Applying Software Model Checking Techniques for Behavioral UML Models

Orna Grumberg[1], Yael Meller[1], and Karen Yorav[2]

[1] Computer Science Department, Technion, Haifa, Israel
{orna,ymeller}@cs.technion.ac.il
[2] IBM Haifa Research Laboratory, Haifa, Israel
yorav@il.ibm.com

Abstract. This work presents a novel approach for the verification of Behavioral UML models, by means of software model checking.

We propose adopting *software model checking* techniques for verification of UML models. We translate UML to *verifiable* C code which preserves the high level structure of the models, and abstracts details that are not needed for verification. We combine of static analysis and bounded model checking for verifying LTL safety properties and absence of livelocks.

We implemented our approach on top of the bounded software model checker CBMC. We compared it to an IBM research tool that verifies UML models via a translation to IBM's hardware model checker RuleBasePE. Our experiments show that our approach is more scalable and more robust for finding long counterexamples. We also demonstrate the usefulness of several optimizations that we introduced into our tool.

1 Introduction

This work presents a novel approach for the verification of Behavioral UML models, by means of software model checking.

The *Unified Modeling Language* (UML) [4] is a widely accepted modeling language that is used to visualize, specify, and construct systems. It provides means to represent a system as a collection of objects and to describe the system's internal structure and behavior. UML has been accepted as a standard object-oriented modeling language by the Object Management Group (OMG) [12]. It is becoming the dominant modeling language for embedded systems. As such, the correct behavior of systems represented as UML models is crucial and verification techniques for such models are required.

Model checking [6] is a successful automated verification technique for checking whether a given system satisfies a desired property. Model checking traverses *all* system behaviors, and either confirms that the system is correct w.r.t. the checked property, or provides a *counterexample* demonstrating an erroneous behavior.

Model checking tools expect the checked system to be presented in an appropriate description language. Previous works on UML model checking translate UML models to SMV [5,7] or VIS[1] [25], both particularly suitable for hardware; to PROMELA (the input language of SPIN) [17,16,20,10,1,14,11]), which is mainly suitable for communication protocols; or to IF^3 [18], which is oriented to real-time systems.

[1] These works were developed as part of the European research project OMEGA [19].

We believe that behavioral UML models mostly resemble high-level software systems. We therefore choose to translate UML models to C and adopt *software model checking* techniques for their verification. Our translation indeed preserves the high-level structure of the UML system: event-driven objects communicate with each other via an *event queue*. An execution consists of a sequence of *Run To Completion* (RTC) steps. Each RTC step is initiated by the event queue by sending an event to its target object, which in turn executes a maximal series of enabled transitions.

Model checking assumes a finite-state representation of the system in order to guarantee termination with a definite result. One approach for obtaining finiteness is to *bound* the length of the traversed executions by an iteratively increased bound. This is called *Bounded Model Checking* (BMC) [3]. BMC is highly scalable, and widely used, and is particularly suitable for bug hunting. We find this approach most suitable for UML models, which are inherently infinite due to the unbound size of the event queue[2].

We emphasize that our goal is to translate the UML model into *verifiable* C code that suits model checking, rather than produce executable code. Also, we only wish to verify user-created artifacts. When translating to C, we therefore simplify implementation details that are irrelevant for verification. For instance, the event queue is described at a high level of abstraction, and code is sometimes duplicated to avoid pointers and simplify the verification. The resulting code is significantly easier for model checking than automatically generated code produced by UML tools such as Rhapsody [23].

Recall that the verifiable C code will be checked by BMC with some bound k. We choose k to count the number of RTC steps. This implies that along an execution of size k only the first k events in the queue are consumed, even if more were produced. It is therefore sufficient to hold an event queue of size k. We thus obtain a finite-state model without losing any precision. Counterexamples are also returned as a sequence of RTC steps, but zooming in to intermediate states is available upon request.

We verify two types of properties: LTL safety properties and livelocks. Safety properties require that the system never arrives at bad states, such as deadlock states, states violating mutual exclusion, or states from which the execution can continue nondeterministically. *LTL safety properties* can further require that no undesired finite execution occurs. Checking (LTL) safety properties can be reduced to traversing the reachable states of the system while searching for bad states. We apply *Bounded reachability* with increasing bounds for finding bad states. Our method can also be extended to proving the absence of bad states, using k-induction [26].

Another interesting type of properties is the absence of livelocks. *Livelocks* are a generalization of deadlocks. While in *deadlock* states the *full system* cannot progress, in livelock states part of the system is "stuck" forever while other parts continue to run. Livelocks can be hazardous in safety critical systems and often indicate a faulty design.

Scalable bounded model checking tools mostly handle *safety* or linear-time properties. However, absence of livelocks is neither safety nor linear-time property and is therefore not amenable to bounded model checking. We identify an important subclass of livelocks, which we refer to as *cycle-livelocks*, and show that they can be found by combining static analysis and bounded reachability.

[2] Variables are treated as finite width bit vectors and therefore do not hurt the model finiteness.

The property of deadlock has been the subject of many works. In the context of UML, [15] presents model checking for deadlocks via process algebra. The SPIN model checker itself supports checking for deadlocks. To the best of our knowledge, absence of livelocks has never been verified in the context of behavioral UML models.

We implemented our approach to verifying UML models with respect to LTL safety properties and cycle-livelocks in a tool called soft-UMC (**soft**ware-based **U**ML **M**odel **C**hecking). Our tool is built on top of the software model checker CBMC [8] which applies BMC to C programs and safety properties. We ran it on several UML examples and interesting properties, and found erroneous behaviors and livelocks. For safety properties, we also compared soft-UMC with an IBM research tool that verifies UML models via a translation to IBM's hardware model checker RuleBasePE [24]. Our experiments show that soft-UMC is more scalable and more robust for finding long counterexamples. Our experimental results also demonstrate the usefulness of the optimizations applied in the creation of the verifiable C code.

The rest of the paper is organized as follows. In Sec. 2 we present some background. Our translation to verifiable C code is presented in Sec. 3, and our method for verification of (LTL) safety properties and cycle-livelocks is presented in Sec 4. We show our experimental results in Sec 5, and conclude in Sec. 6.

2 Preliminaries

2.1 Behavioral UML Models

We use a running example describing a flight ticket ordering system to explain UML. The *class diagram* in Fig. 1(a)[3] shows the classes *DB* and *Agent* and the connection between them. The *object diagram* in Fig. 1(b) defines four objects, two of each class. These diagrams also show the attributes (variables) of each class and their *event receptions*. E.g., objects of class DB have two attributes ($isMyFlt$ and $space$) and are able to receive events of type $evReqOwnership$, $evReqFlt$, and $evGrantOwnership$.

UML objects process events. Event processing is defined by *statecharts* [13], which extend conventional state machines with hierarchy, concurrency and communication. The statecharts of DB and Agent classes are presented in Fig. 2.

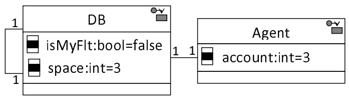
Objects communicate by sending events (asynchronous messages). An event is a pair $(ev, trgt)$, where ev is the type of the sent event and $trgt$ is the target object of the event. Events are kept in an *event queue* (EQ), managed by an *event queue manager* (EQ-mgr). When object A sends an event to object B, the event is inserted into the EQ. The EQ-mgr executes a never-ending event-loop, taking an event from the EQ, and dispatching it to the target object. If the target object cannot process the event, the event is *discarded*. Otherwise, the event is *consumed* and the target object makes a *run-to-completion (RTC) step*, where it processes the event, and continues execution until it cannot continue anymore. Only when the target object finishes its RTC step, the EQ-mgr dispatches the next event available in the EQ[4].

[3] We used Rhapsody [23] to generate the drawings in this paper, and will accordingly use some of Rhapsody's terms and conventions.

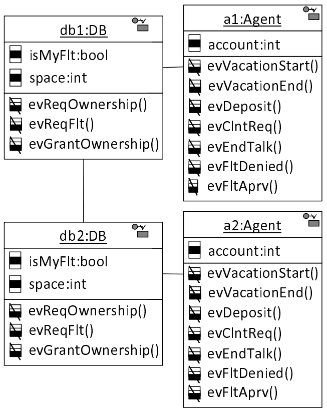
[4] The order in which events are executed is under-specified in UML. We choose to follow the Rhapsody semantics, and implement event processing as a FIFO.

Every object is associated with a single EQ. In a multi-threaded model, there are several EQ-mgrs, and objects from different threads can communicate with each other. In this paper we focus on the case of a single thread, and will henceforth ignore the multi-threaded case.

Objects send events via the operation $GEN()$. For example, in the statechart of DB (Fig. 2(a)), when $db1$ executes the operation $itsA \to GEN(evFltAprv())$, an event of type $evFltAprv$ is sent to the target object on the relevant link. From the object diagram (Fig. 1(b)), we see that the target object is $a1$.



(a) class diagram



(b) object diagram

Fig. 1. Ticket Ordering System

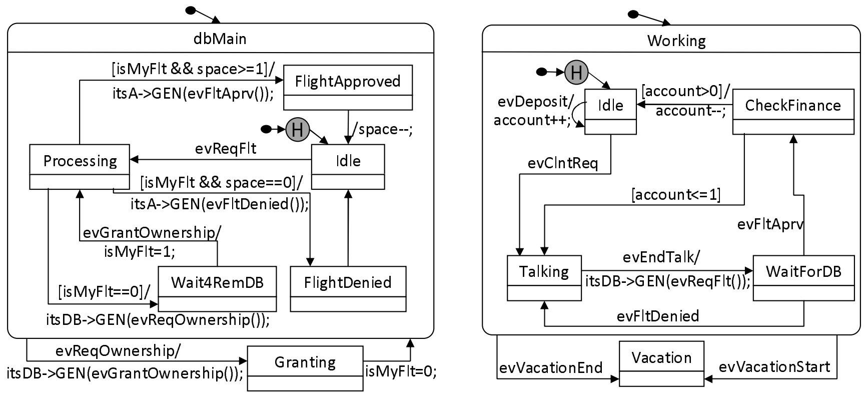
Statecharts: The behavior of each object in the system is described by the *hierarchical statechart* associated with the class of the object. Hierarchical statecharts for classes DB and Agent are given in Fig. 2. For simplicity, in the rest of this section, notions related to statecharts and semantics are first defined for non-hierarchical statecharts. Needed definitions and notations are then extended for hierarchical statecharts as well.

We first define the following notions: A *guard* is a boolean expression over a set of attributes. The trivial guard is $true$. A *trigger* is the name of some event type. An *action* is a possibly empty sequence of statements in some programming language. A statechart of class Cls is a tuple $sc = (Q, T, init)$, where Q is a finite set of states, $init \in Q$ is the initial state, and T is a finite set of transitions. For every $t \in T$, $t = (q, b, e, a, q')$ where $q \in Q$ is the source state, b is a guard, e is either a trigger or nil, a is an action, and $q' \in Q$ is the destination state.

Transitions whose trigger is nil and whose guard is $true$ are referred to as *null-transitions*. In a graphical representation of a statechart, states are marked as squares. Every transition t is marked with $trig[grd]/act$, representing the trigger, guard and action of t. If trigger is nil, guard is $true$ or action is empty then they are omitted from the representation. The initial state is marked with a transition with no source (●▶).

We place a few restrictions on the statecharts language. We assume that every loop in a statechart includes at least one transition with a trigger. We also place restrictions on the action language and disallow dynamic allocation of objects and memory, dynamic pointers, unbounded loops, and recursion. This defines a restricted case of behavioral UML models, which is nevertheless relevant for embedded software. These restrictions enable us to focus on software based verification for UML models, while avoiding orthogonal issues such as termination and pointer analysis.

The Semantics of Behavioral UML Models: Let o be an object with statechart $sc(o)$, and attribute evaluation $\nu(o)$, where $\nu(o)$ is a function mapping all attributes of o to a value in the relevant domain. We say that a transition $t = (q, b, e, a, q')$ in $sc(o)$ is



(a) statechart of class DB (b) statechart of class Agent

Fig. 2. Ticket Ordering System - Statecharts

enabled w.r.t. $\nu(o)$ and an event $ev = (e', o)$ if the following holds: b evaluates to $true$ under $\nu(o)$, and e is either nil or $e = e'$. Let (e, o) be an event that was taken from the queue. Then q, the current state of $sc(o)$, and $\nu(o)$ determine how this event is processed. A transition t can be executed if its source is q and t is enabled w.r.t. $\nu(o)$ and (e, o). If there exists one or more transitions that can be executed from q, then one is executed non-deterministically. When transition t is executed from q, the $action$ of t is executed, and the statechart reaches state q', which is the destination state of t.

Let (e, o) be an event dispatched to o, whose current state is q. An RTC step is a sequence of enabled transitions starting from q. The first transition in the sequence can be marked with a trigger or not[5]. The rest of the transitions are not marked with triggers. An RTC step terminates at a state q' that has no enabled outgoing transitions.

The following terminology will be needed later. Objects that can send some event (ev, o) are called *producers* of (ev, o). In our example, the (only) producer of event $(evReqOwnership, db1)$ is $db2$. Objects that can modify some attribute x of object o are called *modifiers* of (x, o). Let b be a guard in $sc(o)$, where b includes attributes $\{x_1, ..., x_m\}$. The set of modifiers of all attributes in b are called the modifiers of (b, o).

Hierarchical Statecharts: In hierarchical statecharts states can be either *simple* or *composite*. A composite state consists of a set of states, called its *substates*. A simple state has no substates. Every composite state includes an initial state. A composite state can also be defined with *history data*[6], marked by Ⓗ in the statechart. This represents the most recent active substate of q.

Every hierarchical statechart includes a unique *top* state, which is not a substate of any other state. Hierarchical statecharts are denoted by $sc = (Q, T, top)$. A *h-state* $\bar{q} = (q_1, ..., q_n)$ of sc describes a full hierarchical path in sc, where $q_1 = top$, q_n is a simple state and for every $i > 0$, q_i is a substate of q_{i-1}. The initial h-state of sc is $\bar{q} = (q_1, ..., q_n)$ s.t. for every composite state q_i, q_{i+1} is the initial state of q_i.

[5] This point is under-specified in UML. We chose to follow the Rhapsody semantics.

[6] In this work we only consider shallow history.

We disallow transitions to cross hierarchy levels, i.e. for $t = (q, b, e, a, q')$, q and q' are substates of the same state. Given $\nu(o)$ and (e, o), a transition t can execute from h-state $\bar{q} = (q_1, ..., q_n)$ of $sc(o)$ if t is enabled w.r.t. $\nu(o)$ and (e, o) from q_i, $1 \leq i \leq n$, and no t' is enabled from any q_j, $i < j \leq n$. When t is executed, $sc(o)$ reaches the destination h-state $\bar{q}' = (q'_1, ..., q'_m)$ for some $m \geq i$ s.t.: (1) $\forall j.1 \leq j < i$, $q'_j = q_j$, (2) q'_i is the destination state of t, and (3) $\forall j.i < j \leq m$, q'_j is defined according to the history semantics of UML. From now on we consider only hierarchical statecharts.

2.2 LTL and Automata Based Model Checking

A *Kripke structure* is a tuple $M = (S, I_0, R)$, where S is a set of states, $I_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a total transition relation. An *execution* of M is an infinite set of states $s_0, s_1, ...$ s.t. for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. .

The Linear-time Temporal Logic (LTL) [21] is suitable for expressing properties of a system along an execution path. We assume the reader is familiar with LTL. In this work we restrict ourselves to a fragment of LTL, in which only *safety properties* are expressible. These are properties whose violation occurs along a finite execution. [27] gives a syntactic characterization of safety properties.

A Kripke structure M satisfies an LTL formula ψ, denoted $M \models \psi$, if every execution of M starting at an initial state satisfies ψ. A general method for on-the-fly verification of LTL safety properties is based on a construction of a regular automaton $A_{\neg\psi}$, which accepts exactly all the executions that violate ψ. Given M and ψ, we construct $M \times A_{\neg\psi}$ to be the product of M and $A_{\neg\psi}$. A path in $M \times A_{\neg\psi}$ from an initial state (s, q) to a state (s', q') where q' is an accepting state in $A_{\neg\psi}$ represents an execution of M, and a word accepted by $A_{\neg\psi}$. It therefore represents an execution showing why M does not satisfy ψ. Such executions are called *counterexamples* for ψ.

2.3 Bounded Model Checking

Bounded Model Checking (BMC) [3] is an iterative process for checking models against LTL formulas. The transition relations for a Kripke structure M and its specification are jointly unwound for k steps and are represented by a boolean formula that is satisfiable iff there exists an execution of M of length k that violates the specification. The formula is then checked by a SAT solver. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. Otherwise, k is increased.

BMC is widely used for finding bugs in large systems, including software systems ([8,2,9]). BMC for software is performed by unwinding the loops in the program for k times, and verifying the required property. The property is often described by an assertion added to the program text. The model checker then searches for a program execution that violates the assertion. Our method for verifying UML models relies on invoking a software BMC tool. We require that the tool supports assumptions on the program, given as $assume(b)$ commands, where b is some boolean condition. Having $assume(b)$ at location ℓ of the program means that only executions π that satisfy b when passing at ℓ are considered. If b is violated then π is ignored.

2.4 Notations and Abbreviations

Throughout the rest of the paper we will use the following notations and abbreviations. A model includes N objects $\mathcal{M} = \{o_1, ..., o_N\}$. Every object o_i is associated with a statechart $sc(o_i) = (Q_i, T_i, top_i)$. For $t_i = (q, b, e, a, q')$ a transition in T_i: $grd(t_i) = b$, $ev(t_i) = e$ and $act(t_i) = a$. Given a state $q \in Q_i$:

- $trn(q) \subseteq T_i$ is the set of transitions whose source is q.
- $evnts(q) = \bigcup_{t \in trn(q)} \{(ev(t), i)\} \setminus \{(nil, i)\}$ is the set of triggers on $trn(q)$.
- $grds(q) = \bigcup_{t \in trn(q)} \{(grd(t), i)\}$ is the set of guards on $trn(q)$.
- $prod(q) \subseteq \{1, ..., n\}$ denotes indexes of producers of all events in $evnts(q)$. For example, if $evnts(q') = \{(ev, j)\}$, and the producers of (ev, o_j) are $\{o_{i_1}, ..., o_{i_k}\}$, then $prod(q') = \{i_1, ..., i_k\}$.
- $modif(q) \subseteq \{1, ..., n\}$ denotes indexes of modifiers of all guards in $grds(q)$.

These abbreviations are generalized to denote the transitions, events, guards, producers, and modifiers of an h-state and of a *subset* of states.

3 Translation to Verifiable Bounded C

We translate behavioral UML models to C. Our goal is to create code that is most suitable for verification, rather then an efficient implementation of the system. Moreover, we verify our code using a BMC verifier, therefore our code describes a bounded run of the model. In order to create code suitable for verification we avoid as much as possible the use of pointers or of methods called with different parameters. This results in code which is longer in lines-of-code. However, the model created by the verification tool is smaller, and the model checker can then perform optimizations more efficiently.

```
1: method $RunRTCStep_i(ev)$
2: while $(j < maxRTClen)$ do
3:     if $(!enabled(currSt, \nu_i, ev))$ return
4:     $choose\ Transition\ t$
5:     assume($t \in trn(currSt)$)
6:     assume($val(t, currSt, \nu_i, ev)$)
7:     $execute\ \ act(t)$
8:     $ev := nil$
9:     $incr\ \ j$
```

Fig. 3. $RunRTCStep_i$ method of o_i

The atomic unit in our translation is a *single RTC step*, rather than a single transition. Every object is translated into a method, representing the behavior of its associated statechart. When an event ev is dispatched to object o_i, the method associated with o_i executes a single RTC step of o_i.

Fig. 3 presents $RunRTCStep_i$, the pseudo-code for a single RTC step of o_i. $currSt$ is the current h-state of o_i in $sc(o_i)$. $enabled(currSt, \nu_i, ev)$ is $true$ iff there exists an enabled transition $t \in trn(currSt)$ w.r.t. ν_i and event ev. The method terminates when there are no enabled transitions to execute. The `while` loop iterates up to $maxRTClen$ iterations. $maxRTClen$ represents the maximum number of transitions of any RTC step of o_i. If this value cannot be extracted by static analysis, then the condition is replaced by $true$, and the length of the RTC step is bounded by the BMC bound, k.

$val(t, currSt, \nu_i, ev)$ is $true$ iff t can be executed from $currSt$ w.r.t. ν_i and event ev. Lines 4-6 amount to a non-deterministic choice of a transition t, which can be executed from $currSt$. When choosing a transition (line 4), no constraints are assumed on it. Line 5 restricts the program executions to those where t is a transition from $currSt$. Line 6 restricts the remaining program executions to those where t can execute. In line 7 the action of the transition is executed. Executing the action updates the $currSt$ according to the destination state of t. Note line 8, where we set the event to nil. This is done since the event is consumed once, and only in the first transition of the RTC step. The rest of the transitions of the RTC step can be executed only if their trigger is nil.

The EQ is represented as a bounded array. The main method of the program executes the never-ending loop of taking an event from the EQ, and dispatching it to the relevant target object. Fig. 4 presents the pseudo-code for the `main` method. In line 3 an event ev whose target is o_i is taken from the EQ. In line 4 an RTC step of o_i is initiated.

```
1: method $main$
2: while $(true)$ do
3:     $(ev, o_i) := popEv()$
4:     $RunRTCStep_i(ev)$
```

Fig. 4. $main$ method

When applying BMC on the main method in Fig. 4, the `while` loop is unrolled k times, which means that the model is verified for k RTC steps. Generally, placing a bound on the EQ can make the model inaccurate due to overflows. However, k is the exact bound for a k-bounded verification over k RTC steps, since only the first k events that are sent will be dispatched during k RTC steps.

Another verification oriented optimization we introduce is in the implementation of the environment. The array is initialized with k environment events, but with $head = tail = 1$. When a system event evS is sent, the tail is incremented non-deterministically, after which evS is added to the EQ, overriding the environment event there. This models inserting to the EQ a non-deterministic number of environment events that arrive prior to the addition of evS to the EQ.

C code can be automatically generated by UML tools such as Rhapsody, but this code would not be suitable for verification. Automatically generated code includes generic code, and means for communicating with different libraries and with the operating system. We, on the other hand, are interested in verifying only the user-created behavior of the system, and therefore we can abstract the event queue and the operating system. We exploit features of the model-checker, such as the `assume` construct, to make the verification more efficient. Assuming a static model allows us to implement links by direct calls rather than using pointers.

4 Model Verification

We now describe our method for verification of a given behavioral UML model. The model includes N objects $\mathcal{M} = \{o_1, ..., o_N\}$. Verification is done using assertions on the code describing the model. We support verification in a granularity of transition level or RTC level. First, we define the notion of *configuration* (CONF) of a UML model.

Definition 1. *A* configuration *(CONF) of $\mathcal{M}$ is $C = (q, \nu, EQ)$, where:*

– *$q = (\bar{q}_1, ..., \bar{q}_N)$ is a* system state *where $\bar{q}_i$ is a h-state of $sc(o_i)$.*

– $\nu = (\nu(o_1), ..., \nu(o_N))$ *is an* evaluation vector*;* $\nu(o_i)$ *is attribute evaluation of* o_i.
– $EQ = ((e_1, i_1), ..., (e_m, i_m))$ *is an event queue with* m *elements, where* (ev_j, i_j) *represents event* ev_j *whose target is* o_{i_j}. (e_1, i_1) *is the top, denoted* $top(EQ)$.

A behavioral UML model $\mathcal{M}$ can be viewed as a Kripke structure $M = (S, I_0, R)$, where S is the set of all possible CONFs in $\mathcal{M}$. R can be defined either at the *RTC level* (denoted R_{RTC}) or at the *transition level* (denoted R_t). $(C, C') \in R_{RTC}$ iff C' is reachable from C in a single RTC step. $(C, C') \in R_t$ iff C' is reachable from C in an execution of a single transition. Executions are defined at *RTC* or *transition* level.

Definition 2. $\pi_r = C_0, C_1, ...$ *is an* execution at the RTC level (RTC-execution) *iff for every* $n > 0$*,* $(C_{n-1}, C_n) \in R_{RTC}$.

Definition 3. $\pi_t = C_0, C_1, ...$ *is an* execution at the transition level (t-execution) *iff for every* $n > 0$*,* $(C_{n-1}, C_n) \in R_t$*, and* π_t *represents an execution of RTC steps. That is, for every* $i \geq 0$*, there exist* $j \leq i$ *and* $m \geq i$ *s.t.* $C_j, ..., C_m$ *represents a single RTC step.*

For the rest of the paper, when an execution is either a t-execution or an RTC-execution, we refer to it as an *execution*. In the following we first present how model checking of an LTL safety property over a given behavioral UML model is done. We then continue to present our algorithm for verifying cycle-livelocks.

4.1 Verifying LTL Safety Properties

We now show how to verify safety LTL properties over behavioral UML models using an automata based approach. We assume the atomic propositions of the property are predicates over the CONFs of the model. We extend the C program created from $\mathcal{M}$ with a method representing the automaton $A_{\neg\psi}$. The method runs in lock step with the system, and identifies property violations.

A safety property can be verified either at the RTC level or at the transition level, by placing the call to the automaton method either at the end of each RTC step (within the method $main$) or at the end of each transition (within the method $RunRTCStep_i$). The choice of the level for verification depends on the property to be verified. For example, in our running example we might want to guarantee that, at the end of RTC steps $isMyFlt$ cannot be $true$ for both $db1$ and $db2$ at the same time. This property must not necessarily hold during an RTC step. We would therefore verify AG[7]$(db1.isMyFlt = 0 \vee db2.isMyFlt = 0)$ at the RTC level. If we want to check for dead states (unreachable states) we need to work at the transition level in order to recognize as reachable also those states that are passed through during the RTC step.

Note that our method for BMC can be extended to proof by k-induction [26] in a straightforward manner. The base case is a BMC of k steps, which is done in the way we described above. The step is a BMC run of $k + 1$ steps with the initial state completely non-deterministic, looking for a run in which a property violation occurs at the $k + 1$ step after k steps with no violation. In the initial state of the step case we assume there may already be any number of events in the queue, of any type. We can

[7] G is the temporal operator with the meaning of "globally".

still bound the event queue to $k + 1$ entries because no more than $k + 1$ events will be dispatched in $k + 1$ steps, making it sound to ignore the content of the queue beyond $k + 1$ entries.

4.2 Verify Cycle-Livelocks

A Livelock describes the case where *part of the system* cannot progress, even though the other parts of the system do. In this section we focus on finding livelocks in behavioral UML models. As mentioned before, absence of livelocks in neither safety nor LTL property and therefore cannot be handled by scalable bounded model checking tools. For that reason, we identify a subclass of livelocks, and present a method for finding such livelocks within our framework. This is done by a reduction to a safety property, which requires a preceding syntactic analysis of the UML model.

We first define the notion of a *livelock CONF* in behavioral UML models.

Definition 4. *Given a CONF $C = (q, \nu, EQ)$, where $q = (\bar{q}_1, ..., \bar{q}_N)$. We say that o_i is* disabled under *C if no transition $t \in trn(\bar{q}_i)$ is enabled.*

Definition 5. *Given a CONF C, object o_i is* stuck at *C if for every RTC-execution $\pi = C_0, C_1, ...$ s.t. $C_0 = C$ the following holds: for every $C_j = (q, \nu, EQ)$ s.t. $j \geq 0$, if $top(EQ) = (ev, i)$ then o_i is disabled under C_j.*

Thus, an object o_i is stuck if whenever the event at the top of the queue is targeted at o_i, meaning it is o_i's turn to execute, o_i is disabled and cannot make any progress.

Definition 6. *A CONF C is a* livelock CONF *if at least one object is stuck at C.*

Following, we present a characterization for a subclass of livelock CONFs, which we call *cycle-livelocks*. Intuitively, a CONF C is a cycle-livelock if there is a subset of objects that are stuck at C, and for every object o in the subset all of the producers of events that o is stuck on, and all of the modifiers of the guards that o is stuck on, are in the subset as well.

Definition 7. *Let $C = (q, \nu, EQ)$ where $q = (\bar{q}_1, ..., \bar{q}_N)$. A $q' = (\bar{q}'_1, ..., \bar{q}'_N)$ is a* partial state *of C if for every $1 \leq i \leq N$, $\bar{q}'_i = nil$ or $\bar{q}'_i = \bar{q}_i$.*

Definition 8. *Let C be a livelock CONF, and let $q' = (\bar{q}'_1, ..., \bar{q}'_N)$ be partial state of C. q' is a* livelock state of *C if $\forall i.1 \leq i \leq N$, if $\bar{q}'_i \neq nil$ then o_i is stuck at C.*

Definition 9. *CONF C is a* cycle-livelock *if there exists a livelock state of C, $q' = (\bar{q}'_1, ..., \bar{q}'_N)$ s.t. for all $j \in prod(q') \cup modif(q')$, $q'_j \neq nil$.*

Intuitively, the partial state describes a set of objects that are stuck and will stay stuck forever. This is because all objects that may "release" a stuck object by producing an event or changing a guard are in the same set. That is, they are stuck as well.

Our goal is to find *reachable* cycle-livelock CONFs. To achieve scalability, we use SAT-based BMC and only find livelock CONFs that are reachable within k RTC steps. Our method for finding reachable cycle-livelocks consists of two stages. We first identify system states that are *cycle-states*. This is a *syntactic* identification and can thus be checked independently of a CONF. This stage is performed during the analysis of the UML model. We then search for a reachable cycle-livelock CONF. This is done by adding an assertion describing the fact that the current CONF is a cycle-livelock. We then apply BMC to search for a violation of the assertion. Next we define the syntactic notion of cycle-state.

Finding Cycle-States: An object o_i cannot be stuck at $C = (q, \nu, EQ)$ if $\bar{q}_i \in q$ has a null-transition, or if q_i has a transition that can be enabled by an environment event.

Definition 10. *An h-state $\bar{q}$ is* potentially stuck *if for every $t \in trn(\bar{q})$, t is not a null-transition, and if $ev(t)$ is an environment event, then $grd(t) \neq true$.*

Intuitively, a *cycle-state* represents a subset of objects that are all potentially stuck and dependant on each other, i.e. all the necessary producers are inside this subset.

Definition 11. *A* cycle-state *is a vector $q = (\bar{q}_1, ..., \bar{q}_N)$ s.t. $\forall 1 \leq i \leq N$, $\bar{q}_i = nil$ or $\bar{q}_i$ is a h-state of $sc(o_i)$, and the following holds for every $\bar{q}_i \neq nil$: (1) $\bar{q}_i$ is a potentially stuck h-state, and (2) There is no $j \in prod(\bar{q}_i) \cup modif(\bar{q}_i)$ s.t. $\bar{q}_j = nil$, and (3) q is minimal. That is, let $q' = (\bar{q}'_1, ..., \bar{q}'_N)$ be a system state vector where $\forall 1 \leq i \leq N$, $\bar{q}'_i \neq nil \Rightarrow \bar{q}'_i = \bar{q}_i$. If $q' \neq q$ then req. 2 does not hold for q'.*

The requirement of minimality (requirement (3)) is introduced for the sake of efficiency. It reduces the number of states to be considered and also simplifies the encoding in BMC. Further, it reduces the number of similar counterexamples returned to the user.

Note that this definition is *syntactic*. That is, it depends only on the system state vector. It does not depend on the evaluation vector or the event queue, which can be determined along an execution. As a result, the set of all cycle-states can be identified independently of any configuration. We generate this set from the syntactic structure of the model, as part of the analysis of the UML model.

Lemma 1. *The set of cycle-states is* complete. *Meaning for every cycle-livelock configuration C there exists a partial state of C, q, that is a cycle-state.*

The set of CONFs is infinite, because the size of the EQ is not limited, and the domain of the evaluation vector can be infinite. However, the set of cycle-states is finite.

Bounded Search for Cycle-Livelocks: We observe that if a given CONF includes a cycle-state s.t. for every transition in the cycle-state either the guard is false or the trigger is a system event which is not in the EQ, then this CONF is a cycle-livelock.

We adapt the translation of UML models to C (Sec. 3) to allow checking whether a cycle-livelock CONF is reachable by adding assertions at the RTC level. When the model checker finds an execution violating the assertion, the last CONF in the execution is a cycle-livelock CONF. Fig. 5 presents the pseudo-code of the modified method. Line 5 and 6 show the added code.

$currC = (q, \nu, EQ)$ represents the current CONF of the system. At every iteration of the `while` loop $currC$ changes (due to the RTC step). The method $partSt(q, C)$ receives a cycle-state q and a CONF C, and returns $true$ iff q is a partial state of C. The method $grdFalse(grd, \nu)$ returns $true$ iff grd is $false$ w.r.t. ν_i. The method $notInQ(ev, EQ)$ returns $true$ iff ev is a system event which is not in the EQ. The assertion is violated on C if C is a cycle-livelock.

There is one subtle point that still needs to be solved: We need a finite representation of the queue. Recall that for verifying safety properties, for k-bounded executions we bound the queue to k. However, when searching for cycle-livelocks this is incorrect because a configuration is a cycle-livelock if there are *no future* executions that can release the stuck states. Thus, we must keep track of *all* events inserted into the queue (within k RTC steps). However, only the first k events are dispatched, and therefore their relative order is important. For the rest of the events, we only need to know whether they were sent or not. indicating whether or not an instance of that event exists in the "actual" queue. The method $notInQ(ev, EQ)$ returns $true$ iff the flag of event ev is $false$, indicating that no such event is in the "actual" queue.

```
1: method $FindCycleLivelock()$
2: while $(true)$ do
3:     $(ev, i) := popEv()$
4:     $RunRTCStep_i(ev)$
5:     for each cycle-state $q'$ do
6:         assert$(!(partSt(q', currC) \wedge$
               for all $t \in trn(q')$ :
                   $notInQ(ev(t), EQ) \vee$
                   $grdFalse(grd(t), \nu)))$
```

Fig. 5. $FindCycleLivelock$ method

We exemplify our method on our running example. The events $evVacationStart$ and $evVacationEnd$, which are consumed by class Agent, are both environment events. Note that none of the h-states associated with the statechart of Agent are potentially stuck h-states. Thus, $a1$ and $a2$ can never be stuck. The system state vector $(Wait4RemDB, Wait4RemDB, nil, nil)$ is a cycle-state because the producer of state $Wait4RemDB$ of $db1$ is $db2$, and vice-versa. Note that for finding $prod(Wait4RemDB)$, we include the producers of both $Wait4RemDB$ and $dbMain$, since $Wait4RemDB$ is a substate of $dbMain$. For this cycle-state, we add the following assertion:

$$\begin{aligned}&assert(!(!InEQ(evGrantOwnership, 1) \wedge !InEQ(evGrantOwnership, 2) \wedge \\ &\quad !InEQ(evReqOwnership, 1) \wedge !InEQ(evReqOwnership, 2) \wedge \\ &\quad partSt((Wait4RemDB, Wait4RemDB, nil, nil), currC)))\end{aligned}$$

Note that it is possible to skip the first stage of our algorithm, that finds the set of cycle-states, and incorporate it within the second stage. However, this would be inefficient due to the number of checks that would need to be done during the model checking stage. Further, since the first stage is applied to the UML model, it is quite "light weight". Model checking, on the other hand, is applied to a low-level description and is a heavy task. Thus, the first stage is essential for the scalability of our method.

5 Experimental Results

We have implemented the algorithm described above in a tool called Soft-UMC (**soft**ware-based **U**ML **M**odel **C**hecking). The implementation reads a UML (version 2.0) model, and translates it to verifiable C code. Static analysis is applied at this stage, according to the type of property to be checked: (LTL) safety or livelock. We then apply CBMC[8] (version 4.1) as our C verifier.

First, we compared our implementation to one translating the model to the input language of RuleBasePE[24], IBM's hardware model checker (we call this solution HWMC). HWMC represents the EQ as a bounded FIFO, where the size of the FIFO is relative to the maximum number of events generated in a single RTC step. It also preserves the hierarchical structure of the model.

To compare the performance of Soft-UMC and HWMC we used the following four examples. (1) A variant of the railroad crossing system from [22], including a gate object and three track objects that communicate with the gate, (2) The ticket ordering model (Figs. 1,2), (3) A dishwasher machine (inspired by the example provided with Rhapsody), (4) A locking model, including a manager and three lock clients. We have checked several safety properties on the models. In Fig. 6 we present a comparison of the runtime for finding a counterexample in Soft-UMC and HWMC. It can be seen that HWMC is better on short counterexamples. However, on long ones Soft-UMC achieves results in shorter times. This can be explained by the initialization time of CBMC which is significant for short counterexamples but becomes negligible on long ones.

	Soft-UMC		HWMC	
prop.	time	#RTCs	time	# trans
RC1	155	10	**44**	34
RC2	198	11	**145**	39
RC3	**868**	17	2315	57
TO1	17	6	**14**	8
TO2	23	7	**14**	13
TO3	51	10	**28**	31
TO4	**514**	22	1425	67
DW1	263	12	**58**	37
DW2	304	18	**40**	95
DW3	**986**	30	1345	155
LM1	18	7	**12**	19
LM3	101	16	**79**	86
LM2	**158**	14	1320	37
LM4	**555**	34	645	176

Fig. 6. Soft-UMC vs. HWMC. time in secs. ♯RTC and ♯trans is number of RTC steps and transitions in counterexamples.

To check the scalability of our tool compared to HMWC, we considered three parameterized examples: The ticket ordering model, and variations of the dishwasher machine and the locking model. E.g., for the ticket ordering model, the attribute *account* of *Agent* is used as the parameter, and the checked property is non-determinism. For increasing initial values of *account*, the counterexample leading to a non-deterministic state is of increasing length. This allows us to experiment on the same model with different lengths of counterexamples. In all examples, a counterexample for a model with parameter i is of length $\sim 2*i$ RTC steps. Each RTC step is composed of 3-5 transitions. We used a timeout of 1 hour. Results are presented in Fig 7. From the comparison it is clear that HWMC is better for shallow examples, however our tool is more scalable.

We also evaluated the performance impact of two of our optimizations, the EQ (Sec. 3) and the hierarchical model. We compared a naive implementation of the EQ against our optimized implementation. To analyze the impact of maintaining the hierarchy of the model we created a flat model of the ticket ordering model. The flat model has 24 states and 54 transitions, whereas the hierarchical model has 26 states and 36 transitions. The flat model is missing the hierarchical states. However, it has an additional attribute for maintaining the history. Fig 8 shows the results of the comparison. We compared the runtime of 4 different implementations: Hierarchical model with optimized EQ (H-OP-EQ), flat model with

param	Soft-UMC TO	HWMC TO	Soft-UMC DW	HWMC DW	Soft-UMC LM	HWMC LM
5	49	**21**	82	**23**	34	**30**
8	113	**92**	242	**34**	101	**71**
11	**202**	380	475	**66**	192	**180**
14	**364**	1830	825	**254**	328	**187**
17	**693**	3470	1326	**810**	555	**613**
20	**1740**	T.O	**1964**	T.O	766	**789**
23	T.O	T.O	**2900**	T.O	1153	**889**
26	T.O	T.O	T.O	T.O	**1657**	1876
29	T.O	T.O	T.O	T.O	**1859**	2142
32	T.O	T.O	T.O	T.O	**3049**	T.O

Fig. 7. Compare scalability. time in secs.

optimized EQ (F-OP-EQ), hierarchical model with naive EQ (H-NV-EQ) and flat model with naive EQ (F-NV-EQ).

We verified three different properties, and modified the model s.t. counterexample is reached at different bounds. 1,3 are safety properties. 2 is a livelock check, checked on a slightly modified model: the guard of transition from $Processing$ to $FlightApproved$ of DB (Fig. 2(a)) is modified to $[isMyFlt\ \&\&\ (space > 1)]$. This introduces a reachable livelock state, when $db1$ and $db2$ are in state $Processing$, $space = 1$ and $isMyFlt = true$ for both objects. Each row in Fig 8 represents a different setting defined by the property and the initial values of the attributes, which determine the length of the counterexample (in RTC steps). Time limit is set to 1 hour. It is clear that the optimized implementation of the EQ scales much better w.r.t. the naive EQ implementation. This is because the naive implementation includes a loop representing the addition of a non-deterministic number of environment events to the EQ. In the optimized implementation this amounts to a non-deterministic increment of the tail. The comparison also shows that the hierarchical implementation scales better than the flat one. Our conjecture is that flattening increases the number of transitions in the model, and therefore increases the search space. [11] presents similar results when comparing verification of hierarchical UML models to flat models. The above shows the significance of optimizations. We expect to be able to further improve performance of our solution with other optimizations.

	#RTC	H-OP-EQ	F-OP-EQ	H-NV-EQ	F-NV-EQ
#1	6	21	31	369	396
	10	63	94	3362	T.O
	18	224	420	T.O	T.O
	26	524	1235	T.O	T.O
#2	10	88	133	T.O	T.O
	20	818	3157	T.O	T.O
#3	6	21	32	371	420
	10	72	103	T.O	T.O
	14	275	550	T.O	T.O

Fig. 8. Optimizations on ticket ordering. Bound in RTC steps; time in secs.

6 Conclusions

This work is a first step in exploiting software model checking techniques for the verification of behavioral UML models. By translating UML models to C we could preserve the high-level structure of the model. We intend to further exploit this structure in techniques such as abstraction and modularity in order to enhance UML verification.

Our translation to *verifiable* C code rather than executable one significantly eased the workload of the model checker. This is demonstrated, for instance, by the comparison of our optimized representation of the event queue with a naive one. In our translation we also took advantage of the fact that *bounded* model checking is applied, and obtained a finite representation in spite of the unbounded size of the queue. Nevertheless, our method can be extended to *unbounded* model checking by means of k-induction.

The comparison with IBM's hardware oriented tool for UML verification demonstrates that our approach is superior for long counterexamples.

Our approach to finding cycle-livelocks in UML models is novel. Static analysis identifies *syntactically* potential cycle-livelock states. A suitable finite representation of the event queue then enables to apply BMC for finding reachable such states. We expect similar approaches to be useful for proving additional non-safety properties.

References

1. Majzik, I., Darvas, A., Beny, B.: Verification of UML Statechart Models of Embedded Systems. In: DDECS (2002)
2. Armando, A., Mantovani, J., Platania, L.: Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. STTT 11(1) (2009)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Booch, G., Rumbaugh, J.E., Jacobson, I.: The Unified Modeling Language User Guide. J. Database Manag. 10(4) (1999)
5. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model Checking Large Software Specifications. IEEE Trans. Software Eng. 24(7) (1998)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
7. Clarke, E.M., Heinle, W.: Modular Translation of Statecharts to SMV. Technical Report, CMU (2000)
8. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In: ASE (2009)
10. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In: ASE (2002)
11. Dubrovin, J., Junttila, T.A.: Symbolic Model Checking of Hierarchical UML State Machines. In: ACSD (2008)
12. Object Management Group. OMG Unified Modeling Language (UML) Infrastructure, version 2.4. ptc/2010-11-16 (2010)
13. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Sci. Comp. Prog. 8(3) (1987)
14. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I.: Model Checking Dynamic and Hierarchical UML State Machines. In: MoDeVa (2006)
15. Kaveh, N.: Using Model Checking to Detect Deadlocks in Distributed Object Systems. In: Emmerich, W., Tai, S. (eds.) EDO 2000. LNCS, vol. 1999, pp. 116–128. Springer, Heidelberg (2001)
16. Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Formal Asp. Comput. 11(6) (1999)
17. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing Statecharts in PROMELA/SPIN. In: WIFT (1998)
18. Ober, I., Graf, S., Ober, I.: Validating Timed UML Models by Simulation and Verification. STTT 8(2) (2006)
19. IST-2001-33522 OMEGA (2001), `http://www-omega.imag.fr`
20. Lilius, J., Paltor, I.P.: Formalising UML State Machines for Model Checking. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 430–444. Springer, Heidelberg (1999)
21. Pnueli, A.: The Temporal Logic of Programs. In: FOCS (1977)
22. Prashanth, C.M., Shet, K.C., Elamkulam, J.: An Efficient Event Based Approach for Verification of UML Statechart Model for Reactive Systems. In: ADCOM (2008)
23. Rhapsody, `http://www-01.ibm.com/software/awdtools/rhapsody`

24. RuleBasePE,
`http://www.haifa.ibm.com/projects/verification/RB_Homepage/`
25. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML Verification Environment. In: SEFM (2004)
26. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
27. Sistla, A.P.: Safety, Liveness and Fairness in Temporal Logic. Formal Asp. Comput. 6(5) (1994)