# Verifying Behavioral UML Systems via CEGAR

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

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

**Abstract.** This work presents a novel approach for applying abstraction and refinement in the verification of behavioral UML models.

The *Unified Modeling Language* (UML) is a widely accepted modeling language for embedded and safety critical systems. As such the correct behavior of systems represented as UML models is crucial. *Model checking* is a successful automated verification technique for checking whether a system satisfies a desired property. Nevertheless, its applicability is often impeded by its high time and memory requirements. A successful approach to avoiding this limitation is *CounterExample-Guided Abstraction-Refinement (CEGAR)*. We propose a CEGAR-like approach for UML systems. We present a model-to-model transformation that generates an *abstract UML system* from a given concrete one, and formally prove that our transformation creates an *over-approximation*.

The abstract system is often much smaller, thus model checking is easier. Because the abstraction creates an over-approximation we are guaranteed that if the abstract model satisfies the property then so does the concrete one. If not, we check whether the resulting abstract counterexample is *spurious*. In case it is, we automatically *refine* the abstract system, in order to obtain a more precise abstraction.

## 1 Introduction

This work presents a novel approach for applying abstraction and refinement for the verification of behavioral UML models. The *Unified Modeling Language* (UML) [2] is a widely accepted modeling language that can be used to specify and construct systems. It provides means to represent a system in terms of classes and their relationships, and to describe the systems' internal structure and behavior. UML has been developed as a standard object-oriented modeling language by the Object Management Group (OMG) [11]. It is becoming the dominant modeling language for embedded and safety critical systems. As such, the correct behavior of systems represented as UML models is crucial and verification techniques applicable to such models are required.

*Model checking* [6] is a successful automated verification technique for checking whether a given system satisfies a desired property. It traverses *all* of the system behaviors, and either confirms that the system is correct w.r.t. the checked property, or provides a *counterexample (CEX)* that demonstrates an erroneous behavior. Model checking is widely recognized as an important approach to increasing reliability of hardware and software systems and is vastly used in industry.

E. Albert and E. Sekerinski (Eds.): IFM 2014, LNCS 8739, pp. 139–154, 2014.

Unfortunately, the applicability of model checking is impeded by its high time and memory requirements. One of the most successful approaches for fighting these problems is *abstraction*, where some of the system details are hidden. This results in an *over-approximated* system that has *more behaviors and less states* than the concrete (original) system. The abstract system has the feature that if a property holds on the abstract system, then it also holds on the concrete system. However, if the property does not hold, then nothing can be concluded of the concrete system. *CounterExample-Guided Abstraction Refinement (CEGAR)* approach [4] provides an automatic and iterative framework for abstraction and refinement, where the refinement is based on a spurious CEX. When model checking returns an abstract CEX, a matching concrete CEX is searched. If there exists one, then a real bug on the concrete system is found. Otherwise, the CEX is *spurious* and a refinement is needed. During refinement, more details are added to the abstract system, in order to eliminate the spurious CEX.

In this paper we focus on behavioral systems that rely on *UML state machines*. UML state machines are a standard graphical language for modeling the behavior of event-driven software components. We propose a CEGAR-like framework for verifying such systems. We present a model-to-model transformation that generates an *abstract system* from a given concrete one. Our transformation is done on the UML level, thus resulting in a *new UML behavioral system* which is an *over-approximation* of the original system. We adapt the CEGAR approach to our UML framework, and apply refinement if needed. Our refinement is also performed as a model-to-model transformation. It is important to note that by defining abstraction and refinement in terms of model-to-model transformation, we avoid the translation to lower level representation (such as Kripke structures). This is highly beneficial to the user, since both the property, the abstraction, and the abstract CEX are given on the UML level and are therefore more meaningful.

Our abstraction is obtained by abstracting some (or all) of the state machines in the concrete system. When abstracting a state machine, we over-approximate its *interface behavior* w.r.t. the rest of the system. In the context of behavioral UML systems, the interface includes the events generated/consumed and the (non-private) variables. We thus abstract part of the system's variables, and maintain an *abstract view* of the events generated by the abstracted state machines. In particular, the abstract state machines may change the number and order of the generated events. Further, abstracted variables are assigned the "don't-know" value. Our abstraction does not necessarily replace an *entire* state machine. Rather, it enables abstracting *different parts* of a state machine whose behavior is irrelevant to the checked property. We present our abstraction construction in section 4.

We show that the abstract system is an over-approximation by proving that for every concrete system computation there exists an abstract system computation that "behaves similarly". This is formally defined and proved in section 5. To formalize the notion of *system computation*, we present in section 3 a formal semantics for behavioral UML systems that rely on state machines. Works such as [7,10,15] also give formal semantics to state machines, however they all

differ from our semantics: e.g. [7] defines the semantics on flat state machines and present a translation from hierarchical to flat state machines, whereas we maintain the hierarchical structure of the state machines. [10] define the semantics of a *single* state machine. Thus it neither addresses the semantics of the full system, nor the communication between state machines. [15] addresses the communication of state machines, however their notion of run-to-completion step does not enable context switches during a run-to-completion step. Our formal semantics is defined for a *system*, possibly multi-threaded, where the atomicity level is a transition execution (formally defined later).

Our CEGAR framework is suitable for verifying $LTL_x$, which is the Linear-time Temporal Logic (LTL)[22] without the next-time operator. Also, we assume the existence of a model checker for behavioral UML systems. Extensive work has been done in the last years to provide such model checkers by translating the system into an input language of some model checker. [3,5] present translation of state machines to SMV. Several works [18,14,21,1,8] translate state machines to PROMELA, which is the input language of the model checker SPIN. A verification environment for UML behavioral models was developed in the context of the European research project OMEGA [20], and works such as [25,19] apply different methods for model checking these models. [12,16] translate a UML behavioral model to C code, and apply bounded model checking via CBMC. We add the special value "don't-know" to the domain of the variables. This results in a 3-valued semantics for UML systems, as shown in section 4. To model check abstract systems we need a 3-valued model checker. Extending a model checker to support the 3-valued semantics (e.g., [27,13]) is straightforward.

Many works such as [26,28,24,9,23] address *semantic refinement* of state machines, which is adding details to a partially defined state machine while preserving behavior of the original (abstracted) model. Though we also address an abstraction-refinement relation between state machines, these works are very different from ours. These works look at manual refinement as part of the modeling process, whereas we are suggesting an *automatic* abstraction and refinement, and our goal is improving scalability of the verification tool. Moreover, these works handle a single state machine level, where we consider a system which includes possibly many state machines that interact with each other. To the best of our knowledge, this is the first work that addresses the abstraction for a behavioral UML system at the UML level.

## 2 Preliminaries - UML Behavioral Systems

Behavioral UML systems include objects (instances of classes) that process events. Event processing is defined by state machines, which include complex features such as hierarchy, concurrency and communication. UML objects communicate by sending each other events (asynchronous messages) that are kept in *event queues* (EQs). Every object is associated with a single EQ, and several objects can be associated with the same EQ. In a multi-threaded system there are several EQs, one for each thread. Each thread executes a never-ending loop,

taking an event from its EQ, and dispatching it to the target object. The target object makes a *run-to-completion (RTC)* step, where it processes the event and continues execution until it cannot continue anymore. RTCs are composed of a series of *steps*, formally defined later. Only when the target object finishes its RTC, the thread dispatches the next event available in its EQ. Steps of different threads are interleaved. Next we formally define state machines, UML systems, and the set of behaviors associated with them. The following definitions closely follow the UML2 standard.

### 2.1 UML State Machines

We first define the following notions: $EV = EV_{env} \cup EV_{sys}$ is a fixed set of events, where $EV_{sys}$ includes events sent by a state machine in the system. $EV_{env}$ includes events which are considered to be sent by the "environment" of the system. An event $e$ is a pair $(type(e), trgt(e))$, where $type(e)$ denotes the event name (or type), and $trgt(e)$ denotes the state machine to which the event was sent (formally defined later). $V$ is a fixed set of variables over finite domains.

We use a running example to present state machines and behavioral UML systems. Fig. 1 describes the state machine of class $DB$. A *state machine* is a tuple $SM = (S, R, \Omega, init, TR, L)$ where $S$ and $R$ are sets of states and regions respectively. We assume $TOP \in R$. States are graphically represented as squares. $\Omega : S \cup R \rightarrow S \cup R \cup \{\epsilon\}$ represents the hierarchical structure of states and regions: for every $s \in S$, $\Omega(s) \in R$, $\Omega(TOP) = \epsilon$ and for every other $r \in R$, $\Omega(r) \in S$. E.g., in Fig 1, $\Omega(Working) = \Omega(Vacation) = TOP$. The transitive closure of $\Omega$ is irreflexive and induces a partial order. $u' \in \Omega^+(u)$ if $u'$ *contains* $u$ (possibly transitively). This is denoted $u \lhd u'$. Two different regions $r_1, r_2 \in R$ are *orthogonal*, denoted $ORTH(r_1, r_2)$, if $\Omega(r_1) = \Omega(r_2)$. Regions are graphically represented only if they are orthogonal. Orthogonal regions are denoted by a dashed line. E.g., state $Working$ contains two orthogonal regions. $init \subseteq S$ is a set of initial states, s.t. there is one initial state in each region. Initial states are marked with a transition with no source state. $TR$ is a set of transitions. Each $t \in TR$ connects a single source state, denoted $src(t)$, with a single target state, denoted $trgt(t)$. $L$ is a function that labels each transition $t$ with a trigger $(trig(t))$, a guard $(grd(t))$, and an action $(act(t))$. A trigger is a type of an event from $EV$. $\epsilon \in EV$ represents no trigger. A guard is a Boolean expression over $V$. An action is a sequence of statements in some programming language where *skip* is an empty statement. Actions can include "GEN(e)" statements, representing a generation of an event. In the graphical representation, a transition $t$ is labeled with $tr[g]/a$ where $tr = trig(t)$, $g = grd(t)$ and $a = act(t)$. If $tr = \epsilon$, $g = true$ or $a = skip$ they are omitted from the representation. We assume there exists a macro $GEN(\{e_1, ..., e_h\})$, representing a generation of *one of the events* from $\{e_1, ..., e_h\}$ non-deterministically. Given an action $act$, by abuse of notation we write $GEN(e) \in act$ iff $GEN(e)$ is one of the statements in $act$. $modif(act)$ denotes the set of variables that may be modified on $act$ (are in the left hand side of an assignment statement). By abuse of notation, $modif(t)$ denotes the set of variables that may be modified by $act(t)$.

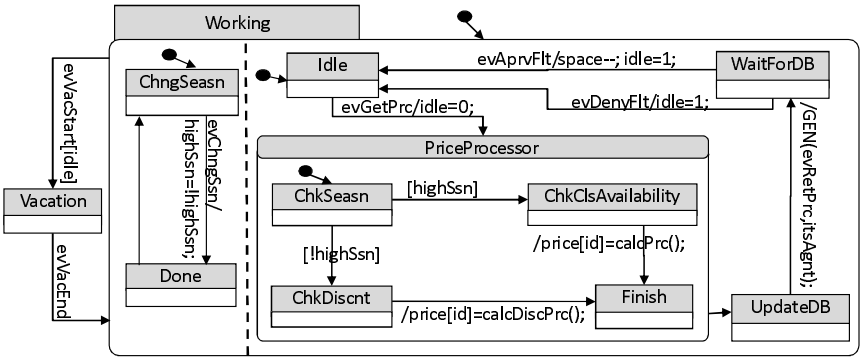


**Fig. 1.** DB State Machine

Let $SM$ be a state machine, a *state machine configuration (SM-conf)* is a tuple $c = (\omega, \rho)$ where $\rho \in type(EV) \cup \{\epsilon\}$ holds the type of an event currently *dispatched* to SM and not yet consumed, and $\omega \subseteq S$ is the set of currently active states. $\omega$ always contains a single state $s$ s.t. $\Omega(s) = TOP$. It also has the property that for every $s \in \omega$ and for every $r \in R$ s.t. $\Omega(r) = s$ there exists a *single* $s' \in S$ s.t. $\Omega(s') = r$ and $s' \in \omega$.

From here on, we assume the following restrictions on $SM$:
(1) An action includes at most one "$GEN(e)$". In addition, an action that includes "$GEN(e)$" is a non-branching sequence of statements. If either one of these restrictions does not hold, then $SM$ can be preprocessed s.t. the transition is replaced with a series of states and transitions, each executing part of the original action.
(2) $SM$ does not include the following complex UML syntactic features: history, cross-hierarchy transitions, fork, join, entry and exit actions. It is straightforward to eliminate these features, at the expense of additional states, transitions and variables. Note that the hierarchical structure of the state machines is maintained, thus avoiding the exponential blow-up incurred by flattening.

### 2.2 Systems

Next we define UML systems and their behavior. UML2 places no restrictions on the implementation of the EQ and neither do we. A finite sequence $q = (e_1, ..., e_l)$ of events $e_i \in EV$ represents the EQ at a particular point in time. We assume functions $top(q)$, $pop(q)$ and $push(q, e)$ are defined in the usual way.

A *system* is a tuple $\Gamma = (SM_1, ..., SM_n, Q_1, ..., Q_m, thrd, V)$ s.t. $SM_1, ..., SM_n$ are state machines, $Q_1, ..., Q_m$ $(m \leq n)$ are EQs (one for each thread), $thrd : \{1, ..., n\} \rightarrow \{1, ..., m\}$ assigns each $SM_i$ to a thread, and $V$ is a collection of variables over finite domains. A *system configuration ($\Gamma$-conf)* is a tuple $C = (c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \sigma)$ s.t. $c_i$ is a SM-conf of $SM_i$, $q_j$ is the contents of $Q_j$, $id_j \in \{0, ..., n\}$ is the id of the SM associated with thread $j$ that is currently executing a RTC ($id_j = 0$ means that all SMs of thread $j$ are *inactive*), and $\sigma$ is a legal assignment to all variables in $V$.

From now on we fix a given system $\Gamma = (SM_1, ..., SM_n, Q_1, ..., Q_m, thrd, V)$. We use $c$ for SM-confs and $C$ for $\Gamma$-confs. We use $k$ as a superscript to range over steps in time, making $c_i^k$ the SM-conf of $SM_i$ at time $k$. For every $e \in EV$, we define $trgt(e) \in \{0, ..., n\}$ to give the index of the $SM$ that is the target of $e$. $trgt(e) = 0$ means the event is sent to the environment of $\Gamma$.

A transition $t \in TR_i$ can be executed in $C$ if $SM_i$ is currently executing a RTC, and $t$ is *enabled* in $C$, denoted $enabled(t, C)$. $t$ is enabled in $C$ if the source state is active ($src(t) \in \omega_i$), the trigger is the currently dispatched event ($\rho_i = trig(t)$) or no trigger on $t$ if $\rho_i = \epsilon$, the guard is satisfied under the current variable assignment, and all transitions from states $s$ s.t. $s \lhd src(t)$ are not enabled. When $t$ executes, $SM_i$ moves to $c_i' = (\omega_i', \rho_i')$, denoted $dest(c_i, t)$, where $\rho_i' = \epsilon$ (an event is consumed once). $\omega'$ is obtained by removing from $\omega$ $src(t)$ and states contained in it and then adding $trgt(t)$ and states contained in it, based on *init*.

Let $C$ be a $\Gamma$-conf, $SM_i$ be a state machine in $\Gamma$, and let $s_1, s_2 \in S_i$ and $t, t_1, ..., t_y \in TR_i$. We use the following notations. $Qpush(t, (q_1, ..., q_m)) = (q_1', ..., q_m')$ denotes the effect of executing $t$ on the different EQs of the system: if for some event $e$, $GEN(e) \in act(t)$, then executing $t$ pushes $e$ to the relevant EQ (to $Q_{thrd(trgt(e))}$). The rest of the EQs remain unchanged. $act(t)(\sigma, C) = \sigma'$ represents the effect of executing the assignments in $act(t)$ on the valuation $\sigma$ of $C$, which results in a new assignment, $\sigma'$. States are *orthogonal* iff they are contained (possibly transitively) in orthogonal regions. $maxORTH((t_1, ..., t_y), C) = true$ iff $(t_1, ..., t_y)$ is a *maximal set* of *enabled* orthogonal transitions. $t_1, ..., t_y$ are *orthogonal* iff their sources are pairwise orthogonal. In Fig 1, the transition from *ChkSeasn* to *ChkDiscnt* is orthogonal to the transition from *ChngSeasn* to *Done*.

The function *apply* defines the effect of executing a sequence of transitions on a $\Gamma$-conf $C$. $apply((t_1, ..., t_y), C) = C'$ represents the effect of executing $t_1$ on $C$ followed by $t_2$ on the result etc. until executing $t_y$, which results in $C' = (c_1, ..., c_i', ..., c_n, q_1', ..., q_m', id_1, ..., id_m, \sigma')$ where: $c_i' = dest(...dest(c_i, t_1)..., t_y)$, $q_1', ..., q_m' = Qpush(t_y, ...Qpush(t_1, (q_1, ..., q_m)))$, $\sigma' = act(t_y)(...act(t_1)(\sigma, C), C)$.

## 3 System Computations

**Def. 1 (System Computations).** *A* computation *of a system $\Gamma$ is a maximal sequence $\pi = C^0, step^0, C^1, step^1, ...$ s.t.: (1) each $C^k$ is a $\Gamma$-conf, (2) each step $C^k \xrightarrow{step^k} C^{k+1}$ can be generated by one of the inference rules detailed below, and (3) each $step^k$ is a pair $(thid^k, t^k)$ where $thid^k \in \{1, ..., m\}$ represents the id of the thread executing the step ($t^k$ is described in the inference rules).*

We now define the set of inference rules describing $C \xrightarrow{step} C'$. We specify only the parts of $C'$ that change w.r.t. $C$ due to *step*.

**Initialization.** In the initial configuration $C^0$ all EQs are empty, and each $SM_i$ is inactive (for every $j$, $id_j^0 = 0$) and is in its initial state (for every $c_i^0$, $\rho_i^0 = \epsilon$ and $\omega_i^0 = \{s \in S_i | s \in init_i \wedge \forall s' \in S_i . s \lhd s' \rightarrow s' \in init_i\}$).

**Dispatch.** An event can be dispatched from thread $j$'s EQ only if the previous RTC on thread $j$ ended and the EQ is not empty.

$$DISP(j,e): \quad \frac{id_j = 0 \quad q_j \neq \phi \quad top(q_j) = e \quad trgt(e) = l}{id'_j = l \quad q'_j = pop(q_j) \quad c'_l = (\omega_l, type(e))}$$

**Transition.** UML2 defines a single case where transitions are executed simultaneously, when the transitions are in orthogonal regions and all simultaneously consume an event (on the first step of a RTC). Since it is not clear how to define simultaneous execution, we define an interleaved execution of these transitions. Only after all transitions have executed, the next step is enabled.

$$TRANS(j,(t_1,...,t_y)): \quad \frac{\begin{array}{c} id_j = l > 0 \qquad t_1,...,t_y \in TR_l \\ \rho_l \neq \epsilon \rightarrow (maxORTH((t_1,...,t_y),C) = true) \\ \rho_l = \epsilon \rightarrow (y = 1 \wedge enabled(t_1,C)) \end{array}}{C' = apply((t_1,...,t_y),C)}$$

**EndRTC.** If the currently running state machine on thread $j$ has no enabled transitions, then the RTC is complete.

$$EndRTC(j,\epsilon): \quad \frac{id_j = l > 0 \qquad \forall t \in TR_l.enabled(t,C) = false}{id'_j = 0 \qquad c'_l = (\omega_l, \epsilon)}$$

**ENV.** The behavior of the environment is not precisely described in the UML standard. We assume the most general definition, where the environment may insert events into the EQs at any step.

$$ENV(j,e): \quad \frac{e \in EV_{env} \qquad thrd(trgt(e)) = j}{q'_j = push(q_j,e)}$$

Intuitively, a computation is a series of steps that follow the RTC semantics per-thread, where RTCs of different threads are interleaved.

## 4 Abstracting a Behavioral UML System

### 4.1 Abstracting a State Machine

Let $SM$ be a concrete state machine. The abstraction of $SM$ is defined w.r.t. a *collection* $A = \{A^1,...,A^k\}$, where for every $i$, the *abstraction set* $A^i$ is a set of states from $S$ s.t. for every $s, s' \in A^i$, $\Omega(s) = \Omega(s')$. Intuitively, our abstraction replaces every $A^i$ (and all states contained in $A^i$) with a *different construct* that ignores the details of $A^i$ and maintains an over-approximated behavior of the events generated by $A^i$. For simplicity, from here on we assume the collection contains a single abstraction set $A$. A description of the framework for any collection size is available in [17].

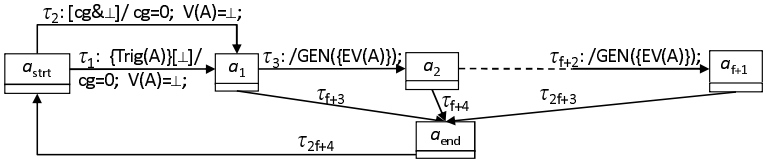


**Fig. 2.** $\Delta(A)$: The abstraction construct created for $A$

We add the value $don't - know$, denoted $\perp$, to the domain of all variables in $V$, where $\perp$ represents any value in the domain. The semantics of boolean operations is extended to 3-valued logic in the usual way: $\perp \wedge false = false$, $\perp \wedge true = \perp$ and $\neg\perp = \perp$. An expression is evaluated to $\perp$ if one of its arguments is $\perp$. For simplicity of presentation, we enable $trig(t)$ to be a *set of triggers*. I.e. $trig(t) = \{e_1, ..., e_q\} \cup \epsilon$, and $enabled(t, C) = true$ if *one of the events* from $trig(t)$ matches $\rho$.

Next, we define several notions that are concrete and are defined w.r.t. $A$:
- $S(A) = \{s \in S | \exists s' \in A.(s \lhd s')\}$ are the *abstracted states*.
- $R(A) = \{r \in R | \exists s \in A.(r \lhd s)\}$ are the *abstracted regions*.
- $TR(A) = \{t \in TR | src(t), trgt(t) \in S(A)\}$ are the *abstracted transitions*.
- $EV(A) = \{e \in EV | \exists t \in TR(A).(GEN(e) \in act(t))\}$
- $Trig(A) = \{tr | \exists t \in TR(A).(trig(t) = tr)\} \setminus \{\epsilon\}$
- $V(A) = \{v \in V | \exists t \in TR(A).(v \in modif(t))\}$
- $GRDV(A) = \{v \in V | \exists t \in TR(A).(trig(t) = \epsilon \wedge v \in grd(t))\}$

We require the following restrictions on $A$ of $SM_i$:
(1) For every $v \in GRDV(A)$, if $v$ can be modified by several SMs in $\Gamma$, then all these SMs are assigned to the same thread. This is needed for correctness of the construction (details in [17]).
(2) There are no loops without triggers within $S(A)$. Further, there are no self loops without a trigger on states containing $S(A)$. This is needed to enable static analysis described next.

In order to explain our abstraction we introduce the notion of an *A-round*. Let $\pi$ be a computation on the concrete system $\Gamma$, an A-round is a maximal, possibly non-consecutive, sequence of steps, $step_{i_1}, ..., step_{i_d}$ from $\pi$, s.t. all the steps are part of a single RTC, every step executes a transition from $TR(A)$, and the SM remains in an abstracted state throughout the A-round. I.e., for every $j \in \{i_1, i_1 + 1, ..., i_d\}$: $\omega^j \cap S(A) \neq \phi$. Due to the above requirement (2), we can easily apply static analysis in order to determine the maximal number of events that can be generated by any single A-round. We denote this number by $f$.

Given an abstraction set $A$, our abstraction replaces $S(A)$, $R(A)$ and $TR(A)$ with *a new construct*, referred to as $\Delta(A)$, demonstrated in Fig. 2. $\Delta(A)$ includes an initial state $a_{strt}$ and a final state $a_{end}$. Every A-round over states from $S(A)$ is represented by a computation from $a_{strt}$ to $a_{end}$. $\Delta(A)$ includes computations that can generate any sequence of size 0 to $f$ events from $EV(A)$. Also, all the variables that *can* be modified in the A-round are given the value $\perp$.

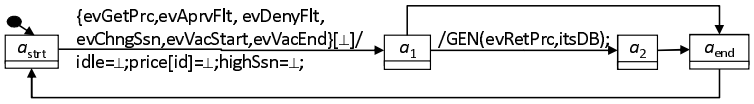


**Fig. 3.** Abstract DB State Machine

An A-round whose first transition consumes an event, is represented by a computation that starts with transition $\tau_1$ from $a_{strt}$ to $a_1$, which can consume any single event from $Trig(A)$. The guard $\perp$ on $\tau_1$ and $\tau_2$ represents a non-deterministic choice between "true" or "false". If the first transition on an A-round does not consume an event, it will be represented by transition $\tau_2$, which is not marked with a trigger. Since $\Delta(A)$ contains a loop of transitions without triggers we must ensure that all RTCs through $\Delta(A)$ are finite. We introduce a new Boolean variable $cg$. A trace on $\Delta(A)$ can be initiated without a trigger only if $cg$ is 1. $\Delta(A)$ then sets $cg$ to 0 on the transitions exiting $a_{strt}$.

When $cg$ is set to 1 it signals that it is possible to execute an A-round that does not consume an event. Such a situation abstracts a concrete execution in which the RTC that includes the A-round starts at a state that is not abstracted and continues within the abstraction. The situation can also occur if an abstracted transition becomes enabled due to some variable change. I.e., execution of some transition $t$, which is either orthogonal to $A$ or is in a different state machine, and $t$ modifies a variable $v \in GRDV(A)$.

If by static analysis we can conclude that the first transition of every A-round *consumes an event*, then $cg$ is redundant (and $\tau_2$ can be removed). All the A-rounds are then represented by computations that start by traversing $\tau_1$.

We now formally define our abstract state machines. Given $SM = (S, R, \Omega, init, TR, L)$ and an abstraction set $A \subseteq S$, $SM(A) = (S^A, R^A, \Omega^A, init^A, TR^A, L^A)$ is the abstraction of $SM$ w.r.t. $A$. We denote functions over the abstraction ($src$, $trgt$, $trig$, $grd$, and $act$) with a superscript $A$.

- $S^A = (S \setminus S(A)) \cup \{a_{strt}, a_1, ..., a_{f+1}, a_{end}\}$ and $R^A = (R \setminus R(A))$
- For every $s \in (S^A \cap S) \cup R^A$: $\Omega^A(s) = \Omega(s)$.
  For every $s \in \{a_{strt}, a_1, ..., a_{f+1}, a_{end}\}$: $\Omega^A(s) = \Omega(s')$ for some $s' \in A$.
- $init^A = init \cap S^A$ or $init^A = (init \cap S^A) \cup \{a_{strt}\}$ if $\exists s \in A$ s.t. $s \in init$
- $TR^A = (TR \setminus TR(A)) \cup \{\tau_1, ..., \tau_{2f+4}\}$.

The $src^A$, $trgt^A$, $trig^A$, $grd^A$ and $act^A$ functions are redefined as follows:

Transitions $\tau_1, ..., \tau_{2f+4}$ are defined according to Fig. 2. Every transition $t \in TR \setminus TR(A)$ has a representation (*matching transition*) in $SM(A)$. Note that for every such transition, at least one of $src(t)$ and $trgt(t)$ are not abstracted. If $src(t)$ or $trgt(t)$ are abstracted, then $src^A(t)$ or $trgt^A(t)$ respectively are in $\Delta(A)$. The handling of $cg$ is added to the relevant actions, as discussed above. In the following we present only the values of $src^A$, $trgt^A$, $trig^A$ $grd^A$ and $act^A$ that change in $SM(A)$ w.r.t. $SM$. For every $t \in TR \setminus TR(A)$:

1. $trgt(t) \in S(A)$ (the target of $t$ is abstracted): we define $trgt^A(t) = a_{strt}$.
   If there exists an abstracted transition from $trgt(t)$ whose trigger is $\epsilon$ then $act^A(t) = act(t); cg = 1$ (otherwise, $act^A(t) = act(t)$).

2. $src(t) \in S(A)$ (the source of $t$ is abstracted): we define $src^A(t) = a_{strt}$, $act^A(t) = cg = 0; act(t)$ and $grd^A(t) = grd(t)\&\bot$. We add $\bot$ to the guard in order to ensure that executions of possibly enabled transitions from states containing the abstraction remain (possibly) enabled.
3. Otherwise (neither $src(t)$ nor $trgt(t)$ are abstracted):
   **Case a:** $A \lhd trgt(t)$. If an execution of $t$ results in a new $\omega$ that includes an abstracted state $s \in S(A)$, and there exists an abstracted transition from $s$ whose trigger is $\epsilon$. Then: $act^A(t) = act(t); cg = 1$ (otherwise, $act^A(t) = act(t)$).
   **Case b:** $src(t)$ and $a_{strt}$ are contained in orthogonal regions ($t$ can be executed orthogonally to the abstraction). Then: $act^A(t) = act(t)$ with the following modifications: If $\exists v \in GRDV(A)$ s.t. $v \in modif(t)$ then $cg = 1$ is added to $act^A(t)$. In addition, if $SM$ is in an abstracted state, then variables that *can* be modified by abstracted transitions should remain $\bot$. For that, every assignment $x = e$ in $act(t)$, if $x \in V(A)$ then $x = e$ is replaced with: "if $(isIn(A))$ $x = \bot$; else $x = e$;" in $act^A(t)$. The current state is checked using the macro $isIn(U)$, that checks whether a certain state from $U$ is active.

Fig. 3 shows the state machine created by abstracting the $DB$ state machine (Fig. 1) with $A = \{Working, Vacation\}$. Note that in this state machine, by static analysis we can conclude that every A-round first consumes an event, and therefore we do not need the $cg$ flag and transition $\tau_2$. Also, on every A-round no more than one event can be generated, therefore $f = 1$.

### 4.2 Abstracting a System

Next we define an abstract system. This is a system in which some of the state machines are abstract. For $SM_i$ and an abstraction set $A_i$, $SM_i^A$ denotes the abstraction of $SM_i$ w.r.t. $A_i$. We denote the $cg$ variable in $SM_i^A$ as $cg_i$.

**Def. 2.** *Let $\Gamma$ and $\Gamma'$ be two systems, each with $n$ SMs and $m$ EQs. We say that $\Gamma'$ is an* abstraction *of $\Gamma$, denoted $\Gamma^A$, if the following holds. (1) For $i \in \{1, ..., n\}$, $SM_i' = SM_i$ or $SM_i' = SM_i^A$, (2) $thrd = thrd'$, (3) $V' = V \cup \{cg_i | SM_i' = SM_i^A\}$, and (4) for every $i, j \in \{1, ..., n\}$ s.t. $i \neq j$, and for every $t \in TR_j'$: if there exists a variable $v \in GRDV(A_i)$ and $v \in modif(t)$ then $cg_i = 1$ is added to $act'(t)$.*

Recall that setting $cg_i$ to 1 on $SM_i^A$ signals that it is possible to execute an A-round on $SM_i$ without consuming an event. Req. (4) in Def. 2 handles the case where a guard of an abstracted transition of $SM_i$ may change by a transition $t$ of $SM_j$. It ensures that $cg_i$ is set to 1 on such transitions of $TR_j'$.

Adding the value $\bot$ to the domain of all variables in $V$ affects the cases when a transition is enabled, since now $grd(t)(\sigma) \in \{true, false, \bot\}$. Intuitively, if $grd(t)(\sigma) = \bot$ then we assume it *can* be either $true$ or $false$. We thus consider both cases in the analysis. Therefore, $enabled(t, C) = true$ iff $t$ *can be enabled* w.r.t. $C$ ($grd(t)(\sigma) \in \{true, \bot\}$) and all transitions $t'$ from states contained in $src(t)$ *can be not enabled* ($grd(t')(\sigma) \in \{false, \bot\}$). Note that when enabling

3-valued semantics, a transition may be enabled, even though lower level transitions may be enabled as well.

## 5 Correctness of the Abstraction

In this section we prove that $\Gamma^A$ is an over-approximation of $\Gamma$ by showing that every computation of $\Gamma$ has a "matching" computation in $\Gamma^A$.

**Def. 3 (Abstraction of SM-conf).** *Let $c = (\omega, \rho)$ and $c^A = (\omega^A, \rho^A)$ be SM-confs of $SM$ and $SM^A$ respectively. $c^A$* abstracts *$c$, denoted $c \preceq c^A$, if $\rho = \rho^A$, and $c$, $c^A$ agree on the joint states: $\omega \neq \omega^A$ iff $\omega \setminus \omega^A \subseteq S(A)$ and $\omega^A \setminus \omega \subseteq \Delta(A)$.*

**Def. 4 (Abstraction of $\Gamma$-conf).** *Let $C$ and $C'$ be two $\Gamma$-confs of $\Gamma$ and $\Gamma^A$ respectively. We say that $C'$* abstracts *$C$, denoted $C \preceq C'$, if the $\Gamma$-confs agree on the EQs and id elements, and the SM-confs and $\sigma'$ of $\Gamma^A$ are abstraction of the matching elements in $\Gamma$: for $j \in \{1, ..., m\}$, $q_j = q'_j$ and $id_j = id'_j$, for $i \in \{1, ..., n\}$, $c_i \preceq c'_i$, and for every $v \in V$ either $\sigma(v) = \sigma'(v)$ or $\sigma'(v) = \perp$.*

We now define *stuttering computation inclusion*, which is an extension of stuttering-trace inclusion ([6]) to system computations. For simplicity of presentation, we assume that computations are infinite. However, all the results presented hold for finite computations as well. Intuitively, there exists stuttering inclusion between $\pi$ and $\pi'$ if they can be partitioned into infinitely many finite intervals, s.t. *every* configuration in the $k$th interval of $\pi'$ abstracts *every* configuration in the $k$th interval of $\pi$.

**Def. 5 (Stuttering Computation Inclusion).** *Let $\pi = C^0, step^0, C^1, step^1, ...$ and $\pi' = C'^0, step'^0, C'^1, step'^1, ...$ be two computations over $\Gamma$ and $\Gamma^A$ respectively. There exists a* stuttering computation inclusion *between $\pi$ and $\pi'$, denoted $\pi \preceq_s \pi'$, if there are two infinite sequences of integers $0 = i_0 < i_1 < ...$ and $0 = i'_0 < i'_1 < ...$ s.t. for every $k \geq 0$:*
*For every $j \in \{i_k, ..., (i_{k+1}) - 1\}$ and for every $j' \in \{i'_k, ..., (i'_{k+1}) - 1\}$: $C^j \preceq C'^{j'}$*

Fig. 4 illustrates two computations where $\pi \preceq_s \pi'$. Def. 4 implies that steps of type $DISP$, $ENV$ and $EndRTC$ cannot be steps within an interval, due to the effect of these steps on $\Gamma$-conf. For example, in Fig. 4, $C^6 \preceq C'^5$. Assume $step^6 = EndRTC(j, \epsilon)$, then by the definition of $EndRTC$ step, the value of $id_j$ changes from $C^6$ to $C^7$. Since $\Gamma$-conf abstraction requires equality of the $id$ elements, then clearly $C^7 \not\preceq C'^5$. Thus $C^6$ and $C^7$ cannot be in the same interval. For a similar reason, a step of type $DISP$, $ENV$ or $EndRTC$ on $\pi$ implies a step of the same type on $\pi'$, and vice versa. Steps of type $TRANS$ that are either the first step in a RTC or a step that generates events are also steps that cannot be part of an interval, due to the effect of these steps on the $\rho$ elements and the EQs.

An immediate consequence of the above is that an interval can be of size greater than one only if the steps in the interval are $TRANS$ steps that are neither a first step in a RTC nor a step generating an event. Recall that Def. 4 requires a correlation between the current states of the state machines. It can

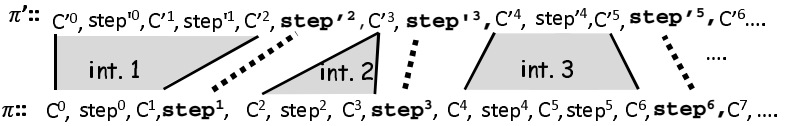


**Fig. 4.** Stuttering Computation Inclusion

therefore be shown that if $step^i = TRANS(j,(t))$ is a step between two configurations in the same interval, then one the following holds: (1) If $step^i \in \pi$ then $t$ is an abstracted transition, (2) If $step^i \in \pi'$ then $t \in \Delta(A)$.

We extend the notion of stuttering inclusion to systems, and say that there exists a *stuttering inclusion* between $\Gamma$ and $\Gamma^A$, denoted $\Gamma \preceq_s \Gamma^A$, if for each computation $\pi$ of $\Gamma$ from an initial configuration $C_{init}$, there exists a computation $\pi'$ of $\Gamma^A$ from an initial configuration $C'_{init}$ s.t. $\pi \preceq_s \pi'$.

The following theorem captures the relation between $\Gamma$ and $\Gamma^A$, stating that there *exists* stuttering inclusion between $\Gamma$ and $\Gamma^A$.

**Theorem 6.** *If $\Gamma^A$ is an abstraction of $\Gamma$ then $\Gamma \preceq_s \Gamma^A$.*

Every system $\Gamma$ can be viewed as a Kripke structure $K$, where the K-states are the set of $\Gamma$-confs, and there exists a K-transition $(C, C')$ iff $C'$ is reachable from $C$ within a single *step*. Thus, every computation of $\Gamma$ corresponds to a trace in $K$. Let $\Gamma$ be a system, and let $A\psi$ be an $LTL$ formula, where the atomic propositions are predicates over $\Gamma$. Then $\Gamma \models A\psi$ iff for every computation $\pi$ of $\Gamma$ from an initial configuration, $\pi \models \psi$. By preservation of $LTL_x$ over stuttering-traces inclusion we conclude:

**Corollary 7.** *Let $\Gamma$ and $\Gamma^A$ be two systems, s.t. $\Gamma \preceq_s \Gamma^A$, and let $A\psi$ be an $LTL_x$ formula over joint elements of $\Gamma$ and $\Gamma^A$. If $\Gamma^A \models A\psi$ then $\Gamma \models A\psi$.*

Due to the stuttering-inclusion, $\Gamma^A$ preserves $LTL_x$ and not $LTL$. It is important to note that since $\Gamma$ itself is a multi-threaded system, properties of interest are commonly defined without the next-time operator.

The proof of Theorem 6 is available in [17]. We give here an intuitive explanation to why for every $\pi$ of $\Gamma$ from $C_{init}$, there exists $\pi'$ of $\Gamma^A$ from $C'_{init}$ s.t. $\pi \preceq_s \pi'$. For every step executed on $\Gamma$ that does not include execution of an abstracted transition it is possible to execute the same step on $\Gamma^A$. More specifically, for every transition $t$ executed on $\Gamma$, if $t$ has a matching transition $t_a$ in $\Gamma^A$, then $t_a$ can be executed on $\pi'$. For every step of type $ENV$, $DISP$ and $EndRTC$ on $\pi$ it is possible to execute the same step on $\pi'$. This holds since matching configurations $C^r$ and $C'^p$ of $\pi$ and $\pi'$ respectively agree on their joint elements, and $\sigma'^p$ might assign $\perp$ to variables. Thus, if a transition $t$ is enabled, then its matching transition $t_a$ *can* be enabled.

For execution of abstracted transitions on $\Gamma$, every A-round $\chi$ on some concrete state machine $SM_i$ can be matched to a trace from $a_{strt}$ to $a_{end}$ on $SM_i^A$. The matching is as follows: every transition $t$ that is traversed on $\chi$ and where $t$ generates an event $(GEN(e) \in act(t))$ matches a transition from $a_i$ to $a_{i+1}$ (for some $i$). Every transition $t$ that is traversed on $\chi$ and where $t$ does not generate

or consume an event, matches an interval of length one on $\pi'$ ($\Gamma^A$ does not execute a matching step). Since $\chi$ can generate at most $f$ events, then indeed we can match the transitions as described. All variables that *can* be modified on $\chi$ are given the value $\perp$ upon execution of the first transition in $\Delta(A)$ (transitions from $a_{strt}$ to $a_1$). This value is maintained in the variables throughout the traversal on $\Delta(A)$.

## 6 Using Abstraction

We now present the applicability of our abstraction framework by an example. Consider a system $\Gamma$ describing a travel agent (of class *Agent*) that books flights and communicates with both airline databases (of class $DB$) and clients. Assume $\Gamma$ includes $n$ different $DB$ objects, where the behavior of each $DB$ is defined in Fig. 1. The single *Agent* object in $\Gamma$ communicates with clients (modeled as the environment) and with all of the $DBs$. The *Agent* behavior is as follows: upon receiving a flight request from a client, it requests a price offer from all $DBs$ by sending them event *evGetPrc*. After getting an answer from the $DBs$ (via *evRetPrc*), it chooses an offer, reserves the flight from the relevant $DB$ (via *evAprvFlt*) and rejects the offers from the rest of the $DB$ (via *evDenyFlt*).

Assume now we create an abstract system $\Gamma^A$, where the $DBs$ are abstracted as in Fig. 3 (the *Agent* remains concrete). If *Agent* state machine includes $x$ states, then $\Gamma$ has $(12*n+x)$ states, whereas $\Gamma^A$ has $(4*n+x)$ states. Moreover, $\Gamma^A$ does not include the pieces of code in the actions of the transitions of $DBs$, which may be complicated. E.g., the method *calcPrc*() is not part of the abstract state machine of $DB$, and this method might include complex computations.

Assume we want to verify the property describing that on all computations of $\Gamma$, if *Agent* orders a flight from some $DB$, then all the $DBs$ returned an answer to the *Agent* before the *Agent* chooses an offer. For this property it is enough to consider only the *interface* of the $DBs$. The property is not affected, for example, by the calculation of a price by the $DBs$. It is an outcome only of the information that every $DB$ can consume an event *evGetPrc*, and can send an event *evRetPrc*. We can therefore verify the property on $\Gamma^A$. If the property holds, then we can conclude that $\Gamma$ also satisfies the property.

Consider another property: we want to verify that due to a single request from the client, *space* decreases by at most 1. Clearly, when verifying the property on $\Gamma^A$, the result is $\perp$, since $\Gamma^A$ abstracts the variable *space*. This means that we cannot conclude whether or not the property holds on $\Gamma$ by model checking $\Gamma^A$. However, it might be possible to *refine* $\Gamma^A$, and create a different abstraction $\Gamma'^A$ for which this property can be verified. Following, in section 7 we present how to refine an abstract system when the verification does not succeed.

## 7 Refinement

Once we have an abstract system $\Gamma^A$, we model check our $LTL_x$ property $A\psi$ over the abstract system. Since variables in $\Gamma^A$ can have the value $\perp$, then $(\Gamma^A \models A\psi) \in \{true, false, \perp\}$. If $(\Gamma^A \models A\psi) = true$, then from Theorem 6 the

property holds on $\Gamma$ as well. If $(\Gamma^A \models A\psi) \in \{false, \bot\}$ then due to $\Gamma^A$ being an over-approximation we cannot determine whether or not the property holds on $\Gamma$. Typical model checkers provide the user with a CEX in case verification does not succeed. A CEX $\pi^A$ on $\Gamma^A$ is either a finite computation or a lasso computation s.t. either $(\pi^A \models \psi) = false$ or $(\pi^A \models \psi) = \bot$.

Next we present a CEGAR-like algorithm for refining $\Gamma^A$ based on $\pi^A$. The refinement step suggests how to create a new abstract system $\Gamma'^A$, where one or more of the abstracted states of $\Gamma$ are removed from the abstraction sets. Since the concrete system $\Gamma$ is finite, the $CEGAR$ algorithm ultimately terminates and returns a correct result.

If $(\pi^A \models \psi) = \bot$ then we cannot determine the value of the property. If $(\pi^A \models \psi) = false$, then this CEX might be spurious. In both cases we search for a computation $\pi$ on $\Gamma$ s.t. $\pi \preceq_s \pi^A$. Given $\pi^A$, we inductively construct $\pi$ w.r.t. $\pi^A$. Note that if the concrete model enables non-determinism, then there might be more than one matching concrete CEXs. In this case, all the matching concrete CEXs are simultaneously constructed. Intuitively, the construction of $\pi$ follows the steps of $\pi^A$, maintaining the stuttering inclusion. During the construction, if for some prefix of $\pi^A$: $C'^0, step'^0, ..., step'^{p-1}, C'^p$ it is not possible to extend any of the matching concrete computations based on $step'^p$, then $\pi^A$ is a spurious CEX and we should refine the system. Detailed description of the construction of $\pi$ is presented in [17]. There are three cases where we cannot extend a concrete computation $\pi = C^0, step^0, ..., C^r$ $(C^r \preceq C'^p)$ based on $step'^p$: (1) $step'^p$ is an $EndRTC$ step on $SM'_l$ but there exists an enabled transition in $TR_l$ w.r.t. $C^r$. (2) $step'^p$ is a $TRANS$ step on $SM'_l$ that executes a transition $t_a \notin \Delta(A)$, and the concrete transition $t$ that matches $t_a$ is not enabled. (3) $step'^p$ is a $TRANS$ step on $SM'_l$ that executes a transition $t_a \in \Delta(A)$ that generates an event $e$, and there is no enabled concrete transition $t \in TR(A)$ where $GEN(e) \in act(t)$.

We call the configuration $C'^p \in \pi^A$ from which we cannot extend a matching concrete computation *failure-conf*. Following, we distinguish between two reasons that can cause a failure-conf, and show how to refine the system in each case.

**Case 1:** $step'^p$ executes a transition that does not have a matching behavior in $\Gamma$. E.g., when $step'^p = TRANS(j, (t^a))$, $id'^p_j = l$, and the concrete $t$ that matches $t_a$ is not enabled since $src(t) \notin \omega^r_l$. This is possible only if $src(t) \in S(A)$ and $trgt(t) \notin S(A)$. Another example for such a failure is when $\Gamma^A$ generates an event $e$ as part of the action of $t_a$, but $e$ cannot be generated from $C^r$ on any possible *step*. This can happen only if $t_a \in \Delta(A)$. In both cases we refine by removing a state $s \in S(A)$ s.t. $s \in \omega^r_l$ from the abstraction.

**Case 2:** There exists $v \in V$ for which $\sigma'^p(v) = \bot$ and the value of $\sigma^r(v)$ causes the failure-conf. For example, when $step'^p = TRANS(j, (t_a))$ and the concrete $t$ that matches $t_a$ is not enabled since $grd(t)(\sigma^r) = false$. Since $C^r \preceq C'^p$ and $grd(t_a) = grd(t)$, then clearly $grd(t_a)(\sigma'^p) = \bot$ and for some $v$, $\sigma'^p(v) = \bot$ and $v$ affects the value of $grd(t_a)$. We refine $\Gamma^A$ to obtain a concrete value on $v$:
We trace $\pi^A$ back to find the variable that gave $v$ the value $\bot$. The only place where a variable is initially assigned the value $\bot$ is a transition from $a_{strt}$ to $a_1$

in some $\Delta(A_i)$. Thus, the tracing back of $\pi^A$ terminates at $C'^{\alpha}$ s.t. $a_{strt} \in \omega_i^{\alpha}$. We find the matching $\Gamma$-conf $C^{\beta}$ in $\pi$ s.t. $C^{\beta} \preceq C'^{\alpha}$, and refine the model by removing from the abstraction a state $s \in S(A_i)$ s.t. $s \in \omega_i^{\beta}$.

If we are able to construct $\pi$ s.t. $\pi \preceq \pi^A$, then one of the following holds: (a) If $(\pi^A \models \psi) = false$ then no need to check $\pi$. By construction, $\pi \not\models \psi$, and we can conclude that $\Gamma \not\models A\psi$, (b) If $(\pi^A \models \psi) = \perp$ then we check $\pi$ w.r.t. $\psi$. If $\pi \not\models \psi$ then again $\pi$ is a concrete CEX and we conclude that $\Gamma \not\models A\psi$. Otherwise $(\pi \models \psi)$, the abstraction is too coarse and we need to refine. Notice that in the latter case, since $(\pi^A \models \psi) = \perp$ then there exists $v \in V$ which affects the value of $\psi$, and $v$ has the value $\perp$. We then refine $\Gamma^A$ in order to have a concrete value on $v$, as described above (Case 2).

Consider the example system presented in section 6, and consider a property that addresses the variable *space*. Recall that under the abstraction presented for this example, such a property is evaluated to $\perp$, since the variable *space* is abstracted. During the refinement, state $WaitForDB$ is suggested for refinement, and is removed from the abstraction. We can then create a *refined* system $\Gamma'^A$, where $DB$ objects are abstracted w.r.t. a new abstraction set $A' = \{Idle, PriceProcessor, UpdateDB\}$. The property can then be verified on $\Gamma'^A$, and we can conclude that it holds on the concrete system.

## 8 Conclusion

In this work we presented a CEGAR-like method for abstraction and refinement of behavioral UML systems. It is important to note that our framework is completely automatic. An initial abstraction can be one that abstracts entire state machines, based on the given property. We presented a basic and automatic refinement method. Heuristics can be applied during the refinement stage in order to converge in less iterations. For example, when refining due to a variable $v$ whose value is $\perp$, we can refine by adding all abstracted transitions that modify $v$ (or $v$'s cone-of-influence). Note, however, that there always exists a tradeoff between quick convergence and the growth in size of the abstract system.

## References

1. Majzik, I., Darvas, A., Beny, B.: Verification of UML statechart models of embedded systems. In: DDECS 2002 (2002)
2. Booch, G., Rumbaugh, J.E., Jacobson, I.: The unified modeling language user guide. J. Database Manag. 10(4), 51–52 (1999)
3. 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), 498–520 (1998)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. Journal of the ACM 50(5), 752–794 (2003)
5. Clarke, E.M., Heinle, W.: Modular translation of statecharts to SMV. Tr, CMU (2000)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)

7. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: Understanding UML: A formal semantics of concurrency and communication in real-time UML. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 71–98. Springer, Heidelberg (2003)
8. Dubrovin, J., Junttila, T.A.: Symbolic model checking of hierarchical UML state machines. In: ACSD 2008 (2008)
9. Fecher, H., Huth, M., Schmidt, H., Schönborn, J.: Refinement sensitive formal semantics of state machines with persistent choice. Electron. Notes Theor. Comput. Sci. 250(1), 71–86 (2009)
10. Fecher, H., Schönborn, J.: UML 2.0 state machines: Complete formal semantics via core state machine. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 244–260. Springer, Heidelberg (2007)
11. Object Management Group. OMG Unified Modeling Language (UML) Infrastructure, version 2.4. ptc/2010-11-16 (2010)
12. Grumberg, O., Meller, Y., Yorav, K.: Applying software model checking techniques for behavioral UML models. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 277–292. Springer, Heidelberg (2012)
13. Gurfinkel, A., Chechik, M.: Why waste a perfectly good abstraction? In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)
14. 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), 637–664 (1999)
15. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.S.: A formal semantics for complete UML state machines with communications. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 331–346. Springer, Heidelberg (2013)
16. Madhukar, K., Metta, R., Singh, P., Venkatesh, R.: Reachability verification of rhapsody statecharts. In: ICSTW 2013 (2013)
17. Meller, Y., Grubmerg, O., Yorav, K.: Verifying behavioral UML systems via CEGAR. TR CS-2014-01, Dept. of Computer Science. Technion - Israel Institute of Technology (2014)
18. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in promela/spin. In: WIFT 1998 (1998)
19. Ober, I., Graf, S., Ober, I.: Validating timed UML models by simulation and verification. STTT 8(2), 128–145 (2006)
20. IST-2001-33522 OMEGA (2001), `http://www-omega.imag.fr`
21. 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)
22. Pnueli, A.: The temporal logic of programs. In: FOCS 1977 (1977)
23. Prehofer, C.: Behavioral refinement and compatibility of statechart extensions. Electron. Notes Theor. Comput. Sci. 295(5), 65–78 (2013)
24. Reeve, G., Reeves, S.: Logic and refinement for charts. In: ACSC 2006 (2006)
25. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The rhapsody UML verification environment. In: SEFM 2004 (2004)
26. Scholz, P.: Incremental design of statechart specifications. Sci. Comput. Program. 40(1), 119–145 (2001)
27. Seger, C.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. Form. Methods Syst. Des. 6(2), 147–189 (1995)
28. Simons, A.J.H., Stannett, M.P., Bogdanov, K.E., Holcombe, W.M.L.: Plug and play safely: Rules for behavioural compatibility. In: SEA 2002 (2002)