underspecification and Nondeterminism. Nordic J. Comp. **12** (2005) 157–188
10. Klose, J., Wittke, H.: An Automata Based Interpretation of Live Sequence Charts. In Margaria, T., Yi, W., eds.: Proc. 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01). Volume 2031 of Lect. Notes Comp. Sci., Springer (2001) 512–527
11. 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
12. Hugo/RT web site: http://www.pst.ifi.lmu.de/projekte/hugo(06/07/18) (2000)
13. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (2003)
14. UPPAAL web site: http://www.uppaal.com(06/07/18) (1995)
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
16. Uchitel, S., Kramer, J., Magee, J.: Incremental Elaboration of Scenario-based Specifications and Behavior Models using Implied Scenarios. ACM Trans. Softw. Eng. Methodol. **13** (2004) 37–85
17. 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
18. 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
19. Safra, S.: On the Complexity of omega-Automata. In: Proc. 29th IEEE Symp. Foundations of Computer Science (FOCS'88), IEEE Computer Society (1988) 319–327
20. 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
21. 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.
22. Klose, J.: Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behaviour. PhD thesis, Carl von Ossietzky-Universität Oldenburg (2003)
23. 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
24. Pelliccione, P.: Charmy: A Framework for Software Architecture Specification and Analysis. PhD thesis, Università degli Studi dell'Aquila (2005)

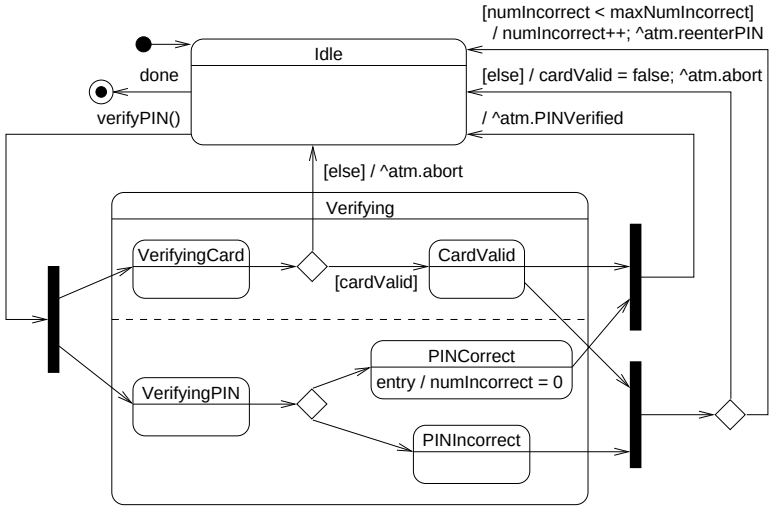
A ATM Example



(a) Class diagram



(b) State machine diagram for class ATM



(c) State machine diagram for class Bank

Fig. 6. UML model of an ATM