Interactive Verification of UML State Machines

Michael Balser[1], Simon Bäumler[1], Alexander Knapp[2],
Wolfgang Reif[1], and Andreas Thums[1]

[1] Institut für Informatik, Universität Augsburg
{balser, baeumler, reif, thums}@informatik.uni-augsburg.de
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
knapp@pst.ifi.lmu.de

Abstract. We propose a new technique for interactive formal verification of temporal properties of UML state machines. We introduce a formal, operational semantics of UML state machines and give an overview of the proof method which is based on symbolic execution with induction. Usefulness of the approach is demonstrated by example of an automatic teller machine. The approach is implemented in the KIV system.

1 Introduction

The "Unified Modeling Language" (UML [14]), the de facto standard notation for specifying object-oriented software systems, offers state machine diagrams as an operational view of the dynamic aspects of a system, which are a variant of the statechart notations introduced by Harel [5]. Most attempts to verify properties of UML state machines use model checking (see e.g. [11, 10, 16]). Model checking allows a fast and simple way to check, whether a property holds or not. However, the major disadvantage of model checking is that it needs to search the whole state space of the state machine for a violation of the property and thus the verification success highly depends on the size of the state space. If the state space of a state machine is unlimited, e.g. by using an unbounded integer variable, verification of true properties normally fails.

We pursue another approach to the verification of UML state machines. In contrast to model checking, interactive verifiers such as KIV [3] are designed for systems with an infinite state space. Our goal is to provide a uniform, interactive, intuitive and efficient proof method for verifying properties of UML state machines. We have chosen symbolic execution with induction as proof method, because this method has been successfully applied to the verification of sequential programs (e.g. [7, 17]) and gives very intuitive proofs. Furthermore, symbolic execution can be automated to a very large extent. We have also shown how to carry over the proof method to the verification of parallel programs [2]. Here, we demonstrate how to apply the method to the verification of UML state machines.

To our knowledge, this is the first such approach to interactive verification of UML state machines. In contrast to other approaches, we do not formalise the semantics of UML in the existing logic, but derive a calculus to directly execute state machines. We have been able to efficiently support not only a subset but all of the main features of

UML state machines such as hierarchical and concurrent states, compound transitions, and a rich action language. Thus, our approach is more complete than e.g. the Omega project [15]. Compared to other interactive calculi for the verification of similar concurrent systems (e.g. [12, 4]), the proof method of symbolic execution promises a higher degree of automation, a uniform treatment of safety and liveness properties, and the use of invariants which are easier to come up with.

We assume the reader to be familiar with UML state machines and to have at least basic knowledge in temporal logic and sequent calculus. The remainder of this paper is structured as follows: In Sect. 2, we give a short introduction to constructing temporal proofs with symbolic execution and induction. Section 3 describes UML state machines and introduces an example. Section 4 formally defines an operational semantics of UML state machines. How to turn the operational semantics of UML into a calculus rule for symbolic execution is sketched in Sect. 5. A safety property of the example is verified in Sect. 6. Section 7 summarizes the results and outlines future work.

2 Temporal Logic Framework

The logical formalism is a variant of ITL (Interval Temporal Logic [13]). It is based on (finite or infinite) linear sequences π of valuations which we call *intervals*. A single valuation (which we also call state) is described by first-order predicate logic formulae over static variables v and dynamic variables V. Different valuations of an interval π may assign different values to the dynamic variables V. In addition, a valuation gives values to primed variables V' and double-primed variables V''; the relation between unprimed and primed variables is interpreted as *system transitions*, the relation between primed and double-primed variables as *environment transitions*. System and environment transitions alternate, the value of V'' being equal to V in the next valuation. Temporal operators, we support, include $\Box\varphi$ (φ always holds), $\Diamond\varphi$ (eventually φ holds), φ **until** ψ, φ **unless** ψ, $\circ\varphi$ (there is a next state which satisfies φ), $\bullet\varphi$ (if there is a next state, it satisfies φ), and **last** (the current state is the last) with their standard semantics. The proof method is based on a sequent calculus.

We focus on a subset of proof obligations, here: Our goal is to take an initial variable condition I, a system S, a configuration C, and a temporal property φ and to prove that $S, C, I \vdash \varphi$. In other words, all runs of system S which satisfy C and I in its initial state must satisfy φ. The proof method is symbolic execution with induction. In the following, we show how to symbolically execute a temporal formula φ, how to execute an arbitrary system description S, and how to use induction to prove properties for infinite system runs. In Sect. 5 we show how to apply this general proof method to the verification of UML state machines.

2.1 Symbolic Execution

Our principle of symbolic execution is to rewrite an arbitrary temporal formula φ to an equivalent formula

$$\varphi \leftrightarrow \tau_0 \wedge \textbf{last} \vee \bigvee_{1 \le i \le n} (\tau_i \wedge \circ\varphi_i)$$

where τ_i are formulae in first-order predicate logic. Either the system represented by φ terminates with condition τ_0, or one of n possible transitions τ_i is executed and execution continues with φ_i in the next step. As an example, consider rewrite rule

(alw) $\Box\varphi \leftrightarrow \varphi \wedge \bullet\Box\varphi$

which unwinds an always operator such that φ must hold now and always in the future. Our calculus provides rewrite rules to similarly unwind all of the temporal operators. By unwinding the single temporal operators with these rules, it is possible to rewrite arbitrary temporal formulae to receive an equivalent formula of the form stated above. Details can be found in [2].

2.2 Executing System Descriptions

We apply the same strategy of symbolic execution to almost arbitrary system descriptions S which describe the operational behaviour of systems as follows: a function $exec(S, C)$ is implemented which returns the possible transitions τ with configurations C_0 for the next state. This function can be used in the rule (to be read bottom-up)

$$\text{(execute)}\quad \frac{\{\tau \wedge \circ(S \wedge C_0), I \vdash \varphi \mid \langle \tau, C_0 \rangle \in exec(S, C)\}}{S, C, I \vdash \varphi}$$

to execute a system S in configuration C. The transition τ relates variable condition I to the variable condition in the next state. We use the *execute* rule to integrate UML state machines into our formalism (see Sect. 5). An implementation for $exec(S, C)$ is rather straightforward if the operational semantics of S is formally defined. The rule (execute) is sound if implementation of $exec(S, C)$ is sound.

This approach avoids the overhead in proof size, which would result if we specify the semantics of UML within our logic. In this case, we would need several basic rules to execute a single state machine step. In our experience, application of rule (execute) is very efficient.

2.3 Taking a Step

After system and temporal formula have been rewritten and execution does not terminate, rule (step) can be applied.

$$\text{(step)}\quad \frac{S, C, I_0 \vdash \varphi}{\tau, \circ(S \wedge C), I \vdash \bullet\varphi} \quad \text{where } I_0 := (I \wedge \tau)[V, V', V''/v_0, v_1, V]$$

Condition I_0 in the next state is received by replacing in the original I and transition τ all unprimed and primed dynamic variables V and V' with fresh static variables v_0 and v_1, the double-primed variables V'' correspond to the unprimed variables in the next state. In addition, the leading operators $\circ$ and $\bullet$ are eliminated. Symbolic execution continues in the next state.

2.4 Induction

Symbolic execution needs not to terminate. Therefore, we use noetherian induction over a supplied induction term T

$$\text{(ind)}\ \frac{\Gamma, T = n, \Box(T < n \to (\bigwedge \Gamma \to \bigvee \Delta)) \vdash \Delta}{\Gamma \vdash \Delta}$$

The fresh static variable n is assigned to the initial value of T. Always, if the value of T is less than the initial value n, the induction hypothesis $\bigwedge \Gamma \to \bigvee \Delta$ can be applied. This general induction principle can be applied to prove arbitrary temporal properties. Special derived rules are defined to prove certain temporal properties without induction term. A derived rule for proving safety properties is as follows.

$$\text{(ind alw)}\ \frac{\Gamma, \bullet(\bigwedge \Gamma \to \bigvee \Delta)\ \textbf{until}\ \neg\varphi \vdash \Delta}{\Gamma \vdash \Box\varphi, \Delta}$$

Formula $\Box\varphi$ in the succedent corresponds to a liveness condition $\Diamond\neg\varphi$ in the antecedent and therefore the induction is performed over the number of steps it takes to satisfy $\neg\varphi$. Special rules without induction terms can be given for all safety properties; a term T must only be provided to establish liveness conditions.

2.5 Sequencing

Executing concurrent systems is costly, because in every step typically a number of nondeterministic transitions can be executed leading to large proofs. However, executing several transitions often leads to the same state, no matter in which order they are executed. In this case, an extended sequent rule

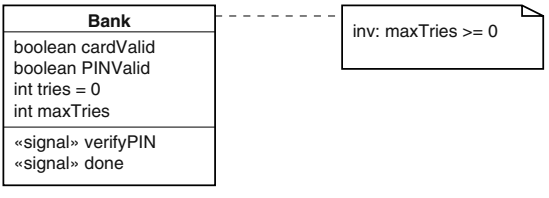
$$\text{(seq)}\ \frac{\Gamma, \rho_1 \vee \rho_2 \vdash \Delta}{\Gamma, \rho_1 \vdash \Delta \qquad \Gamma, \rho_2 \vdash \Delta}$$

can be applied which contains two conclusions to contract two proof obligations with the same temporal formulae. Additional predicate logical formulae ρ_1 and ρ_2 are combined as disjunction. This approach leads to proof graphs instead of proof trees and works best, if automatic simplification is able to significantly reduce $\rho_1 \vee \rho_2$. An example proof graph can be found in Fig. 4.

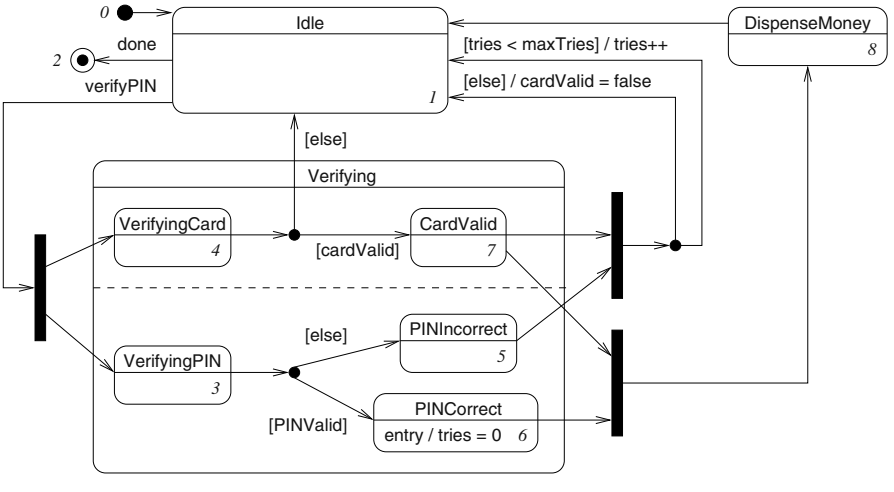
3 UML State Machines: An Example

We use a simple UML model of an automatic teller machine (ATM), shown in Fig. 1, as our running example: The class diagram in Fig. 1(a) specifies an (active) class Bank. Classes define *attributes*, i.e., local variables of its instances, and *operations* and *signals* that may be invoked on instances by call and send actions, respectively. Additionally, *invariants* restrict the state space of class instances.

The state machine for class Bank is shown in Fig. 1(b), consisting of *states* and *transitions* between states (we number the states for short reference later on). States can be *simple* (such as Idle and PINCorrect) or *composite* (such as Verifying); a *concurrent* composite state contains several *orthogonal regions*, separated by dashed lines. Moreover, *fork* and *join* (pseudo-)states, shown as bars, synchronize several transitions to and from orthogonal regions; *junction* (pseudo-)states, represented as filled circles, chain together multiple transitions. Transitions between states are triggered by *events*. Transitions may also be guarded by conditions and specify actions to be executed or events to be emitted when the transition is fired. For example, the transition leading from state Idle to the fork pseudostate requires signal verifyPIN to be present; the transition branch from VerifyingCard to CardValid requires the guard cardValid to be true; the transition branches from CardValid to Idle set the Bank attributes tries and cardValid. Events may also be emitted by *entry* and *exit* actions that are executed when a state is activated or deactivated. Transitions without an explicit trigger (e.g. the transition leaving DispenseMoney), are called *completion transitions* and are triggered by *completion events* which are emitted when a state completes all its internal activities.



(a) Class diagram



(b) State machine diagram SC_{Bank} for class Bank

Fig. 1. UML model of an ATM

The actual state of a state machine is given by its *active state configuration* and by the contents of its *event queue*. The active state configuration is the tree of active states; in particular, for every active concurrent composite state each of its orthogonal regions is active. The event queue holds the events that have not yet been handled. The *event dispatcher* dequeues the first event from the queue; the event is then processed in a *run-to-completion* (RTC) step. First, a maximally consistent set of enabled transitions is chosen: a transition is *enabled* if all of its source states are contained in the active state configuration, if its trigger is matched by the current event, and if its guard is true; two enabled transitions are *consistent* if they do not share a source state. For each transition in the set, its *least common ancestor* (LCA) is determined, i.e. the lowest composite state that contains all the transition's source and target states. The transition's main source state, that is the direct substate of the LCA containing the source states, is deactivated, the transition's actions are executed, and its target states are activated.

The example state machine simulates card and PIN validation of a bank computer. After initialization the bank computer is in state Idle. The reception of signal done leads to finalizing the state machine, whereas on reception of signal verifyPIN the verification process is started in state Verifying. If the card is invalid, the bank computer immediately returns to state Idle. If the PIN is invalid, it is checked whether the maximum number of trials is exceeded. If this is the case, the card is marked invalid; otherwise the number of trials is incremented by one. In both cases, the bank computer returns to state Idle. If the PIN is valid, the number of trials is reset to zero. If both the PIN and the card are valid, state DispenseMoney is entered from which the bank computer returns to state Idle.

4 Semantics of UML State Machines

The semantics of UML state machines is defined by an execution algorithm. This algorithm forms the basis for embedding UML state machines into temporal logic and, in particular, the symbolic execution technique. Our semantical account of UML state machines follows the semantics definition of the UML 1.5 specification [14] as closely as possible, but fills in some of the gaps of the specification. We take up the ideas presented by Lilius and Porres [11]; however, we use a simplified notion of compound transitions and correct the definition of maximally consistent sets of compound transitions.

We first define the abstract syntax of the sublanguage of UML state machines from the UML specification [14] for which our semantics is valid. Apart from those language constructs which we do not discuss here (i.e. history, sync, and choice pseudostates, call and deferred events, and internal transitions), this definition introduces the following restriction: A transition from an initial pseudostate must target a non-pseudostate contained in the same composite state as the initial pseudostate. Abandoning this restriction would lead to a more intricate definition of compound transitions, which, however, has no relevance in most practical applications.

The semantics of UML state machines is defined in two steps: First, we describe a procedure for statically computing the configurations and the compound transitions of a state machine. Due to our syntactical restriction, compound transitions are trees of state machine transitions (describing the chaining of junctions, forks, and entries of states) with a possible fan-in prefix (describing joins) that represent a move from a configura-

tion into another configuration. Second, we define an algorithm for run-to-completion steps which first computes a maximally consistent set of compound transitions for a given event and then executes the set of compound transitions.

4.1 Abstract Syntax of UML State Machines

We assume an expression language Exp that at least includes boolean expressions (like true, false, $e_1 \wedge e_2$, etc.) and an action language Act that at least includes a skip statement, and sequential (;) and parallel ($\|$) composition of statements. Furthermore, we assume a set of events $Event$ which includes $*$ denoting a completion event.

A *state* s has a kind $kind(s) \in \{\mathsf{initial}, \mathsf{final}, \mathsf{simple}, \mathsf{composite}, \mathsf{concurrent}, \mathsf{junction}, \mathsf{join}, \mathsf{fork}\}$, an entry action $entry(s) \in Act$, and an exit action $exit(s) \in Act$. A *pseudostate* is a state s with $kind(s) \in \{\mathsf{initial}, \mathsf{junction}, \mathsf{join}, \mathsf{fork}\}$; we require that $entry(s) = \mathsf{skip}$ and $exit(s) = \mathsf{skip}$ for each pseudostate s. A *composite* state is a state s with $kind(s) \in \{\mathsf{composite}, \mathsf{concurrent}\}$.

A *state hierarchy* is given by a tree (S, E) where S is a finite set of states and $E \subseteq S \times S$ a non-empty substate relation such that the constraints below are satisfied. We write $substates(s) = \{s' \in S \mid (s, s') \in E\}$ for the substates of state s:

1. If $substates(s) \neq \emptyset$ then $kind(s) \in \{\mathsf{composite}, \mathsf{concurrent}\}$.
2. If $kind(s) = \mathsf{concurrent}$ then $\#substates(s) \geq 2$ and $kind(s') = \mathsf{composite}$ for all $s' \in substates(s)$.
3. If $kind(s) = \mathsf{composite}$ then $\#\{s \in substates(s) \mid kind(s') = \mathsf{initial}\} \leq 1$.

We further write $container(s)$ for the container state of state s if s is not the root state; $substates^+(s) = \{s' \in S \mid (s, s') \in E^+\}$ and $substates^*(s) = substates^+(s) \cup \{s\}$ denote the set of (reflexive) transitive substates of s; and $initial(s)$ is the initial state contained in the composite state s if it exists. The *least common ancestor* of a set of states $M \subseteq S$ not containing the root state, denoted by $lca(M)$, is the least composite state c w.r.t. E such that $M \subseteq substates^+(c)$; the *least common reflexive ancestor* of $M \subseteq S$, written $lca^=(M)$, is the least state s w.r.t. E such that $M \subseteq substates^*(s)$.

Given a state hierarchy $H = (S, E)$, a *transition* t over H has a source state $source(t) \in S$, a target state $target(t) \in S$, a triggering event $trigger(t) \in Event$, a guard expression $guard(t) \in Exp$, and an effect action $effect(t) \in Act$, such that the following constraints are satisfied:

1. $kind(source(t)) \neq \mathsf{final}$ and $kind(target(t)) \neq \mathsf{initial}$.
2. If $kind(source(t)) \in \{\mathsf{initial}, \mathsf{fork}\}$ then $target(t)$ is not a pseudostate.
3. If $kind(source(t)) = \mathsf{initial}$ then $container(target(t)) = container(source(t))$.
4. If $kind(target(t)) = \mathsf{join}$ then $source(t)$ is not a pseudostate.
5. If $kind(source(t)) = \mathsf{composite}$ then $kind(container(source(t))) \neq \mathsf{concurrent}$.
6. If $kind(target(t)) = \mathsf{composite}$ then $kind(container(target(t))) \neq \mathsf{concurrent}$.
7. If $kind(source(t)) \in \{\mathsf{initial}, \mathsf{fork}, \mathsf{join}\}$ then $guard(t) = \mathsf{true}$.
8. If $kind(target(t)) = \mathsf{join}$ then $guard(t) = \mathsf{true}$.
9. If $kind(source(t)) = \mathsf{initial}$ then $effect(t) = \mathsf{skip}$.
10. If $source(t)$ is a pseudostate, then $trigger(t) = *$.

A *state machine* (for a class C) is given by a pair (H, T) where $H = (S, E)$ is a state hierarchy and T a finite set of transitions over H such that the constraints below are satisfied for all $t \in T$. We write $outgoings(s)$ for the set $\{t \in T \mid source(t) = s\}$; $incomings(s)$ for the set $\{t \in T \mid target(t) = s\}$; $sources(M)$ for the set $\{source(t) \mid t \in M\}$; and $targets(M)$ for the set $\{target(t) \mid t \in M\}$:

1. If $kind(s) =$ initial then $\#outgoings(s) = 1$.
2. If $kind(s) =$ junction then $\#incomings(s) = 1$ and $\#outgoings(s) \geq 1$.
3. If $kind(s) =$ fork then $\#incomings(s) = 1$ and $\#outgoings(s) \geq 2$.
4. If $kind(s) =$ fork then there is an $s' \in S$ with $kind(s') =$ concurrent such that $targets(outgoings(s)) \subseteq substates^+(s') \setminus substates(s')$ and the following holds: if $t, t' \in outgoings(s)$ such that $\{target(t), target(t')\} \subseteq substates^+(s'')$ for some $s'' \in substates^+(s')$ then $t = t'$.
5. If $kind(s) =$ join then conditions (3) and (4) hold likewise with replacing $target$ by $source$ and $outgoings$ by $incomings$.

Conditions (4) and (5) require forks and joins to come from and go to different orthogonal regions of a concurrent composite state.

4.2 Configurations and Compound Transitions

The *configurations* of a state machine $((S, E), T)$ are given by the smallest subsets C of S that satisfy the following conditions:

1. The root state of S is in C.
2. No state $s \in C$ is a pseudostate.
3. If $kind(s) =$ composite then there is a single $s' \in C$ such that $container(s') = s$.
4. If $kind(s) =$ concurrent then all states $s' \in S$ with $container(s') = s$ are in C.

In particular, composite states are or-states, concurrent states are and-states.

The *compound transitions* of a state machine $((S, E), T)$ represent semantically complete transition paths that originate from a set of non-pseudostates and target a set of simple states. More precisely, a compound transition consists of three parts: The optional tail part of a compound transition may have multiple transitions in T originating from a set of mutually orthogonal regions that are joined by a join pseudostate. The middle part of a compound transition is a finite chain of transitions in T joined via junction pseudostates. Finally, the optional head part of a compound transition is a tree of transitions in T: If a transition in the middle part of a compound transition or in its head part itself targets a composite state the head part continues at the initial state of this composite transition; if a transition targets a concurrent composite state the head part continues at all initial states of the orthogonal regions of the concurrent composite state; if a transition targets a fork pseudostate the head part continues with the transitions outgoing from the fork pseudostate which target mutually orthogonal regions and simultaneously continues at the initial states of all those orthogonal regions that are not targeted by transitions outgoing from the fork pseudostate.

In the ATM example, the compound transitions outgoing from VerifyingCard just consist of middle parts:

$\langle\{\}, \langle \mathsf{VerifyingCard} \rightarrow \mathsf{junction}, \mathsf{junction} \rightarrow \mathsf{Idle}\rangle, \{\}\rangle,$
$\langle\{\}, \langle \mathsf{VerifyingCard} \rightarrow \mathsf{junction}, \mathsf{junction} \rightarrow \mathsf{CardValid}\rangle, \{\}\rangle.$

The fork transition from Idle consists of a middle part and a tail part:

$$\langle\{\}, \langle \mathsf{Idle} \rightarrow \mathsf{fork}\rangle, \{\langle \mathsf{fork} \rightarrow \mathsf{VerifyingCard}\rangle, \langle \mathsf{fork} \rightarrow \mathsf{VerifyingPIN}\rangle\}\rangle \ .$$

The join transition to DispenseMoney consists of a head part and a middle part:

$$\langle\{\langle \mathsf{CardValid} \rightarrow \mathsf{join}\rangle, \langle \mathsf{PINCorrect} \rightarrow \mathsf{join}\rangle\}, \langle \mathsf{join} \rightarrow \mathsf{DispenseMoney}\rangle, \{\}\rangle \ .$$

The algorithm for computing the compound transitions outgoing from a non-pseudostate in S of a state machine $((S, E), T)$ relies on a procedure that computes the middle and head parts of compound transitions, also called *forward trees*, outgoing from an arbitrary state in S; the details of the algorithms can be found in [8]. Note that our definition of compound transitions deviates from the explanations in the UML specification [14]. There, compound transitions are not required to target simple states only, but may as well stop at composite states. The proper initialization of composite and concurrent composite states is left to the entry procedure for composite states.

The notions of source states, target states, trigger, guard, and effect are transferred from transitions to compound transitions in the following way: The *source states* of a compound transition τ, written $sources(\tau)$, are the source states of the transitions in the tail part of τ, if τ shows a tail part, and the source state of the middle part, otherwise. Analogously, the target states of τ, written $targets(\tau)$, are the target states of the transitions in the head part of τ, if τ shows a head part, and the target state of the middle part, otherwise. The *trigger* of τ is the set of triggers of the transitions in the tail part of τ, if τ shows a tail part, and the trigger of the first transition in the middle part otherwise. The *guard* of τ is the conjunction of all guards of transitions in τ. Finally, the *effect* of τ is the sequential composition of the effects of the tail, the middle, and the head part of τ, where the effects in the tail and the head are conjoined in parallel whereas the effects in the middle part are composed sequentially. These definitions are naturally extended to sets of compound transitions which show the same trigger.

We recall some notions on compound transitions τ from the UML specification that will be used for the definition of the execution semantics of state machines, in particular, when computing maximally conflict free sets of compound transitions in a given configuration C: The *main source state* of τ, $mainSource(\tau)$, is given by the state $s = lca(lca^{=}(sources(\tau)), lca^{=}(targets(\tau)))$ if $kind(s) = \mathsf{concurrent}$, and it is given by the state $s' \in substates(s)$ with $lca^{=}(sources(\tau)) \in substates^{*}(s')$, otherwise. The *main target state* of τ, $mainTarget(\tau)$ is defined analogously, but exchanging $sources(\tau)$ and $targets(\tau)$. The set of states *exited* by τ in configuration C, $exited(C, \tau)$, consists of $substates^{*}(mainSource(\tau)) \cap C$. The set of states *entered* by τ in configuration C, $entered(\tau)$, is $substates^{*}(mainTarget(\tau)) \cap C$. Again, the definitions for $entered$ and $exited$ are naturally extended to sets of compound transitions.

Two compound transitions τ_1 and τ_2 are *in conflict* in configuration C, written $\tau_1 \sharp_C \tau_2$, if $exited(C, \tau_1) \cap exited(C, \tau_2) \neq \emptyset$; more generally, a compound transition τ is in conflict with a set of compound transitions T in configuration C, written $\tau \sharp_C T$, if

$\tau \sharp_C \tau'$ for some $\tau' \in T$. If $\tau_1 \sharp_C \tau_2$ let S_1 and S_2 be the sets of states in $sources(\tau_1)$ and $sources(\tau_2)$, resp., that show the maximal numerical distance from the root state of (S, E); τ_1 is *prioritized* over τ_2 in configuration C, written $\tau_1 \prec_C \tau_2$, if $S_1 \subseteq substates^+(S_2)$. Again, $\tau \prec_C T$ for a compound transition τ and a set of compound transitions T in configuration C if $\tau \prec_C \tau'$ for some $\tau' \in T$.

4.3 Run-to-Completion Semantics

The execution semantics of a UML state machine is described in the UML specification as a sequence of *run-to-completion steps*. Each such step is a move from a configuration of the state machine to another configuration. The sequence of steps starts in the *initial configuration* of the state machine, i.e., the configuration that is targeted by the forward tree outgoing from the initial state of the root state of the state hierarchy. In a run-to-completion step from some configuration, first, an event is fetched from the event queue. Second, a maximally consistent set of enabled compound transitions outgoing from the states of the current configuration and whose guards are satisfied is chosen. If such a set, called a *step*, exists, all its compound transitions are fired simultaneously: First, all states that are exited by the step are deactivated in an inside-out manner, executing the exit actions of these states; each such that is marked to be not completed, as it is not part of the configuration any more. Second, the gathered effect of the step is executed. Third, all states entered by the step are activated in an outside-in manner, executing the entry actions of these states. Furthermore, after executing the entry action of a state this state is marked as complete, i.e. a completion event for this state is generated.

More formally, let $((S, E), T)$ be a state machine. We assume a structure of *environments* η for state machines that provides the following primitive operations: An event can be fetched by $\texttt{fetch}(\eta)$; the completion of a state s can be recorded by $\texttt{complete}(\eta, s)$; the revocation of a state s from being completed can be recorded by $\texttt{uncomplete}(\eta, s)$; a statement a can be executed by $\texttt{exec}(\eta, a)$; given a configuration C and an event e all compound transitions of $((S, E), T)$ that are triggered by e can be computed by $\texttt{enabled}(\eta, C, e)$; and, finally, the validity of an expression g can be checked by $\eta \models g$. The enabledness of compound transitions in a configuration C by an event e is indeed solely defined on the basis of the triggers of compound transitions and thus only involves the completed states that have been previously recorded with the environment. The fireable sets of compound transitions, which are maximally consistent sets of enabled compound transitions are computed by the steps algorithm in Fig. 2(a). The execution of a state machine in some configuration and some environment is defined by the RTC algorithm in Fig. 2(b) which uses the algorithm for firing a compound transitions step in Fig. 2(c).

5 Embedding of UML State Machines

Our goal is to make use of symbolic execution as interactive proof method for UML state machines. Embedding state machines into our temporal framework of Sect. 2 requires first to define state machines as temporal formulae and to extend our calculus with rules for their symbolic execution.

```
steps(env, conf, event) ≡
  ⌈transitions ← enabled(env, conf, event)
    {step | ⟨guard, step⟩ ∈ steps(conf, transitions) ∧ env ⊨ guard}⌋

steps(conf, transitions) ≡
  ⌈steps ← {⟨false, ∅⟩}
   for transition ∈ transitions do
      for ⟨guard, step⟩ ∈ steps(transitions \ {transition}) do
         if transition ♯conf step
           then if transition ≺conf step
                   then guard ← guard ∧ ¬ guard(transition) fi
           else step ← step ∪ {transition}
                guard ← guard ∧ guard(transition) fi
         steps ← steps ∪ {⟨guard, step⟩} od od
   steps⌋
```

(a) Transition selection algorithm

```
RTC(env, conf) ≡
  ⌈⟨event, env⟩ ← fetch(env)
   steps ← steps(env, conf, event)
   if steps ≠ ∅
     then choose step ∈ steps
          ⟨env, conf⟩ ← fire(env, conf, step) fi
   ⟨env, conf⟩⌋
```

(b) Run-to-completion step algorithm

```
fire(env, conf, step) ≡
  ⌈for state ∈ insideOut(exited(conf, step)) do
      env ← exec(env, exit(state))
      conf ← conf \ {state}
      env ← uncomplete(env, state) od
   env ← exec(env, effect(step))
   for state ∈ outsideIn(entered(conf, step)) do
      env ← exec(env, entry(state))
      conf ← conf ∪ {state}
      env ← complete(env, state) od
   ⟨env, conf⟩⌋
```

(c) Transition firing algorithm

Fig. 2. State machine execution algorithms

State machines $((S, E), T)$ are embedded into the temporal logic as a special formula $[((S, E), T)]$ with the following semantics:

$$\pi \models [((S, E), T)] \quad \text{iff} \quad \pi \text{ is a valid trace of } ((S, E), T)$$

for sequences of valuations π. The definition of a valid trace is not given here, as it is straightforward to derive from the definition of a single step as was defined in Sect. 4. A state machine configuration is a formula *conf* which represents the active states: for each state, a corresponding dynamic boolean variable is true if and only if the state is part of the current active state configuration of the state machine. Attributes and the event queue are represented as algebraic datatypes.

To define the calculus, we implement the function $exec$ of rule (execute), which we described in Sect. 2.2, by setting

$$\begin{aligned} exec([((S, E), T)], conf) = \{\langle guard \wedge \langle action \rangle_{\mathrm{DL}} V' = V, [((S, E), T)] \wedge conf_0 \rangle \mid \\ \langle guard, step \rangle \in \mathsf{steps}(conf, enabled(T)), \\ \langle action, conf_0 \rangle = \mathsf{fireaction}(conf, step)\} \ . \end{aligned}$$

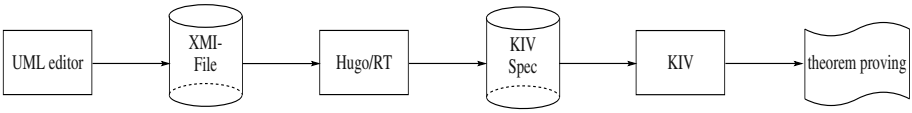


Fig. 3. Tool chain to translate UML models into KIV

Function $exec$ provides configuration *conf* and the enabled transitions in T to the function steps of Fig. 2(a) which returns pairs of possible steps *step* and their corresponding guards *guard*. The function $enabled$ uses the head element of the event queue as the current triggering event. We cannot use function fire of Fig. 2(c) to execute *step* as we cannot provide an explicit environment env. Therefore, we define a similar function fireaction which returns instead of a modified environment an action *action* which combines all of the environment updates as a sequential program. The guard, the action and the modified configuration $conf_0$ are combined to form the result of $exec$. (The special formula $\langle action \rangle_{\text{DL}} V' = V$ of Dynamic Logic (DL [6]) takes the unprimed values of variables V as input to *action*, modifies the unprimed variables according to the sequential program and uses the modified variables to evaluate formula $V' = V$. This formula is used to "copy" the new values of the unprimed variables to the primed variables V'.)

This algorithm can be further optimized to ensure that the number of steps which are returned by steps is as small as possible: the context formulae which represent the symbolic environment can be used to automatically simplify guards, and the state hierarchy is exploited to faster detect transition priorities.

We implemented the calculus described above in the interactive verifier KIV. To complete the integration, we use a standard UML modelling tool (e.g. ArgoUML [1]) to draw state machines, and translate the resulting XMI-files to KIV with the model translator Hugo/RT [9]. The complete tool chain is shown in Fig. 3. This translation is fully automatic.

6 An Example Proof

The proof method for UML state machines is very simple: we repeatedly execute steps and test whether we have already encountered the current active state configuration earlier in the proof; in this case, we have executed a cycle and the current goal can be closed with an inductive argument.

As example proof we show for state machine SC_{Bank} that the state DispenseMoney can only be entered, if tries $\leq$ max-tries and PINValid is true. So the property to show is

(prop) $\Box(\mathsf{DispenseMoney} \rightarrow \mathsf{tries} \leq \mathsf{maxTries} \wedge \mathsf{PINValid})$.

With initial state machine configuration $C \equiv \mathsf{Idle}$ and variable condition $I \equiv \mathsf{tries} = 0 \wedge \mathsf{maxTries} \geq 0$ the proof obligation is

(init) $[SC_{\mathsf{Bank}}], \mathsf{Idle}, \mathsf{tries} = 0 \wedge \mathsf{maxTries} \geq 0 \vdash$ (prop) .

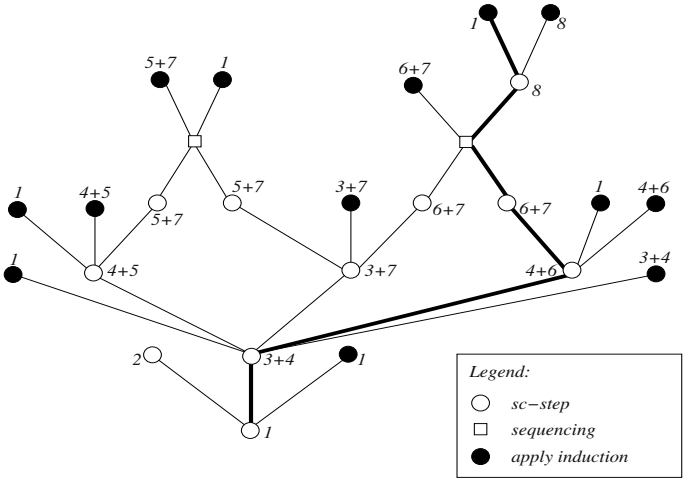


Fig. 4. Proof graph for the example proof

As precondition, we assume that the state machine SC_{Bank} of Fig. 1 is in state Idle and the initial conditions of the class diagram hold. The event queue is unspecified; we make no demands on the events in the initial queue and the queue of the following steps. In other words, the environment behaves arbitrarily. Note that the safety property of (prop) is trivially true for all states with ¬DispenseMoney.

The complete proof is shown in Fig. 4. To demonstrate our proof scheme we explain only the highlighted path. This exemplary run corresponds to the main use case scenario of the automatic teller machine. All other runs of the state machine can be shown following the same pattern. For better understanding of the proof graph, we denoted the nodes with the numbers of the actual states. The state numbers can be found in the state machine diagram (see Fig. 1).

The state machine will always return to state Idle (1). To apply our induction scheme, we must choose an appropriate invariant, because the variable tries may increase in each cycle. We use

(inv) $\mathsf{CardValid} \rightarrow \mathsf{tries} \leq \mathsf{maxTries}$

which can be easily shown with (init). This invariant is preserved in each cycle. Then, to use the current sequent later as induction hypothesis, the rule (ind alw) is applied.

Now we can apply the rule *execute* to advance one step in our state machine:

$$\frac{\begin{array}{ll}(a) & \mathsf{done} \wedge \langle \mathbf{skip} \rangle_{\mathrm{DL}} \cdots \wedge \circ([SC_{\mathsf{Bank}}] \wedge \mathsf{Finalized}), (\mathrm{inv}) \vdash (\mathrm{prop}) \\ (b) & \mathsf{verifyPIN} \wedge \langle \mathbf{skip} \rangle_{\mathrm{DL}} \cdots \wedge \circ([SC_{\mathsf{Bank}}] \wedge \\ & \mathsf{Verifying} \wedge \mathsf{VerifyingCard} \wedge \mathsf{VerifyingPIN}), (\mathrm{inv}) \vdash (\mathrm{prop}) \\ (c) & \neg(\mathsf{verifyPIN} \wedge \mathsf{done}) \wedge \langle \mathbf{skip} \rangle_{\mathrm{DL}} \cdots \wedge \circ([SC_{\mathsf{Bank}}] \wedge \mathsf{Idle}), (\mathrm{inv}) \vdash (\mathrm{prop})\end{array}}{[SC_{\mathsf{Bank}}], \mathsf{Idle}, (\mathrm{inv}) \vdash (\mathrm{prop})}$$

The lowest node in the proof graph (Fig. 4) depicts the application of this rule. As expected from the state machine diagram, we get three premises: Either the state

machine is finalized (2) with the event done (premise (a)) or the transition to state Verifying $(3+4)$ is taken with event verifyPIN (premise (b)) or the state machine remains in state Idle (1) with any other event (premise (c)). In all cases the action is **skip** and a special formula (omitted) assigns the values of all unprimed variables to their primed complement. Premise (a) terminates and the proof is complete, premise (c) loops and induction is applied. The only nontrivial case is premise (b). After simplification and application of rule (step) we receive

$$[SC_{\mathsf{Bank}}], \mathsf{Verifying} \wedge \mathsf{VerifyingCard} \wedge \mathsf{VerifyingPIN}, (\mathrm{inv}) \vdash (\mathrm{prop}) \ .$$

The new active states are Verifying and its substates VerifyingCard (4) and VerifyingPIN (3) and (inv) is preserved. Symbolic execution continues with the next step where we receive four different states for each possible transition and one state if no transition is enabled. We focus on the state Verifying, VerifyingCard (4) and PINCorrect (6). This transition is only taken, if the attribute PINValid is true, so we can assume PINValid to be true for the following steps. Like before, we invoke the next step. One of the next states is Verifying, CardValid (7) and PINCorrect (6). This transition is only possible if the attribute CardValid is true and due to the invariant we can conclude that tries $\leq$ maxTries. The same state is also reachable by first entering CardValid (7) and then entering PINCorrect (6). Both cases can be merged by sequencing, so there is only one open goal. From this state we reach DispenseMoney (8). Because we have shown PINValid and tries $\leq$ maxTries in the previous two steps property (prop) holds in this state. The next step leads again to the state Idle (1), where we use the induction hypothesis and close the current goal.

This proof scheme can be further automated. Our goal is to adapt the heuristics, so that the only user interaction is the specification of an adequate invariant.

7 Conclusion

We have demonstrated how to integrate UML state machines into a proof method to interactively verify temporal properties of state machines. As a first step, we have defined a formal semantics of UML state machines and derived a proof rule for their symbolic execution. As can be seen in the example, the resulting proofs are very intuitive as they follow the different possible system runs. However, only small examples have been tackled so far.

Symbolic execution in general turns out to be a very intuitive approach not only to the verification of sequential programs but also to the construction of proofs for concurrent systems, with the potential to dispel the prejudice that interactive proofs in temporal logic are difficult. Proof rules for the execution of (most of) the temporal operators are invertible ensuring that the calculus can be automated to a large extent. A generic induction principle can be used to establish both safety and liveness properties. Sequencing is used to avoid exponential growth of proof trees for nondeterministic systems. A strategy to automatically apply sequencing is the key to efficiently use the approach for larger case studies. As has been demonstrated with the integration of UML state machines, proof rules for the symbolic execution of different formalisms can be implemented, provided that a formal operational semantics is defined.

Next steps are to further automate proofs, especially to automate sequencing, and to look into compositional proofs. We expect that proofs like the one in Sect. 6 require the invariant as the only user interaction. Very important for the future is to gain more experience in larger case studies.

References

1. ArgoUML homepage. argouml.tigris.org.
2. M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying Concurrent Systems with Symbolic Execution. *J. Logic Computat.*, 12(4):549–560, 2002.
3. M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for Provably Correct Systems. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Proc. Int. Wsh. Applied Formal Methods (FM-Trends'98)*, volume 1641 of *Lect. Notes Comp. Sci.*, pages 330–337. Springer, Berlin, 1999.
4. Dove homepage. http://www.dsto.defence.gov.au/isl/dove/dove.html.
5. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comp. Program.*, 8(3):231–274, 1987.
6. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. The MIT Press, Cambridge, Mass.–London, 2000.
7. M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. R. Meyer and M. A. Taitslin, editors, *Proc. Symp. Logical Foundations of Computer Science (Logic at Botik '89)*, volume 363 of *Lect. Notes Comp. Sci.*, pages 134–145. Springer, Berlin, 1989.
8. A. Knapp. Semantics of UML State Machines. Technical Report 0408, Institut für Informatik, Ludwig-Maximilians-Universität München, 2004.
9. A. Knapp and S. Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In D. Haneberg, G. Schellhorn, and W. Reif, editors, *Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64. Technical Report 2002-11, Institut für Informatik, Universität Augsburg, 2002.
10. D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.
11. J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 430–445. Springer, Berlin, 1999.
12. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer, Berlin—&c., 1995.
13. B. Moszkowski. A Temporal Logic for Multilevel Reasoning about Hardware. *IEEE Computer*, 18(2):10–19, 1985.
14. Object Management Group. Unified Modeling Language Specification, Version 1.5. Specification, OMG, 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.
15. Omega homepage. www-omega.imag.fr.
16. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. In S. Stoller and W. Visser, editors, *Proc. Wsh. Software Model Checking*, volume 55(3) of *Elect. Notes Theo. Comp. Sci.*, Paris, 2001. 13 pages.
17. K. Stenzel. A Formally Verified Calculus for Full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proc. 10th Int. Conf. Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *Lect. Notes Comp. Sci.* Springer, Berlin, 2004.