Model Checking Techniques for Behavioral UML Models

Yael Meller

Model Checking Techniques for Behavioral UML Models

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Yael Meller

Submitted to the Senate of
the Technion — Israel Institute of Technology
Shvat 5776 Haifa January 2016

The research thesis was done under the supervision of Prof. Orna Grumberg and Dr. Karen Yorav in the Computer Science Department.

First and foremost I would like to thank my supervisor Orna Grumberg for her incredible guidance and support throughout my studies. Thank you for guiding me how to be a researcher, and for showing me the beauty and fun of research. You have always provided me with inspiration and encouragement, and knew how to always do it with a smile. More importantly, I thank you for your friendship: for the coffee breaks, for the talks, and for the parties and dancing. These all have made my studies so enjoyable. I feel lucky to have had you as my advisor.

I would also like to thank my other supervisor, Karen Yorav. Thank you for introducing me to the world of UML. For always finding the time to guide and help me, and to share your ideas with me. I feel privileged to have had the opportunity to work and learn from you.

I would like to thank my parents Kobi and Tami Kalka, for your love and support in every aspect of my life and studies, from elementary school to graduate school. I thank my parents in law, Daniella and Isaac Meller, who went out of their way to help me during my studies.

Last but not least I thank my husband, Nimrod and my daughters, Adi, Maya and Noga, for your love, for making me so happy, and for showing me each and every day what are the most important things in my life. Nimrod, thank you for your friendship and endless support throughout my studies. I could not have made it without you.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

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. In this thesis, we present several approaches to enhancing model checking to behavioral UML systems.

The applicability of model checking is often impeded by its high time and memory requirements. The first approach we propose aims at avoiding this limitation by adopting *software model checking* techniques for verification of UML models. We translate UML to *verifiable* C code which preserves the high level structure of the models, and abstracts details that are not needed for verification. We combine static analysis and bounded model checking for verifying LTL safety properties and absence of livelocks. We implemented our approach on top of the bounded software model checker CBMC. We compared it to an IBM research tool that verifies UML models via a translation to IBM's hardware model checker RuleBasePE. Our experiments show that our approach is more scalable and more robust for finding long counterexamples. We also demonstrate the usefulness of several optimizations that we introduced into our tool.

A successful approach to avoiding the high time and memory requirements of model checking is *CounterExample-Guided Abstraction-Refinement (CEGAR)*. In the second approach we propose a CEGAR-like method 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.

Another successful approach to tackle the limitations of model checking is *compositional verification*. Recently, great advances have been made in this direction via automatic learning-based Assume-Guarantee reasoning. In the last approach we present a framework for automatic Assume-Guarantee reasoning for behavioral UML systems. We apply an off-the-shelf learning algorithm for incrementally generating assumptions on the environment, that guarantee satisfaction of the property. A unique feature of our approach is that the generated assumptions are UML state machines. Moreover, our Teacher works at the UML level: All queries from the learning algorithm are answered by generating and verifying behavioral UML systems.

Abbreviations and Notations

S	—	Set of states
R	—	Set of regions
$\Omega : S \cup R \rightarrow S \cup R \cup \{\epsilon\}$	—	Mapping function from states and regions to their container
$s \lhd s'$	—	s' contains s
V	—	Set of variables
λ	—	Variable assignment
EV_{sys}	—	Set of system events
EV_{env}	—	Set of environment events
$EV = EV_{sys} \cup EV_{env}$	—	Set of events
act	—	Action (sequence of statements)
$modif(act)$	—	Set of variables modified on *act*
TR	—	Set of transitions
$trig(t)$	—	Trigger of transition t
$grd(t)$	—	Guard of transition t
$act(t)$	—	Action of transition t
$L : TR \rightarrow EV \times \mathcal{B} \times Actions$	—	Labeling function for transitions
$\mathcal{H}$	—	History marker
$SM = (S, R, \Omega, init, TR, L, \mathcal{H})$	—	State machine
$\omega \subseteq S$	—	Set of currently active states
ρ	—	Event currently dispatched
$H : R \rightarrow S$	—	History information function
$c = (\omega, \rho, H)$	—	State machine configuration
EQ	—	Event queue
Q_i	—	Event queue instance
q_i	—	Contents of Q_i

$\Gamma = (SM_1, ..., SM_n, Q_1, ..., Q_m, \, thread, V)$	—	System
$C = (c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \lambda)$	—	System configuration
$\pi = C^0, step^0, C^1, step^1, ...$	—	Computation of a system
RTC	—	Run-to-completion
LTL	—	Linear time logic
LTL_x	—	Linear time logic without the next-time operator

Chapter 1

Introduction

Computerized systems dominate almost every aspect of our lives and their correct behavior is crucial. *Model checking* [11] is a successful automated verification technique for checking whether a given system satisfies a desired property. The system is usually described as a finite-state model and the specification is given as a formula in temporal logic. The process of model checking considers *all* of the system behaviors, and either confirms that the system is correct w.r.t. the checked property, or provides a *counterexample* that demonstrates an erroneous behavior.

Model checking is widely recognized as an important approach to increasing reliability of hardware and software systems and is widely used in industry. Unfortunately, the applicability of model checking is often impeded by its high time and memory requirements, referred to as the *state explosion problem*. Much of the research in this area is dedicated to increasing model checking applicability and scalability.

The *Unified Modeling Language* (UML) [6] is a widely accepted modeling language that is used to visualize, specify, and construct systems. It provides means to represent a system as a collection of objects and to describe the system's internal structure and behavior. UML has been accepted as a standard object-oriented modeling language by the Object Management Group (OMG) [25]. It is becoming the dominant modeling language for embedded systems. As such, the correct behavior of systems represented as UML systems is crucial and verification techniques for such models are required.

In this work we present new techniques for improving model checking of behavioral UML systems. Our main goal is to keep the model checking process *at the UML level*. That is, instead of translating the behavioral UML system to some low level representation (e.g., Kripke structure) and applying optimizations on the low level representation, our goal is to apply optimizations on the UML system directly. This approach enables us to exploit high level information, which results from the unique structure and behavior of such models, in our optimizations, information which is otherwise lost. It is important to note that remaining at the UML level is also highly beneficial to the user, since the property, the optimizations and the counterexamples are all given at the UML level and are therefore more meaningful.

There are two orthogonal challenges to tackle when addressing model checking of behavioral UML systems. The first is how to apply existing model checking tools for verification of UML systems. The second challenge is, given a model checker for behavioral UML systems, how to fight the state explosion problem in the context of behavioral UML systems. Two of the most promising approaches for fighting the state explosion problem are *abstraction* and *compositional verification*. We propose applying these approaches for behavioral UML systems.

Following, we describe these challenges and our techniques for fighting them.

Model Checking Behavioral UML Systems

Model checking tools expect the checked system to be presented in an appropriate description language. Previous works on UML model checking translate UML systems to SMV [8, 12] or VIS[1] [52], both particularly suitable for hardware; to PROMELA (the input language of SPIN) [38, 34, 42, 17, 1, 31, 19]), which is mainly suitable for communication protocols; or to IF[3] [40], which is oriented to real-time systems.

We believe that behavioral UML systems most resemble high-level software systems. We therefore choose to translate UML systems to C and adopt *software model checking* techniques for their verification. Our translation preserves the high-level structure of the UML system: event-driven objects communicate with each other via an event queue. An execution con-

[1]These works were developed as part of the European research project OMEGA [41].

sists of a sequence of *Run To Completion* (RTC) steps. Each RTC step is initiated by the event queue by sending an event to its target object, which in turn executes a maximal series of enabled transitions. In Chapter 4 we present our approach for verifying behavioral UML systems by applying software model checking techniques. This work was published in [27].

Abstraction and Refinement for Behavioral UML Systems

Abstractions hide some of the system details in order to result in an *over-approximated* system that has *more behaviors and fewer 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. The *CounterExample-Guided Abstraction Refinement (CEGAR)* approach [10] provides an automatic and iterative framework for abstraction and refinement, where the refinement is based on a spurious counterexample. When model checking returns an abstract counterexample, a search is make for a matching concrete counterexample. If one exists, then a real bug on the concrete system is found. Otherwise, the counterexample is *spurious* and a refinement is needed. In the refinement stage, more details are added to the abstract system, in order to eliminate the spurious counterexample.

In Chapter 5 we propose a CEGAR-like framework for verifying behavioral UML systems. We present a model-to-model transformation that generates an *abstract model* from a given concrete one. Our transformation is done on the UML level, thus resulting in a *new behavioral UML system* which is an *over-approximation* of the original model. 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. This work was published in [36].

Compositional Verification for Behavioral UML Systems

Another promising solution to the state explosion problem is *compositional model checking*, where parts of the system are verified separately in order to avoid the construction of the entire system and to reduce the model checking cost. Due to dependencies among components' behaviors, it is usually impossible to verify one component in complete isolation from the rest of the system. To take such dependencies into account the Assume-Guarantee (**AG**) paradigm [30, 44, 26] suggests how to verify a component based on an *assumption* on the behavior of its environment, which consists of the other system components. The environment is then verified in order to guarantee that the assumption is actually correct.

Learning [2] has been a major technique to construct assumptions for the **AG** paradigm automatically. An automated *learning-based* **AG** *framework* was first introduced in [15]. It uses iterative **AG** reasoning, where in each iteration an assumption is constructed and checked for suitability, based on learning and on model checking. Many works suggest optimizations of the basic framework and apply it in the context of different **AG** rules ([7, 23, 57, 20, 39, 28, 5, 14, 43, 9]).

In Chapter 6 we propose a framework for automated learning-based **AG** reasoning *for behavioral UML systems*. Our framework is similar to the one presented in [15], with the main difference being that our framework remains at the state machine level. That is, the system's components are state machines, and the learned assumptions are *state machines* as well. This is in contrast to [15], where the system's components and the learned assumptions are all presented as Labeled Transition Systems (LTSs). This work was published in [37].

Chapter 2

Preliminaries

2.1 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 single-threaded system there is one EQ, while 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. Only when the target object finishes its RTC step, the thread dispatches the next event available in its EQ. Next we formally define state machines, UML systems, and the set of behaviors associated with them. The following definitions closely follow the UML2 standard.

2.1.1 UML State Machines

Definition 2.1 (States and Regions) *Let S denote a set of states partitioned into disjoint subsets according to two types: simple states S_{sim} and compound states S_{com}. Let R be a non-empty set of regions. We assume R contains the region TOP. Let $\Omega : S \cup R \rightarrow S \cup R \cup \{\epsilon\}$ be a function that* *associates regions to their containing states, and states to their containing regions. We assume the following constraints on Ω:*

- *For every $s \in S$, $\Omega(s) \in R$ (the container of a state is a region).*
- *For every $r \in R$ if $r = TOP$ then $\Omega(r) = \epsilon$, otherwise $\Omega(r) \in S$ (the container of a region is a state and TOP has no container).*
- *For every $r \in R$ s.t. $r \neq TOP$, $\Omega(r) \in S_{com}$ (only compound states contain regions)*
- *For every $r \in R$ there exists at least one $s \in S$ such that $\Omega(s) = r$*
- *The transitive closure of Ω is irreflexive*

The function Ω induces a partial order on $S \cup R$: $u \lhd u'$ denotes that u' *contains* u.

We say that two different regions $r_1, r_2 \in R$ are *orthogonal*, denoted $ORTH(r_1, r_2)$, if they are contained in the same state.
Formally, $ORTH(r_1, r_2) = true$ iff $r_1 \neq r_2$ and $\Omega(r_1) = \Omega(r_2)$.

From here on we assume a fixed set V of variables over finite domains. We use Λ to denote the set of all possible valuations for the variables in V, and λ or λ_i to denote specific assignments. We use $\mathcal{B}$ to denote the set of Boolean expressions over V. We also assume a fixed set of environment events EV_{env} and a fixed set of system events EV_{sys}, and we denote $EV = EV_{env} \cup EV_{sys}$. 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).

Definition 2.2 (Actions) *An action is a sequence of statements in some programming language. A simple statement is either an assignment "$x = e$" over variables in V, or "$GEN(e)$", which is the generation of an event from EV_{sys}. skip represents an empty sequence of statements. A compound statement is a sequence of statements, "$a_1; a_2$" or a branching statement "if b then a_1 else a_2", for actions a_1 and a_2 and $b \in \mathcal{B}$.*

Given an action *act*, we denote by $modif(act)$ the set of variables that may be modified on *act*. Formally, $x \in modif(act)$ if statement "$x = e$" is part of *act*.

Note that we restrict the action language and disallow dynamic allocation of objects and memory, dynamic pointers, unbounded loops, and recursion. These restrictions enable us to focus on the model checking of UML systems, while avoiding orthogonal issues such as termination and pointer analysis.

Definition 2.3 (State Machines) *A* state machine *is a tuple* $(S, R, \Omega, init, TR, L, \mathcal{H})$ *such that:*

- *S, R, and Ω are the sets of states and regions and the Ω function, as defined above.*
- *$init \subseteq S$ are initial states, such that there is exactly one initial state in each region.*
- *$TR \subseteq S \times S$ is the set of transitions. Each transition t connects a single source state $src(t)$ with a single target state $trgt(t)$.*
- *$L : TR \to EV \times \mathcal{B} \times Actions$ is a function that labels each transition with a trigger (an event from EV), a guard, and an action. Since none of these components are mandatory we assume $\epsilon \in EV$ representing no trigger, $true \in \mathcal{B}$ representing an empty guard, and $skip \in Actions$ representing no action. We use $trig(t)$, $grd(t)$, and $act(t)$ to refer to the trigger, guard, and action of t respectively.*
- *$\mathcal{H} \subseteq R$ is the history marker, marking those regions that have history (these would have a history pseudostate in them in the graphical representation).*

Transitions t where $trig(t) = \epsilon$ and $grd(t) = true$ are referred to as *null transitions.* Recall that $modif(act)$ denotes the set of variables that may be modified on act. By abuse of notation, $modif(t)$ denotes the set of variables that may be modified on $act(t)$.

Figure 2.1 describes a state machine. States are denoted as squares. Regions are graphically represented only if they are orthogonal. Orthogonal regions are denoted by a dashed line. For example, state $Work$ contains two orthogonal regions, where one region contains states s_4, s_5 and s_6, and the other region contains states s_7, s_8, s_9 and the compound state $process$. Assume these regions are r_1 and r_2, then $ORTH(r_1, r_2) = true$

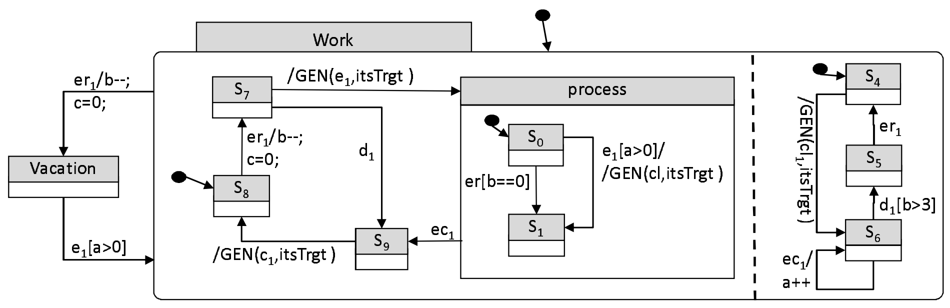


Figure 2.1: Example State Machine

(since $\Omega(r_1) = \Omega(r_2) = Work$). Note that these are the only orthogonal regions in the state machine.

A transition t is denoted 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. For example, in Figure 2.1, for transition t from s_0 to s_1 (i.e., $src(t) = s_0$ and $trgt(t) = s_1$), $trig(t) = er$, $grd(t) = (b == 0)$, and $act(t) = skip$ (thus the action is omitted from the representation). The transition from s_7 to *process* is a null transition whose action is $GEN(e_1, itsTrgt)$.

Definition 2.4 (State Machine Configurations) *Let* $SM = (S, R, \Omega, init, TR, L, \mathcal{H})$ *be a state machine. An* SM-configuration *is a tuple* (ω, ρ, H) *such that:*

- $\omega \subseteq S$ *is the set of currently active states.* ω *has the property that for every* $s \in \omega$ *and for every* $r \in R$ *such that* $\Omega(r) = s$ *there exists a* single $s' \in S$ *such that* $\Omega(s') = r$ *and* $s' \in \omega$*. Also, there exists a state* $s \in \omega$ *such that* $\Omega(s) = TOP$*.*

- $\rho \in type(EV) \cup \{\epsilon\}$ *holds an event currently* dispatched *(formally defined later) to the state machine and not yet consumed (and* ϵ *if there is no event to be consumed).*

- $H : R \to S$ *is the history information. It records the last active state in each region marked with history (*$r \in \mathcal{H}$*), or the initial state if either the region has not yet been visited or the region is not marked with history.*

The requirements on ω ensure that for every compound state s in ω, and for every region r contained in s (i.e., $\Omega(r) = s$), there exists a single state s' contained in r ($\Omega(s') = r$) such that s' is in ω as well. For example, $\{Vacation\}$ and $\{Work, s_6, process, s_0\}$ are both possible sets of currently active states of the state machine in Figure 2.1

From here on, we assume that state machines do not include complex UML syntactic features: 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 flatenning.

2.1.2 Systems

Next we define UML systems and their behavior. UML2 places no restrictions on the implementation of the event queue and neither do we. We use a finite sequence $q = (e_1, ..., e_l)$ of events $e_i \in EV$ to represent the contents of an event queue at a particular point in time (thus the set of all possible values for an event queue is EV^*). We assume the functions *pop*(q), *top*(q), and *push*(q, e) are correctly defined with respect to the semantics of the event queue.

Definition 2.5 (System) *A* system *is a tuple* $(SM_1, ..., SM_n, Q_1, ..., Q_m, thread, V)$ *s.t.* $SM_1, ..., SM_n$ *are state machines,* $Q_1, ..., Q_m$ *(*$m \leq n$*) are event queues (one for each thread),* $thread : \{1, ..., n\} \rightarrow \{1, ..., m\}$ *assigns each state machine to a thread, and* V *is a collection of variables over finite domains.*

Note that in the original UML system variables are partitioned into private attributes, public attributes, and global variables. These definitions govern the constraints on which variables each state machine may read or write to. For the semantic model we bundle all variables together into a single vector V and assume that all accesses are legal.

Definition 2.6 (System Configuration) *Let* $\Gamma = (SM_1, ..., SM_n, Q_1, ..., Q_m, thread, V)$ *be a system. A* system-configuration *is a tuple* $(c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \lambda)$ *such that:*

- *c_i is an SM-configuration of SM_i*
- *q_j is the contents of Q_j*
- *$id_j \in \{0, ..., n\}$ is the id of the state machine associated with thread j that is currently executing a run-to-completion step. $id_j = 0$ means that all the state machines of thread j are* inactive*.*
- *λ is an assignment giving each variable in V a value from its legal domain.*

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

Next we define computations of a system. In principle, a computation is a series of transitions fired according to certain constraints and following the run-to-completion semantics per-thread. The main difference between our definition and the majority of formal semantic theories suggested for UML state machines is that we differentiate between the extraction of an event from the event queue and the state machine transition that is fired as a result of this event being dispatched.

In order to define computations we require a few more definitions.

Definition 2.7 (Enabled Transition) *A transition t of a state machine SM_i is* enabled *in a configuration $C = (c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \lambda)$ (where $c_i = (\omega_i, \rho_i, H_i)$), denoted enabled$(t, C)$, if the following conditions hold:*

- *$src(t) \in \omega_i$ (the source state of t is active)*
- *$trig(t) = \rho_i$ (the trigger is the currently dispatched event, or no trigger on the transition if $\rho_i = \epsilon$)*
- *$\lambda \models grd(t)$ (the guard of the transition is satisfied under the current assignment to variables)*

- *For every $t' \in TR_i$ such that $src(t') \in \omega_i$ and $src(t') \lhd src(t)$: $trig(t') \neq \rho_i$ or $\lambda \not\models grd(t')$ (a transition is enabled if all transitions from states contained in $src(t)$ are not enabled)*

By abuse of notation we say that a state machine SM_i is enabled in configuration C, denoted $enabled(i, C)$, if SM_i has an enabled transition in C. That is, $enabled(i, C)$ is *true* iff there exists $t \in TR_i$ such that $enabled(t, C)$.

We say that a state machine configuration c_i is *stable* in a configuration $C = (c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \lambda)$ if there are no enabled transitions in SM_i.

Example 2.8 *Assume the state machine in Figure 2.1, denoted SM_1, is part of a system Γ. Assume a system-configuration C of Γ, where the SM-configuration of SM_1 is $c_1 = (\omega_1, \rho_1, H_1)$, $\omega_1 = \{Work, s_6, process, s_0\}$, and $\rho_1 = er$. Assume also that for the variable assignment λ in C, $\lambda(b) = 0$. Let $t \in TR_1$ be the transition from s_0 to s_1, then $enabled(t, C) = true$. Moreover, for every other transition $t' \in TR_1$ such that $t' \neq t$, $enabled(t', C) = false$, since either $src(t') \notin \omega_1$ or $trig(t') \neq \rho_1$.*

Definition 2.9 (Transition Execution on state machine) *When a transition t of a state machine SM_i in state machine configuration $c_i = (\omega_i, \rho_i, H_i)$ is executed, SM_i moves to a new state machine configuration $c_i' = (\omega_i', \rho_i', H_i')$, denoted $dest(c_i, t)$, which is defined as follows:*

- $\omega_i' = (\omega_i \setminus \{s \in \omega_i | s = src(t) \vee s \lhd src(t)\}) \cup \{s \in S | s = trgt(t) \vee (s \lhd trgt(t) \wedge s = H_i(\Omega(s)) \wedge \forall s' \in S : s \lhd s' \lhd trgt(t) \rightarrow s' = H_i(\Omega(s')))\}$ *(ω_i' is obtained by removing from ω_i states contained in $src(t)$ and then adding states contained in $trgt(t)$, based on the history).*

- $\rho_i' = \epsilon$ *(an event is consumed once)*

- *For every region $r \in R_i$ where $r \in \mathcal{H}_i$: If there exists $s \in S_i$ s.t. $s \in \omega_i'$ and $\Omega(s) = r$ then $H_i'(r) = s$ (if region r is an active region that has history marker, then we update the history according to the current state). Otherwise, $H_i'(r) = H_i(r)$.*

Example 2.10 *Let SM_1 be the state machine in Figure 2.1. Let $c_1 = (\omega_1, \rho_1, H_1)$ be a SM-configuration of SM_1 where $\omega_1 = \{Work, s_6, process, s_0\}$,* *and $\rho_1 = er$. Let $t \in TR_1$ be the transition from s_0 to s_1, then execution of t results in a new SM-configuration $dest(c_1, t) = (\omega'_1, \rho'_1, H'_1)$, where $\omega'_1 = \{Work, s_6, process, s_1\}$, $\rho'_1 = \epsilon$, and $H'_1 = H_1$ (since no region with history marker in SM_1).*

Let C be a system-configuration, SM_i be a state machine in Γ, and let $s_1, s_2 \in S_i$ and $t, t_1, ..., t_y \in TR_i$. We will further use the following notations:

- $Qpush(t, (q_1, ..., q_m)) = (q'_1, ..., q'_m)$ denotes the effect of executing transition t on the different queues of the system; if for some event e, $GEN(e) \in act(t)$, then executing t pushes e to the relevant event queue (to $Q_{thrd(trgt(e))}$). The rest of the event queues remain unchanged.
- $act(t)(\lambda, C) = \lambda'$ represents the effect of executing the assignments in $act(t)$ on the valuation λ of C, which results in a new assignment, λ'.
- $ORTH(s_1, s_2)$ is $true$ if the states are contained in orthogonal regions, and $false$ otherwise. Formally, $ORTH(s_1, s_2) = true$ iff $\exists r_1, r_2 \in R_i$ s.t. $ORTH(r_1, r_2)$ and for $k \in \{1, 2\}$: $s_k \lhd r_k$. For example, in Figure 2.1, $ORTH(s_0, s_4) = true$ since s_0 and s_4 are each contained in a region of state $Work$.
- $ORTH(t_1, ..., t_y)$ is $true$ iff $t_1, ..., t_y$ are pairwise orthogonal. I.e., for every $k, l \in \{1, ..., y\}$ s.t. $k \neq l$: $ORTH(src(t_k), src(t_l))$.
- $maxORTH((t_1, ..., t_y), C)$ is $true$ iff $(t_1, ..., t_y)$ is a *maximal set* of *enabled* orthogonal transitions. Formally $maxORTH((t_1, ..., t_q), C) = true$ iff (1) for every $i \in \{1, ..., q\}$, $enabled(t_i, C)$, and (2) $ORTH(t_1, ..., t_y)$, and (3) there is no $t \in TR_i$ such that $enabled(t, C)$ and $ORTH(t, t_1, ..., t_y)$. Note that for some configuration C and state machine SM_i there can be several *different* sets transitions for which $maxORTH$ is $true$.

Example 2.11 *Assume the state machine in Figure 2.1, denoted SM_1, is part of a system Γ. Assume a system-configuration C of Γ, where the SM-configuration of SM_1 is $c_1 = (\omega_1, \rho_1, H_1)$, $\omega_1 = \{Work, s_4, s_7\}$, and $\rho_1 = \epsilon$. Let $t_1 \in TR_1$ be the transition from s_7 to process, and let t_2 be the transition from s_4 to s_6. Then $orth(t_1, t_2) = true$, and $enabled(t_1, C) = enabled(t_2, C) = true$. Therefore $maxORTH((t_1), C) = false$ and $maxORTH((t_1, t_2), C) = true$.*

Definition 2.12 (Transition Execution on System) *Let $C = (c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \lambda)$ be a configuration on Γ, and let $t_1, ..., t_q \in TR_i$ (possibly $q = 1$) be a set of transitions. $apply((t_1, ..., t_q), C) = C'$ represents the effect of executing t_1 of C followed by t_2 on the result etc. until executing t_y, which results in configuration $C' = (c_1, ..., c'_i, ..., c_n, q'_1, ..., q'_m, id_1, ..., id_m, \lambda')$ defined as follows:*

- $c'_i = dest(...dest(dest(dest(c_i, t_1), t_2), t_3)..., t_q)$
- $\lambda' = act(t_q)(...act(t_3)(act(t_2)(act(t_1)(\lambda, C), C), C)..., C)$
- $q'_1, ..., q'_m = Qpush(t_q, (...Qpush(t_3, (Qpush(t_2, (Qpush(t_1, (q_1, ..., q_m))))))...))$

2.2 Linear-time Temporal Logic (LTL)

Let AP be a set of atomic propositions. A *Kripke structure* is a tuple $M = (\mathcal{S}, I_0, \mathcal{R}, \mathcal{L})$, where $\mathcal{S}$ is a set of K-states, $I_0 \subseteq \mathcal{S}$ is a set of initial K-states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a total K-transition relation, and $\mathcal{L} : \mathcal{S} \to 2^{AP}$ is a labeling function that maps each K-state to a set of atomic propositions. A *path* of M is an infinite set of K-states $s_0, s_1, ...$ s.t. for every $i \geq 0$, $(s_i, s_{i+1}) \in \mathcal{R}$.

The Linear-time Temporal Logic (LTL) [45] is suitable for expressing properties of a system along a path. Formulas of LTL are constructed from a set AP of atomic propositions using the usual Boolean operators and the temporal operators X ("next time"), and U ("until"). Formally, an LTL formula over AP is defined as follows:

- $true|false|p$ for $p \in AP$
- $\neg\psi_1|\psi_1 \wedge \psi_2|X\psi_1|\psi_1 U \psi_2$ for ψ_1, ψ_2 LTL formulas.

Let $\pi = s_0, s_1, ...$ be a path in a Kripke structure M. $\pi^i = s_i, s_{i+1}, ...$ denotes the suffix of π starting at state s_i. The semantics of LTL is inductively defined as follows:

- $\pi \models true$, $\pi \not\models false$.
- For $p \in AP$: $\pi \models p$ iff $p \in \mathcal{L}(s_0)$.
- $\pi \models \neg\psi_1$ iff $\pi \not\models \psi_1$

- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$
- $\pi \models X\psi_1$ iff $\pi^1 \models \psi_1$
- $\pi \models \psi_1 U \psi_2$ iff there exists $k \geq 0$ s.t. $\pi^k \models \psi_2$ and for all $0 \leq i < k$, $\pi^i \models \psi_1$.

We use the following abbreviations in writing formulas:

- $\vee, \rightarrow, \leftrightarrow$ are interpreted in the usual way.
- $F\psi \equiv true U \psi$ ("eventually").
- $G\psi \equiv \neg F \neg \psi$ ("always").

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

Chapter 3

Semantics of System Computations

In this chapter we formalize the notion of *system computation*, and present formal semantics for behavioral UML systems that rely on state machines. Works such as [18, 22, 35] also give formal semantics to state machines, however they all differ from our semantics: [18] 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. [22] define the semantics of a *single* state machine. Thus it neither addresses the semantics of the full system, nor the communication between state machines. [35] 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.

Definition 3.1 (System Computations) *A* computation *of a system* Γ *is a maximal sequence* $C^0, step^0, C^1, step^1, ...$ *such that: (1) each* C^k *is a system-configuration, (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 all event queues are empty, and the state machines are in their initial state and are inactive. Formally: C^0 is the initial system configuration, such that for every j: $q_j^0 = \phi$ and $id_j^0 = 0$. c_i^0 is the initial configuration of SM_i ($\rho_i^0 = \epsilon$ 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 event queue only if the processing of the previous event on thread j has terminated (i.e. the run-to-completion step ended) and the queue is not empty. A dispatch step pops the event out of the thread's queue and places it in the target object's ρ element. It also updates the corresponding id_j^{k+1} with the index of the state machine that is the target of the event. Formally:

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

Transition A transition can be fired if it is enabled and the state machine containing it is currently executing a RTC step. If the state machine's ρ element is not empty then the fired transition has ρ as its trigger. After firing the transition ρ is set to ϵ (so that an event cannot be consumed twice). There is a single case where more than one transition can be fired together. It is the case where transitions are in orthogonal regions and several transitions simultaneously consume the event (the state machine's ρ element is not empty). UML2 defines a simultaneous execution of the transitions in this case. Since it is not clear how to define simultaneous execution of actions, we define an interleaved execution of these transitions. Only after all transitions have executed, the next step is enabled. Note that the transitions are executed according to their order in the $TRANS$ step (t_1 executed first, followed by t_2 etc.). However, since the step itself can be defined with any order of transitions, then if from a given configuration $step = TRANS(j, (t_1, ..., t_q))$ is possible, then also $step = TRANS(j, (t'_1, ..., t'_q))$ is possible for any permutation $(t'_1, ..., t'_q)$ of $(t_1, ..., t_q)$. Formally:

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

EndRTC If the currently running state machine on thread j is stable, then the RTC step is complete, id_j is set to zero, and the ρ element of the state machine that finished the RTC is cleared. Formally:

$$EndRTC(j,\epsilon) : \quad \frac{id_j = l > 0 \qquad stable(c_l, C)}{id'_j = 0 \qquad c'_l = (\omega_l, \epsilon, H_l)}$$

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

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

Let π be a computation. A run-to-completion (RTC) step w.r.t. π on thread j is a maximal sequence of $TRANS$ steps of state machine i where $thread(i) = j$, s.t. the $TRANS$ steps appear between a $DISP$ step (initiating the RTC step) and a $EndRTC$ step (terminating the RTC step). Note that between each $DISP$ step and its following $EndRTC$ step on thread j, the currently active state machine remains the same (the value of id_j does not change).

Definition 3.2 (Run-to-Completion Steps) *Given a computation* $\pi = C^0, step^0, C^1, step^1, ...,$ *a* run-to-completion (RTC) step *is a maximal series of steps* $\chi = step^{i_0}, step^{i_1}, ..., step^{i_d}$ *where for every* $r \in \{1,...,d\}$*:* $i_{r-1} < i_r$*, and for some thread* j *the following holds:*

- $step^{i_0} = DISP(j,e)$
- *For every* $r \in \{1,...,d-1\}$*:* $step^{i_r} = TRANS(j,(t_1,...,t_q))$

- *$step^{i_d} = EndRTC(j, \epsilon)$*
- *For every $r \in \{i_0, i_0 + 1, ..., i_d\}$ s.t. $r \neq i_0, i_1, ..., i_d$: if $step^r$ is on thread j, then $step^r == ENV(j, e')$ (for some event e'). This item ensures the maximality of χ.*

Chapter 4

Applying Software Model Checking Techniques For Behavioral UML Systems

In this chapter we present a novel approach for the verification of Behavioral UML systems by means of software model checking.

We translate UML systems to C and adopt software model checking techniques for their verification. Our translation preserves the high-level structure of the UML system: event-driven objects communicate with each other via an event queue. The hierarchical structure of the objects is maintained. An execution consists of a sequence of RTC steps. Each RTC step is initiated by the event queue by sending an event to its target object, which in turn executes a maximal series of enabled transitions. Therefore, we maintain the granularity of transitions as well as the RTC step semantics.

Model checking assumes a finite-state representation of the system in order to guarantee termination with a definite result. One approach for obtaining finiteness is to *bound* the length of the traversed executions by an iteratively increased bound. This is called *Bounded Model Checking* (BMC) [4]. BMC is highly scalable, and widely used, and is particularly suitable for bug hunting. We find this approach most suitable for UML systems, which are inherently infinite due to the unbound size of the event queue[1].

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

We emphasize that our goal is to translate the UML system into *verifiable* C code that suits model checking, rather than produce executable code. Also, we only wish to verify user-created artifacts. When translating to C, we therefore simplify implementation details that are irrelevant for verification. For instance, the event queue is described at a high level of abstraction, and code is sometimes duplicated to avoid pointers and simplify the verification. The resulting code is significantly easier for model checking than automatically generated code produced by UML tools such as Rhapsody [49]. It is important to note that the automatically generated code produced by tools such as Rhapsody are very complex to analyze, and the relevant parts for verification are tightly tangled along with parts not relevant for verification. Thus, trying to slice relevant parts from the automatically generated code is a task that cannot be done automatically.

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

We verify two types of properties: LTL safety properties and livelocks. Safety properties require that the system never arrives at bad states, such as deadlock states, or states violating mutual exclusion. *LTL safety properties* can further require that no undesired finite execution occurs. Checking (LTL) safety properties can be reduced to traversing the reachable states of the system while searching for bad states. We apply *Bounded reachability* with increasing bounds for finding bad states. Our method can also be extended to proving the absence of bad states, using k-induction [55].

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

Scalable bounded model checking tools mostly handle *safety* or linear-time properties. However, absence of livelocks is neither safety nor linear-

finiteness.

time property and is therefore not amenable to bounded model checking. We identify an important subclass of livelocks, which we refer to as *mutually-dependent livelocks*, and show that they can be found by combining static analysis and bounded reachability.

The property of deadlock has been the subject of many works. In the context of UML, [32] presents model checking for deadlocks via process algebra. The SPIN model checker itself supports checking for deadlocks. To the best of our knowledge, the property of livelocks has never been studied in the context of behavioral UML systems.

We implemented our approach to verifying behavioral UML systems with respect to LTL safety properties and mutually-dependent livelocks in a tool called soft-UMC (**soft**ware-based **U**ML **M**odel **C**hecking). Our tool is built on top of the software model checker CBMC [13] which applies BMC to C programs and safety properties. We ran it on several UML examples and interesting properties, and found erroneous behaviors and livelocks. For safety properties, we also compared soft-UMC with an IBM research tool that verifies behavioral UML systems via a translation to IBM's hardware model checker RuleBasePE [51]. Our experiments show that soft-UMC is more scalable and more robust for finding long counterexamples. Our experimental results also demonstrate the usefulness of the optimizations applied in the creation of the verifiable C code.

The rest of the chapter is organized as follows. In Section 4.1 we present some background. Our translation to verifiable C code is presented in Section 4.2, and our method for verification of (LTL) safety properties and mutually-dependent livelocks is presented in Section 4.3. We show our experimental results in Section 4.4, and conclude in Section 4.5.

4.1 Preliminaries

4.1.1 Bounded Model Checking

Bounded Model Checking (BMC) [4] is an iterative process for checking models against LTL formulas. The transition relations for a Kripke structure M and its specification are jointly unwound for k steps and are represented by a boolean formula that is satisfiable iff there exists an execution of M of length k that violates the specification. The formula is then checked by a

SAT solver. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. Otherwise, k is increased.

BMC is widely used for finding bugs in large systems, including software systems ([13, 3, 16]). BMC for software is performed by unwinding the loops in the program k times, and verifying the required property. The property is often described by an assertion added to the program text. The model checker then searches for a program execution that violates the assertion. Our method for verifying UML models relies on invoking a software BMC tool. We require that the tool supports assumptions on the program, given as $assume(b)$ commands, where b is some boolean condition. Having $assume(b)$ at location ℓ of the program means that only executions π that satisfy b when passing at ℓ are considered. If b is violated then π is ignored.

4.1.2 Restrictions, Notations and Abbreviations

In the rest of the chapter we focus on systems that run on a single thread. Thus, a system (Definition 2.5) is $\Gamma = (SM_1, ..., SM_n, Q, V)$ and a system configuration (Definition 2.6) is $C = (c_1, ..., c_n, q, id, \lambda)$, where $c_i = (\omega_i, \rho_i, H_i)$. As described in Section 2.1.2, UML2 places no restrictions on the implementation of the event queue. In this work we choose to follow the Rhapsody semantics, and implement event processing as a FIFO.

We use a flight ticket ordering system as a running example throughout the rest of the chapter. The system includes two DB objects and two $Agent$ objects. The system is therefore represented as $(a1, a2, db1, db2, Q, V)$, where $a1$ and $a2$ are state machines of type $Agent$, presented in Figure 4.2, and $db1$ and $db2$ are state machines of type DB, presented in Figure 4.1. Each DB object communicates with a single $Agent$ object, and with the other DB object. These are denoted as $itsA$ and $itsDB$ respectively in the state machine. Each $Agent$ object communicates with a single DB object, denoted as $itsDB$ in the state machine. Formally, for $i, j \in \{1, 2\}$, $itsDB$ of ai is dbi, and $itsA$ of dbi is ai. Also, $itsDB$ of dbi is dbj, where $i \neq j$.

The definition of enabled transitions (Definition 2.7) requires that the trigger of the transition matches the dispatched event, or no trigger on the transition if the value of the dispatched event is ϵ (i.e., this is not the first transition executed in the RTC step). In this chapter we follow the Rhapsody semantics and require that either the transition has no trigger or

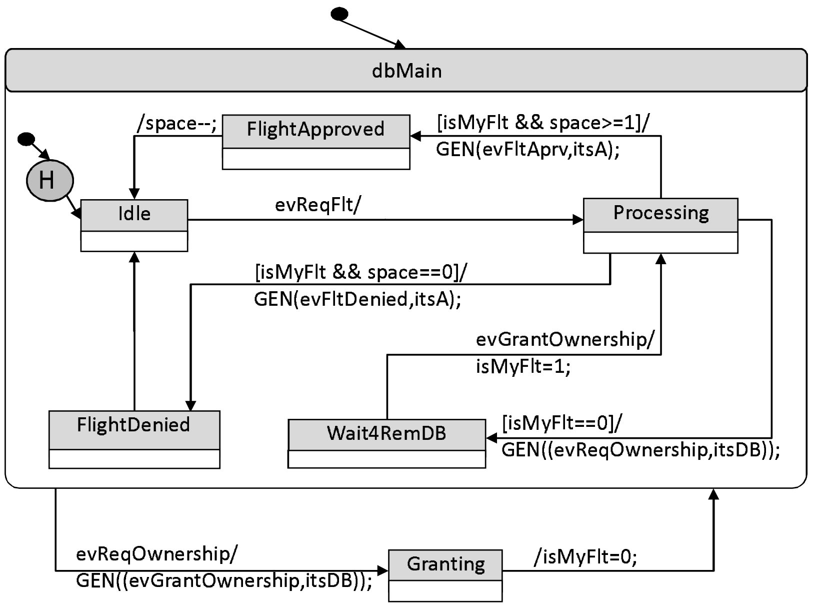


Figure 4.1: State Machine for Class DB

the trigger matches the dispatched event (i.e., the first transition in the RTC step might not be marked with a trigger). Note that if the first transition in the RTC step of state machine SM is not marked with a trigger, then the transition is marked with a guard whose value was $false$ in the previous RTC step of SM (if such RTC step exists). That is, the value of some variable was modified by a state machine different from SM.

The following terminology will be needed later. State machines that can send some event (ev, i) are called *producers* of (ev, i). In our example, the (only) producer of event $(evReqOwnership, db1)$ is $db2$. State machines that can modify some variable x of state machine SM are called *modifiers* of (x, SM). In our example, the (only) modifier of variable $isMyFlt$ of $db1$ is $db1$. Let b be a guard in a state machine SM, where b includes variables $\{x_1, ..., x_m\}$. The set of modifiers of all variables in b are called the modifiers of (b, SM).

Throughout the rest of the chapter we will use the following notations and abbreviations. Given a state machine $SM_i = (S_i, R_i, \Omega_i, init_i, TR_i, L_i, \mathcal{H}_i)$ and a state $s \in S_i$:

- $trans(s) \subseteq T_i$ is the set of transitions whose source is s.

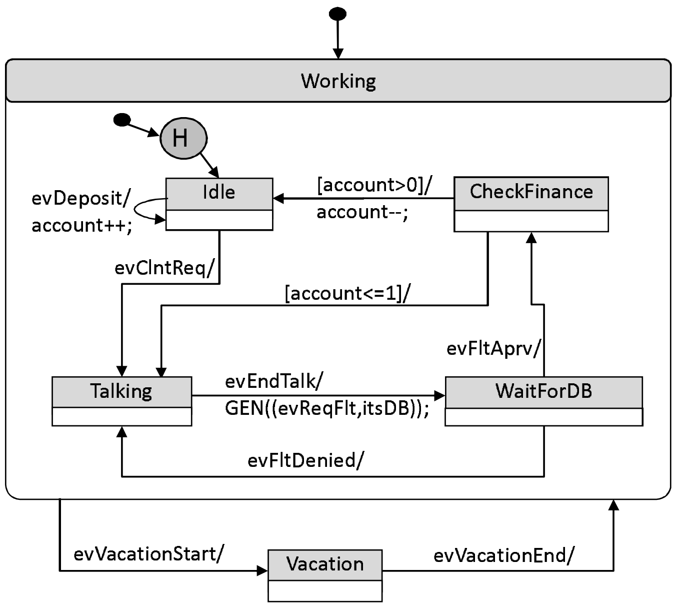


Figure 4.2: State Machine for Class *Agent*

- $evnts(s) = \bigcup_{t \in trans(s)} \{(trig(t), i)\} \setminus \{(\epsilon, i)\}$ is the set of triggers on $trans(s)$.

- $grds(s) = \bigcup_{t \in trans(s)} \{(grd(t), i)\}$ is the set of guards on $trans(s)$.

- $prod(s) \subseteq \{1, ..., n\}$ denotes indexes of producers of all events in $evnts(s)$. For example, if $evnts(s') = \{(ev, j)\}$, and the producers of (ev, SM_j) are $\{SM_{i_1}, ..., SM_{i_k}\}$, then $prod(s') = \{i_1, ..., i_k\}$.

- $modifier(s) \subseteq \{1, ..., n\}$ denotes indexes of modifiers of all guards in $grds(s)$.

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

Given a system $\Gamma = (SM_1, ..., SM_n, Q, V)$ and a system configuration C, we say that $enabled(i, C)$ is $true$ if there exists a transition $t \in TR_i$ such that $enabled(t, C)$ is $true$, and $false$ otherwise.

```
1: method $RunRTCStep_i()$
2: while $(j < maxRTClen)$ do
3:    if $(!enabled(i, currC))$ return
4:    $choose\ Transition\ t$
5:    assume$(t \in trans(\omega_i))$
6:    assume$(enabled(t, currC))$
7:    $execute\ \ act(t)$
8:    $\rho_i := \epsilon$
9:    $incr\ \ j$
```

Figure 4.3: $RunRTCStep_i$ method of state machine SM_i

4.2 Translation to Verifiable Bounded C

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

Every object is translated into a method, representing the behavior of its associated state machine. When an event *ev* is dispatched to object o_i, the method associated with o_i executes a single RTC step of o_i.

Figure 4.3 presents $RunRTCStep_i$, the pseudo-code for a single RTC step of o_i. *currC* is the current system configuration. The method terminates when there are no enabled transitions to execute. The `while` loop iterates up to *maxRTClen* iterations. *maxRTClen* represents the maximum number of transitions of any RTC step of o_i. If this value cannot be extracted by static analysis, then the condition is replaced by *true*, and the length of the RTC step is bounded by the BMC bound, k.

Lines 4-6 amount to a non-deterministic choice of a transition t, which is enabled in *currC*. When choosing a transition (line 4), no constraints are assumed on it. Line 5 restricts the program executions to those where

```
1: method $main$
2: while $(true)$ do
3:     $(ev, i) := pop(q)$
4:     $\rho_i := ev$
5:     $RunRTCStep_i()$
```

Figure 4.4: *main* method

t is a transition from ω_i (the active states). Line 6 restricts the remaining program executions to those where t is enabled. In line 7 the action of the transition is executed. Executing the action updates ω_i according to the destination state of t. Note line 8, where we set ρ_i to ϵ. This is done since the event is consumed once, and only in the first transition of the RTC step. The rest of the transitions of the RTC step can be executed only if their trigger is ϵ.

The EQ is represented as a bounded array. The main method of the program executes the never-ending loop of taking an event from the EQ, and dispatching it to the relevant target object. Figure 4.4 presents the pseudo-code for the `main` method. In line 3 an event ev whose target is o_i is taken from the EQ. Line 4 updates ρ_i according to ev, and in line 5 an RTC step of o_i is initiated.

When applying BMC on the main method in Figure 4.4, the `while` loop is unrolled k times, which means that the model is verified for k RTC steps. Generally, placing a bound on the EQ can make the model inaccurate due to overflows. However, k is the exact bound for a k-bounded verification over k RTC steps, since only the first k events that are sent will be dispatched during k RTC steps.

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

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

4.3 System Verification

We now describe our method for verification of a given behavioral UML system. We assume a system $\Gamma = (SM_1, ..., SM_n, Q, V)$. Verification is done using assertions on the code describing the system. We support verification in a granularity of transition level or RTC level.

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

Definition 4.1 $\pi_r = C_0, C_1, ...$ *is an* execution at the RTC level (RTC-execution) *iff for every* $n > 0$*,* $(C_{n-1}, C_n) \in \mathcal{R}_{RTC}$*.*

Definition 4.2 $\pi_t = C_0, C_1, ...$ *is an* execution at the transition level (t-execution) *iff for every* $n > 0$*,* $(C_{n-1}, C_n) \in \mathcal{R}_t$*.*

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

4.3.1 Verifying LTL Safety Properties

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

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

Note that our method for BMC can be extended to proof by k-induction [55] in a straightforward manner. The base case is a BMC of k steps, which is done in the way we described above. The step is a BMC run of $k + 1$ steps with the initial state completely non-deterministic, looking for a run in which a property violation occurs at the $k + 1$ step after k steps with no violation. In the initial state of the step case we assume there may already be any number of events in the queue, of any type. We can still bound the event queue to $k + 1$ entries because no more than $k + 1$ events will be dispatched in $k+1$ steps, making it sound to ignore the content of the queue beyond $k + 1$ entries.

4.3.2 Verify Mutually-Dependent Livelocks

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

We first define the notion of a *livelock-configuration* in behavioral UML systems.

Definition 4.3 *Let $C = (c_1, ..., c_n, q, id, \lambda)$ be a system configuration of* Γ. *We say that SM_i is* disabled under C *if for every $t \in TR_i$, enabled$(t, C) = false$. That is, no transition $t \in TR_i$ is enabled.*

Definition 4.4 *Let $C = (c_1, ..., c_n, q, id, \lambda)$ be a system configuration of* Γ. *State machine SM_i is* stuck at C *if for every RTC-execution $\pi = C_0, C_1, ...$ s.t. $C_0 = C$ the following holds: for every $C_j = (c_1^j, ..., c_n^j, q^j, id^j, \lambda^j)$ s.t. $j \geq 0$, if top$(q^j) = (ev, i)$ then SM_i is disabled under C_j.*

Thus, SM_i is stuck if whenever the event at the top of the queue is targeted at SM_i, meaning it is SM_i's turn to execute, SM_i is disabled and cannot make any progress. Intuitively, whenever it is SM_i's turn to execute, SM_i is either waiting for a different event, or the guard on its transitions is $false$ under the current system-configuration.

Definition 4.5 *A system configuration C is a* livelock-configuration *if at least one state machine is stuck at C.*

Following, we present a characterization for a subclass of livelock configurations, which we call *mutually-dependent livelocks (MD-livelocks)*. Intuitively, a system configuration C is an MD-livelock if there is a subset of state machines that are stuck at C, and for every state machine SM in the subset all of the producers of events that SM is stuck on, and all of the modifiers of the guards that SM is stuck on, are in the subset as well.

Definition 4.6 *Let $C = (c_1, ..., c_n, q, id, \lambda)$ be a system configuration of* Γ. *A vector $\bar{\omega} = (\omega'_1, ..., \omega'_n)$ is a* partial state *of C if for every $1 \leq i \leq n$, $\omega'_i = nil$ or $\omega'_i = \omega_i$.*

Intuitively, a partial state of C represents the current state of some of the state machines in Γ. These are the state machines for which $\omega'_i \neq nil$.

Definition 4.7 *Let C be a livelock-configuration, and let $\bar{\omega} = (\omega'_1, ..., \omega'_n)$ be a partial state of C. $\bar{\omega}$ is a* livelock state of C *if for every $i \in \{1, ..., n\}$, if $\omega'_i \neq nil$ then SM_i is stuck at C.*

Definition 4.8 *Configuration C is an* MD-livelock *if there exists a livelock state of C, $\bar{\omega} = (\omega'_1, ..., \omega'_n)$ s.t. for all $j \in prod(\bar{\omega}) \cup modifier(\bar{\omega})$, $\omega'_j \neq nil$.*

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

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

Finding Mutually-Dependent States:

A state machine SM_i cannot be stuck at $C = (c_1, ..., c_n, q, id, \lambda)$, where $c_i = (\omega_i, \rho_i, H_i)$, if ω_i, the set of currently active states of SM_i, has a null-transition, or if ω_i has a transition that can be enabled by an environment event.

We first define the set of *possible-active-states*. Intuitively, this set over-approximates the possible currently active states of a state machine SM.

Definition 4.9 *Let $SM = (S, R, \Omega, init, TR, L, \mathcal{H})$ be a state machine. $\nu \subseteq S$ is a* possible-active-state *if the following hold.*

- *For every $s \in \nu$ and for every $r \in R$ s.t. $\Omega(r) = s$ there exists a single $s' \in S$ such that $\Omega(s') = r$ and $s' \in \nu$.*
- *There exists $s \in \nu$ such that $\Omega(s) = TOP$.*

Note that this definition follows exactly the definition of active states as part of state machine configuration (Definition 2.4).

Definition 4.10 *A possible-active-state ν is* potentially stuck *if for every $t \in trans(\nu)$, t is not a null-transition, and if $ev(t)$ is an environment event, then $grd(t) \neq true$.*

Following, we define *mutually-dependent states.* Intuitively, a *mutually-dependent state* represents a subset of state machines that are all potentially stuck and the state machines depend on each other, i.e. all the necessary producers are inside this subset.

Definition 4.11 *A* mutually-dependent state *is a vector $\bar{\nu} = (\nu_1, ..., \nu_n)$ s.t. for every $i \in \{1, ..., n\}$, $\nu_i = nil$ or ν_i is a possible-active-state of SM_i, and the following holds for every $\nu_i \neq nil$:*

1. *ν_i is a potentially stuck possible-active-state, and*
2. *There is no $j \in prod(\nu_i) \cup modifier(\nu_i)$ such that $\nu_j = nil$, and*
3. *$\bar{\nu}$ is minimal. That is, let $\bar{\nu}' = (\nu'_1, ..., \nu'_n)$ such that for every $i \in \{1, ..., n\}$, either $\nu'_i = nil$ or $\nu'_i = \nu_i$. If $\bar{\nu}' \neq \bar{\nu}$ then requirement 2 does not hold for $\bar{\nu}'$.*

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

Note that this definition is *syntactic*. That is, it depends only on the possible-active-states of the system. It does not depend on the variable assignment, the history or the event queue, which can be determined along a computation. As a result, the set of all mutually-dependent states can be identified independently of any configuration. We generate this set from the syntactic structure of the system, as part of the analysis of the UML system.

Lemma 4.12 *The set of mutually-dependent states is* complete. *Meaning for every MD-livelock configuration C there exists a partial state of C, $\bar{\nu}$, that is a mutually-dependent state.*

The set of system configurations is infinite, because the size of the EQ is not limited. However, the set of mutually-dependent states is finite.

```
1: method $FindMDLivelock()$
2: while $(true)$ do
3:     $(ev, i) := pop(q)$
4:     $\rho_i := ev$
5:     $RunRTCStep_i()$
6:     for each mutually-dependent state $\bar{\nu}'$ do
7:         assert$(!(partSt(\bar{\nu}', currC) \wedge$
                   for all $t \in trans(\bar{\nu}')$ :
                       $notInQ(trig(t), q) \vee$
                       $grdFalse(grd(t), \lambda)))$
```

Figure 4.5: *FindMDLivelock* method

Bounded Search for Mutually-Dependent Livelocks:

We observe that if a given system configuration includes a mutually-dependent state s.t. for every transition in the mutually-dependent state either the guard is false or the trigger is a system event which is not in the EQ, then this system configuration is a MD-livelock.

We adapt the translation of UML systems to C (Section 4.2) to allow checking whether a MD-livelock configuration is reachable by adding assertions at the RTC level. When the model checker finds an execution violating the assertion, the last system configuration in the execution is a MD-livelock configuration. Figure 4.5 presents the pseudo-code of the modified method. Line 6 and 7 show the added code.

currC represents the current system configuration of the system. At every iteration of the `while` loop *currC* changes (due to the RTC step). The method $partSt(\bar{\nu}, C)$ receives a mutually-dependent state $\bar{\nu}$ and a configuration C, and returns *true* iff $\bar{\nu}$ is a partial state of C (i.e., $partSt(\bar{\nu}, C)$ returns *true* iff for every $\nu_i \in \bar{\nu}$, if $\nu_i \neq nil$ then $\nu_i = \omega_i$). The method $grdFalse(grd, \lambda)$ returns *true* iff *grd* is *false* w.r.t. the variable assignment λ. The method $notInQ(ev, q)$ returns *true* iff *ev* is a system event which is not in the EQ q. The assertion is violated on C if C is a MD-livelock.

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

We exemplify our method on our running example. The events $evVacati$ $onStart$ and $evVacationEnd$, which are consumed by class Agent, are both environment events. Note that none of the possible-active-states associated with the state machine of Agent are potentially stuck possible-active-states. Thus, $a1$ and $a2$ can never be stuck. The vector $(\{Wait4RemDB, dbMain\}$, $\{Wait4RemDB,\ dbMain\}, nil, nil)$ is a mutually-dependent state because the producer of the possible-active-state $\{Wait4RemDB, dbMain\}$ of $db1$ is $db2$, and vice-versa. For this mutually-dependent state, we add the following assertion:

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

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

4.4 Experimental Results

We have implemented the algorithm described above in a tool called Soft-UMC (**soft**ware-based **U**ML **M**odel **C**hecking). The implementation reads a UML (version 2.0) system, and translates it to verifiable C code. Static analysis is applied at this stage, according to the type of property to be

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

Figure 4.6: Soft-UMC vs. HWMC. time in secs. ♯RTC and ♯trans is number of RTC steps and transitions in counterexamples

checked: (LTL) safety or livelock. We then apply CBMC[13] (version 4.1) as our C verifier.

First, we compared our implementation to the one translating the system to the input language of RuleBasePE[51], IBM's hardware model checker (we call this solution HWMC). HWMC represents the EQ as a bounded FIFO, where the size of the FIFO is relative to the maximum number of events generated in a single RTC step. It also preserves the hierarchical structure of the state machines.

To compare the performance of Soft-UMC and HWMC we used the following four examples. (1) A variant of the railroad crossing system from [46], including a gate object and three track objects that communicate with the gate, (2) The ticket ordering system (Figures 4.1 and 4.2), (3) A dishwasher machine (inspired by the example provided with Rhapsody), (4) A locking system, including a manager and three lock clients. We have checked several safety properties on the systems. In Figure 4.6 we present a comparison of the runtime for finding a counterexample in Soft-UMC and HWMC. It can be seen that HWMC is better on short counterexamples. However, on long ones Soft-UMC achieves results in shorter times. This can be explained by the initialization time of CBMC which is significant for short counterexamples but becomes negligible on long ones.

To check the scalability of our tool compared to HMWC, we considered

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

Figure 4.7: Compare scalability. time in secs.

three parameterized examples: The ticket ordering system, and variations of the dishwasher machine and the locking system. E.g., for the ticket ordering system, the attribute *account* of *Agent* is used as the parameter, and the checked property is non-determinism. For increasing initial values of *account*, the counterexample leading to a non-deterministic state is of increasing length. This allows us to experiment on the same system with different lengths of counterexamples. In all examples, a counterexample for a system with parameter i is of length $\sim 2 * i$ RTC steps. Each RTC step is composed of 3-5 transitions. We used a timeout of 1 hour. Results are presented in Fig 4.7. From the comparison it is clear that HWMC is better for shallow examples, however our tool is more scalable.

We also evaluated the performance impact of two of our optimizations, the EQ (Sec. 4.2) and the hierarchical system. We compared a naive implementation of the EQ against our optimized implementation. To analyze the impact of maintaining the hierarchy of the state machines we created a flat system from the ticket ordering system. The flat system has 24 states and 54 transitions, whereas the hierarchical system has 26 states and 36 transitions. The flat system is missing the hierarchical states. However, it has an additional attribute for maintaining the history. Figure 4.8 shows the results of the comparison. We compared the runtime of 4 different implementations: Hierarchical system with optimized EQ (H-OP-EQ), flat system with optimized EQ (F-OP-EQ), hierarchical system with naive EQ (H-NV-EQ) and flat system with naive EQ (F-NV-EQ).

We verified three different properties, and modified the system s.t. coun-

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

Figure 4.8: Optimizations on ticket ordering. Bound in RTC steps; time in secs.

terexample is reached at different bounds. 1,3 are safety properties. 2 is a livelock check, checked on a slightly modified system: the guard of transition from *Processing* to *FlightApproved* of *DB* (Figure 4.1) is modified to $[isMyFlt \ \&\& \ (space > 1)]$. This introduces a reachable livelock state, when $db1$ and $db2$ are in state $Processing$, $space = 1$ and $isMyFlt = true$ for both objects. Each row in Fig 4.8 represents a different setting defined by the property and the initial values of the attributes, which determine the length of the counterexample (in RTC steps). Time limit is set to 1 hour. It is clear that the optimized implementation of the EQ scales much better w.r.t. the naive EQ implementation. This is because the naive implementation includes a loop representing the addition of a non-deterministic number of environment events to the EQ. In the optimized implementation this amounts to a non-deterministic increment of the tail. The comparison also shows that the hierarchical implementation scales better than the flat one. Our conjecture is that flattening increases the number of transitions in the system, and therefore increases the search space. [19] presents similar results when comparing verification of hierarchical UML systems to flat systems. The above shows the significance of optimizations. We expect to be able to further improve performance of our solution with other optimizations.

4.5 Conclusions

This work is a first step in exploiting software model checking techniques for the verification of behavioral UML systems. By translating UML systems to C we could preserve the high-level structure of the system.

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

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

Our approach to finding MD-livelocks in UML models is novel. Static analysis identifies *syntactically* mutually-dependent states. During the model checking phase, we check whether these mutually-dependent states are reachable and represent a real MD-livelock. A suitable finite representation of the event queue then enables to apply BMC for finding such states that are reachable. We expect similar approaches to be useful for proving additional non-safety properties.

Chapter 5

Verifying Behavioral UML Systems via CEGAR

In this chapter we present a novel approach for applying abstraction and refinement for the verification of behavioral UML systems.

The *CounterExample-Guided Abstraction Refinement (CEGAR)* approach [10] provides an automatic and iterative framework for abstraction and refinement, where the refinement is based on a spurious counterexample. The concrete system is initially *abstracted*, which results in an abstract, *over-approximated* system. When model checking returns an abstract counterexample, a search is made for a matching concrete counterexample. If one exists, then a real bug on the concrete system is found. Otherwise, the counterexample is *spurious* and a refinement is needed. In the refinement stage, more details are added to the abstract system, in order to eliminate the spurious counterexample.

In this chapter we propose a CEGAR-like framework for verifying behavioral systems that rely on UML state machines. 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 transformations, we avoid the translation to lower level representation (such as Kripke structures). This is highly beneficial to the user, since the property, the abstraction, and the abstract counterexample are all given at 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, an 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. Our abstraction construction is presented in Section 5.1. We show that the abstract system is an over-approximation by proving that for every computation of the concrete system there exists a computation of the abstract system that "behaves similarly". This is formally defined and proved in Section 5.2.

Our CEGAR framework is suitable for verifying LTL_x, which is LTL without the next-time operator. Also, we assume the existence of a model checker for behavioral UML systems. As mentioned before, 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 5.1. To model check abstract systems we need a 3-valued model checker. Extending a model checker to support the 3-valued semantics (e.g., [54, 29]) is straightforward.

Many works such as [53, 56, 48, 21, 47] address *semantic refinement* of state machines. Semantic refinement is a method for top-down design, where details are added to a partially defined state machine, while behavior of the original (abstracted) model is preserved. We, on the other hand, remove details from a given model during the abstraction stage, in order to obtain a smaller 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 to improve 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.

From here on, we assume the following restriction on the actions of state machines: An action includes at most one "$GEN(e)$" statement. 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 the state machine can be preprocessed such that the transition is replaced with a series of states and transitions, each executing part of the original action.

We also assume that state machines do not include history. Therefore, a state machine SM is a tuple $(S, R, \Omega, init, TR, L)$, and a state machine configuration is a tuple $c = (\omega, \rho)$. That is, we exclude the $\mathcal{H}$ element in SM and the H element in c. Note that it is straightforward to eliminate history at the expense of additional states, transitions and variables.

5.1 Abstract State Machines

We now present the construction of abstract UML systems.

5.1.1 Abstracting a State Machine

The abstraction of a state machine $SM = (S, R, \Omega, init, TR, L)$ is defined w.r.t. a disjoint *abstraction collection* $ABS = \{\mathcal{A}^1, ..., \mathcal{A}^g\} \subseteq 2^S$. Every $\mathcal{A}^\beta$ is referred to as an *abstraction set.* Every abstraction set $\mathcal{A}^\beta$ is a set of states (simple or composite) for which the following holds.

1. For every $s, s' \in \mathcal{A}^\beta$: $\Omega(s) = \Omega(s')$, and
2. For every $s \in \mathcal{A}^\beta$ and $s' \in \mathcal{A}^\gamma$, if $\beta \neq \gamma$ then $s \not\trianglelefteq s'$ and $s' \not\trianglelefteq s$.

The first requirement states that an abstraction set $\mathcal{A}^\beta$ includes states from a single region. The second requirement states that the abstraction sets are disjoint: there are no two states in different abstraction sets s.t. one state contains the other state. Intuitively, our abstraction replaces every $\mathcal{A}^\beta$ (and all states contained in $\mathcal{A}^\beta$) with a *different construct* that ignores the details of $\mathcal{A}^\beta$ and maintains an over-approximated behavior of the events generated by $\mathcal{A}^\beta$.

We add the value $don't-know$, denoted $\bot$, to the domain of all variables in V, where $\bot$ represents any value in the domain. The semantics of boolean operations is extended to 3-valued logic as follows: $\bot \wedge false = false$, $\bot \wedge true = \bot$ and $\neg\bot = \bot$. An expression is evaluated to $\bot$ if one of its arguments is $\bot$. 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 ρ (the event dispatched to the state machine).

Next, we define several notions that are concrete and are defined w.r.t. an abstraction set $\mathcal{A} \in ABS$:

- $S(\mathcal{A}) = \{s \in S | \exists s' \in \mathcal{A}.(s \lhd s')\}$ are the *abstracted states.*
- $R(\mathcal{A}) = \{r \in R | \exists s \in \mathcal{A}.(r \lhd s)\}$ are the *abstracted regions.*
- $TR(\mathcal{A}) = \{t \in TR | src(t)\ and\ trgt(t) \in S(\mathcal{A})\}$ are the *abstracted transitions.*
- $EV(\mathcal{A}) = \{e \in EV | \exists t \in TR(\mathcal{A}).(GEN(e) \in act(t))\}$.
- $Trig(\mathcal{A}) = \{tr | \exists t \in TR(\mathcal{A}).(trig(t) = tr)\} \setminus \{\epsilon\}$.
- $V(\mathcal{A}) = \{v \in V | \exists t \in TR(\mathcal{A}).(v \in modif(t))\}$.
- $GRDV(\mathcal{A}) = \{v \in V | \exists t \in TR(\mathcal{A}).(trig(t) = \epsilon \wedge v \in grd(t))\}$.

Let $\Gamma = (SM_1, ..., SM_n, Q_1, ..., Q_m, thread, V)$ be a system, let ABS be an abstraction collection of SM_i, and let $\mathcal{A} \in ABS$ be an abstraction set. We require the following restrictions of $\mathcal{A}$:

1. For every $v \in V(\mathcal{A})$, if v can be modified by several state machines in Γ, then all these state machines are assigned to the same thread. Formally: if $v \in modif(TR_i) \cap modif(TR_j)$ then $thread(i) = thread(j)$. This is needed for correctness of the construction (details in the proof of theorem 5.11). Intuitively, this ensures that the value of v cannot be changed by a different state machine during the execution of the abstract state machine.
2. For every $t \in TR(\mathcal{A})$, if $trig(t) \neq \epsilon$ then for every $e \in EV$, $GEN(e) \notin act(t)$.

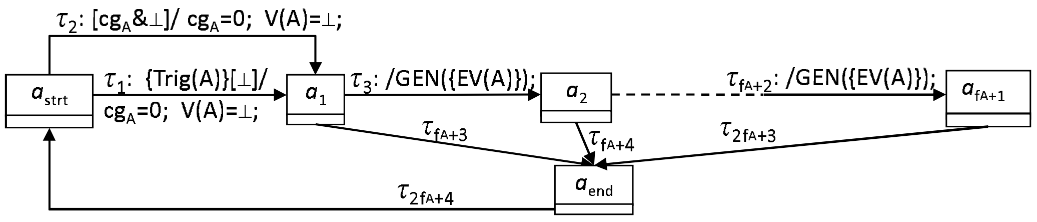


Figure 5.1: $\Delta(\mathcal{A})$: The abstraction construct created for abstraction set $\mathcal{A}$

3. There are no loops without triggers within $S(\mathcal{A})$. Further, there are no self loops without a trigger on states containing $S(\mathcal{A})$. This is needed to enable the static analysis described next.

In order to explain our abstraction we introduce the notion of an Abs-round on abstraction set $\mathcal{A}$, which is a maximal, possibly non-consecutive, sequence of steps from a computation π, s.t. all the steps are part of a single RTC, every step executes an abstracted transition, and the state machine remains in an abstracted state throughout the Abs-round. Formally,

Definition 5.1 (Abs-Round) *Let $\mathcal{A}$ be an abstraction set of state machine SM_i from system Γ, and let $\tau = step^{i_0}, ..., step^{i_d}$ be a RTC step of Γ on SM_i. An* Abs-round *on $\mathcal{A}$ is a maximal sub-sequence of τ, $\tilde{\tau} = step^{i_{j_1}}, ..., step^{i_{j_k}}$ s.t. the following holds:*

1. *For every $j \in \{j_1, j_2, ..., j_k\}$: $step^{i_j} = TRANS(j, (t_1, ..., t_y))$ (possibly $y = 1$), and there exists a transition $t \in \{t_1, ..., t_y\}$ s.t. $t \in TR(\mathcal{A})$ (the step executes an abstracted transition), and*

2. *For every $\iota \in \{i_{j_1}, (i_{j_1}) + 1, ..., i_{j_k}\}$, $\omega_i^{\iota} \cap S(\mathcal{A}) \neq \phi$ (the state machine remains in the abstracted states throughout $\tilde{\tau}$.*

Since there are no loops without triggers that include abstracted states, we can easily apply static analysis in order to determine the maximal number of events that can be generated by any single Abs-round of abstraction set $\mathcal{A}$. We denote this number by $f_{\mathcal{A}}$.

Given an abstraction set $\mathcal{A} \in ABS$, our abstraction replaces $S(\mathcal{A})$,$R(\mathcal{A})$ and $TR(\mathcal{A})$ with *a new construct*, referred to as $\Delta(\mathcal{A})$, demonstrated in Figure 5.1. $\Delta(\mathcal{A})$ includes an initial state a_{strt} and a final state a_{end}. Every

Abs-round over states from $S(\mathcal{A})$ is represented by a computation that includes a single loop on $\Delta(\mathcal{A})$ from a_{strt} to a_{strt}. $\Delta(\mathcal{A})$ includes computations that can generate any sequence of size 0 to $f_{\mathcal{A}}$ events from $EV(\mathcal{A})$. also, all the variables that *can* be modified in the Abs-round are given the value $\perp$.

An Abs-round whose first transition consumes an event, is represented by a computation that starts with transition τ_1 from a_{strt} to a_1, which can consume any single event from $Trig(\mathcal{A})$. The guard $\perp$ on τ_1 and τ_2 represents a non-deterministic choice between "true" or "false". If the first transition on an Abs-round does not consume an event, it will be represented by transition τ_2, which is not marked with a trigger. Since $\Delta(\mathcal{A})$ contains a loop of transitions without triggers we must ensure that all RTCs through $\Delta(\mathcal{A})$ are finite. We introduce a new Boolean variable $cg_{\mathcal{A}}$. A trace on $\Delta(\mathcal{A})$ can be initiated without a trigger only if $cg_{\mathcal{A}}$ is 1. $\Delta(\mathcal{A})$ then sets $cg_{\mathcal{A}}$ to 0 on both transitions exiting a_{strt}.

When $cg_{\mathcal{A}}$ is set to 1 it signals that it is possible to execute an Abs-round that does not consume an event. Such a situation abstracts a concrete execution in which the RTC step that includes the Abs-round starts at a state that is not abstracted and continues within the abstraction. I.e., execution of a transition t whose source is outside of $S(\mathcal{A})$ and whose target is a state s that either contains $\mathcal{A}$ or $s \in \mathcal{A}$. 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 $\mathcal{A}$ or is in a different state machine, and t modifies a variable v and $v \in GRDV(\mathcal{A})$.

If by static analysis we can conclude that the first transition of every Abs-round *consumes an event*, then $cg_{\mathcal{A}}$ is redundant (and τ_2 can be removed). All the Abs-rounds are then represented by computations that start by traversing τ_1.

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

- $S^A = (S \setminus S(\mathcal{A})) \cup \{a_{strt}, a_1, ..., a_{f_{\mathcal{A}}+1}, a_{end}\}$
- $R^A = (R \setminus R(\mathcal{A}))$
- For every $s \in (S^A \cap S) \cup R^A$: $\Omega^A(s) = \Omega(s)$.

For every $s \in \{a_{strt}, a_1, ..., a_{f_\mathcal{A}+1}, a_{end}\}$: $\Omega^A(s) = \Omega(s')$ for some $s' \in \mathcal{A}$ (recall that all states in A are contained in the same region).

- If there exists $s \in \mathcal{A}$ s.t. $s \in init$ then $init^A = (init \cap S^A) \cup \{a_{strt}\}$. Otherwise, $init^A = init \cap S^A$.
- $TR^A = (TR \setminus TR(\mathcal{A})) \cup \{\tau_1, ..., \tau_{2f_\mathcal{A}+5}\}$.

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 Figure 5.1. Every transition $t \in TR \setminus TR(\mathcal{A})$ has a representation (*matching transition*) in $SM(\mathcal{A})$. Note that for every such transition, either $src(t)$ or $trgt(t)$ are not abstracted (are in $S \cap S^A$). In $SM(\mathcal{A})$, the connection to the$\Delta(\mathcal{A})$ is only through a_{strt}. Thus, if $src(t)$ or $trgt(t)$ are abstracted, then $src^A(t)$ or $trgt^A(t)$ respectively is $a_{strt} \in \Delta(\mathcal{A})$. The handling of $cg_\mathcal{A}$ 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 $SM(\mathcal{A})$ w.r.t. SM. For every $t \in TR \setminus TR(\mathcal{A})$:

1. $trgt(t) \in S(\mathcal{A})$ (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 ϵ then $act^A(t)$ is $act(t); cg_\mathcal{A} = 1$ (otherwise, $act^A(t)$ is $act(t)$). This describes the case that the RTC can start outside the abstraction and continue within the abstraction.
2. $src(t) \in S(\mathcal{A})$ (the source of t is abstracted): we define $src^A(t) = a_{strt}$, $act^A(t)$ is $cg_\mathcal{A} = 0; act(t)$ and $grd^A(t) = grd(t)\&\perp$. We add $\perp$ 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: $\mathcal{A} \lhd trgt(t)$: An execution of t may result in a new ω (current active state) that includes an abstracted state $s \in S(\mathcal{A})$. If there exists an abstracted transition from s whose trigger is ϵ, then $act^A(t)$ is $act(t); cg_\mathcal{A} = 1$ (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:

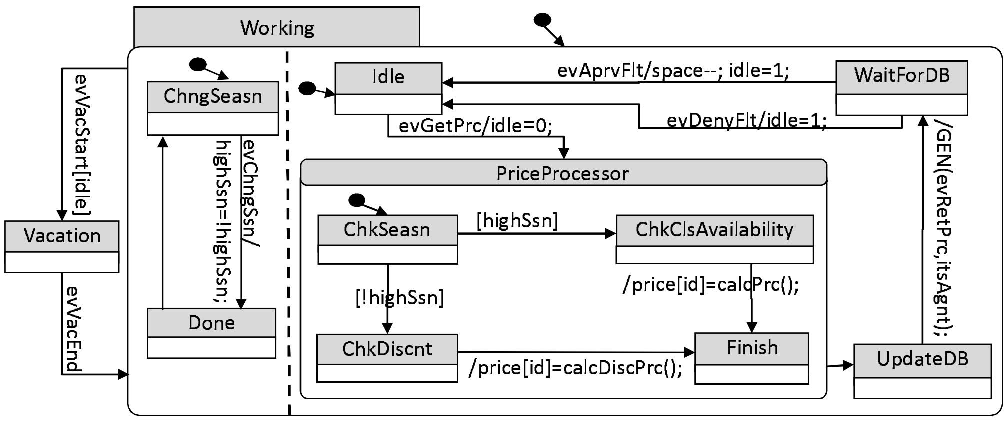


Figure 5.2: DB State Machine

- If there exists $v \in GRDV(\mathcal{A})$ such that $v \in modif(t)$ then $cg_{\mathcal{A}} = 1$ is added to $act^{A}(t)$, and
- If the current state of SM includes an abstracted state, then variables that *can* be modified by abstracted transitions are given the value $\perp$ on the first transition executed on $\Delta(\mathcal{A})$). The value on these variables should remain $\perp$ as long as the current state of SM includes an abstracted state. In order to ensure that the value remains $\perp$ even if t is executed orthogonally to the abstraction, every assignment $x = e$ in $act(t)$, if $x \in V(\mathcal{A})$ then $x = e$ is replaced with: "if $(isIn(\{a_{strt}, a_1, ..., a_{f_{\mathcal{A}}+1}, a_{end}\}))$ $x = \perp$; else $x = e$;" in $act^{A}(t)$. The current state is checked using the macro $isIn(U)$ where U is a set of states, that checks whether a certain state from U is active.

Example 5.2 *Consider the DB state machine presented in Figure 5.2. Abstracting the state machine with $A = \{Working, Vacation\}$ results in the state machine in Figure 5.3. Note that in this state machine, by static analysis we can conclude that every Abs-round first consumes an event, and therefore we do not need the cg_A flag and transition τ_2. Also, on every Abs-round no more than one event can be generated, therefore $f_A = 1$.*

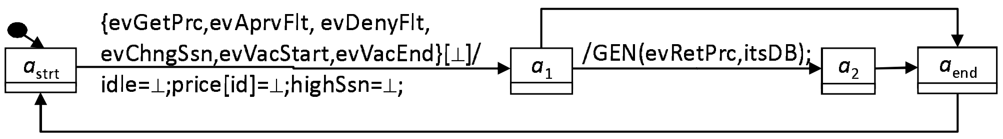

{evGetPrc,evAprvFlt, evDenyFlt,
evChngSsn,evVacStart,evVacEnd}[⊥]/
idle=⊥;price[id]=⊥;highSsn=⊥;
/GEN(evRetPrc,itsDB);
a_{strt}
a_1
a_2
a_{end}

Figure 5.3: Abstract DB State Machine

The above definitions enable us to define several different abstractions over a concrete state machine, by defining them one after the other. Given an abstraction collection $ABS = \{\mathcal{A}^1, ..., \mathcal{A}^g\}$, the abstraction of SM w.r.t. ABS is defined as $SM^A = (((SM(\mathcal{A}^1))(\mathcal{A}^2))....)(\mathcal{A}^g)$.

5.1.2 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 collection $ABS_i = \{\mathcal{A}^1, ..., \mathcal{A}^g\}$, SM_i^A denotes the abstraction of SM_i w.r.t. ABS_i. We denote the cg variable in SM_i^A that was added when abstracting SM_i w.r.t. abstraction set $\mathcal{A}^\beta$ as cg_i^β.

Definition 5.3 *Let Γ and Γ' be two systems, each with n state machines and m event queues. We say that Γ' is an* abstraction *of Γ w.r.t. $\{ABS_1, ..., ABS_n\}$, denoted Γ^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. *$V' = V \cup \{cg_i^\beta | SM_i' = SM_i^A$ and $\mathcal{A}^\beta \in ABS_i\}$*
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 $v \in GRDV(\mathcal{A}^\beta)$ where $\mathcal{A}^\beta \in ABS_i$, and $v \in modif(t)$ then $cg_i^\beta = 1$ is added to $act'(t)$ (in SM_j').*

Recall that setting cg_i^β to 1 on SM_j^A signals that it is possible to execute an Abs-round on SM_i that does not consume an event. Requirement (4) in Definition 5.3 handles the case where a guard of an abstracted transition of

SM_i changes by a transition t of SM_j, by ensuring that cg_i^β is set to 1 on such transitions of TR'_j.

Adding the value $\perp$ to the domain of all variables in V affects the cases when a transition is enabled, and when a state machine is stable, since now $grd(t)(\lambda) \in \{true, false, \perp\}$. Intuitively, if $grd(t)(\lambda) = \perp$ 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 ($grd(t)(\lambda) \in \{true, \perp\}$) and all transitions from states contained in $src(t)$ *can be not enabled* ($grd(t')(\lambda) \in \{false, \perp\}$). Similarly, $stable(c_i, C)$ if c_i *can* be stable in C. I.e., for every $t \in TR_i$, s.t. $src(t) \in \omega_i$, either $trig(t) \neq \rho_i$ or $grd(t)(\lambda) \in \{false, \perp\}$.

Note that when enabling 3-valued semantics, a transition may be enabled, even though lower level transitions may be enabled as well. Note also that in the 3-valued context it still holds that for a SM-configuration c_i, if there exists a transition $t \in TR_i$ s.t. $src(t) \in \omega_i$, $trig(t) = \epsilon$ and $grd(t) = true$, then c_i is not stable. Thus, when a state machine SM_i finishes an RTC step, if SM_i is in an abstract state, then that state can only be a^β_{strt} (i.e., the start state of the abstraction construct replacing $\mathcal{A}^\beta$). Similarly, when an event is dispatched on some thread j, then for every state machine SM_i associated with thread j: if SM_i is in an abstract state, then that state can only be a^β_{strt}.

5.2 Correctness of The Abstraction

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

Definition 5.4 (Abstraction relation of SM-configuration) *Let $c = (\omega, \rho)$ and $c^A = (\omega^A, \rho^A)$ be SM-configurations of a state machine SM and its abstraction SM^A respectively. c^A* abstracts *c, denoted $c \preceq c^A$, if the following holds:*

- $\rho = \rho^A$
- *c and c^A agree on the joint states: $\omega \neq \omega^A$ iff $\omega \setminus \omega^A \subseteq S(A)$ and $\omega^A \setminus \omega \subseteq \Delta(\mathcal{A})$.*

Definition 5.5 (Abstraction relation of λ) *Let λ and λ' be variable assignments over V of Γ and V' of Γ^A respectively. We say that λ' abstracts λ, denoted $\lambda \preceq \lambda'$ if for every $v \in V$ either $\lambda(v) = \lambda'(v)$ or $\lambda'(v) = \bot$.*

Definition 5.6 (Abstraction relation of system-configuration) *Let C and C' be two system configurations of Γ and Γ^A respectively. We say that C'* abstracts *C, denoted $C \preceq C'$, if C and C' agree on the event queues and id elements, and the state machine configurations and λ' of Γ^A are abstraction of the matching elements in Γ:*

- *For $j \in \{1, ..., m\}$: $q_j = q'_j$ and $id_j = id'_j$*
- *For $i \in \{1, ..., n\}$: $c_i \preceq c'_i$*
- *$\lambda \preceq \lambda'$*

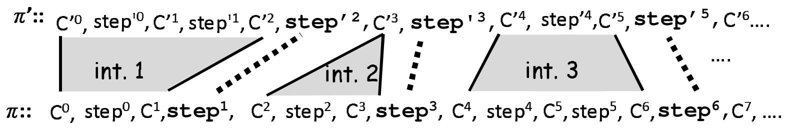
We will further need the following lemma, stating that when executing two matching transitions t and t_a from two computations C and C', where C' is an abstraction of C, then the resulting variable assignments are related.

Lemma 5.7 *Let C and C' be system-configurations of Γ and Γ^A respectively, such that $C \preceq C'$. For every $l \in \{1, ..., n\}$, for every $t \in TR_l$ and $t_a \in TR_l^A$: if t_a matches t then $act(t)(\lambda, C) \preceq act(t_a)(\lambda', C')$.*

Proof. We show that for every $v \in V$: either $act(t)(\lambda, C)(v) = act(t_a)(\lambda', C')(v)$ or $act(t_a)(\lambda', C')(v) = \bot$.

Since $C \preceq C'$, then for every variable $v \in V$, either $\lambda(v) = \lambda'(v)$ or $\lambda'(v) = \bot$. Note that by the definition of matching transitions, $modif(t) = modif(t_a) \cap V$. For every $v \in V$:

- If $v \notin modif(t)$ then $v \notin modif(t_a)$. Therefore $act(t)(\lambda, C)(v) = \lambda(v)$ and $act(t_a)(\lambda', C')(v) = \lambda'(v)$, and clearly the requirement holds.
- If $v \in modif(t)$: If $act(t_a)(\lambda', C')(v) \neq \bot$ then the value of v is determined by an evaluation of an expression over V for variables whose value is not $\bot$. These variables have the same value in λ, and the evaluating expression is the same. Therefore, $act(t)(\lambda, C)(v) = act(t_a)(\lambda', C')(v)$. Otherwise, $act(t_a)(\lambda', C')(v) = \bot$, and clearly the requirement holds.



π'::
C'0, step'0, C'1, step'1, C'2, step'2, C'3, step'3, C'4, step'4, C'5, step'5, C'6....
int. 1
int. 2
int. 3
....
π::
C0, step0, C1, step1, C2, step2, C3, step3, C4, step4, C5, step5, C6, step6, C7, ...

Figure 5.4: Stuttering Computation Inclusion

□

We now define *stuttering computation inclusion*, which is an extension of stuttering-trace inclusion ([11]) to system computations. For simplicity of presentation, we assume from now on that computations are infinite. however, all the results presented hold for finite computations as well. Intuitively, there exists stuttering inclusion between π and π' if they can be partitioned into infinitely many finite intervals, s.t. *every* configurations in the kth interval of π' abstracts *every* configuration in the kth interval of π, and vice versa.

Definition 5.8 (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* Γ *and* Γ^A *respectively. There exists a* stuttering computation inclusion *between* π *and* π'*, denoted* $\pi \preceq_s \pi'$*, if there are two infinite sequences of integers* $0 = i_0 < i_1 < i_2 < ...$ *and* $0 = i'_0 < i'_1 < i'_2 < ...$ *such that for every* $k \geq 0$ *the following holds. For every* $j \in \{i_k, ..., (i_{k+1}) - 1\}$ *and for every* $j' \in \{i'_k, ..., (i'_{k+1}) - 1\}$*:* $C^j \preceq C'^{j'}$

Note that corresponding intervals in π and π' may have different lengths. Figure 5.4 illustrates two computations where $\pi \preceq_s \pi'$. Definition 5.6 implies that steps of type $DISP$, ENV and $EndRTC$ cannot be steps within an interval, due to the effect of these steps on system-configuration. For example, in Figure 5.4, $C^6 \preceq C'^5$. 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 system-configuration 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 π implies a step of the same type on π', 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 ρ elements and the event queues.

The above is captured in the following lemma.

Lemma 5.9 *Let $\pi = C^0, step^0, C^1, step^1, ...$ and $\pi' = C'^0, step'^0, C'^1, step'^1, ...$ be two computations over Γ and Γ^A respectively, s.t. $\pi \preceq_s \pi'$. Let $0 = i_0 < i_1 < i_2 < ...$ and $0 = i'_0 < i'_1 < i'_2 < ...$ be two infinite sequences of positive integers describing the intervals of the stuttering inclusion. Then for every $k \geq 0$:*

- $step^{i_k} = DISP(j, e)$ *iff* $step'^{i'_k} = DISP(j, e)$
- $step^{i_k} = ENV(j, e)$ *iff* $step'^{i'_k} = ENV(j, e)$
- $step^{i_k} = EndRTC(j, \epsilon)$ *iff* $step'^{i'_k} = EndRTC(j, \epsilon)$
- $step^{i_k} = TRANS(j, (t_1, ..., t_y))$ *where* $id_j^{i_k} = l$ *and* $\rho_l^{i_k} \neq \epsilon$ *iff* $step'^{i'_k} = TRANS(j, (t'_1, ..., t'_{y'}))$ *where* $id_j^{i'_k} = l$ *and* $\rho_l^{i'_k} \neq \epsilon$
- $step^{i_k} = TRANS(j, (t_1, ..., t_y))$ *where* $id_j^{i_k} = l$ *and* $GEN(e) \in act(t)$ *for some* $t \in \{t_1, ..., t_y\}$ *iff* $step'^{i'_k} = TRANS(j, (t'_1, ..., t'_{y'}))$ *where* $id_j^{i'_k} = l$ *and* $GEN(e) \in act(t')$ *for some* $t' \in \{t'_1, ..., t'_{y'}\}$

An immediate result 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 Definition 5.6 requires a correlation between the current states of the state machines. It can therefore be shown (for a similar reason as above) that if $step^i = TRANS(j, (t))$ is a step inside an interval, i.e. between two configurations in the same interval, then one of the following holds: (1) If $step^i \in \pi$ then t is an abstracted transition, (2) If $step^i \in \pi'$ then $t \in \Delta(\mathcal{A})$.

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

Every system Γ can be viewed as a Kripke structure K, where the K-states are the set of system-configurations, and there exists a K-transition (C, C') iff C' is reachable from C within a single *step*. Thus, every computation of Γ corresponds to a path in K. Let Γ be a system, and let $A\psi$

be an LTL formula, where the atomic propositions are predicates over Γ. Then $\Gamma \models A\psi$ iff for every computation π of Γ from an initial configuration, $\pi \models \psi$. By preservation of LTL_x over stuttering traces we conclude:

Corollary 5.10 *Let Γ and Γ^A be two systems, s.t. $\Gamma \preceq_s \Gamma^A$, and let $A\psi$ be an LTL_x formula over joint elements of Γ and Γ^A. If $\Gamma^A \models A\psi$ then $\Gamma \models A\psi$.*

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

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

Theorem 5.11 *If Γ^A is an abstraction of Γ then $\Gamma \preceq_s \Gamma^A$.*

The proof of the above theorem is presented in Section 5.2.1. We give here an intuitive explanation to why for every π of Γ from C_{init}, there exists π' of Γ^A from C'_{init} such that $\pi \preceq_s \pi'$. For every step executed on Γ that does not include execution of an abstracted transition it is possible to execute the same step on Γ^A. More specifically, for every transition t executed on Γ, if t has a matching transition t_a in Γ^A, then t_a can be executed on π'. For every step of type ENV, $DISP$ and $EndRTC$ on π it is possible to execute the same step on π'. This holds since matching configurations C^r and C'^p of π and π' respectively agree on their joint elements, and λ'^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 an abstracted transition on Γ, every Abs-round χ of abstraction set $\mathcal{A}^\beta$ on some concrete state machine SM_i can be matched to a trace from a^β_{strt} to a^β_{end} on SM^A_i. The matching is as follows: every transition t that is traversed on χ and where t generated an event ($GEN(e) \in act(t)$) matches a transition from a^β_i to a^β_{i+1} (for some i). Every transition t that is traversed on χ and where t does not generate or consume an event, matches an interval of length one on π' (Γ^A does not execute a matching step). Since χ can generate at most $f_{\mathcal{A}}$ events, then indeed we can match the transitions as described. All variables that *can* be modified on χ are given the value $\perp$ upon execution of the first transition in $\Delta(\mathcal{A})$ (transitions from a^β_{strt} to a^β_1). This value is maintained in the variables throughout the traversal on $\Delta(\mathcal{A})$.

5.2.1 Proving Correctness of the Abstraction

This section includes the full proof of Theorem 5.11, which states that if Γ^A is an abstraction of Γ then $\Gamma \preceq_s \Gamma^A$. We will further use the following lemma, which captures the fact that when an event is dispatched on some thread j, then for all of the state machines associated with thread j: if the state machine is in an abstract state, then that state can only be a^{β}_{strt}.

Lemma 5.12 *Let $\pi = C^0, step^0, C^1, step^1, ...$ be some computation. For every $1 \leq j \leq m$, for every r s.t. $step^r = DISP(j, e)$, and for every l s.t. $thread(l) = j$: If $\mathcal{A}^{\beta} \in ABS_l$ then $\{a^{\beta}_1, ..., a^{\beta}_{f_{\beta}+1}, a^{\beta}_{end}\} \cap \omega^r_l = \phi$.*

For simplicity of the proof, we assume the following on Γ: For every $i \in \{1, ..., n\}$ and for every $t \in TR_i$, if $trig(t) \neq \epsilon$ then $act(t) = skip$.
Proof. Assume a computation $\pi = C^0, step^0, C^1, step^1, ...$ on Γ such that C^0 is an initial configuration. We prove by induction on the number of steps in π that there exists a computation $\pi' = C'^0, step'^0, C'^1, step'^1, ...$ on Γ^A such that $\pi \preceq_s \pi'$.
Base: Given $C^0 = (c_1, ..., c_n, q_1, ..., q_m, id_1, ..., id_m, \lambda)$, the initial configuration of π. We define the following initial configuration for π': $C'^0 = (c'_1, ..., c'_n, q'_1, ..., q'_m, id'_1, ..., id'_m, \lambda')$ and show that it is an initial configuration on Γ^A.

- For every $i \in \{1, ..., n\}$ c'_i is defined as follows. For every $s \in c_i$:
  - If $s \in S^A_i$ then $s \in c'_i$ (if state s from SM_i exists also in SM^A_i, then it is part of c'_i)
  - If $s \notin S^A_i$ and $s \in \mathcal{A}^{\beta}$ (i.e, s is part of abstraction set $\mathcal{A}^{\beta}$ of SM_i) then $a^{\beta}_{strt} \in c'_i$
- For every $1 \leq j \leq m$ $q'_j = q_j = \phi$ ($q_j = \phi$ since C^0 is an initial configuration)
- For every $1 \leq j \leq m$ $id'_j = id_j = 0$ ($id_j = 0$ since C^0 is an initial configuration)
- For every $v \in V$, $\lambda'(v) = \lambda(v)$
- For every $cg^{\beta}_i \in V^A$:

- If there exists $s \in c_i$ s.t. $s \in S(A^\beta)$ and there exists $t \in TR(A^\beta)$ s.t. $src(t) = s$, $trig(t) = \epsilon$ and $grd(t)(\lambda^0) = true$ then $cg_i^\beta = 1$.
- Otherwise, $cg_i^\beta = 0$.

The above captures the case that an abstracted transition on SM_i can be executed without consumption of an event, and without a modification of some variable effecting its guard. This situation can occur if the guard is *true* under the initial configuration. In this case we initialize the matching cg_i^β to 1.

Clearly, C'^0 is an initial configuration and also that $C^0 \preceq C'^0$

Step: We assume that for the first r steps of π: $C^0, step^0, C^1, step^1, ..., C^{r-1}, step^{r-1}, C^r$ $(r > 0)$ there exists a partial computation $\pi' = C'^0, step'^0, C'^1, step'^1, ..., C'^p$ over Γ^A s.t. there are two sequences of positive integers $0 = i_0 < i_1 < i_2 < ... < i_l = r$ and $0 = i'_0 < i'_1 < i'_2 < ... < i'_l = p$ and for every $0 \leq k < l$, $C^{i_k}, C^{(i_k)+1}, ..., C^{(i_{k+1})-1} \preceq C'^{i'_k}, C'^{(i'_k)+1}, ..., C'^{(i'_{k+1})-1}$, and also $C^r \preceq C'^p$.

We define the matching extension of π' based on $step^r$:

- $step^r = DISP(j, ev)$
  By definition, $id_j^r = 0$, $q_j^r \neq \phi$ and $top(q_j^r) = ev$. Since $C^r \preceq C'^p$, then $id_i^r = id_i'^p$ and $q_i^r = q_i'^p$ for every i. Therefore, $id_j'^p = 0$, $q_j'^p \neq \phi$ and $top(q_j'^p) = ev$ as well, and it is possible to make a step where $step'^p = DISP(j, ev)$ from C'^p.
  By definition of $DISP$ step, $C^{r+1} = (c_1^r, ..., c_n^r, q_1^r, ..., q_j^{r+1}, ..., q_m^r, id_1^r, ..., id_j^{r+1}, ...id_m^r, \lambda^r)$, $q_j^{r+1} = pop(q_j^r)$, $id_j^{r+1} = trgt(ev)$ and $\rho_{trgt(ev)}^{r+1} = type(ev)$.
  By definition of $DISP$ step, $C'^{p+1} = (c_1'^p, ..., c_n'^p, q_1'^p, ..., q_j'^{p+1}, ..., q_m'^p, id_1'^p, ..., id_j'^{p+1}, ...id_m'^p, \lambda'^p)$, $q_j'^{p+1} = pop(q_j'^p)$, $id_j'^{p+1} = trgt(ev)$ and $\rho_{trgt(ev)}'^{p+1} = type(ev)$.
  Since $C^r \preceq C'^p$, it is clear that $C^{r+1} \preceq C'^{p+1}$ as well.

- $step^r = EndRTC(j, \epsilon)$
  Assume $id_j^r = id_j'^p = l > 0$. Since $stable(c_l^r, C^r)$, then for every $t \in TR_l$ s.t. $src(t) \in \omega_l^r$ either $trig(t) \neq \rho_l^r$ or $grd(t)(\lambda^r) = false$.
  For every $s \in \omega_l'^p$:

1. If $s \in \{a^{\beta}_{1}, ..., a^{\beta}_{f_{\beta}+1}, a^{\beta}_{end}\}$: If $\rho'^{p}_{l} \neq \epsilon$, then by definition of the semantics there exists $step^{p'}$ is π' such that $p' < p$ and $step'^{p'} = DISP(j, ev)$ where $id'^{p'}_{j} = l$ (an $EndRTC$ step must appear after a matching $DISP$ step). Also, for every $p' < p'' < p$, if $step'^{p''} = TRANS(j, (t_1, ..., t_q))$ then $id'^{p''}_{j} \neq l$ (otherwise, the $TRANS$ step would have set the ρ element to ϵ). This means that $s \in \omega'^{p'}_{l}$ (since only $TRANS$ steps can change the current state of a state machine). However, it is not possible that $s \in \omega'^{p'}_{l}$, since there exists a null transition from s. We therefore conclude that $\rho'^{p}_{l} = \epsilon$.
   For $s \in \{a^{\beta}_{1}, ..., a^{\beta}_{f_{\beta}+1}\}$ ($s = a^{\beta}_{end}$): There exists an enabled transition t' s.t. $trgt(t') = a^{\beta}_{end}$ (a^{β}_{strt}) (this is a null transition). We define $step'^{p} = TRANS(j, (t'))$. Clearly, if $C^{r}_{l} \preceq C'^{p}_{l}$, then also $C^{r}_{l} \preceq C'^{p+1}_{l}$. Note that we match C'^{p+1} to C^{r} and not to C^{r+1}. We prove stuttering simulation, and this step of π' is part of the matching interval, continuing in the handling of a^{β}_{end} (a^{β}_{strt}).
2. If $s = a^{\beta}_{strt}$ then for every t' s.t. $src(t') = a^{\beta}_{strt}$, by construction of SM^{A}_{i}, $grd(t')(\lambda) = \perp$ (for any λ), since $\perp$ is included in $grd(t')$.
3. Otherwise, since $c^{r}_{l} \preceq c'^{p}_{l}$, then $s \in \omega^{r}_{l}$. By definition of SM^{A}_{l}, for every transition $t \in TR_l$ s.t. $src(t) = s$ there exists a transition $t_a \in TR^{A}_{l}$ s.t. $src(t_a) = s$, $trig(t) = trig(t_a)$ and $grd(t) = grd(t_a)$. Thus, if $trig(t) \neq \rho^{r}_{l}$, then $trig(t_a) \neq \rho'^{p}_{l}$, and if $grd(t)(\lambda^{r}) = false$ then $grd(t_a)(\lambda'^{p}) \in \{false, \perp\}$.

The above means that for cases (2) and (3), for every $t_a \in TR^{A}_{l}$ s.t. $src(t_a) \in \omega'^{p}_{l}$ either $trig(t_a) \neq \rho'^{p}_{l}$ or $grd(t_a)(\lambda'^{p}) \in \{false, \perp\}$. Therefore, an $EndRTC$ step is possible s.t. $step'^{p} = EndRTC(j, \epsilon)$, and clearly $C^{r+1} \preceq C'^{p+1}$.

- $step^{r} = ENV(j, ev)$.
  Since the environment is always enabled, then an ENV step s.t. $step'^{p} = ENV(j, ev)$ is possible from C'^{p}, and clearly $C^{r+1} \preceq C'^{p+1}$.

- $step^{r} = TRANS(j, \{t_1, ..., t_q\})$.
  Assume $id^{r}_{j} = id'^{p}_{j} = l$. By definition of the semantics, $\rho^{r}_{l} = \rho'^{p}_{l} \neq \epsilon$. Thus, there exists $r' < r$ s.t. $step^{r'} = DISP(j, ev)$ and $id^{r'}_{j} = l$ (only

$DISP$ steps can set ρ to a value not ϵ) and there are no $TRANS$ or $EndRTC$ steps on thread j between $step^{r'}$ and $step^{r}$ (since these steps set the value of ρ to ϵ). From lemma 5.9 we know that there exists a matching step $step'^{p'} = DISP(j, ev)$ where $id_j'^{p'} = l$. From lemma 5.9 we also know that there are no $TRANS$ or $EndRTC$ steps on thread j between $step'^{p'}$ and $step'^{p}$ (if there was such a $TRANS$ step, then it had to be the first step in the RTC, in which case it had to have a matching step on π between $step^{r'}$ and $step^{r}$).

From lemma 5.12 we know that none of $a_1^{\beta}, ...a_{f_\beta+1}^{\beta}, a_{end}^{\beta}$ are in $\omega_l'^{p'}$. Since only $TRANS$ steps on thread j can change the current state of state machine SM_l^A, then we can conclude that none of $a_1^{\beta}, ...a_{f_\beta+1}^{\beta}, a_{end}^{\beta}$ are in $\omega_l'^{p}$.

For every $i \in \{1, ..., q\}$ we match transition t_i with a transition $t_i^a \in TR_l^A$. Assume $s = src(t_i)$.

1. If $src(t_i) \in \omega_l'^{p}$ and for every $\mathcal{A}^{\beta} \in ABS_l$, $a_{strt}^{\beta} \not\lhd s$: For every $s' \in \omega_l'^{p}$ s.t. $s' \lhd s$, and for every $t_a' \in TR_l^A$ s.t. $src(t_a') = s'$, since s does not contain any abstracted state then $s' \in \omega_l^{r}$. Consider the transition $t' \in TR_l$ that matches t_a': $trig(t') = trig(t_a')$ and $grd(t') = grd(t_a')$. Since $enabled(t_i, C^r) = true$ and $src(t') \lhd src(t_i)$ then $enabled(t', C^r) = false$. Thus either $trig(t') \neq \rho_l^r$ or $grd(t')(\lambda_l^r) = false$. Since $\rho_l'^{p} = \rho_l^r$ then $trig(t_a') \neq \rho_l'^{p}$ as well. Since $\lambda^r \preceq \lambda'^{p}$ then $grd(t_a')(\lambda'^{p}) \in \{false, \perp\}$. Therefore, we conclude that $enabled(t_a', C'^{p}) \in \{false, \perp\}$.

   By definition of the abstraction, there exists $t_i^a \in TR_l^A$ that matches t_i. Thus, $src(t_i^a) = s$, $trig(t_i^a) = trig(t_i)$ and $grd(t_i^a) = grd(t_i)$. For similar reasons, since $trig(t_i) = \rho_l^r$ and $grd(t_i) = true$ then $trig(t_i^a) = \rho_l'^{p}$ and $grd(t_i^a) \in \{\perp, true\}$.
   Therefore, since $enabled(t_i, C^r) = true$ then $enabled(t_i^a, C'^{p}) \in \{\perp, true\}$.

   Since $trig(t_i) \neq \epsilon$, then $act(t_i) = skip$. By definition of the abstraction $act(t_i^a) = skip$ as well. Therefore, since $\lambda^r \preceq \lambda'^{p}$ then $act(t_i)(\lambda^r, C^r) \preceq act(t_i^a)(\lambda'^{p}, C'^{p})$

2. If $s \in \omega_l'^{p}$ and for some $\mathcal{A}^{\beta} \in ABS_l$, $a_{strt}^{\beta} \lhd s$: By definition of the abstraction, there exists a transition $t_i^a \in TR_l^A$ that matches t_i.

Thus, $src(t_i^a) = src(t_i)$, $trig(t_i^a) = trig(t_i)$ and $grd(t_i^a) = grd(t_i)$. For every transition $t_i'^a \in TR_l^A$ s.t. $src(t_i'^a) = s' \in \omega_l'^p$ and $s' \lhd s$:

(a) If $s' = a_{strt}^{\beta}$ then by definition of the abstraction structure $grd(t_i'^a) = \bot$ or $grd(t_i'^a) = grd\&\bot$. Therefore $enabled(t_i'^a, C'^p) = \bot$.

(b) If $s' \neq a_{strt}^{\beta}$ then by definition of the simulation, $s' \in \omega_l^r$, and by the definition of the abstraction there exists $t_i' \in TR_l$ that matches $t_i'^a$. Thus, $src(t_i') = src(t_i'^a)$, $trig(t_i') = trig(t_i'^a)$ and $grd(t_i') = grd(t_i'^a)$. Since $enabled(t_i', C^r) = false$ (otherwise t_i is not enabled), and since $C^r \preceq C'^p$ then $enabled(t_i'^a, C^p) \in \{false, \bot\}$ as well.

From the above we conclude that $enabled(t_i^a, C'^p) \in \{\bot, true\}$.

Since $trig(t_i) \neq \epsilon$, then $act(t_i) = skip$. By definition of the abstraction $modif(t_i^a) \cap V = \phi$ (since t_i^a may only modify cg variables). Therefore, if $\lambda \preceq \lambda'$ then $act(t_i)(\lambda, C^r) \preceq act(t_i^a)(\lambda', C'^p)$

3. If $s \notin \omega_l'^p$: Notice that if $s \notin \omega_l'^p$, and since $c_l^r \preceq c_l'^p$, then $s \notin S_l^A$. This means that the transition t_i taken is from an abstracted state s.

   (a) If for some $\mathcal{A}^{\beta} \in ABS_l$, $s \in S_l(\mathcal{A}^{\beta})$ and $trgt(t_i) \in S_l^A$: This means that the transition taken is from an abstracted state $s \in S_l(\mathcal{A}^{\beta})$ and the target is not an abstracted state.
   Since $enabled(t_i, C^r) = true$ and $\rho_l^r \neq \epsilon$, then $trig(t_i) \neq \epsilon$. By definition of the abstraction there exists a matching transition $t_i^a \in TR_l^A$. Thus, $src(t_i^a) = a_{strt}^{\beta}$, $trig(t_i^a) = trig(t_i)$, $grd(t_i^a) = grd(t_i)\&\bot$, and $trgt(t_i^a) = trgt(t_i)$. Since $C^r \preceq C'^p$, then if $enabled(t_i, C^r) = true$ then $enabled(t_i^a, C'^p) = \bot$. Since $trig(t_i) \neq \epsilon$, then $act(t_i) = skip$. By definition of the abstraction $modif(t_i^a) = \{cg_l^{\beta}\}$. Therefore, if $\lambda \preceq \lambda'$ then $act(t_i)(\lambda, C^r) \preceq act(t_i^a)(\lambda', C'^p)$

   (b) If there exist $\mathcal{A}^{\beta}, \mathcal{A}^{\gamma} \in ABS_l$ $(\beta \neq \gamma)$ where $s \in S_l(\mathcal{A}^{\beta})$ and $trgt(t_i) \in S_l(\mathcal{A}^{\gamma})$: This means that the transition taken is from an abstracted state $s \in S_l(\mathcal{A}^{\beta})$ and the target is an abstracted state $s' \in S_l(\mathcal{A}^{\gamma})$.
   Since $enabled(t_i, C^r) = true$ and $\rho_l^r \neq \epsilon$, then $trig(t_i) \neq \epsilon$. By definition of the abstraction there exists a transition

$t_i^a \in TR_l^A$ s.t. $src(t_i^a) = a_{strt}^{\beta}$, $trig(t_i^a) = trig(t_i)$, $grd(t_i^a) = grd(t_i)\&\bot$, and $trgt(t_i^a) = a_{strt}^{\gamma}$. Since $C^r \preceq C'^p$, then if $enabled(t_i, C^r) = true$ then $enabled(t_i^a, C'^p) = \bot$.
Since $trig(t_i) \neq \epsilon$, then $act(t_i) = skip$. By definition of the abstraction $modif(t_i^a) = \{cg_l^{\beta}, cg_l^{\gamma}\}$. Therefore, if $\lambda \preceq \lambda'$ then $act(t_i)(\lambda, C^r) \preceq act(t_i^a)(\lambda', C'^p)$

(c) Otherwise (for some $\mathcal{A}^{\beta} \in ABS_l\ s, trgt(t_i) \in S_l(\mathcal{A}^{\beta})$): This means that the the transition t_i taken is a transition within the abstraction of $\mathcal{A}^{\beta}$. Since $enabled(t_i, C^r) = true$ and $\rho_l^r \neq \epsilon$, then $trig(t_i) \neq \epsilon$. By definition of the abstraction, this means that $\rho_l^r \in Trig(\mathcal{A}^{\beta})$. By the definition of the abstraction, $modif(t_i) \subseteq V(\mathcal{A}^{\beta})$. Consider transition $\tau_1^{\beta} \in \Delta(\mathcal{A}^{\beta})$. It holds that $enabled(\tau_1^{\beta}, C'^p) = \bot$. We define $t_i^a = \tau_1^{\beta}$.
Since $trig(t_i) \neq \epsilon$, then $act(t_i) = skip$. By definition of the abstraction, for every $v \in V(\mathcal{A}^{\beta})$, $act(t_1^{\beta})(\lambda', C'^p)(v) = \bot$. Therefore, we can conclude that if $\lambda \preceq \lambda'$ then $act(t_i)(\lambda, C^r) \preceq act(t_a^i)(\lambda', C'^p)$.

Note that there can be *several* transitions from $(t_1, ..., t_y)$ that are abstracted by a *single* τ_1^{β}. A single occurrence of τ_1^{β} replaces all of these transitions.

We define $step'^p = TRANS(j, (t_1^a, ..., t_{y'}^a))$: If there are several transitions $t_{i_1}, ..., t_{i_d}$ executed in the current $TRANS$ step for which $t_{i_j}^a = \tau_1^{\beta}$ (for a specific $\mathcal{A}^{\beta} \in ABS_l$), then τ_1^{β} replaces the first occurrence of abstracted transition in $(t_1, ..., t_q)$. Since the first execution of τ_1^{β} sets all variables in $V(\mathcal{A}^{\beta})$ to $\bot$, then the effect of executing only the first occurrence of τ_1^{β} λ^p is the same as executing several occurrences of τ_1^{β}. Since for every $t \in \{t_1, ..., t_q\}$, $trig(t') \neq \epsilon$ then $act(t) = skip$, then this step does not change the event queues. We can conclude that $C^{r+1} \preceq C'^{p+1}$.

- $step^r = TRANS(j, (t))$.
Assume $id_j^r = id_j'^p = l$, and $src(t) = s$. By definition of the semantics, $\rho_l^r = \rho_l'^p = \epsilon$. We want to show that if $enabled(t, C^r)$, then there exists $t' \in TR_l'$ s.t. $enabled(t', C'^p)$. We then define $step'^p = TRANS(j, (t'))$

and show that $C^{r+1} \preceq C'^{p+1}$. We separate the proof to the different cases described below.

1. For every $\mathcal{A}^\beta \in ABS_l$, $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p = \phi$

Otherwise (for some $\mathcal{A}^\beta \in ABS_l$, $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p \neq \phi$), and:

2. $s \in \omega_l'^p$ and for every $\mathcal{A}^\beta \in ABS_l$, $a^\beta_{strt} \not\lhd s$)
3. $s \in \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, and for some $i \in \{1, ..., f_\beta + 1\}$: $a^\beta_i \in \omega_l'^p$, and $a^\beta_i \lhd s$
4. $s \in \omega_l'^p$, and for some $\mathcal{A}^\beta \in ABS_l$, $a^\beta_{end} \in \omega_l'^p$ and $a^\beta_{end} \lhd s$
5. $s \in \omega_l'^p$, for every $\mathcal{A}^\beta \in ABS_l$, if $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p \neq \phi$ and $a^\beta_{strt} \lhd s$ then $a^\beta_{strt} \in \omega_l'^p$ (possibly more than one such β)
6. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, for some $i \in \{1, ..., f_\beta + 1\}$, $a^\beta_i \in \omega_l'^p$, and $trgt(t) \notin S_l(\mathcal{A}^\beta)$
7. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a^\beta_{end} \in \omega_l'^p$, and $trgt(t) \notin S_l(\mathcal{A}^\beta)$
8. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a^\beta_{strt} \in \omega_l'^p$, and $trgt(t) \notin S_l(\mathcal{A}^\beta)$
9. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a^\beta_{strt} \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^\beta)$
10. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, for some $i \in \{1, ..., f_\beta\}$: $a^\beta_i \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^\beta)$
11. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a^\beta_{f_\beta+1} \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^\beta)$
12. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a^\beta_{end} \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^\beta)$

1. For every $\mathcal{A}^\beta \in ABS_l$, $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p = \phi$: This means that the current state of SM^A_l does not include any abstraction.
   Since $c^r_l \preceq c'^p_l$ then $\omega_l'^p = \omega^r_l$. $enabled(t, C^r) = true$, therefore for every $s' \in \omega^r_l$ s.t. $s' \lhd s$, and for every $t' \in TR_l$ s.t. $src(t') = s'$, either $trig(t') \neq \rho^r_l$ or $grd(t')(\lambda) = false$.
   For every $s' \in \omega_l'^p$ where $s' \lhd s$, it holds that $s' \in \omega^r_l$ as well (since $\omega_l'^p = \omega^r_l$). By definition of the abstraction, for every $t'_a \in TR^A_l$

s.t. $src(t'_a) = s'$ there exists a matching $t' \in TR_l$ where $src(t') = s'$. Also, by definition of the abstraction $trig(t'_a) = trig(t')$ and $grd(t'_a) = grd(t')$. Since $\rho_l^r = \rho_l'^p$ and $\lambda^r \preceq \lambda'^p$, then either $trig(t'_a) \neq \rho_l'^p$ or $grd(t'_a)(\lambda'^p) \in \{false, \bot\}$.
By definition of the abstraction, there exists a matching $t_a \in TR_l^A$ s.t. $src(t_a) = s$, $trig(t_a) = trig(t)$ and $grd(t_a) = grd(t)$. For similar reasons, since $trig(t) = \rho_l^r$ and $grd(t) = true$ then $trig(t_a) = \rho_l'^p$ and $grd(t_a) \in \{\bot, true\}$.
Therefore, since $enabled(t, C^r)$ then also $enabled(t_a, C'^p)$.
Thus, $step^p = TRANS(j, (t_a))$ is possible from C'^p.
By definition of the abstraction, $act(t_a)$ either equals to $act(t)$, or equals to $act(t)$ with addition of manipulation of cg variables. Thus, by lemma 5.7, and since $\lambda^r \preceq \lambda'^p$, $act(t)(\lambda^r, C^r) \preceq act(t_a)(\lambda'^p, C'^p)$. We can then conclude that $C^{r+1} \preceq C'^{p+1}$.

For some $\mathcal{A}^\beta \in ABS_l$, $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p \neq \phi$, and:

2. $s \in \omega_l'^p$ and for every $\mathcal{A}^\beta \in ABS_l$, $a^\beta_{strt} \not\lhd s$): This means that the current state of SM_l^A includes the abstraction, and the transition taken is from a state s which is in an orthogonal region to each of the abstractions. For the same reasoning as in the previous item, there exists $t_a \in TR_l^A$ s.t. $src(t_a) = s$ and $enabled(t_a, C'^p) \in \{\bot, true\}$.

   By definition of the abstraction, $act(t_a)$ either equals to $act(t)$, or equals to $act(t)$ with addition of manipulation of cg variables. Thus, by lemma 5.7, and since $\lambda^r \preceq \lambda'^p$, $act(t)(\lambda^r, C^r) \preceq act(t_a)(\lambda'^p, C'^p)$. We can then conclude that $C^{r+1} \preceq C'^{p+1}$.

3. For some $\mathcal{A}^\beta \in ABS_l$, and for some $i \in \{1, ..., f_\beta + 1\}$: $a_i^\beta \in \omega_l'^p$, $s \in \omega_l'^p$ and $a_i^\beta \lhd s$: This means that the current state of SM_l^A includes the abstraction, and the transition taken is from a state s which includes the abstraction (possibly more than one abstraction). By definition of the abstraction, there exists a transition $t_a \in TR_l^A$ s.t. $src(t_a) = a_i^\beta$, $trig(t_a) = \epsilon$, $grd(t_a) = true$, and $trgt(t_a) = a^\beta_{end}$. This means that $enabled(t_a, C'^p)$.
   We define $step'^p = TRANS(j, (t_a))$ and clearly $C^r \preceq C'^{p+1}$.
   Note that we match C'^{p+1} to C^r and not to C^{r+1}. We prove

stuttering simulation, and this step of π' is part of the matching interval, continuing in the handling of a^{β}_{end}.

4. For some $\mathcal{A}^{\beta} \in ABS_l$, $a^{\beta}_{end} \in \omega'^{p}_{l}$, $s \in \omega'^{p}_{l}$ and $a^{\beta}_{end} \lhd s$: This means that the current state of SM^{A}_{l} includes the abstraction, and the transition taken is from a state s which includes the abstraction. By definition of the abstraction, there exists a transition $t_a \in TR^{A}_{l}$ s.t. $src(t_a) = a^{\beta}_{end}$, $trig(t_a) = \epsilon$, $grd(t_a) = true$, and $trgt(t_a) = a^{\beta}_{strt}$. This means that $enabled(t_a, C'^{p})$.
   We define $step'^{p} = TRANS(j, (t_a))$ and clearly $C^{r} \preceq C'^{p+1}$.
   Note that we match C'^{p+1} to C^{r} and not to C^{r+1}. We prove stuttering simulation, and this step of π' is part of the matching interval, continuing in the handling of a^{β}_{strt}.

5. $s \in \omega'^{p}_{l}$, for every $\mathcal{A}^{\beta} \in ABS_l$, if $\Delta(\mathcal{A}^{\beta}) \cap \omega'^{p}_{l} \neq \phi$ and $a^{\beta}_{strt} \lhd s$ then $a^{\beta}_{strt} \in \omega'^{p}_{l}$ (there can possibly be more than one such β): This means that the current state of SM^{A}_{l} includes the abstraction, and the transition taken is from a state s which includes the abstraction. By definition of the abstraction, there exists a transition $t_a \in TR^{A}_{l}$ s.t. $src(t_a) = s$, $trig(t_a) = trig(t)$ and $grd(t_a) = grd(t)$.
   For every transition $t'_a \in TR^{A}_{l}$ s.t. $src(t'_a) = s' \in \omega'^{p}_{l}$ and $s' \lhd s$:
   (a) If $s' = a^{\beta}_{strt}$ then by definition of the abstraction structure, $grd(t'_a) = \perp$ or $grd(t'_a) = grd\&\perp$. Therefore $enabled(t'_a, C'^{p}) = \perp$.
   (b) If $s' \neq a^{\beta}_{strt}$ then $s' \in \omega^{r}_{l}$. By definition of the abstraction there exists $t' \in TR_l$ s.t. $src(t') = s'$, $trig(t') = trig(t'_a)$ and $grd(t') = grd(t'_a)$. Since $enabled(t', C^{r}) = false$ (otherwise t is not enabled), and since $C^{r} \preceq C'^{p}$ then $enabled(t'_a, C^{p}) \in \{false, \perp\}$ as well.

   From the above we conclude that $enabled(t_a, C'^{p}) \in \{\perp, true\}$. We define $step'^{p} = TRANS(j, (t_a))$ and $C^{r+1} \preceq C'^{p+1}$ (reasoning regarding the correctness w.r.t. the value of the variables is similar to the previous case).

6. $s \notin \omega'^{p}_{l}$, for some $\mathcal{A}^{\beta} \in ABS_l$, $s \in S_l(\mathcal{A}^{\beta})$, for some $i \in \{1, ..., f_{\beta}+1\}$, $a^{\beta}_{i} \in \omega'^{p}_{l}$, and $trgt(t) \notin S_l(\mathcal{A}^{\beta})$: Notice that if $s \notin \omega'^{p}_{l}$, and since $c^{r}_{l} \preceq c'^{t}_{l}$, then $s \notin S^{A}_{l}$, and thus for some $\mathcal{A}^{\beta} \in ABS_l$,

$s \in S_l(\mathcal{A}^\beta)$. This means that the current state of SM_l^A includes the abstraction, the transition taken is from a state s which is abstracted by $\Delta(\mathcal{A}^\beta)$, and the target is not a state abstracted by $\Delta(\mathcal{A}^\beta)$. By definition of the abstraction, there exists a transition $t_a \in TR_l^A$ s.t. $src(t_a) = a_i^\beta$, $trig(t_a) = \epsilon$, $grd(t_a) = true$, and $trgt(t_a) = a_{end}^\beta$. This means that $enabled(t_a, C'^t)$.
We define $step'^t = TRANS(j, (t_a))$ and clearly $C^r \preceq C'^{p+1}$.
Note that we match C'^{p+1} to C^r and not to C^{r+1}. We prove stuttering simulation, and this step of π' is part of the matching interval, continuing in the handling of a_{end}^β.

7. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a_{end}^\beta \in \omega_l'^p$, and $trgt(t) \notin S_l(\mathcal{A}^\beta)$: Notice that if $s \notin \omega_l'^p$, and since $c_l^r \preceq c_l'^p$, then $s \notin S_l^A$, and thus for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$. This means that the current state of SM_l^A includes the abstraction, the transition taken is from a state s which is abstracted by $\Delta(\mathcal{A}^\beta)$, and the target is not a state abstracted by $\Delta(\mathcal{A}^\beta)$.
By definition of the abstraction, there exists a transition $t_a \in TR_l^A$ s.t. $src(t_a) = a_{end}^\beta$, $trig(t_a) = \epsilon$, $grd(t_a) = true$, and $trgt(t_a) = a_{strt}^\beta$. This means that $enabled(t_a, C'^p)$.
We define $step'^p = TRANS(j, (t_a))$ and clearly $C^r \preceq C'^{p+1}$.
Note that we match C'^{p+1} to C^r and not to C^{r+1}. We prove stuttering simulation, and this step of π' is part of the matching interval, continuing in the handling of a_{strt}^β.

8. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a_{strt}^\beta \in \omega_l'^p$, and $trgt(t) \notin S_l(\mathcal{A}^\beta)$: Notice that if $s \notin \omega_l'^p$, and since $c_l^r \preceq c_l'^p$, then $s \notin S_l^A$, and thus for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$. This means that the current state of SM_l^A includes the abstraction, the transition taken is from a state s which is abstracted by $\Delta(\mathcal{A}^\beta)$, and the target is not a state abstracted by $\Delta(\mathcal{A}^\beta)$.
By definition of the abstraction there exists a matching transition $t_a \in TR_l^A$ s.t. $src(t_a) = a_{strt}^\beta$, $trig(t_a) = trig(t)$, $grd(t_a) = grd(t)\&\bot$. Since $C^r \preceq C'^p$, then if $enabled(t, C^r) = true$ then $enabled(t_a, C'^p) = \bot$. We define $step^p = TRANS(j, (t_a))$.
By definition of the abstraction, $act(t_a)$ equals to $act(t)$ with addition of manipulation of cg variables. Thus, based on lemma 5.7, and since $\lambda^r \preceq \lambda'^p$, $act(t)(\lambda^r, C^r) \preceq act(t_a)(\lambda'^p, C'^p)$.

By definition of the abstraction either $trgt(t_a) = trgt(t)$ (if $trgt(t) \in S_l^A$) or $trgt(t_a) = a^{\gamma}_{strt}$ for $\mathcal{A}^{\gamma} \in ABS_l$ and $\gamma \neq \beta$ (if $trgt(t) \in S_l(\mathcal{A}^{\gamma})$).
We conclude that $C^{r+1} \preceq C'^{p+1}$.

The following lemma is a result of the above items. The lemma states that if a transition t was taken in Γ during some computation π, and that transition is not abstracted by a single abstraction construct $\Delta(\mathcal{A}^{\beta})$, then the matching transition t_a is also taken in the matching interval in the matching computation π' of Γ^A.

Lemma 5.13 *Let $C^r \in \pi$ be a configuration s.t. $step^r = TRANS(j,(t_1,...,t_q))$ (possibly $q = 1$) and $id^r_j = l$. For every transition $t \in \{t_1,...,t_q\}$ s.t. there is no $\mathcal{A}^{\beta} \in ABS_l$ for which $src(t), trgt(t) \in S_l(\mathcal{A}^{\beta})$:*

Assume $C^r \preceq C'^p$ and $C^{r+1} \preceq C'^{p+1}$. Then $step'^p = TRANS(j,(t^a_1,...,t^a_{q'}))$ and there exists $t^a \in \{t^a_1,...,t^a_{q'}\}$ s.t. t_a is the matching transition of t. Thus the following properties hold:

- *If $src(t) \in S_l^A$ then $src(t_a) = src(t)$. Otherwise ($src(t) \in S_l(\mathcal{A}^{\beta})$ for some β) then $src(t_a) = a^{\beta}_{strt}$.*
- *If $trgt(t) \in S_l^A$ then $trgt(t_a) = trgt(t)$. Otherwise ($trgt(t) \in S_l(\mathcal{A}^{\beta})$ for some β) then $trgt(t_a) = a^{\beta}_{strt}$.*
- *$trig(t_a) = trig(t)$*
- *either $grd(t_a) = grd(t)$ or $grd(t_a) = grd(t)\&\bot$*
- *$act(t_a)$ includes $act(t)$ with possible additional manipulation of cg variables.*

We will further need the following definition: Given some active state s in a state machine SM_l, we define the previous step in a computation which caused reaching to state s.

Definition 5.14 *Assume some computation $\pi = C^0, step^0, C^1, step^1, ...$ over a system Γ, a configuration $C^r \in \pi$, and a state $s \in \omega^r_l$. We define the* previous configuration *leading to s as $prev(\pi, r, l, s) = C^{r'}$ where $r' < r$ if the following hold:*

1. For every $r' < r'' < r$: $s \in \omega_l^{r''}$

2. $step^{r'} = TRANS(j, (t_1, ..., t_q))$ where $id_j^{r'} = l$, and there exists $i \in \{1, ..., q\}$ s.t. $s \leq trgt(t_i)$

3. For every $r' < r'' < r$: if $step^{r''} = TRANS(j, (t_1, ..., t_q))$ and $id_j^{r''} = l$, then for every $i \in \{1, ..., q\}$: $s \not\leq src(t_i)$

If no such $C^{r'}$ exists then $prev(\pi, r, l, s) = \epsilon$

Notice that when $prev(\pi, r, l, s) = C^{r'}$ is not ϵ, then we assure that every transition of state machine SM_l that was executed between $step^{r'}$ and $step^r$ was either in an orthogonal region to s or in a state inside s. The execution of $step^{r'}$ caused the current state of SM_l to move to state s, since the target of one of the transitions executed in $step^{r'}$ is either s or a state containing s.

9. $s \notin \omega_l'^p$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, $a_{strt}^\beta \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^\beta)$: Notice that if $s \notin \omega_l'^p$, and since $c_l^r \preceq c_l'^p$, then $s \notin S_l^A$, and thus for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$. This means that the current state of SM_l^A includes the abstraction, the transition taken is from a state s which is abstracted by $\Delta(\mathcal{A}^\beta)$, and the target is a state abstracted by $\Delta(\mathcal{A}^\beta)$ as well. By definition of the current $TRANS$ step, $\rho_l^r = \epsilon$.

   The definition of the matching interval for this step includes a small induction proof which states the following: For every configuration $C^r \in \pi$ s.t. $step^r = TRANS(j, (t_1, ..., t_q))$, and a matching abstract configuration C'^p s.t. $C^r \preceq C'^p$: If for some $t \in \{t_1, ..., t_q\}$ and for some $\beta \in \{1, ..., g\}$, $a_{strt}^\beta \in \omega_l'^p$, and $src(t), trgt(t) \in S_l(\mathcal{A}^\beta)$, then $step'^p = TRANS(j, (t_1^a, ..., t_{q'}^a))$, and there exists $t^a \in \{t_1^a, ..., t_{q'}^a\}$ s.t. $C^{r+1} \preceq C'^{p+1}$, $src(t^a) = a_{strt}^\beta$, and $trgt(t^a) = a_1^\beta$.

   For the case where $q > 1$ the property holds by the definition of matching interval for $TRANS$ step with multiple transitions. The case where $q = 1$ is handled in this item.

   This property can be added to the entire induction proof, but it is relevant only in this item, where we define the interval matching such a step. In the following we define the matching interval for $step^r$, and show that the matching interval terminates at abstracted state a^{β}_1.

Assume we can determine that $\lambda'^p(cg^{\beta}_l) \neq 0$. We can then define the matching interval as follows. It holds that $enabled(\tau^{\beta}_2, C'^p) = \perp$. We define $step'^p = TRANS(j, \tau^{\beta}_2)$, and differentiate according to $act(t)$:

(a) If $GEN(..) \notin act(t)$ then clearly $C^{r+1} \preceq C'^{p+1}$.

(b) If $GEN(ev') \in act(t)$ then $C^r \preceq C'^{p+1}$. We define stuttering inclusion, and this step is part of the matching interval, continuing in the handling of a^{β}_i.

The correctness of $C^{r+1} \preceq C'^{p+1}$ or $C^r \preceq C'^{p+1}$ w.r.t. the value of variables holds since by definition of the abstraction, all variables whose value might modify on $act(t)$ are set to $\perp$ on $act(\tau^{\beta}_2)$. For any other variable, since $C^r \preceq C'^p$ then the correct relation remains after $step^r$ and $step'^p$.

It now remains to be shown that indeed $\lambda'^p(cg^{\beta}_l) \neq 0$. We mark the the step representing the beginning of the current RTC step on thread j as $step^{r'} = DISP(j, ev)$. We denote the matching $DISP$ step on π' as $step'^{p'} = DISP(j, ev)$. Consider $prev(\pi, r, l, s)$. If $prev(\pi, r, l, s) = \epsilon$ then for every $\alpha \in \{0, ..., r\}$, $s \in \omega^{\alpha}_l$. Thus $s \in init_l$. By our requirement from abstraction sets, and since $trig(t) = \epsilon$, we know that $grd(t) \neq true$. Since $enabled(t, C^r)$, we know that $grd(t)(\lambda^r) = true$. We denote by r'' s.t. $r'' < r$ the maximal index for which $grd(t)(\lambda^{r''}) = false$ and $grd(t)(\lambda^{r''+1}) = true$.

- If $r'' = 0$: this means that $grd(t)(\lambda^{\alpha}) = true$ for every $\alpha \in \{0, ..., r\}$. Assume there exists $t' \in \Delta(\mathcal{A}^{\beta})$ that was executed in some step r''' prior to the current step. By the property resented at the beginning of this item, this step did not occur in the current RTC step (otherwise, the interval matching that execution should traverse from a^{β}_{strt} to a^{β}_1). Consider the $EndRTC$ step on thread j that followed $step^{r'''}$, denoted $step^{r^*} = EndRTC(j, \epsilon)$. Since a transition executed on that RTC step, then $\rho^{r^*}_l = \epsilon$. Also, $grd(t)(\lambda^{r^*}) = true$. Since $s \in \omega^{r^*}_l$, then clearly $enabled(t, C^{r^*})$. This is in contradiction

to the fact that an *EndRTC* step occurred. We can then conclude that no transition $t' \in \Delta(\mathcal{A}^{\beta})$ has been executed prior to the current step.
The above means that for every $\alpha' \in \{0, ..., p\}$, $a^{\beta}_{strt} \in \omega'^{\alpha'}_{l}$. Since $grd(t)(\lambda^0) = true$, $trig(t) = \epsilon$ and $s \in c^0_l$, then $\lambda'^0(cg^{\beta}_l) = 1$ (by the base definition). The only transition that can change the value of cg^{β}_l to 0 is a transition from a^{β}_{strt} to a^{β}_1, and we know such transition was not taken. Thus we can conclude that $\lambda'^p(cg^{\beta}_l) \neq 0$.

– If $r'' > 0$. Assume $step^{r''}$ is executed on state machine $SM_{l'}$ (possibly $l' = l$). We first notice if $grd(t)(\lambda^{r''}) = false$ and $grd(t)(\lambda^{r''+1}) = true$, then $step^{r''} = TRANS(j', (t'_1, ..., t'_y))$ and for some $t' \in \{t'_1, ..., t'_h\}$, $v \in modif(t')$. Assume $id^{r''}_{j'} = l'$ ($SM_{l'}$ executes $step^{r''}$). By the definition of abstraction of systems, this means that $thread(l') = thread(l)$, thus $j' = j$. We separate between two cases, based on the relation between r'' and r':

(a) If $r'' < r'$: We show that for every $\alpha \in \{r'', ..., r'\}$, SM_l did not execute any transition in $step^{\alpha}$. Assume, by contradiction, that such a transition did execute. Consider the *EndRTC* step on thread j that followed $step^{\alpha}$ (denoted $step^{r'''}$). Since a $TRANS$ step occurred in this RTC step, then $\rho^{r'''}_l = \epsilon$. We also know that $grd(t)(\lambda^{r'''}) = true$. Since $s \in \omega^{r'''}_l$, then clearly $enabled(t, C^{r^*})$. This is in contradiction to the fact that an *EndRTC* step occurred in $step^{r'''}$. We can then conclude that $l' \neq l$. Since SM_l and $SM_{l'}$ are on the same thread, then they cannot both execute a RTC step simultaneously.
Denote the beginning of the RTC step that ended on $step^{r'''}$ with $step^{r^*} = DISP(j, e)$. Denote the step on π' that matches $step^{r^*}$ as $step'^{p^*} = DISP(j, e)$. From the above, we can conclude that for every $\alpha' \in \{p^*, ..., p'\}$, $a^{\beta}_{strt} \in \omega'^{\alpha'}_l$. Moreover, from the property presented in the beginning of the item we can conclude that for every $\alpha' \in \{p^*, ..., p\}$, $a^{\beta}_{strt} \in \omega'^{\alpha'}_l$.
Since for some $v \in GRDV(A^{\beta})$, $v \in modif(t')$, then we

know that for some $\beta \in \{p^*, ..., p'\}$, cg_l^β was set to 1 on $step'^\beta$ (If t' has a matching transition t'_a, then during the execution of t'_a (by lemma 5.13). If t' is abstracted by $\Delta(\mathcal{A}^\gamma)$, then during execution of transition from a^γ_{strt} to a^γ_1 that occurred due to the property in the beginning of this item).
The only transition that can set cg_l^β to 0 is a transition from a^β_{strt} to a^β_1. Since we know that for every $\alpha' \in \{p^*, ..., p\}$, $a^\beta_{strt} \in \omega_l'^{\alpha'}$, then no such transition executed. We can then conclude that $\lambda^p(cg_l^\beta) \neq 0$.

(b) If $r'' > r'$: This means that $l' = l$ (no other SM on thread j can execute between $step^{r'}$ and $step^r$). We know that $a^\beta_{strt} \in \omega_l'^p$. By lemma 5.16 and by the property from the beginning of this item we can conclude that for every $\alpha' \in \{p', ..., p\}$, $a^\beta_{strt} \in \omega_l'^{\alpha'}$.
Since for some $v \in GRDV(A^\beta)$, $v \in modif(t')$, then we know that for some $\xi \in \{p', ..., p\}$, cg_l^β was set to 1 on $step'^\xi$ (If t' has a matching transition t'_a, then during the execution of t'_a (by lemma 5.13). If t' is abstracted by $\Delta(\mathcal{A}^\gamma)$, then during execution of transition from a^γ_{strt} to a^γ_1 that occurred due to the property in the beginning of this item).
The only transition that can set cg_l^β to 0 is a transition from a^β_{strt} to a^β_1. Since we know that for every $\alpha' \in \{p^*, ..., p\}$, $a^\beta_{strt} \in \omega_l'^{\alpha'}$, then no such transition executed. We can then conclude that $\lambda^p(cg_l^\beta) \neq 0$.

If $prev(\pi, r, l, s) = C^{r''} \neq \epsilon$. By definition of $prev$, $step^{r''} = TRANS(j, (t_1, ..., t_q))$ s.t. for some $t' \in \{t_1, ..., t_q\}$, $s \lhd trgt(t')$. We separate between two cases, and show that under both cases $\lambda'^p(cg^\beta) \neq 0$:

(a) $r'' > r'$: This means that the execution of the transition leading to s occurred after the beginning of the RTC step. We first show that it is not possible that $t' \in \Delta(\mathcal{A}^\beta)$. Assume it is (i.e., both $src(t')$ and $trgt(t')$ are in $S_l(\mathcal{A}^\beta)$). Consider the first transition in $\Delta(\mathcal{A}^\beta)$ that was executed in the current RTC step. By the property presented in the beginning of this item, that transition caused a traversal from a^k_{strt} to a^k_1. By lemma 5.16 and since $a^\beta_{strt} \in \omega'^p_l$ we can conclude that no such abstracted transition was executed.

If $trgt(t') \notin S_(\mathcal{A}^\beta)$ (i.e., $trgt(t')$ contains s and is not abstracted): By definition of the abstraction, t' has a matching transition $t'_a \in TR^A_l$, and by lemma 5.13, t'_a was executed at $step'^{p''}$ that matches $step^{r''}$. Since we have no history and $s \in \omega^{r''+1}_l$, then $a^\beta_{strt} \in \omega'^{p''+1}_l$. By lemma 5.16 we can conclude that for every $\alpha' \in \{p'', ..., p\}$, $a^\beta_{strt} \in \omega'^{\alpha'}_l$. Since only transitions exiting a^β_{strt} can set cg^β_l to 0, then we can conclude that $\lambda^p(cg^\beta_l) \neq 0$.

If $trgt(t') \in S_l(\mathcal{A}^\beta)$ then $src(t') \notin S_l(\mathcal{A}^\beta)$. Since t' is not an abstracted transition, then there exists a matching transition $t'_a \in TR^A_l$. By lemma 5.13, there exists $step'^{p''}$ that matches $step^{r''}$ and t'_a is executed on $step'^{p''}$. By definition of the abstraction, $cg^\beta_l = 1 \in act(t'_a)$. Since $src(t') \notin S_(\mathcal{A}^\beta)$ then $src(t'_a) \notin \Delta(\mathcal{A}^\beta)$. Since there are no cross hierarchy transitions, then $a^\beta_{strt} \notin \omega'^{p''}_l$. The only transitions that can set cg^β_l to 0 are transitions exiting a^β_{strt}. We can then conclude that $\lambda'^{p''+1}(cg^\beta_l) \neq 0$.

We now show that if $\lambda'^{p''+1}(cg^\beta_l) \neq 0$ then $\lambda'^p(cg^\beta_l) \neq 0$ as well. The only transitions that can set cg^β_l to 0 are transitions from a^β_{strt} to a^β_1. We know that $a^\beta_{strt} \in \omega'^{p''+1}_l$ and $a^\beta_{strt} \in \omega'^p_l$. If we show that for every $\alpha' \in \{p''+1, ..., p\}$, $a^\beta_{strt} \in \omega'^{\alpha'}_l$ then clearly no such transition is taken and $\lambda'^p(cg^\beta_l) \neq 0$. Assume this is not the case, and that for some $\alpha' \in \{p''+1, ..., p-1\}$, $a^\beta_{strt} \notin \omega'^{\alpha'}_l$. By the definition of the abstraction, in order to return to a^β_{strt} the transition from a^β_{end} to a^β_{strt} had to be executed at some $step^\xi$ s.t. $\alpha' < \xi < p$. However, by lemma 5.16, the interval that includes $step'^\xi$ either leaves the abstraction or is part of an $EndRTC$ step. It is not possible that $step'^\xi$ is part of an $EndRTC$ step, since by lemma 5.9 there had to be a matching $EndRTC$ step in π, and we know that the last $EndRTC$ step on thread j occurred at $step^{r'}$, which is prior to $step^{r''}$ that matches $step^{p''}$. Also, we know that $prev(\pi, r, l, s) = C^{r''}$, thus for every $r'' < r^* \leq r$, $s \in$

$\omega_l^{r^*}$. Therefore, since we assume stuttering simulation on the prefix of the computations, then abstraction $\Delta(\mathcal{A}^\beta)$ must be part of the configuration for every $\omega_l'^{p^*}$, for $p'' + 1 < p^* \leq p$ and it is not possible that $step'^{\xi}$ leaves the abstraction. Thus, for every $p'' + 1 \leq \alpha' \leq p$, $a_{strt}^{\beta} \in \omega_l'^{\alpha'}$.
We conclude that $\lambda'^p(cg_l^\beta) \neq 0$.

(b) $r' > r''$: This means that the execution of the transition leading to s occurred before the beginning of the RTC step. By definition of $prev$, this means that for every $r'' < r^* \leq r$, $s \in \omega_l^{r^*}$. Since the transition execution ($step^{r''}$) occurred before the $DISP$ step ($step^{r'}$), then by the semantics there exists $r'' < r''' < r'$ s.t. $step^{r'''} = EndRTC(j, \epsilon)$ where $id_j^{r'''} = l$. Consider the last such $EndRTC$ step on state machine SM_l (meaning: for every $r''' < r^* < r'$, if $step^{r^*} = EndRTC(j, \epsilon)$ then $id_j^{r^*} \neq l$). By the semantics of $EndRTC$ step and since $s \in \omega_l^{r'''}$, we know that $enabled(t, C^{r'''}) = false$. We also know that $enabled(t, C^r) = true$. This means that there exists some variable $v \in GRDV(\mathcal{A}^\beta)$ s.t. the value of v was modified between steps r''' and r. Formally, there exists $r''' < \tilde{r} < r$ s.t. $step^{\tilde{r}} = TRANS(j, (t_1, ..., t_q))$, and for some $\tilde{t} \in \{t_1, ..., t_q\}$, $v \in modif(\tilde{t})$ and $\lambda^{\tilde{r}}(v) \neq \lambda^{\tilde{r}+1}(v)$.
Consider the configuration that matches $C^{r''+1}$ on π', denoted $C'^{p''+1}$. By definition of $prev$, for every $r^* \in \{r'' + 1, ..., r\}$, $s \in \omega_l^{r^*}$. Thus $\Delta(\mathcal{A}^\beta) \cap \omega'^{p^*} \neq \phi$ for every $p^* \in \{p'' + 1, ..., p\}$. Assume $step'^{p'''} = EndRTC(j, \epsilon)$ matches $step^{r'''}$. Thus, $a_{strt}^{\beta} \in \omega_l'^{p'''}$ (this is the only state in by $\Delta(\mathcal{A}^\beta)$ without a null transition).
We want to show that for every $p^* \in \{p''', ..., p\}$: $a_{strt}^{\beta} \in \omega_l'^{p^*}$. Assume this is not the case. Since for every $p^* \in \{p''', ..., p\}$, $\Delta(\mathcal{A}^\beta) \cap \omega'^{p^*} \neq \phi$, and $a_{strt}^{\beta} \in \omega_l'^{p'''}$, then this means that a transition from a_{strt}^{β} to a_1^{β} was executed at some $step'^{p^*}$, for $p''' < p^* \leq p$. Since $a_{strt}^{\beta} \in \omega_l'^{p}$ then the transition from a_{end}^{β} to a_{strt}^{β} had to be executed on some $step'^{p^{**}}$ for $p^* < p^{**} \leq p$. By lemma 5.16, this means that $step'^{p^{**}}$ is either part of an $EndRTC$ step or a step leaving the abstraction. It is not part of an $EndRTC$ step, since the last $EndRTC$ step on

the current state machine is in $step'^{p'''}$ (and $p''' < p^{**}$). It is not possible that $step'^{p^{**}}$ leaves the abstraction, since we know $\Delta(\mathcal{A}^{\beta})$ is part of the configuration for all the steps until $step'^{p}$. We can then conclude that for every $p^* \in \{p''', ..., p\}$: $a^{\beta}_{strt} \in \omega'^{p^*}_{l}$.
Recall that $r''' < \tilde{r} < r$, $step^{\tilde{r}} = TRANS(j, (t_1, ..., t_q))$ and for some $\tilde{t} \in \{t_1, ..., t_q\}$, $v \in modif(\tilde{t})$ and $\lambda^{\tilde{r}}(v) \neq \lambda^{\tilde{r}+1}(v)$. Consider the last such $\tilde{r}$ (meaning, for every $r^* \in \{\tilde{r}+1, ..., r\}$: $\lambda^{r^*}(v) = \lambda^{\tilde{r}+1}(v)$
We separate between two cases, whether or not $\tilde{t}$ is an abstracted transition.

i. If $\tilde{(t)}$ is not an abstracted transition. Assume $\tilde{(t)} \in TR_{l'}$ (possibly $l' = l$). Then by definition of the abstraction, there exists a matching transition $\tilde{(t)}_a \in TR^{A}_{l'}$. By lemma 5.13 there exists a $step'^{\tilde{p}}$ on π' that matches $step^{\tilde{r}}$ and $\tilde{(t)}_a$ is executed on $step'^{\tilde{p}}$. By the definition of the abstraction, cg^{β}_{l} is set to 1 on $act(\tilde{(t)}_a$. Thus, $\lambda'^{\tilde{p}+1}(cg^{\beta}_{l}) \neq \epsilon$. The only transitions that can set cg^{β}_{l} to 0 are transitions exiting a^{β}_{strt}. However, we know that such transition was not taken, since for every $p^* \in \{p''', ..., p\}$: $a^{\beta}_{strt} \in \omega'^{p^*}_{l}$, and $p''' < \tilde{p} < p$. We can therefore conclude that $\lambda'^{p}(cg^{\beta}_{l}) \neq 0$.

ii. If $\tilde{t}$ is an abstracted transition. First of all, note that $\tilde{t}' \notin \Delta(\mathcal{A}^{\beta})$. This holds as a result of the property presented at the beginning of this item. Otherwise, by the property, the matching interval should traverse from a^{β}_{strt} to a^{β}_{1}, which did not occur, since for every $p^* \in \{p''', ..., p\}$: $a^{\beta}_{strt} \in \omega'^{p^*}_{l}$
Since we assume variables can be modified only by state machines under the same thread, then we know that $\tilde{t} \in TR_{l'}$ s.t. $thread(l') = j$. Under a single thread j RTC steps are executed one after the other. Assume that the $DISP$ step initiating the RTC step that includes $step^{\tilde{r}}$ is $step^{\tilde{r}'''} = DISP(j, \epsilon)$ and $id^{\tilde{r}'''}_{j} = l'$. Since $r''' < \tilde{r} < r$, then $r''' < \tilde{r}''' < \tilde{r}$.
Assume $\tilde{t} \in \Delta(\mathcal{A}^{\gamma})$ ($\mathcal{A}^{\gamma} \in ABS_{l'}$), and consider the

first transition $\tilde{t}'$ executed in the RTC step initiated by $step^{\tilde{r}'''}$ that is abstracted by $\Delta(\mathcal{A}^\gamma)$. This means that $\tilde{t}' \in TR(\mathcal{A}^\gamma)$, and for some $\tilde{r}' \in \{\tilde{r}''', ..., \tilde{r}\}$, $step^{\tilde{r}'} = TRANS(j, (t_1, ..., t_q))$, and $\tilde{t}' \in \{t_1, ..., t_q\}$. Also, for every $r^* \in \{\tilde{r}''', ..., \tilde{r}' - 1\}$, if $step^{r^*} = TRANS(j, (t'_1, ..., t'_q))$ then for every $t \in \{t'_1, ..., t'_q\}$, $t \notin TR(\mathcal{A}^\gamma)$.
Consider the step matching $step^{\tilde{r}'}$ on π', $step'^{\tilde{p}'}$. Since $\tilde{t}'$ is the first executed transition in the RTC step that is abstracted by $\mathcal{A}^\gamma$ then $a^\gamma_{strt} \in \omega'^{\tilde{p}'}_l$. Therefore $a^\gamma_1 \in \omega'^{\tilde{p}'+1}_l$ (this is a result of the property presented at the beginning of this item, stating that if we execute an abstracted transition and the matching abstract configuration is at a^β_{strt}, then the matching step executes a transition from a^β_{strt} to a^β_1).
Since $v \in modif(\tilde{t})$ and $v \in GRDV(\mathcal{A}^\beta)$, then by the definition of the abstraction, $cg^\beta_l = 1 \in act(t^a)$ for every transition t^a where $src(t^a) = a^\gamma_{strt}$ and $trgt(t^a) = a^\gamma_1$. The only transitions that can set cg^β_l to 0 are transitions from a^β_{strt}. We know that such transitions were not taken, since for every $p^* \in \{p', ..., p\}$: $a^\beta_{strt} \in \omega'^{p^*}_l$, and $p' < \tilde{p} < p$. We can therefore conclude that $\lambda'^p(cg^\beta_l) \neq 0$.

From the above we conclude that indeed $\lambda'^p(cg^\beta_l) \neq 0$.

We formalize the property presented in the previous item in the following lemma:

Lemma 5.15 *For every configuration $C^r \in \pi$ s.t. $step^r = TRANS(j, (t_1, ..., t_q))$, and a matching abstract configuration C'^p s.t. $C^r \preceq C'^p$: If for some $t \in \{t_1, ..., t_q\}$ and for some $\beta \in \{1, ..., g\}$, $a^\beta_{strt} \in \omega'^p_l$, and $src(t), trgt(t) \in S(\mathcal{A}^\beta)$, then $step'^p = TRANS(j, (t^a_1, ..., t^a_{q'}))$, and there exists $t^a \in \{t^a_1, ..., t^a_{q'}\}$ s.t. $C^{r+1} \preceq C'^{p+1}$, $src(t^a) = a^\beta_{strt}$, and $trgt(t^a) = a^\beta_1$.*

10. $s \notin \omega'^p_l$, for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$, for some $i \in \{1, ..., f_\beta\}$: $a^\beta_i \in \omega'^p_l$, and $trgt(t) \in S_l(\mathcal{A}^\beta)$: Notice that if $s \notin \omega'^p_l$, and since $c^r_l \preceq c'^p_l$, then $s \notin S^A_l$, and thus for some $\mathcal{A}^\beta \in ABS_l$, $s \in S_l(\mathcal{A}^\beta)$. This means that the current state of SM^A_l includes the abstraction, the transition taken is from a state s which is

abstracted by $\Delta(\mathcal{A}^\beta)$, and the target is a state abstracted by $\Delta(\mathcal{A}^\beta)$ as well. By definition of the current $TRANS$ step, $\rho_l^r = \epsilon$. The difference between C^r and C^{r+1} is in the value of variables and/or in the value of some event queue (if $GEN(e) \in act(t)$). We first consider the variables and show that $\lambda^{r+1} \preceq \lambda'^p$:

For every variable $v \notin modif(t)$, $\lambda^r(v) = \lambda^{r+1}(v)$, and thus clearly either $\lambda'^p(v) = \bot$ or $\lambda'^p(v) = \lambda^{r+1}(v)$.

For every variable $v \in modif(t)$, we show that $\lambda'^p(v) = \bot$: Since $v \in modif(t)$ then by definition $v \in V(\mathcal{A}^\beta)$. By the definition of the abstract model, if $a_i^\beta \in \omega_l'^p$, then there exists $C'^{p'} \in \pi'$ s.t. $a_{strt}^\beta \in \omega_l'^{p'}$ and $step'^{p'} = TRANS(j, (t'_1, ..., t'_q))$ s.t. there exists $t'_a \in \{t'_1, ..., t'_q\}$ where $src(t'_a) = a_{strt}^\beta$ and $trgt(t'_a) = a_1^\beta$. This means that for every $v \in V(\mathcal{A}^\beta)$, $\lambda'^{p'+1}(v) = \bot$. Since $\Delta(\mathcal{A}^\beta)$ can be entered only through a_{strt}^β, we can conclude that $step'^{p'}$ occurred in the current RTC step (meaning, there is no $p'' \in \{p', ..., p\}$ s.t. $step'^{p''} = EndRTC(j, \epsilon)$). We also know that for every $p'' \in \{p', ..., p\}$, $\omega_l'^{p''} \cap \Delta(\mathcal{A}^\beta) \neq \phi$.
Since variables can be modified only by state machines on the same thread, then if v is modified in some step p'' where $p'' \in \{p', ..., p\}$, then $step'^{p''} = TRANS(j, (t_1, ..., t_q))$ and $id_j'^{p''} = l$. By the definition of the abstraction, if for some $t'_a \in \{t_1, ..., t_q\}$, t'_a modifies $v \in V^A$ then one of the following holds:

- t'_a is an abstracted transition, in this case $v = \bot \in act(t'_a)$, or
- t'_a is not an abstracted transition, in this case consider t' the matching transition of t'_a. By definition of the abstraction, $v = e \in act(t')$ and "if $(isIn(\mathcal{A}^\beta))$ $v = \bot$; else $v = e$;"$\in act(t'_a)$. Since $isIn(\mathcal{A}^\beta) = true$, then $act(t'_a)(\lambda'^{p''}, C'^{p''}) = \bot$.

We can then conclude that $\lambda'^p(v) = \bot$.

We define the matching step based on $act(t)$:

(a) If $GEN(e) \notin act(t)$ (for some event e) - this means that the difference between C^r and C^{r+1} is only in the value of variables. We do not define a matching step in π', and clearly $C^{r+1} \preceq C'^p$.

(b) If $GEN(e) \in act(t)$ (for some event e) then by definition of the abstraction $e \in EV(\mathcal{A}^{\beta})$. By definition of the abstraction there exists a transition $t_a \in TR_l^A$ s.t. $src(t_a) = a_i^{\beta}$, $trgt(t_a) = a_{i+1}^{\beta}$, $trig(t_a) = \epsilon$ and $grd(t_a) = true$. Since $\rho_l'^p = \epsilon$, then $enabled(t_a, C'^p) = true$. We define $step'^p = TRANS(j, t_a)$. By definition of the abstraction it is possible that the action on t_a is $GEN(e)$, therefore $C^{r+1} \preceq C'^{p+1}$.

11. $s \notin \omega_l'^p$, for some $\mathcal{A}^{\beta} \in ABS_l$, $s \in S_l(\mathcal{A}^{\beta})$, $a_{f_\beta+1}^{\beta} \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^{\beta})$: Notice that if $s \notin \omega_l'^p$, and since $c_l^r \preceq c_l'^p$, then $s \notin S_l^A$, and thus for some $\mathcal{A}^{\beta} \in ABS_l$, $s \in S_l(\mathcal{A}^{\beta})$. This means that the current state of SM_l^A includes the abstraction, the transition taken is from a state s which is abstracted by $\Delta(\mathcal{A}^{\beta})$, and the target is a state abstracted by $\Delta(\mathcal{A}^{\beta})$ as well.

    Every computation on Γ^A can enter one of $\Delta(\mathcal{A}^{\beta})$ states only by entering a_{strt}^{β}. Recall that there can be at most f_β events generated in a single RTC step within the states abstracted by $\Delta(\mathcal{A}^{\beta})$ in Γ. If we reach $a_{f_\beta+1}$ then by the definition of the abstraction, f_β events were generated in the current RTC step within the states abstracted by $\Delta(\mathcal{A}^{\beta})$ in Γ. Therefore, $GEN(e) \notin act(t)$. For the same reasoning as in the previous item, we do not define a matching step in π', and $C^{r+1} \preceq C'^p$.

12. $s \notin \omega_l'^p$, for some $\mathcal{A}^{\beta} \in ABS_l$, $s \in S_l(\mathcal{A}^{\beta})$, $a_{end}^{\beta} \in \omega_l'^p$, and $trgt(t) \in S_l(\mathcal{A}^{\beta})$: Notice that if $s \notin \omega_l'^p$, and since $c_l^r \preceq c_l'^p$, then $s \notin S_l^A$, and thus for some $\mathcal{A}^{\beta} \in ABS_l$, $s \in S_l(\mathcal{A}^{\beta})$. This means that the current state of SM_l^A includes the abstraction, the transition taken is from a state s which is abstracted by $\Delta(\mathcal{A}^{\beta})$, and the target is a state abstracted by $\Delta(\mathcal{A}^{\beta})$ as well.

    This situation is not possible. Every computation on Γ^A can enter the one of $\Delta(\mathcal{A}^{\beta})$ states only by entering a_{strt}^{β}. Recall that there can be at most f_β events generated in a single RTC step within the abstracted states in Γ. The simulation is defined s.t. an execution of an abstracted transition t in Γ matches an execution of a transition in Γ^A only if $GEN(e) \in act(t)$. If $GEN(e) \notin act(t)$ then there is no execution of a matching transition in Γ^A. Therefore, it is not possible that t is an abstracted transition and

$a^{\beta}_{end} \in \omega'^{p}_{l}$.

The following lemma is based on the definition of the simulation relation:

Lemma 5.16 *The interval that includes a move from state* a^{β}_{i} *for* $1 \leq i \leq f_{\beta} + 1$ *to* a^{β}_{end} *matches a concrete transition that either leaves the abstraction or an EndRTC step.*

□

5.3 Using Abstraction

We now present the applicability of our abstraction framework through an example. We consider a system Γ describing a travel agent (of class *Agent*) that books flights and communicates with both airline databases (of class *DB*) and clients. We assume Γ includes n different *DB* objects, where the behavior of each *DB* is defined in Fig. 5.2. The single *Agent* object in Γ 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 event *evGetPrc* to them. 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 Γ^{A}, where the *DBs* are abstracted as in Fig. 5.3 (the *Agent* remains concrete). If *Agent* state machine includes x states, then Γ has $(12 * n + x)$ states, whereas Γ^{A} has $(4 * n + x)$ states. Moreover, Γ^{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 Γ, 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 Γ^A. If the property holds, then we can conclude that Γ 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 Γ^A, the result is $\perp$, since Γ^A abstracts the variable *space*. This means that we cannot conclude whether or not the property holds on Γ by model checking Γ^A. However, it might be possible to *refine* Γ^A, and create a different abstraction Γ'^A for which this property can be verified. Following, in section 5.4 we present how to refine an abstract system when the verification does not succeed.

5.4 Refinement

Once we have an abstract system Γ^A, we model check our LTL_x property $A\psi$ over the abstract system. Since variables in Γ^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 5.11 the property holds on Γ as well. If $(\Gamma^A \models A\psi) \in \{false, \perp\}$ then due to Γ^A being an over-approximation we cannot determine whether or not the property holds on Γ. Typical model checkers provide the user with a counterexample in case verification does not succeed. A counterexample π^A on Γ^A is either a finite computation or a lasso computation s.t. either $(\pi^A \models \psi) = false$ or $(\pi^A \models \psi) = \perp$.

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

If $(\pi^A \models \psi) = \perp$ then we cannot determine the value of the property. If $(\pi^A \models \psi) = false$, then this counterexample might be spurious. In both cases we search for a computation π on Γ s.t. $\pi \preceq_s \pi^A$. Given π^A, we inductively construct π w.r.t. π^A. Note that if the concrete model enables non-determinism, then there might be more than one matching concrete counterexample. In this case, all the matching concrete counterexamples are simultaneously constructed. Intuitively, the construction of π follows the steps of π^A, maintaining the stuttering inclusion. During the construction, if for some prefix of π^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 π^A is a spurious counterexample and we should refine the system. We present a formal construction of π based on the counterexample π^A in Section 5.4.1. 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 $t_a \notin \Delta(\mathcal{A}^\beta)$ (for some $\mathcal{A}^\beta \in ABS_l$), 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 $t_a \in \Delta(\mathcal{A}^\beta)$ (for some $\mathcal{A}^\beta \in ABS_l$) that generates an event e, and there is no enabled concrete transition $t \in TR(\mathcal{A}^\beta)$ where $GEN(e) \in act(t)$.

We call the configuration $C'^p \in \pi^A$ from which we cannot extend a matching concrete computation a *failure-configuration.* Following, we distinguish between two reasons that can cause a failure-configuration, 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 Γ. For example, when $step'^p = TRANS(j, (t_a))$, $id'^p_j = l$, and the concrete transition $t \in TR_l$ that matches t_a is not enabled, since $src(t) \notin \omega^r_l$. This is possible only if $src(t) \in S_l(\mathcal{A}^\beta)$ and $trgt(t) \notin S_l(\mathcal{A}^\beta)$ (for some $\mathcal{A}^\beta \in ABS_l$). Another example for such failure is where Γ^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 $t_a \in \mathcal{A}^\beta$ (for some $\mathcal{A}^\beta \in ABS_l$). In both cases we refine by removing a state $s \in S_l(\mathcal{A}^\beta)$ such that $s \in \omega^r_l$ from the abstraction.
Case 2: There exists $v \in V$ for which $\lambda'^p(v) = \perp$ and the value of $\lambda^r(v)$ causes the failure-configuration. For example, when $step'^p = TRANS(j, (t_a))$ and the concrete t that matches t_a is not enabled w.r.t. C^r, since $grd(t)(\lambda^r) = false$. Since $C^r \lhd C'^p$ and $grd(t_a) = grd(t)$, then clearly $grd(t_a)(\lambda'^p) = \perp$, and for some v, $\lambda'^p = \perp$ and v affects the value of $grd(t_a)$. We refine Γ^A to obtain a concrete value of v:
We trace π^A back to find the assignment that gave v the value $\perp$. The only place where a variable is initially assigned the value $\perp$ is a transition from a^β_{strt} to a^β_1 in some $\mathcal{A}^\beta$ (for some $\mathcal{A}^\beta \in ABS_l$). Thus, the tracing back of

π^A terminates as $C'^{p'}$ such that $a^{\beta}_{strt} \in \omega^{\alpha}_{l}$. We find the matching system-configuration $C^{r'}$ in π s.t. $C^{r'} \lhd C'^{p'}$, and refine the model by removing from the abstraction a state $s \in S_l(\mathcal{A}^{\beta})$ such that $s \in \omega^{r'}_{l}$.

If we are able to construct π s.t. $\pi \lhd \pi^A$, then one of the following holds:

1. If $(\pi^A) \models \psi) = \mathit{false}$ then no need to check π. By construction, $\pi \not\models \psi$, and we can conclude that $\Gamma \not\models A\psi$.

2. If $(\pi^A) \models \psi) = \perp$ then we check π w.r.t. ψ. If $\pi \not\models \psi$ then again π is a concrete counterexample 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 ψ, and v has the value $\perp$. We then refine Γ^A in order to have a concrete value on v, as described above (Case 2).

Consider the example system presented in Section 5.3, 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 Γ'^A, where DB objects are abstracted w.r.t. a new abstraction collection $ABS = \{\{\mathit{Idle}, \mathit{PriceProcessor}, \mathit{UpdateDB}\}\}$. The property can then be verified on Γ'^A, and we can conclude that it holds on the concrete system.

5.4.1 Constructing π From π^A

Following we present the inductive construction of π w.r.t. the given counterexample π^A:

Base: Given $C'^0 = (c'^0_1, ..., c'^0_n, q'^0_1, ..., q'^0_m, id'^0_1, ..., id'^0_m, \lambda'^0)$, the initial configuration of π^A. We define the following initial configuration for π: $C^0 = (c^0_1, ..., c^0_n, q^0_1, ..., q^0_m, id^0_1, ..., id^0_m, \lambda^0)$.

- For every $i \in \{1, ..., n\}$: $\omega^0_i = \{s | s \in \mathit{init}_i \land \forall s' : s \lhd s' \rightarrow s \in \mathit{init}_i\}$
- For every $i \in \{1, ..., n\}$: $\rho^0_i = \epsilon$.
- For every $j \in \{1, ..., m\}$: $q^0_j = q'^0_j = \phi$
- For every $j \in \{1, ..., m\}$: $id^0_j = id'^0_j = 0$

- For every $v \in V$: $\lambda^0(v) = \lambda'^0(v)$

It is immediate to see that C'^0 is an initial configuration and also that $C^0 \preceq C'^0$

Step: Assume that for the first p steps of π^A: $C'^0, step'^0, C'^1, step'^1, ..., C'^{p-1}, step'^{p-1}, C'^p$ there exists a partial computation $\pi = C^0, step^0, C^1, step^1, ..., C^r$ over Γ s.t. there are two sequences of integers $0 = i_0 < i_1 < i_2 < ... < i_l = r+1$ and $0 = i'_0 < i'_1 < i'_2 < ... < i'_l = p+1$ and for every $0 \leq k < l$:
For every $j \in \{i_k, ..., (i_{k+1} - 1)\}$ and for every $j' \in \{i'_k, ..., (i'_{k+1}) - 1\}$: $C^j \preceq C'^{j'}$

Note that the requirement on interval $(l-1)$ induces $C^r \preceq C'^p$.

The matching extension of π is defined based on $step'^p$:

- $step'^p = DISP(j, ev)$
  By definition, $id'^p_j = 0$, $q'^p_j \neq \phi$ and $top(q'^p_j) = ev$. Since $C^r \preceq C'^p$, then $id^r_i = id'^p_i$ and $q^r_i = q'^p_i$ for every i. Therefore, $id^r_j = 0$, $q^r_j \neq \phi$ and $top(q^r_j) = ev$ as well, and $step^r = DISP(j, ev)$ is a possible step from C^r.
  By definition of $DISP$ step, C'^{p+1} differs from C'^p in the following: $q'^{p+1}_j = pop(q'^p_j)$, $id'^{p+1}_j = trgt(ev)$ and $\rho'^{p+1}_{trgt(ev)} = type(ev)$.
  By definition of $DISP$ step, C^{r+1} differs from C^r in the following: $q^{r+1}_j = pop(q^r_j)$, $id^{r+1}_j = trgt(ev)$ and $\rho^{r+1}_{trgt(ev)} = type(ev)$.
  Since $C^r \preceq C'^p$, it is clear that $C^{r+1} \preceq C'^{p+1}$ as well.

- $step'^p = ENV(j, ev)$.
  Since the environment is always enabled, then an ENV step s.t. $step^r = ENV(j, ev)$ is possible from C^r, and clearly $C^{r+1} \preceq C'^{p+1}$.

- $step'^p = EndRTC(j, \epsilon)$
  Assume $id^r_j = id'^p_j = l > 0$. If $stable(c^r_l, C^r)$ then we define $step^r = EndRTC(j, \epsilon)$ and clearly $C^{r+1} \preceq C'^{p+1}$. Otherwise, $stable(c^r_l, C^r) = false$, in which case we might need to refine.
  If $stable(c^r_l, C^r) = false$ then there exists $s \in \omega^r_l$ and $t_c \in TR_l$ where $src(t_c) = s$ s.t. $trig(t_c) = \rho^r_l$ and $grd(t_c)(\lambda^r) = true$. Assume SM^A_l is an abstraction of SM_l w.r.t. $ABS_l = \{\mathcal{A}^1, ..., \mathcal{A}^g\}$.

  1. If for every $\beta \in \{1, ..., g\}$, $s \notin S_l(\mathcal{A}^\beta)$ then by definition of SM^A_l, $s \in S^A_l$ and there exists a matching transition $t_a \in TR^A_l$ s.t.

$src(t_a) = s$, $trig(t_c) = trig(t_a)$ and $grd(t_c) = grd(t_a)$. By definition of the simulation, $s \in \omega_l^r$. Since $trig(t_c) = trig(t_a)$ and $stable(c_l'^p, C'^p) \in \{\bot, true\}$ then this means that $grd(t_a)(\lambda'^p) \in \{false, \bot\}$. However, we know that for every $v \in V$, either $\lambda'^p(v) = \bot$ or $\lambda'^p(v) = \lambda^r(v)$. Since $grd(t_c) = grd(t_a)$,then it is not possible that $grd(t_c)(\lambda^r) = true$ and $grd(t_a)(\lambda'^p) = false$. We conclude that $grd(t_a)(\lambda'^p) = \bot$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \bot$ and v effects the evaluation of $grd(t_a)$ and refine Γ^A w.r.t. v. We present how to refine Γ^A w.r.t. some variable v (as described above, in Case 2).

2. If for some $\beta \in \{1, ..., g\}$, $s \in S_l(\mathcal{A}^\beta)$ (s is an abstracted state). Since $c^r \preceq c'^p$ then $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p \neq \phi$. We conclude that $a_{strt}^\beta \in \omega_l'^p$ (this is the only state in $\Delta(\mathcal{A}^\beta)$ without a null transition). We separate between different cases.
   - If $\rho_l^r = \epsilon$ and for every abstracted variable $v \in V(\mathcal{A}^\beta)$, $\lambda'^p(v) = \bot$: this means that the abstraction has been traversed. It is possible that the concrete model might reach a stable state after traversing abstracted transitions without GEN. Recall that these transitions do not have a matching transition in the abstraction. Continue from C^r the RTC step on abstracted transitions (transitions $t' \in TR(\mathcal{A}^\beta)$), as long as it is on transitions without GEN. Since the by definition of the abstraction, these transitions can only modify variables from $V(\mathcal{A}^\beta)$, which have the value $\bot$ in λ'^p, then on all such reachable configurations $C^{r'} \preceq C'^p$. When cannot progress anymore, check $stable(c_l^{r'}, C^{r'})$. If for *one of the reachable configurations* $stable(c_l^{r'}, C^{r'}) = true$ then we define $step^{r'} = EndRTC(j, \epsilon)$ and clearly $C^{r'+1} \preceq C'^{p+1}$. Otherwise, there exists $s' \in \omega_l^{r'}$ and $t'_c \in TR_l$ where $src(t'_c) = s'$ s.t. $trig(t'_c) = \epsilon$ and $grd(t'_c)(\lambda^{r'}) = true$.
     * If $s' \in S_l^A$: then $s' \in \omega_l'^p$, and there exists a transition $t'_a \in TR_l^A$ s.t. $src(t'_a) = s'$, $trig(t'_a) = trig(t'_c) = \epsilon$ and $grd(t'_c) = grd(t'_a)$. Since $stable(c_l'^p, C'^p) \in \{\bot, true\}$ and since for every $v \in V$, either $\lambda'^p(v) = \bot$ or $\lambda'^p(v) = \lambda^{r'}(v)$, then $grd(t'_a)(\lambda'^p) = \bot$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \bot$ and v effects the evaluation of

$grd(t'_a)$ and refine Γ^A w.r.t. v.

  * If $s' \notin S_l^A$: then for some $\gamma \in \{1, ..., g\}$, $s' \in S_l(\mathcal{A}^\gamma)$. We refine the abstraction by removing s' from $S_l(\mathcal{A}^\gamma)$.

– Otherwise: we refine the abstraction by removing s from $S_l(\mathcal{A}^\beta)$.

- $step'^p = TRANS(j, (t_1^a, ..., t_q^a))$

  For every $i \in \{1, ..., q\}$ we match transition t_i^a with (possibly more than one) transition $t_i \in TR_l$.

  1. If $src(t_i^a) \in S_l$: Then $src(t_i^a) \in \omega_l^r$. By definition of the abstraction, there exists a matching transition $t \in TR_l$ s.t. $src(t) = src(t_i^a)$, $trig(t) = trig(t_i^a)$, $grd(i) = grd(t_i^a)$, and $act(t) = act(t_i^a) = skip$ (since $trig(t_i^a) = \rho_l'^p \neq \epsilon$). If $enabled(t, C^r) = true$ then we define $t_i = t$.

     Otherwise, if $enabled(t, C^r) = false$, then we need to refine. We separate between the different cases that can cause $enabled(t, C^r) = false$ and $enabled(t_i^a, C'^p) \in \{\bot, true\}$:

     – For every $t' \in TR_l$ s.t. $src(t') \lhd src(t)$ and $src(t') \in \omega_l^r$ it holds that $enabled(t', C^r) = false$: Since $\rho_l^r = \rho_l'^p$, this means that $grd(t_i^a)(\lambda'^p) \in \{\bot, true\}$ and $grd(t)(\lambda^r) = false$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t_i^a)(\lambda'^p) = \bot$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \bot$ and v effects the evaluation of $grd(t_i^a)$ and refine Γ^A w.r.t. v.

     – There exists $t' \in TR_l$ s.t. $src(t') \lhd src(t)$, $src(t') \in \omega_l^r$ and $enabled(t', C^r) = true$.

       (a) If $src(t') \in \omega_l'^p$, then by definition of the abstraction, there exists $t'_a \in TR_l^A$ s.t. $src(t'_a) = src(t')$, $trig(t') = trig(t'_a)$, $grd(t') = grd(t'_a)$. Since $\rho_l^r = \rho_l'^p$, this means that $grd(t'_a)(\lambda'^p) \in \{false, \bot\}$ and $grd(t')(\lambda^r) = true$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t'_a)(\lambda'^p) = \bot$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \bot$ and v effects the evaluation of $grd(t'_a)$ and refine Γ^A w.r.t. v.

       (b) If $src(t') \notin \omega_l'^p$, then by definition of the abstraction, $src(t') \in S_(\mathcal{A}^\beta)$ for some $\beta \in \{1, ..., g\}$. Also, by definition of the simulation, $\Delta(\mathcal{A}^\beta) \cap \omega_l'^p \neq \epsilon$. We refine the abstraction by removing $src(t')$ from $S_l(\mathcal{A}^\beta)$.

2. If $src(t_i^a) \notin S_l$: Then $src(t_i^a) \notin \omega_l^r$. Since $trig(t_i^a) = \rho_l'^p \neq \epsilon$, then $src(t_i^a) = a_{strt}^\beta$ for some $\beta \in \{1, ..., g\}$.
   - If $trgt(t_i^a) = a_1^\beta$: If there exists a *maximal* set of *orthogonal* and *enabled* transitions $t_1', ..., t_{q'}' \in TR_l(\mathcal{A}^\beta)$, then t_i is defined by $t_1', ..., t_{q'}'$.

     If no such set of transitions exist, then we need to refine. Refine by removing a state $s' \in S_l(\mathcal{A}^\beta) \cap \omega_l^r$ from $S_l(\mathcal{A}^\beta)$.
   - Otherwise, this means that either (1) $trgt(t_i^a) = a_{strt}^\gamma$ for $\gamma \in \{1, ..., g\}$ and $\gamma \neq \beta$, or (2) for every $\gamma \in \{1, ..., g\}$, $trgt(t_i^a) \notin S_l(\mathcal{A}^\gamma)$ ($trgt(t_i^a) \in S_l$): By definition of the abstraction, there exists a matching transition $t' \in TR_l$ s.t. $src(t') \in S_l(\mathcal{A}^\beta)$, $trig(t') = trig(t_i^a)$, and
     * if $trgt(t_i^a) = a_{strt}^\gamma$ then $trgt(t') \in S_l(\mathcal{A}^\gamma)$
     * if $trgt(t_i^a) \in S_l$ then $trgt(t') = trgt(t_i^a)$

     Also, by definition of the abstraction, $grd(t_i^a) = grd(t')\&\bot$.
     (a) If $src(t') \in \omega_l^r$ and $enabled(t', C^r) = true$: then define $t_i = t'$.
     (b) If $src(t') \in \omega_l^r$ and $enabled(t', C^r) = false$: then need to refine.

     We separate between the different cases that can cause $enabled(t', C^r) = false$ and $enabled(t_i^a, C'^p) \in \{\bot, true\}$:
       * For every $t'' \in TR_l$ s.t. $src(t'') \lhd src(t')$ it holds that $enabled(t'', C^r) = false$: Since $\rho_l^r = \rho_l'^p$, this means that $grd(t')(\lambda'^p) \in \{\bot, true\}$ and $grd(t')(\lambda^r) = false$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t')(\lambda'^p) = \bot$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \bot$ and v effects the evaluation of $grd(t')$ and refine Γ^A w.r.t. v.
       * There exists $t'' \in TR_l$ s.t. $src(t'') < src(t')$, $src(t'') \in \omega_l^r$ and $enabled(t'', C^r) = true$. By definition of the abstraction, $src(t'') \in S_a(A^k)$. We refine the abstraction by removing $src(t'')$ from the abstracted states.

     (c) If $src(t') \notin \omega_l^r$, then need to refine. Remove $src(t')$ from $S_(\mathcal{A}^\beta)$.

We define $step^r = TRANS(j,(t_1,...,t_q))$. For every such transition, since $\rho^r_l \neq \epsilon$, then for every $i \in \{1,...,q\}$, either $act(t_i) = skip$ or $act(t_i)$ changes the value of cg variables. By definition of the abstraction, for every $i \in \{1,...,q\}$, $act(t^a_i)$ possibly sets the value of variables from V to $\perp$ and changes the value of cg variables. Since $C^r \preceq C'^p$, then clearly $C^{r+1} \preceq C'^{p+1}$.

- $step'^p = TRANS(j,t_a)$
  Assume $id^r_j = id'^p_j = l$ and $src(t_a) = s$. We know that $enabled(t_a, C'^p) \in \{true, \perp\}$. We separate to the following different cases:

  1. If $src(t_a) \in \omega^r_l$
  2. If $src(t_a) \notin \omega^r_l$ and for some $\beta \in \{1,...,g\}$, $src(t_a) = a^\beta_{end}$
  3. If $src(t_a) \notin \omega^r_l$ and for some $\beta \in \{1,...,g\}$, $src(t_a) = a^\beta_i$
  4. If $src(t_a) \notin \omega^r_l$ and for some $\beta \in \{1,...,g\}$, $src(t_a) = a^\beta_{strt}$

  1. If $src(t_a) \in \omega^r_l$: By definition of the abstraction, there exists a matching transition $t \in TR_l$ s.t. $src(t) = src(t_a)$, $trig(t) = trig(t_a)$, and $grd(t) = grd(t_a)$. If $enabled(t, C^r) = true$ then we define $step^r = TRANS(j,t)$. By definition of the abstraction construction, $act(t_a)$ differs from $act(t)$ in the following:
     - $act(t_a)$ might include manipulation of cg variables
     - assignments of type $v = e$ in $act(t)$ might be replaced with "if $(isIn(\mathcal{A}))$ $v = \perp$; else $v = e$;" in $act(t_a)$.

     Since $C^r \preceq C'^p$ and specifically $\lambda^r \preceq \lambda'^p$, then clearly $act(t)(\lambda^r, C^r) \preceq act(t_a)(\lambda'^p, C'^p)$. By definition of the matching transition, either $trgt(t_a) = trgt(t)$ or $trgt(t_a) = a^\beta_{strt}$ and $trgt(t) \in \mathcal{A}^\beta$. Thus, we can conclude that $C^{r+1} \preceq C'^{p+1}$.

     Otherwise, if $enabled(t, C^r) = false$, then we need to refine. We separate between the different cases that can cause $enabled(t, C^r) = false$ and $enabled(t_a, C'^p) \in \{\perp, true\}$:
     - For every $t' \in TR_l$ s.t. $src(t') \lhd src(t)$ and $src(t') \in \omega^r_l$ it holds that $enabled(t', C^r) = false$: Since $\rho^r_l = \rho'^p_l$, this means that $grd(t_a)(\lambda'^p) \in \{\perp, true\}$ and $grd(t)(\lambda^r) = false$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t_a)(\lambda'^p) = \perp$. We choose

some variable $v \in V$ s.t. $\lambda'^p(v) = \perp$ and v effects the evaluation of $grd(t_a)$ and refine Γ^A w.r.t. v.

- There exists $t' \in TR_l$ s.t. $src(t') \lhd src(t)$, $src(t') \in \omega^r_l$ and $enabled(t', C^r) = true$.
  (a) If $src(t') \in \omega'^p_l$, then by definition of the abstraction, there exists a matching transition $t'_a \in TR^A_l$ s.t. $src(t'_a) = src(t')$, $trig(t') = trig(t'_a)$, $grd(t') = grd(t'_a)$. Since $\rho^r_l = \rho'^p_l$, this means that $grd(t'_a)(\lambda'^p) \in \{false, \perp\}$ and $grd(t')(\lambda^r) = true$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t'_a)(\lambda'^p) = \perp$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \perp$ and v effects the evaluation of $grd(t'_a)$ and refine Γ^A w.r.t. v.
  (b) If $src(t') \notin \omega'^p_l$, then by definition of the abstraction, $src(t') \in S(\mathcal{A}^\beta)$ for $\beta \in \{1, ..., g\}$. Since $c^r_l \preceq c'^p_l$, this means that $a^\beta_{strt} \in \omega'^p_l$ (it is not possible that one of $\Delta(\mathcal{A}^\beta) \setminus \{a^\beta_{strt}\}$ is in ω'^p_l since these states have null outgoing transitions, and thus $enabled(t_a, C'^p) = false$). We separate between two cases, whether or not the abstraction has been traversed.
    * If there exists an abstracted variable $v \in V(\mathcal{A}^\beta)$ s.t. $\lambda'^p(v) \neq \perp$: this means that the abstraction has just been entered. We refine the abstraction by removing $src(t')$ from the abstracted states.
    * If for every abstracted variable $v \in V(\mathcal{A}^\beta)$, $\lambda'^p(v) = \perp$: this means that the abstraction has been traversed. It is possible that the concrete model might reach a state where $enabled(t, C^{r'}) = true$ after traversing abstracted transitions without GEN. Continue from C^r the RTC step on abstracted transitions (transitions $t' \in TR(\mathcal{A}^\beta)$), as long as it is on transitions without GEN. Since by the definition of the abstraction, these transitions can only modify variables from $V(\mathcal{A}^\beta)$, which have the value $\perp$ from the abstraction, then on all such reachable configurations $C^{r'} \preceq C'^p$. For *every* such reachable $C^{r'}$, if $enabled(t, C^{r'})$ then we define $step^{r'} = TRANS(j, t)$.

Otherwise, if on all possible reachable configurations $C^{r'}$ it holds that $enabled(t, C^{r'}) = false$, then we need to refine.

  · If for some reachable configuration $C^{r'}$, and for every $t'' \in TR_l$ s.t. $src(t'') \lhd src(t)$ and $src(t'') \in \omega_l^{r'}$ it holds that $enabled(t'', C^{r'}) = false$, then this means that $grd(t)(\lambda^{r'}) = false$. Since $enabled(t_a, C'^p) = true$, then $grd(t_a)(\lambda'^p) \in \{\perp, true\}$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t_a)(\lambda'^p) = \perp$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \perp$ and v effects the evaluation of $grd(t_a)$ and refine Γ^A w.r.t. v.
  · Otherwise, refine the abstraction by removing $src(t')$ from the abstracted states.

2. If $src(t_a) \notin \omega_l^r$ and for some $\beta \in \{1, ..., g\}$, $src(t_a) = a_{end}^{\beta}$. By definition of the abstraction, $trgt(t_a) = a_{strt}^{\beta}$. Continue on $step'^p$ without matching a step on π. It holds that $C^r \preceq C'^{p+1}$.
3. If $src(t_a) \notin \omega_l^r$ and for some $\beta \in \{1, ..., g\}$, $src(t_a) = a_i^{\beta}$: By definition of the abstraction, $trgt(t_a)$ is either a_{end}^{β} or a_{i+1}^{β}.
   – If $trgt(t_a) = a_{end}^{\beta}$: Continue on $step'^p$ without matching a step on π. It holds that $C^r \preceq C'^{p+1}$.
   – If $trgt(t_a) = a_{i+1}^{\beta}$: By definition of the abstraction, $act(t_a) = GEN(EV(\mathcal{A}^{\beta}))$. Assume the event generated on $step'^p$ is $ev \in EV(\mathcal{A}^{\beta})$. Continue from C^r the RTC step on abstracted transitions (transitions $t' \in TR(\mathcal{A}^{\beta})$), as long as it is on transitions without GEN. Since by the definition of the abstraction, these transitions can only modify variables from $V(\mathcal{A}^{\beta})$, which have the value $\perp$ from the abstraction, then on all of them $C^{r'} \preceq C'^p$. On *every* such reachable $C^{r'}$, if there exists a transition $t'_c \in TR(\mathcal{A}^{\beta})$ where $GEN(ev) \in act(t'_c)$ and $enabled(t'_c, C^{r'}) = true$ then $step^{r'} = TRANS(j, t'_c)$. For same reasoning as before, $C^{r'+1} \preceq C'^{p+1}$.
     Otherwise, if on all possible reachable configurations $C^{r'}$ no such t'_c, then need to refine. For some $s \in \omega_l^r$ s.t. $s \in S_l(\mathcal{A}^{\beta})$, remove s from $S_l(\mathcal{A}^{\beta})$.
4. If $src(t_a) \notin \omega_l^r$ and for some $\beta \in \{1, ..., g\}$, $src(t_a) = a_{strt}^{\beta}$: We

separate between the different cases for $trgt(t_a)$

- $trgt(t_a) = a_1^\beta$: Since $\rho_l'^p = \epsilon$, then $t_a = \tau_2^\beta$. For every abstracted transition $t \in TR(\mathcal{A}^\beta)$ s.t. $enabled(t, C^r)$ define $step^r = TRANS(j,t)$. By definition of the abstraction, $modif(t) \subseteq V(\mathcal{A}^\beta)$. Since for every $v \in V(\mathcal{A}^\beta)$, $\lambda'^{p+1}(v) = \bot$, and since $C^r \preceq C'^p$, then $C^r \preceq C'^{p+1}$.

  Otherwise, if no such t exists, then need to refine. For some $s \in \omega_l^r$ s.t. $s \in S_l(\mathcal{A}^\beta)$, remove s from $S_l(\mathcal{A}^\beta)$.
- Otherwise: This means that either (1) $trgt(t_a) \in S_l$ or (2) $trgt(t_a) = a_{strt}^\gamma$ for $\gamma \in \{1, ..., g\}$ and $\gamma \neq \beta$: By definition of the abstraction, there exists a matching transition $t \in TR_l$ s.t. $src(t) \in S_l(\mathcal{A}^\beta)$, $trig(t) = trig(t_a)$, and $grd(t_a) = grd(t) \& \bot$.

  (a) If $src(t) \in \omega_l^r$ and $enabled(t, C^r) = true$: then define $step^r = TRANS(j,t)$.

  By definition of the abstraction construction, $act(t_a)$ differs from $act(t)$ in the following:

  * $act(t_a)$ might include manipulation of cg variables
  * assignments of type $v = e$ in $act(t)$ might be replaced with "if $(isIn(\mathcal{A}))$ $v = \bot$; else $v = e$;" in $act(t_a)$.

  Since $C^r \preceq C'^p$ and specifically $\lambda^r \preceq \lambda'^p$, then clearly $act(t)(\lambda^r, C^r) \preceq act(t_a)(\lambda'^p, C'^p)$. By definition of the matching transition, either $trgt(t_a) = trgt(t)$ or $trgt(t_a) = a_{strt}^\gamma$ and $trgt(t) \in \mathcal{A}^\gamma$. Thus, we can conclude that $C^{r+1} \preceq C'^{p+1}$.

  (b) If $src(t) \in \omega_l^r$ and $enabled(t.C^r) = false$: then need to refine. We separate between the different cases that can cause $enabled(t, C^r) = false$ and $enabled(t_a, C'^p) \in \{\bot, true\}$:

  * For every $t' \in TR_l$ s.t. $src(t') \lhd src(t)$ it holds that $enabled(t', C^r) = false$: Since $\rho_l^r = \rho_l'^p$, this means that $grd(t)(\lambda'^p) \in \{\bot, true\}$ and $grd(t)(\lambda^r) = false$. Since $\lambda^r \preceq \lambda'^p$ we conclude that $grd(t)(\lambda'^p) = \bot$. We choose some variable $v \in V$ s.t. $\lambda'^p(v) = \bot$ and v effects the evaluation of $grd(t_a)$ and refine Γ^A w.r.t. v.

∗ There exists $t' \in TR_l$ s.t. $src(t') \lhd src(t)$, $src(t') \in \omega_l^r$ and $enabled(t', C^r) = true$. By definition of the abstraction $src(t') \in S_l(\mathcal{A}^\beta)$. We refine the abstraction by removing $src(t')$ from $S_l(\mathcal{A}^\beta)$.

(c) If $src(t) \notin \omega_l^r$, then we need to refine. For some $s \in \omega_l^r$ s.t. $s \in S_l(\mathcal{A}^\beta)$, remove s from $S_l(\mathcal{A}^\beta)$.

5.5 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.

Chapter 6

Learning-Based Compositional Verification of Behavioral UML Systems

In this chapter we present a novel approach for learning-based compositional verification of behavioral UML systems.

One of the most appealing approaches to fighting the high time and memory requirements of model checking is *compositional model checking*, where parts of the system are verified separately in order to avoid the construction of the entire system and to reduce the model checking cost. The Assume-Guarantee (**AG**) paradigm [30, 44, 26] suggests how to verify a component based on an *assumption* on the behavior of its environment, which consists of the other system components. The environment is then verified in order to guarantee that the assumption is actually correct.

Learning [2] has been a major technique to construct assumptions for the **AG** paradigm automatically. An automated *learning-based* **AG** *framework* was first introduced in [15]. It uses iterative **AG** reasoning, where in each iteration an assumption is constructed and checked for suitability, based on learning and on model checking. Many works suggest optimizations of the basic framework and apply it in the context of different **AG** rules ([7, 23, 57, 20, 39, 28, 5, 14, 43, 9]).

In this chapter we propose a framework for automated learning-based **AG** reasoning *for behavioral UML systems.* Our framework is similar to the one presented in [15], with the main difference being that our framework remains at the state machine level. That is, the system's components are state machines, and the learned assumptions are *state machines* as well. This is in contrast to [15], where the system's components and the learned assumptions are all presented as Labeled Transition Systems (LTSs).

A naive implementation of our framework might translate a given behavioral UML system into LTSs and apply the algorithm from [15] on the result. However, due to the hierarchical and orthogonal structure of state machines such translation would result in LTSs that are exponentially larger than the original UML system. Moreover, state machines communicate via event queues. Such translation must also include the event queues, which would also increase the size of the LTSs by an order of magnitude. We therefore choose to define a framework for automated learning-based **AG** reasoning *directly on the state machine level.* Another important advantage of working with state machines is that it enables us to exploit high level information to make the learning much more efficient. It also enables us to apply model checkers designed for *behavioral UML systems.* Such model checkers take into account the specific structure and semantics of UML, and are therefore more efficient than model checkers designed for low-level representations (such as Kripke structures or LTSs).

We use the standard **AG** rule below, where M_1 and M_2 are UML state machines. We replace $\langle A\rangle$ with $[A]$, to emphasize that A is a state machine playing the role of an *assumption* on the environment of M_1. The first premise (*Step* 1) holds iff $A||M_1$ satisfies φ, and the second one (*Step* 2) holds iff every execution of M_2 has a representative in A. Together they guarantee that $M_1||M_2$ satisfies φ.

Rule AG-UML

$$\begin{array}{ll} (\mathit{Step}\ 1) & [A]\ M_1\ \langle\varphi\rangle \\ (\mathit{Step}\ 2) & \langle \mathit{true}\rangle\ M_2\ [A] \\ \hline & \langle \mathit{true}\rangle\ M_1||M_2\ \langle\varphi\rangle \end{array}$$

We assume φ is a safety property, and use the learning algorithm L^* [2, 50] to iteratively construct assumptions A_i until both premises of the rule hold for A_i, implying $M_1||M_2 \models \varphi$, or until a real counterexample is found, demonstrating that $M_1||M_2 \not\models \varphi$.

We exploit the notion of RTC steps for defining the alphabet Σ of the learned assumptions. We define an alphabet over *sequences of events*, where a letter (i.e., a sequence of events) represents a single RTC step of the assumption. A word w over these letters corresponds to an execution of the assumption. It also represents the equivalence class of all executions of the checked system, which are interleaved with w. Our alphabet is defined based on statically analyzing the behavior of M_2.

Learning words over sequences of events makes L^* highly efficient, as it avoids learning sequences that can never occur in M_2 and therefore should not be considered in an assumption. Moreover, our learning is executed w.r.t. *equivalence classes of executions*. Even though our learning process is over equivalence classes, we show that our framework is sound and complete. That is, we do not lose information from grouping executions according to their representative word.

The remainder of the chapter is organized as follows. Some background on **AG** reasoning is given in Section 6.1. Representing UML computations and execution as words is defined in Section 6.2. In Section 6.3 we present our basic framework, implementing **Rule AG-UML** for UML systems. Sections 6.4 and 6.5 extend the framework to a more general setting. We conclude in Section 6.6.

6.1 Preliminaries

6.1.1 Assume Guarantee Reasoning and Compositional Verification

[15] presents a framework for automatically constructing assumption A in an iterative fashion for applying the standard **AG** rule, where M_1 and M_2 are $LTSs$ and φ is a safety property. At each iteration i, an assumption A_i is constructed. Afterwards, *Step* 1 ($\langle A_i \rangle M_1 \langle \varphi \rangle$) is applied in order to check whether M_1 guarantees φ in an environment that satisfies A_i. A $false$ result means that this assumption is too *weak*, i.e., A_i does not restrict the environment enough for φ to be satisfied. Thus, the assumption needs to be *strengthened* (which corresponds to removing behaviors from it) with the help of the counterexample produced by *Step* 1. If *Step* 1 returns $true$ then A_i is strong enough for the property to be satisfied. To complete the proof, *Step* 2 ($\langle true \rangle M_2 \langle A_i \rangle$) must be applied to discharge A_i on M_2. If *Step* 2

returns $true$, then the compositional rule guarantees $\langle true\rangle M_1||M_2\langle\varphi\rangle$. That is, φ holds in $M_1||M_2$. If it returns $false$, further analysis is required to identify whether $M_1||M_2$ violates φ or whether A_i is stronger than necessary. Such analysis is based on the counterexample returned by *Step* 2. If A_i is too strong it must be *weakened* (i.e., behaviors must be added) in iteration $i+1$. The new assumption may be too weak, and thus the entire process must be repeated.

The framework in [15] uses a learning algorithm for generating assumptions A_i and a model checker for verifying the two steps in the rule.

6.1.2 The L^* Algorithm

The learning algorithm used in [15] was developed by [2], and later improved by [50]. The algorithm, named L^*, learns an unknown regular language and produces a minimal deterministic finite automaton (DFA) that accepts it. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L^* needs to interact with a *Minimally Adequate Teacher*, called Teacher. A Teacher must be able to correctly answer two types of questions from L^*. A *membership query*, consists of a string $w \in \Sigma^*$. The answer is *true* if $w \in U$, and $false$ otherwise. A *conjecture* offers a candidate DFA C and the Teacher responds with *true* if $L(C) = U$ (where $L(C)$ denotes the language of C) or returns a counterexample, which is a string w s.t. $w \in L(C) \setminus U$ or $w \in U \setminus L(C)$.

6.2 Representing Executions as Words

A behavioral UML system with n state machines is denoted $\Gamma = M_1||...||M_n$. We assume state machines communicate only through events (all variables are local), and assume also that every RTC step is finite. These assumptions enable us to define sequences of events representing a single RTC step, which will be the letters of our alphabet (formally defined later).

Recall that according to the semantics of RTC steps, only the first transition may consume an event. An exception is the case of orthogonal regions that share the same trigger. As mentioned in Chapter 3, these transitions are executed simultaneously. Since the semantics of simultaneous execution is unclear, we assume that the actions of transitions in orthogonal regions labeled with the same trigger do not affect other transitions. That is, firing them in any order yields the same effect on the system.

For simplicity of presentation, we assume the following restrictions: (a) EQs are implemented as FIFOs, (b) Transitions with triggers do not generate events, and each transition may generate at most one event, (c) A state machine does not generate events to itself, (d) An event e cannot be generated by more than one state machine, and (e) Each state machine runs in a separate thread[1].

Given a state machine M, $Con(M)$ and $Gen(M)$ denote the events that M can consume and generate, respectively. An over-approximation of these sets can be found by static analysis. Recall that the events of a system include events sent by a state machine in the system denoted EV_{sys}, and events sent by the "environment" of the system denoted EV_{env}. For a system Γ, $EV_{sys}(\Gamma) = Gen(M_1) \cup ... \cup Gen(M_n)$, and $EV_{env}(\Gamma) = \{Con(M_1) \cup ... \cup Con(M_n)\} \setminus \{Gen(M_1) \cup ... \cup Gen(M_n)\}$. We denote $EV(\Gamma) = EV_{sys}(\Gamma) \cup EV_{env}(\Gamma)$. We assume the most general environment, that can send any environment event at any time. Note that the environment of a system might send events that will always be discarded by the target state machine. Since we are handling safety properties, such behaviors do not affect the satisfaction of the property, and we can therefore ignore them.

Let $\pi = C^0, step^0, C^1, ...$ be a computation of Γ. Based on the above assumptions on Γ, each $step^i$ in π can be labeled by at most one of $tr(e)$ and $gen(e)$, where $tr(e)$ denotes that when moving from C^i to C^{i+1} event e was dispatched to the target state machine (i.e., $step^i = DISP(j, e)$), and $gen(e)$ denotes that event e was either generated by a state machine in Γ if $e \in EV_{sys}(\Gamma)$ (i.e., $step^i = TRANS(j, t)$ and $GEN(e) \in act(t)$) or sent by the environment of Γ if $e \in EV_{env}(\Gamma)$ (i.e, $step^i = ENV(j, e)$). Note that it is possible that a *step* is denoted with neither (labeled with ϵ).

Note that events are always generated before they are dispatched. Also, since the EQs are FIFOs, then if e was generated before e' and the target of both events is M, then e will be dispatched before e'. Given a set of events EV, a sequence of labels over $\{tr(e), gen(e) | e \in EV\}$ is an *execution* over EV if it adheres to the above ordering requirements. For an execution ex, we define a mapping function that guarantees the ordering requirements.

[1]The case where several state machines run on the same thread is simpler, however presentation of both is cumbersome. We present only the more complex case.

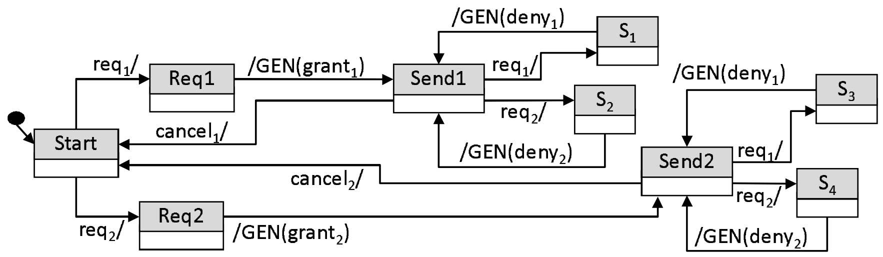


Figure 6.1: Example State Machine for Class *server*

Definition 6.1 *Let EV be a set of events, and let ex be an execution over EV. There exists a one-to-one function* $\gamma : \{i | f_i = tr(e)\} \to \mathbb{N}$ *that maps each* $tr(e)$ *occurrence in ex to its matching* $gen(e)$*:*

1. $\gamma(i) < i$
2. *If* $f_i = tr(e)$ *then* $f_{\gamma(i)} = gen(e)$.
3. *If there exist* $i < i'$ *s.t.* $f_i = tr(e)$, $f_{i'} = tr(e')$, *and* e, e' *are dispatched to the same* M_j, *then* $\gamma(i) < \gamma(i')$.

γ *is the* matching function *of ex.*

A computation matches an execution *ex* if *ex* is the sequence of non-ϵ labels of the computation. We denote the set of executions of Γ by $L_{ex}(\Gamma)$. Note that every computation matches a single execution. However, different computations may match the same execution.

Example 6.2 *Consider the system* $\Gamma = server||client$ *where server and client are presented in Figures 6.1 and 6.2, respectively.*
Then $gen(e_1), tr(e_1), gen(req_1), tr(req_1), gen(grant_1) \in L_{ex}(\Gamma)$. *However,* $gen(e_1), tr(e_1), gen(cancel_1) \notin L_{ex}(\Gamma)$, *since client, when in initial state, cannot generate* $cancel_1$ *after consuming* e_1.

From here on we do not address computations of a system, and consider only executions. We say that "execution *ex* satisfies a property φ" iff *all computations* that match *ex* satisfy φ. Let $EV' \subseteq EV$ be a set of events,

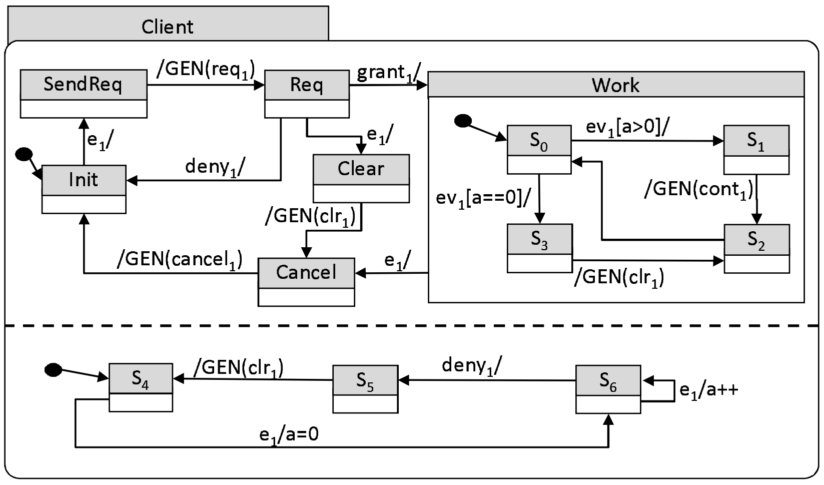


Figure 6.2: Example State Machine for Class *client*

and ex be an execution over EV. The *projection of ex w.r.t. EV'*, denoted $ex\downharpoonright_{EV'}$, is the projection of ex on $\{tr(e), gen(e) | e \in EV'\}$.

A system can include a single state machine M. This is a system where all events consumed by M are generated by the environment. By abuse of notation, we denote by $L_{ex}(M)$ the set of executions of a system that includes the single state machine M. The following lemma is a result of the fact that state machines communicate only through events.

Lemma 6.3 *Let $\Gamma = M_1 || ... || M_n$, let ex be an execution over $EV(\Gamma)$, and let γ be the matching function of ex. Then, $ex \in L_{ex}(\Gamma)$ iff for every $i \in \{1, ..., n\}$, $ex\downharpoonright_{EV(M_i)} \in L_{ex}(M_i)$.*

Proof: $\Longrightarrow$ Since state machines do not send events to themselves, then for every $i \in \{1, ..., n\}$, $EV_{env}(M_i) = Con(M_i)$. Consider $ex\downharpoonright_{EV(M_i)}$. Since state machines communicate only through events, and the events consumed are all generated by the environment, then $ex\downharpoonright_{EV(M_i)} \in L_{ex}(M_i)$.

$\Longleftarrow$ The behavior of each state machine is possible by the assumption. The fact that ex is an execution ensures ordering requirements, since there exists a correct mapping function $\gamma(ex)$. □

The following lemma is a direct result of Lemma 6.3

Lemma 6.4 *Let Sys be a system that includes state machine M. Then, $L_{ex}(Sys) \downharpoonright_{EV(M)} \subseteq L_{ex}(M)$.*

In order to later apply the L^* algorithm for learning assumptions on state machines, we first need to define an alphabet.

Definition 6.5 *Let M be a state machine. $\sigma = (t, (e_1, .., e_n))$ is in* the alphabet of M, *denoted $\Sigma(M)$, if $t \in Con(M)$ and there exists an RTC step of M that starts by consuming or discarding t, and continues by generating a sequence of events $e_1, ..., e_n$.*

Letters in $\Sigma(M)$ where n is 0 are denoted (t, ϵ). The idea behind our definition is that since the state machines in our systems communicate only through events, the alphabet maintains only the event information of the state machines. Since every RTC is finite, then an over-approximation of $\Sigma(M)$ can be found by static analysis (by traversing the graph of M), and the over-approximation is finite.

Example 6.6 *Let M = client (Figure 6.2). Then $\Sigma(M) = \{(e_1, (req_1)), (e_1, (clr_1, cancel_1)), (e_1, (cancel_1)), (e_1, \epsilon), (deny_1, \epsilon), (deny_1, (clr_1)), (grant_1, \epsilon), (ev_1, (clr_1)), (ev_1, (cont_1)), (ev_1, \epsilon)$. For example, $(e_1, (clr_1, cancel_1)) \in \Sigma(M)$, resulting from a possible RTC step that starts when M is in state Req. Also $(ev_1, \epsilon) \in \Sigma(M)$, since client can discard ev_1 (e.g., when in initial state).*

For a letter $\sigma = (t, (e_1, ..., e_n))$, $trig(\sigma) = t$ and $evnts(\sigma) = \{e_1, .., e_n\}$. We extend these notations to the alphabet Σ in the obvious way. Also, $EV(\Sigma) = trig(\Sigma) \cup evnts(\Sigma)$.

Following, we define the relation between executions and words. Intuitively, an execution ex matches a word w if the behavior of M in ex matches w.

Definition 6.7 *Let Γ be a system that includes state machine M, let $ex = f_1, f_2, ... \in L_{ex}(\Gamma)$, and let $w = \sigma_1, \sigma_2, ... \in (\Sigma(M))^*$. Let $\xi_1 = f'_1, f'_2, ...$ be the projection of ex on $\{tr(e) | e \in Con(M))\} \cup \{gen(e) | e \in Gen(M))\}$. Assume also $\xi_2 = f''_1, f''_2, ...$ is the sequence created from w by replacing $\sigma = (t, (e_1, ..., e_n))$ with $tr(t), gen(e_1), ..., gen(e_n)$. Then ex* matches *w, denoted $ex \rhd w$, iff $\xi_1 = \xi_2$.*

Note that an immediate result of the above definition is that if $ex \triangleright w$ where $w \in \Sigma^*$, then adding or removing from ex occurrences of events not in $EV(\Sigma)$ results in a sequence ex' s.t. $ex' \triangleright w$ still holds. Another important thing to note is that different executions can match the same word w. Thus w represents all the different executions under which the behavior of M matches w.

Example 6.8 *Consider execution* $ex = gen(e_1)$, $\boldsymbol{tr(e_1)}, \boldsymbol{gen(req_1)}, tr(req_1)$, $gen(grant_1)$, $gen(ev_1), \boldsymbol{tr(ev_1)} \in L_{ex}(server||client)$. *We denote with* ***bold*** *the parts of the execution that represent behavior of the client. For the word* $w = (e_1, req_1), (ev_1, \epsilon) \in (\Sigma(client))^*$, $ex \triangleright w$.
It also holds that for the execution $ex' = gen(e_1), gen(ev_1), \boldsymbol{tr(e_1)}, \boldsymbol{gen(req_1)}$ $, tr(req_1), \boldsymbol{tr(ev_1)}, gen(grant_1)$, $ex' \triangleright w$.

We consider safety properties over events, based on predicates such as $InQ(e)$, denoting that e is in the EQ, $BeforeQ(e, e')$ indicating that e is before e' in the EQ, and $gen(e)$ (or $tr(e)$), indicating that e is generated (or dispatched). We handle safety properties over LTL_x, which is the Linear-time Temporal Logic (LTL) [45] without the next-time operator. Model checking safety properties can be reduced to handling properties of the form AGp for a state formula p ([33]), which means that along every execution path, p globally holds. That is, every reachable configuration satisfies p. We therefore assume $\varphi = AGp$. The following theorem states that if an execution ex satisfies Gp, then adding or removing occurrences that do not influence p, results in an execution that satisfies Gp.

Theorem 6.9 *Let* ex *be an execution over* EV *and let* p *be a property over events* $EV' \subseteq EV$. *Then* $ex \models Gp$ *iff* $ex \downharpoonright_{EV'} \models Gp$.

Proof: Every occurrence of ex that does not exist in $ex \downharpoonright_{EV'}$ does not address an event in p. p considers properties that describe the contents of the event queues *only w.r.t. the events in* p. Thus, the property can only be affected by occurrences $tr(e)$ and $gen(e)$ where e is in p. $\square$

6.3 AG for State Machines

Our goal is to efficiently adapt the **AG** framework for UML state machines. Following, we first show that **Rule AG-UML** (presented in Section **??**)

holds for UML state machines, and present a framework for applying **Rule AG-UML** for UML state machines (Section 6.3.1). We give a detailed description of the framework in sections 6.3.2 and 6.3.3, discuss its correctness in Section 6.3.4, and present a performance analysis in Section 6.3.5.

6.3.1 A Framework For Employing Rule AG-UML and Its Correctness

First, we formally define the meaning of the two premises in **Rule AG-UML**: $[A]M\langle AGp\rangle$ holds iff for every $ex \in L_{ex}(A||M)$, $ex \models Gp$. $\langle true\rangle M[A]$ holds iff $EV(A) \subseteq EV(M)$ and for every $ex \in L_{ex}(M)$, $ex\downharpoonright_{EV(A)} \in L_{ex}(A)$.

Theorem 6.10 *Let M_1, M_2 and A be state machines s.t. $EV(A) \subseteq EV(M_2)$, let p be a property over events $EV' \subseteq (EV(A) \cup EV(M_1))$, and let $\varphi = AGp$. Then* **Rule AG-UML** *is sound.*

Proof: Assume by means of negation that *Step* 1 and *Step* 2 hold, however $\langle true\rangle M_1||M_2\langle AGp\rangle$ does not hold.
This means that there exists an execution $ex \in L_{ex}(M_1||M_2)$ s.t. $ex \not\models Gp$. By Lemma 6.3, $ex\downharpoonright_{EV(M_2)} \in L_{ex}(M_2)$. Thus, since *Step* 2 holds, $ex\downharpoonright_{EV(A)} \in L_{ex}(A)$. It also holds (by Lemma 6.3) that $ex\downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$.
Since $ex\downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex\downharpoonright_{EV(A)} \in L_{ex}(A)$, then by Lemma 6.3 $ex\downharpoonright_{EV(A)\cup EV(M_1)} \in L_{ex}(A||M_1)$, and since *Step* 1 holds, we can conclude that $ex\downharpoonright_{EV(A)\cup EV(M_1)} \models Gp$. Based on Theorem 6.9, $ex \models Gp$ as well. A contradiction. We then conclude that $\langle true\rangle M_1||M_2\langle AGp\rangle$ holds, which means that **Rule AG-UML** is sound. □

We use L^* to iteratively construct assumptions A, until either both premises of **Rule AG-UML** hold, or until a real counterexample is found. L^* learns a language over *words*, where each word represents an equivalence class of executions.

In order to apply the L^* algorithm we define Σ, the alphabet of the language learned by L^*. Intuitively, Σ includes details of M_2 that are relevant for proving φ with M_1. The alphabet $\Sigma(M_2)$ (Definition 6.5) may include events of M_2 which are irrelevant. We therefore restrict $\Sigma(M_2)$ to Σ by keeping only elements of $EV(M_2)$ that are relevant for the interaction with M_1 and for φ.

Definition 6.11 *Let $M_1||M_2$ be a system and φ be a safety property. Σ,* the assumption alphabet of M_2 w.r.t. M_1 and φ, *is the maximal set, s.t. for every $\sigma = (t, (e_{i_1}, ..., e_{i_n})) \in \Sigma$ there exists $\sigma' = (t, (e_1, ..., e_m)) \in \Sigma(M_2)$ s.t. both requirements hold:*

1. *$(e_{i_1}, ..., e_{i_n})$ is the maximal sub-vector of $(e_1, ..., e_m)$ (i.e., $1 \leq i_1 < i_2 < ... < i_n \leq m$) where each e_{i_j} is consumed by M_1 or part of the property φ.*
2. *If $t \in EV_{env}(M_1||M_2)$ and $n = 0$: (t, ϵ) is included in Σ iff either t is part of φ or there exists $\sigma_1 = (t, (e'_1, ..., e'_k)) \in \Sigma$ s.t. $k > 0$.*

Example 6.12 *Let $\Gamma = server||client$ where server is M_1 and client is M_2, and let $\varphi = AG(\neg(InQ(grant_1) \wedge InQ(deny_1))$. The events of φ are $grant_1$ and $deny_1$. Σ, the assumption alphabet of M_2 w.r.t. M_1 and φ, is $\{(e_1, (req_1)), (e_1, \epsilon), (grant_1, \epsilon), (deny_1, \epsilon), (e_1, (cancel_1))\}$. Note that although $(deny_1, (clr_1)) \in \Sigma(client)$, since clr_1 is not consumed by the server and is not part of φ, then it is not included in Σ.*
Similarly, $(e_1, (clr_1, cancel_1)) \in \Sigma(client)$, but only $(e_1, (cancel_1)) \in \Sigma$. Note also that Σ includes all *the interface information between client and server. Thus, $(e_1, (req_1)) \in \Sigma$, although neither e_1 nor req_1 are part of φ.*

We define the notion of *weakest assumption* in the context of state machines.

Definition 6.13 *A language $A_w \subseteq \Sigma^*$ is the* weakest assumption w.r.t. M_1 and φ *if the following holds: $w \in A_w$ iff for every execution ex over $EV(\Sigma) \cup EV(M_1)$, if $ex \rhd w$ and $ex \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$, then $ex \models Gp$.*

Assume we could construct a state machine M_{A_w} that *represents* A_w. That is, for every execution ex over $EV(\Sigma)$, $ex \in L_{ex}(M_{A_w})$ iff there exists $w \in A_w$ s.t. $ex \rhd w$. Then, M_{A_w} describes exactly those executions over Σ that when executed with M_1 do not violate Gp. The following theorem states that $\langle true \rangle M_1||M_2 \langle \varphi \rangle$ holds iff every execution of M_2 matches a word in A_w.

Theorem 6.14 *$\langle true \rangle M_1||M_2 \langle \varphi \rangle$ holds iff for every execution $ex \in L_{ex}(M_2)$, there exists $w \in A_w$ s.t. $ex \rhd w$, where A_w is the weakest assumption w.r.t. M_1 and φ.*

Proof: $\Longleftarrow$: We assume that for every execution $ex \in L_{ex}(M_2)$, there exists $w \in A_w$ s.t. $ex \triangleright w$ and show that $\langle true \rangle M_1 || M_2 \langle \varphi \rangle$.

Let ex be an execution in $L_{ex}(M_1||M_2)$. We show that $ex \models Gp$ ($\varphi = AGp$).
Since we know that $L_{ex}(M_1||M_2) \downharpoonleft_{EV(M_2)} \subseteq L_{ex}(M_2)$ (Lemma 6.4), then $ex \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$. From the assumption, there exists $w \in A_w$ s.t. $ex \downharpoonleft_{EV(M_2)} \triangleright w$. Therefore it holds that $ex \triangleright w$, and also $ex \downharpoonleft_{EV(\Sigma) \cup EV(M_1)} \triangleright w$. We denote $ex' = ex \downharpoonleft_{EV(\Sigma) \cup EV(M_1)}$, and thus
(1) ex' is an execution over $EV(\Sigma) \cup EV(M_1)$
(2) $ex' \triangleright w$ for $w \in A_w$.
(3) Since $ex \in L_{ex}(M_1||M_2)$, then $ex \downharpoonleft_{EV(M_1)} \in L_{ex}(M_1)$. Clearly, $ex' \downharpoonleft_{EV(M_1)} = ex \downharpoonleft_{EV(M_1)}$ and thus: $ex' \downharpoonleft_{EV(M_1)} \in L_{ex}(M_1)$.
We can then conclude, from the definition of A_w, that $ex' \models Gp$, and based on Theorem 6.9, $ex \models Gp$ as well.

$\Longrightarrow$: Assume by way of contradiction there exists an execution $ex \in L_{ex}(M_2)$ and no word $w \in A_w$ s.t. $ex \triangleright w$. Thus, there exists $w \in \Sigma^* \setminus A_w$ s.t. $ex \triangleright w$ (i.e., $w \notin A_w$). We show that this means that there exists an execution $ex' \in L_{ex}(M_1||M_2)$ s.t. ex' violates Gp.

If $w \notin A_w$ then there exists an execution ex_1 over $EV(\Sigma) \cup EV(M_1)$ s.t. $ex_1 \downharpoonleft_{EV(M_1)} \in L_{ex}(M_1)$, $ex_1 \triangleright w$ and $ex_1 \not\models Gp$.

Recall, $ex \in L_{ex}(M_2)$ and $ex \triangleright w$. We construct the execution ex' as the joint execution of ex_1 and ex. Note that the construction of ex' is not straightforward; ex_1 and ex both match w, however the other parts of the executions might not match, i.e., the interleaving of M_2 and the environment in ex may be different from the interleaving of M_1 and Σ in ex_1. Our construction of ex' actually shows that there exists an interleaving that is possible by both M_1 and M_2, and that still violates Gp.

We first construct the sequence of events generated for M_2 on ex. We denote the γ function that associates the index of the dispatching of event with the index of its generation on execution ex as $\gamma(ex)$.
Let $0 < i_1 < i_2 < ... < i_n$ be the indices in ex where an event was dispatched to M_2 (i.e., for $j \in \{1, ..., n\}$: $f_{i_j} = tr(e)$). Then for every $j \in \{1, ..., n-1\}$, $\gamma(ex)(i_j) < \gamma(ex)(i_{(j+1)})$ (by the definition of $\gamma(ex)$).

Similarly, we construct a sequence from ex_1 that includes the events that are *not dispatched to* M_1. I.e., the events which are triggers of Σ:
Let $0 < k_1 < k_2 < ... < k_m$ be the indices in ex_1 where an event was dis-

patched not to M_1 (i.e., for $j \in \{1, ..., m\}$: $f_{k_j} = tr(e)$ and $e \in trig(\Sigma)$). Then for every $j \in \{1, ..., m-1\}$, $\gamma(ex_1)(k_j) < \gamma(ex_1)(k_{(j+1)})$ (by the definition of $\gamma(ex_1)$).

It is important to note that the sequence $seq(ex_1) = f_{\gamma(ex_1)(k_1)}, ...,$ $f_{\gamma(ex_1)(k_m)}$ is a sub-sequence of $seq(ex) = f_{\gamma(ex)(i_1)}, ..., f_{\gamma(ex)(i_n)}$, since $ex \rhd w$ and $ex_1 \rhd w$. We define a one-to-one function $\hat{\gamma} : \{\gamma(ex_1)(k_1), ..., \gamma(ex_1)(k_m)\} \rightarrow \{\gamma(ex)(i_1), ..., \gamma(ex)(i_n)\}$ that matches each element in $seq(ex_1)$ with its matching element in $seq(ex)$.

Note also that elements in $seq(ex)$ that are not in $seq(ex_1)$ are events that are not generated by M_1 (if they were generated by M_1 then they would have been in Σ). Thus, these events are generated by the *environment of* $M_1||M_2$, and we can therefore assume that they can be generated at any time on an execution of $M_1||M_2$.

Construction of ex': First, we want to have a projection of ex that includes only the behaviors of M_2 (i.e., without the events generated by the environment of M_2). We denote this as $\widetilde{ex}$. $\widetilde{ex}$ is the projection of ex on $\{tr(e)|e \in trig(\Sigma(M_2))\} \cup \{gen(e)|e \in evnts(\Sigma(M_2))\}$. Note that $\widetilde{ex} \rhd w$ (since $\widetilde{ex}$ includes all elements in w).

Intuitively, ex' follows ex_1. When ex_1 executes a behavior of Σ, then we replace that behavior with the behavior of M_2 based on ex (taken from $\widetilde{ex}$). We initiate a counter i to 0 that points to the place in ex_1 we are at. We initiate a counter cnt to 0 that points to the place in $\widetilde{ex}$ we are at. We denote the elements in $\widetilde{ex}$ as f'_i.

For every element f_i from ex_1 execute one of the following:

1. If $f_i = tr(e)$ or $f_i = gen(e)$ and $e \in trig(\Sigma(M_1))$ (that is, e is dispatched to M_1 or generated for M_1): then add f_i to ex'.

2. If $f_i = gen(e)$ and $e \in trig(\Sigma)$: Let $i = \gamma(ex_1)(k_j)$ and let $i' = \gamma(ex_1)(k_{(j-1)})$. Also, let $g = \hat{\gamma}(i)$ and $g' = \hat{\gamma}(i')$. By the definition of $seq(ex)$, the events on $seq(ex)$ between element $f_{g'}$ and element f_g are environment events of $M_1||M_2$. Add to ex' all these elements: for every $j \in \{g'+1, ..., g\}$, if $\tilde{f}_j \in seq(ex)$ then $\tilde{f}_j$ is added to ex'.

3. If $f_i = tr(e)$ and $e \in trig(\Sigma)$ or $f_i = gen(e)$ and $e \in evnts(\Sigma)$: Need to add relevant elements from $\widetilde{ex}$. while ($f'_{cnt} \neq f_i$) { add f'_{cnt} to ex'; $cnt++$ }. When done, add f_i to ex'.

$ex' \downharpoonright_{EV(M_1)} = ex_1 \downharpoonright_{EV(M_1)}$ (by construction). Since $ex_1 \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ then also $ex' \downharpoonright_{E(M_1)} \in L_{ex}(M_1)$.

Note that since a state machine cannot generate events to itself, then if for some execution $ex \in L_{ex}(M_2)$ the following holds: Let $w' \in \Sigma(M_2)^*$ s.t. $ex \triangleright w'$. Then every execution $\hat{ex}$ over $EV(M_2)$, if $\hat{ex} \triangleright w'$ then $\hat{ex} \in L_{ex}(M_2)$. This is since ex and $\hat{ex}$ differ in the interleaving of the environment events sent to M_2.
We can therefore conclude the following: Let $w' \in \Sigma(M_2)^*$ s.t. $ex \triangleright w'$. By definition of $\widetilde{ex}$, $\widetilde{ex} \triangleright w'$. By construction (since we copy exactly $\widetilde{ex}$ to ex'), then $ex' \downharpoonright_{E(M_2)} \triangleright w'$. Also, due to the construction of ex', the order of generated events dispatched to M_2 follows that an event was generated before it was dispatched (item 2 in the construction). Thus $ex' \downharpoonright_{E(M_2)} \in L_{ex}(M_2)$.

By construction of ex', ex' is an execution over $EV(M_1) \cup EV(M_2)$. Thus, by Lemma 6.3, $ex' \in L_{ex}(M_1||M_2)$. Now, recall that $ex_1 \not\models Gp$. ex' adds to ex_1 only behaviors that do not effect Gp. Thus, we can conclude that $ex' \not\models Gp$ as well. $\square$

From the definition of A_w and from the above theorem we conclude the following corollary, which states that **Rule AG-UML** holds if we replace A with M_{A_w}.

Corollary 6.15 *Let A_w be the weakest assumption w.r.t. M_1 and φ. Assume there exists a state machine M_{A_w} that represents A_w. Then* **Rule AG-UML** *holds when replacing A with M_{A_w}.*

The goal of L^* is therefore to learn A_w. To automate L^* in our setting we now show how to construct a Teacher that answers membership and conjecture queries. The Teacher answers queries by "translating" the queries into state machines, and verifying properties on state machines via a model checker for behavioral UML systems. The model checker must be able to always return a definite answer ($true$ or $false$) for properties of type AGp. Also, when answering $false$ it should give a counterexample. Model checkers for behavioral UML systems verify the behavior w.r.t. system configurations. Thus, a counterexample is a computation of the system. It is straightforward to translate the counterexample into a counterexample execution or word. Although our goal is to learn A_w, our automatic framework may stop with a definite $true$ or $false$ answer before A_w is constructed.

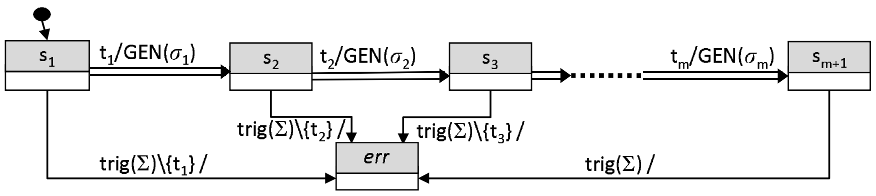


Figure 6.3: $M(w)$ constructed for w

For a membership query on w, the Teacher constructs a state machine for w, and checks if, when executed with M_1, φ is violated. For conjecture queries, the Teacher constructs a state machine $A(C)$ from conjecture C, and verifies *Step* 1 and *Step* 2 of **Rule AG-UML** w.r.t. $A(C)$.

From now on, in our following constructions, we sometimes include an *err* state in state machines. For simplicity of presentation, for a given a system Γ where some of its state machines include *err* state, $L_{ex}(\Gamma)$ represents only the executions that do not reach *err* state on any of its state machines.

6.3.2 Membership Queries

To answer a membership query for $w \in \Sigma^*$, the Teacher must return *true* iff $w \in A_w$. The Teacher creates a state machine $M(w)$ s.t. $\Sigma(M(w)) \subseteq \Sigma$. $M(w)$ is constructed s.t. for every ex over $EV(\Sigma) \cup EV(M_1)$: $ex \in L_{ex}(M(w)||M_1)$ iff $ex \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex \rhd w$. If this holds, then (by the definition of A_w in Definition 6.13) $w \in A_w$ iff for every execution $ex \in L_{ex}(M(w)||M_1)$, $ex \models Gp$.

Let $w = \sigma_1, \sigma_2, ..., \sigma_m$ and let $\sigma_i = (t_i, (e^i_1, e^i_2, ..., e^i_{k_i}))$, for $i \in \{1, ..., m\}$. The state machine $M(w)$ is presented in Figure 6.3. A transition labeled with a set of triggers T (e.g., the transition from s_1 to *err*) is a shorthand for a set of transitions, each labeled with a single trigger $t \in T$. For $\sigma = (t, (e^1, ..., e^k))$, a compound transition, denoted as a double arrow $\Rightarrow$, labeled with $trig[grd]/GEN(\sigma)$ is a shorthand for a sequence of states and transitions, where the first transition is labeled with $trig[grd]$, the second is labeled with action $GEN(e^1)$, the third with action $GEN(e^2)$, etc. The idea behind splitting the compound transition into intermediate states is to enable all possible interleaving between $M(w)$ and M_1, thus ensuring that every execution over $EV(\Sigma) \cup EV(M_1)$ that represents an execution of M_1 and matches w is indeed a possible execution of $M(w)||M_1$.

We explicitly define at each state s_i the behavior of $M(w)$ in response to any possible event $t \in trig(\Sigma)$. Not specifying such a behavior implies that if t is dispatched to $M(w)$ then $M(w)$ discards t and remains in the same state. This is an undesired behavior of $M(w)$, which is supposed to execute w with *no additional intermediate letters*. Thus, transitions that do not match w are sent to state err. The following theorem describes the executions of $M(w)$.

Theorem 6.16 *Let $M(w)$ be the state machine constructed for word $w \in \Sigma^*$. For every execution ex over $EV(\Sigma)$: $ex \in L_{ex}(M(w))$ iff there exists a prefix w' of w s.t. $ex \rhd w'$.*

Proof: $\Longrightarrow$ Recall that by the definition of L_{ex}, if $ex \in L_{ex}(M(w))$ then ex does not reach state err. Thus, for every execution $ex \in L_{ex}(M(w))$, the corresponding behavior of $M(w)$, $ex \downharpoonleft_{M(w)}$, is a prefix of w. Therefore, for every execution ex over $EV(\Sigma)$, if $ex \in L_{ex}(M(w))$ then $ex \rhd w'$ and w' is a prefix of w.

$\Longleftarrow$ Let ex be an execution over $EV(\Sigma)$. Assume $ex \rhd w'$ and w' is a prefix of w. Note that by the definition of $ex \rhd w'$, ex includes exactly the occurrences that match w' and $gen(e)$ occurrences for $tr(e) \in ex$. Also, since ex is an execution over $EV(\Sigma)$, then there exists a mapping function γ on ex.

Clearly, $M(w)$ has an execution ex' s.t. $ex' \rhd w'$. $M(w)$ is constructed s.t. every transition either consumes a single event or generates a single event. Since the environment can sent events at any time, we conclude that every execution over $EV(\Sigma)$ that matches w' is available on $M(w)$. Thus, $ex \in L_{ex}(M(w))$. $\Box$

Once $M(w)$ is constructed, the Teacher model checks $M(w)||M_1 \models AG(p \vee IsIn(err))$, where $IsIn(s)$ denotes that s is a part of the current state of the system. The model checker returns $true$ iff for every execution one of the following holds: (1) the execution does not reach state err, i.e. the execution matches a prefix of w, and p is satisfied along the entire execution, or (2) the execution reaches state err, meaning that the execution does not match w and therefore we do not need to require p. Note that it is ok to require p on a prefix leading to state err, since A_w is prefix closed for safety properties. The Teacher returns $true$, indicating $w \in A_w$ iff the model checker returns $true$. The following theorem defines the correctness of the Teacher.

Theorem 6.17 *$M(w)||M_1 \models AG(p \vee IsIn(err))$ iff $w \in A_w$.*

Proof: Notice that $M(w)||M_1 \models AG(p \vee IsIn(err))$ iff for every execution $ex \in L_{ex}(M(w)||M_1)$, $ex \models Gp$. This is an immediate result of the definition of $L_{ex}(Sys)$ that includes only executions that do not reach state err.

If we show that for every ex over $EV(\Sigma) \cup EV(M_1)$: $ex \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex \triangleright w$ iff $ex \in L_{ex}(M(w)||M_1)$. Then from Definition 6.13 we can conclude that $w \in A_w$ iff for every execution $ex \in L_{ex}(M(w)||M_1)$, $ex \models Gp$, and this is what we need to prove.

Let ex be an execution over $EV(\Sigma) \cup EV(M_1)$. $ex \triangleright w$ iff $ex \downharpoonright_{EV(\Sigma)} \triangleright w$ iff (Theorem 6.16) $ex \downharpoonright_{EV(\Sigma)} \in L_{ex}(M(w))$. Thus, $ex \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex \triangleright w$ iff $ex \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex \downharpoonright_{EV(\Sigma)} \in L_{ex}(M(w))$. From Lemma 6.3 we can conclude that $ex \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex \triangleright w$ iff $ex \in L_{ex}(M(w)||M_1)$. $\square$

6.3.3 Conjecture Queries

A conjecture of the L^* algorithm is a DFA over Σ. Our framework first transforms this DFA, C, into a state machine $A(C)$. Then, *Step* 1 and *Step* 2 are applied in order to verify the correctness of $A(C)$.

Constructing a State Machine From a DFA:

A DFA is a five tuple $C = (Q, \alpha, \delta, q_0, F)$, where Q is a finite non-empty set of states, α is the alphabet, $\delta \subseteq Q \times \alpha \times Q$ is a deterministic transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states. For a string w, $\delta(q, w)$ denotes the state that C arrives at after reading w, starting from state q. A string w is *accepted* by C iff $\delta(q_0, w) \in F$. The language of C, denoted $L(C)$, is the set $\{w|\delta(q_0, w) \in F\}$. The DFAs returned by the L^* algorithm are complete, minimal, and prefix-closed. Thus they contain a single non-accepting state, q_{err}, and for every $\sigma \in \alpha$ and $q \in Q$, $\delta(q, \sigma)$ is defined.

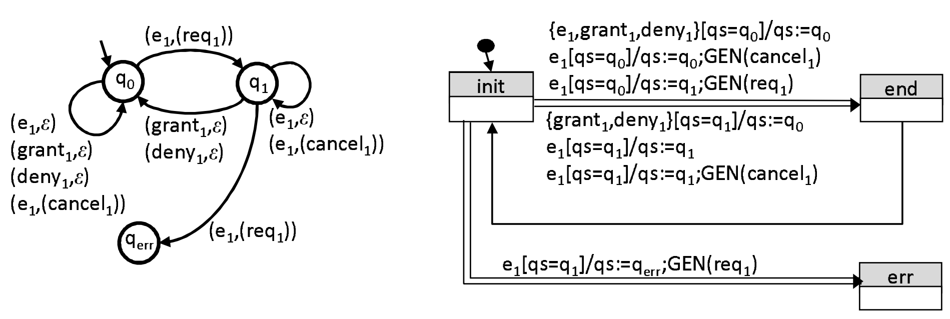

(e1,(req1))
q0
q1
(e1,ε)
(grant1,ε)
(deny1,ε)
(e1,(cancel1))
(grant1,ε)
(deny1,ε)
(e1,ε)
(e1,(cancel1))
qerr
(e1,(req1))
{e1,grant1,deny1}[qs=q0]/qs:=q0
e1[qs=q0]/qs:=q0;GEN(cancel1)
e1[qs=q0]/qs:=q1;GEN(req1)
init
end
{grant1,deny1}[qs=q1]/qs:=q0
e1[qs=q1]/qs:=q1
e1[qs=q1]/qs:=q1;GEN(cancel1)
e1[qs=q1]/qs:=qerr;GEN(req1)
err

Figure 6.4: Conjecture DFA C (left) and state machine $A(C)$ (right)

The alphabet α of the DFA in our framework is exactly Σ. Given a DFA $C = (Q, \Sigma, \delta, q_0, Q \setminus \{q_{err}\})$, we construct a state machine $A(C)$ where $EV(A(C)) = EV(\Sigma)$. We then show that $A(C)$ *represents* $L(C)$, i.e., for every execution ex over $EV(\Sigma)$, $ex \in L_{ex}(A(C))$ iff there exists $w \in L(C)$ s.t. $ex \rhd w$.

Definition 6.18 *[$A(C)$ **Construction**] Let $C = (Q, \Sigma, \delta, q_0, Q \setminus \{q_{err}\})$. $A(C)$ includes 3 states: init, end and err, where init is the initial state. $A(C)$ includes a single variable qs whose domain is Q, initialized to q_0. $A(C)$ has the following transitions:*

1. *For every $q \in Q \setminus \{q_{err}\}$ and $\sigma = (t, (e_1, .., e_n)) \in \Sigma$ where $\delta(q, \sigma) = q'$ and $q' \neq q_{err}$, add a compound transition labeled with $t[qs = q]/qs := q'; GEN(\sigma)$ from init to end*

2. *For every $q \in Q \setminus \{q_{err}\}$ and $\sigma = (t, (e_1, .., e_n)) \in \Sigma$ where $\delta(q, \sigma) = q_{err}$, add a compound transition labeled with $t[qs = q]/qs := q'; GEN(\sigma)$ from init to err*

3. *Add a transition with no trigger, guard or action from end to init.*

Example 6.19 *For $\Gamma = server||client$ and $\varphi = AG(\neg(InQ(grant_1) \wedge InQ(deny_1))$, the conjecture DFA C returned from the L^* algorithm, and state machine $A(C)$ representing $L(C)$, are presented in Figure 6.4.*

The construction ensures that for every $t \in trig(\Sigma)$ and for every $q \in Q \setminus \{q_{err}\}$ there exists a transition from *init* with trigger t and guard $qs = q$.

That is, as long as $A(C)$ is at state *init* in the beginning of an RTC step, it does not discard events. Also, according to the semantics of state machines, every RTC step that starts at state *init*, either moves to state *err*, which is a sink state, or moves to state *end* and returns to state *init*. The following theorem states that $A(C)$ is indeed a state machine representing $L(C)$.

Theorem 6.20 *Let $A(C)$ be the state machine constructed for DFA C. For every execution ex over $EV(\Sigma)$: $ex \in L_{ex}(A(C))$ iff there exists $w \in L(C)$ s.t. $ex \rhd w$.*

Proof: The proof of this theorem is similar to the proof of theorem 6.16:

$\Longleftarrow$ Let ex be an execution over $EV(\Sigma)$. Assume $ex \rhd w$ and $w \in L(C)$. Clearly, by construction of $A(C)$, $A(C)$ has an execution ex' s.t. $ex' \rhd w$. $A(C)$ is constructed s.t. every transition either consumes a single event or generates a single event. Since the environment can send events at any time, we conclude that every execution over $EV(\Sigma)$ that matches w is available on $A(C)$. Thus, $ex \in L_{ex}(A(C))$.

$\Longrightarrow$ Let ex be an execution in $L_{ex}(A(C))$. By definition, it does not pass through state err. Assume by way of contradiction that there exists $w \in \Sigma^*$ s.t. $ex \rhd w$ and $w \notin L(C)$. $L(C)$ is prefix closed. We then look at the longest prefix w' of w s.t. $w' \in L(C)$. Based on the construction of $A(C)$, the RTC step executed after w' matches a transition in C to a non-error state, and thus w' can be extended to a longer prefix of w included in $L(C)$. A contradiction. We conclude that $w \in L(C)$. $\square$

After creating $A(C)$, the Teacher uses two oracles and a counterexample analysis to answer conjecture queries.

Check $[A(C)]M_1\langle\varphi\rangle$:

Oracle 1 performs *Step* 1 in the compositional rule by model checking $A(C)||M_1 \models AG(p \vee IsIn(err))$. If the model checker returns $false$ with a counterexample execution cex, the Teacher informs L^* that the conjecture is incorrect, and gives it the word $w \in \Sigma^*$ s.t. $cex \rhd w$ to witness this fact ($w \in L(C)$ and $w \notin A_w$). If the model checker returns $true$, indicating that $[A(C)]M_1\langle\varphi\rangle$ holds, then the Teacher forwards $A(C)$ to Oracle 2.

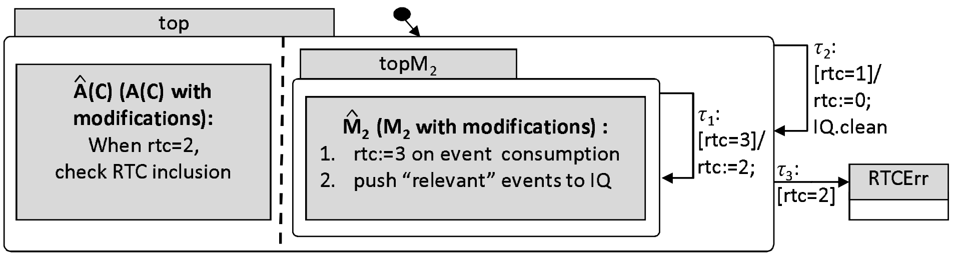

top
Â(C) (A(C) with modifications):
When rtc=2, check RTC inclusion
topM2
M̂2 (M2 with modifications) :
1. rtc:=3 on event consumption
2. push "relevant" events to IQ
τ1: [rtc=3]/ rtc:=2;
τ2: [rtc=1]/ rtc:=0; IQ.clean
τ3: [rtc=2]
RTCErr

Figure 6.5: General scheme for $\mathcal{M}(M_2, A(C))$ created from $A(C)$ and M_2

Check $\langle true\rangle M_2[A(C)]$:

Oracle 2 preforms *Step* 2 in the compositional rule. That is, it checks that for every execution $ex \in L_{ex}(M_2)$, $ex \downharpoonright_{EV(A(C))} \in L_{ex}(A(C))$. Note that this is a language containment check. In state machines there is no known algorithm for checking language containment. We present here a method for this check in the special case where the abstract state machine is the state machine $A(C)$ previously defined. *Step* 2 is done by constructing a single state machine, and applying model checking on the resulting state machine.

Given the state machines M_2 and $A(C)$, Oracle 2 constructs a new state machine, $\mathcal{M}(M_2, A(C))$, that is composed from modifications of M_2 and $A(C)$ as two orthogonal regions. $\mathcal{M}(M_2, A(C))$ is constructed so that the behavior of M_2 is *monitored* by $A(C)$ after every RTC step. $\mathcal{M}(M_2, A(C))$ includes a synchronization mechanism, so that when an event is dispatched, first the region that includes M_2 executes the RTC step. When it finishes, the region that includes $A(C)$ executes its step *only if* $A(C)$ *has a behavior that matches* M_2. If $A(C)$ does not have a matching behavior, then $\mathcal{M}(M_2, A(C))$ moves to an error state, indicating that $\langle true\rangle M_2[A(C)]$ does not hold. The general structure of $\mathcal{M}(M_2, A(C))$ is presented in Figure 6.5.

From here on, we denote the variation of M_2 and $A(C)$ that are regions in $\mathcal{M}(M_2, A(C))$ as $\widehat{M_2}$ and $\widehat{A(C)}$, respectively. We add a local queue, IQ, and two local variables, rtc and tr, to $\mathcal{M}(M_2, A(C))$. tr "records" the event e dispatched to $\mathcal{M}(M_2, A(C))$, if $e \in trig(\Sigma)$. IQ "records" events generated by $\widehat{M_2}$ which are from $evnts(\Sigma)$. Whenever $\widehat{M_2}$ generates an event from $evnts(\Sigma)$, it also pushes the event to IQ. $\widehat{A(C)}$ will, in turn, check if it has a matching behavior by observing IQ. rtc is used for fixing the order of execution along an RTC step of $\mathcal{M}(M_2, A(C))$. It is initialized to 0, and as long as the monitoring is successful, the value of rtc at the end of the RTC step of $\mathcal{M}(M_2, A(C))$ is 0. $rtc = 3$ indicates that $\widehat{M_2}$ is executing an RTC step that should be monitored. $rtc = 2$ indicates that $\widehat{M_2}$ finished its execution, and $\widehat{A(C)}$ can monitor the behavior. $rtc = 1$ indicates that the monitoring step of $\widehat{A(C)}$ was successful, i.e., $\widehat{A(C)}$ has a behavior that matches $\widehat{M_2}$. If the monitoring of $\widehat{A(C)}$ failed, then rtc at the end of the RTC step is 2, indicating an error.

The following modifications are applied to M_2 for constructing $\widehat{M_2}$: Set rtc to 3 on transitions that consume event $e \in trig(\Sigma)$, and add $IQ.push(e')$ on transitions that generate event $e' \in gen(\Sigma)$.
The following modifications are applied to $A(C)$ (Definition 6.18) for constructing $\widehat{A(C)}$:

1. Add a new state called *step* to $A(C)$, and for every $t \in trig(\Sigma)$, add a transition from *init* to *step* labeled $t/tr := t$.

2. Every compound transition from *init* to *end* labeled with:
   $t[qs = q]/qs := q'; GEN(e_1); ...; GEN(e_n)$ s.t. $n > 0$
   is replaced with a transition from *step* to *end* labeled with:
   $[tr = t \wedge qs = q \wedge rtc = 2 \wedge IQ = (e_1, ..., e_n)]/qs := q'; rtc := 1$

3. Every compound transition from *init* to *end* labeled with: $t[qs = q]/qs := q'$ (no event generation), is replaced with a transition from *step* to *end* labeled with: $[tr = t \wedge qs = q \wedge ((rtc = 2 \wedge IQ = ()) \vee rtc = 0)]/\ qs := q'; rtc := 1$

4. Every compound transition from *init* to *err* labeled with:
   $t[qs = q]/qs := q'; GEN(e_1); ...; GEN(e_n)$ s.t. $n > 0$
   is replaced with a transition from *step* to *err* labeled with:
   $[tr = t \wedge qs = q \wedge rtc = 2 \wedge IQ = (e_1, ..., e_n)]/qs := q'; rtc := 2$

5. Every compound transition from *init* to *err* labeled with: $t[qs = q]/qs := q'$ (no event generation), is replaced with a transition from *step* to *err* labeled with: $[tr = t \wedge qs = q \wedge ((rtc = 2 \wedge IQ = ()) \vee rtc = 0)]/\ qs := q'; rtc := 2$

If $\widehat{A(C)}$ is at state *step* and $rtc = 0$ holds, then $\widehat{M_2}$ discarded the event t in the current RTC step. $\widehat{A(C)}$ has a matching behavior if it has a behavior

that consumes t and does not generate events. The transitions described in (3) and (5) monitor RTC steps of $\widehat{M_2}$ that consume event t and do not generate any events, and also RTC steps that discard t. Note that items (2) and (4) (respectively, (3) and (5)) are distinct in the target state (*end* or *err*) and in the assignment to *rtc* on the action. The transitions in (2) and (3) monitor RTC steps that are legal in $\widehat{A(C)}$, and transitions in (4) and (5) monitor RTC steps that are not legal in $\widehat{A(C)}$.
The correctness of our construction is captured in the following theorem.

Theorem 6.21 *Let ex be an execution in $L_{ex}(\mathcal{M}(M_2, A(C)))$, and let ex' be the maximal prefix of ex that does not include the suffix where $IsIn(RTCErr)$ holds (if there exists such a suffix). Then the following holds: ex reaches state $RTCErr$ iff $ex' \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$ and $ex' \downharpoonleft_{EV(A(C))} \notin L_{ex}(A(C))$.*

Note that since *RTCErr* is a sink state, then if an execution *ex* on $\mathcal{M}(M_2, A(C))$ reaches state *RTCErr*, then every event sent to $\mathcal{M}(M_2, A(C))$ will be discarded.

Proof: We first prove by induction on the number of RTC steps that for every execution of $\mathcal{M}(M_2, A(C))$, at the end of every RTC step the following holds: Variable *rtc* is 0 iff the execution is not at state *RTCErr* and $\widehat{A(C)}$ is at state *init*.
Base: For $k = 0$, by construction *rtc* is 0, and also the initial state of $\mathcal{M}(M_2, A(C))$ does not include *RTCErr*. Moreover, the initial state of $\widehat{A(C)}$ is *init*.
Step: For any execution $ex \in \mathcal{M}(M_2, A(C))$, assume the property holds on *ex* after k RTC steps. If $rtc \neq 0$ after k RTC steps, then based on the induction assumption, *ex* is at state *RTCErr*. Since *RTCErr* is a sink state, then *ex* remains at this state. Also, by construction, the value of *rtc* cannot change, and thus remains not 0.

If $rtc = 0$ after k RTC steps: assume the event dispatched to $\mathcal{M}(M_2, A(C))$ is e. By construction, the current state of $\mathcal{M}(M_2, A(C))$ includes a state from $\widehat{M_2}$. Also, based on the assumption $\widehat{A(C)}$ is at state *init*. One of the following behaviors are possible (based on the semantics of state machines):

- If $e \notin trig(\Sigma)$: By construction of $\widehat{M_2}$, *rtc* does not change during the RTC step of $\widehat{M_2}$. Also, there is no transition in $\widehat{A(C)}$ with trigger e, and thus $\widehat{A(C)}$ discards e and remains at state *init*. Thus, after the RTC step terminates on $\widehat{M_2}$ no other transition is enabled in $\mathcal{M}(M_2, A(C))$. The RTC then terminates and rtc remains with value 0, $\mathcal{M}(M_2, A(C))$ does not reach state $RTCErr$, and $\widehat{A(C)}$ is at $init$.

- If $e \in trig(\Sigma)$:
  - If $\widehat{M_2}$ consumes e, then by construction of $\widehat{M_2}$, the transition of $\widehat{M_2}$ that consumes the event sets rtc to 3. Since $e \in trig(\Sigma)$, then by construction if $\widehat{A(C)}$, there exists an enabled transition from $init$ to $step$ in $\widehat{A(C)}$. Since all transitions from $step$ have a guard requiring either that $rtc = 0$ or $rtc = 2$, then no transition is enabled in $\widehat{A(C)}$. Thus the RTC step continues on $\widehat{M_2}$ until it terminates. Transition τ_1 then becomes enabled, setting rtc to 2, after which one of the following holds:
    * $\widehat{A(C)}$ has no enabled transition. Then transition τ_3 becomes enabled, which causes $\mathcal{M}(M_2, A(C))$ to move to state $RTCErr$, and the RTC step terminates with $rtc = 2$.
    * $\widehat{A(C)}$ has an enabled transition t: $\widehat{A(C)}$ executes t. This transition is either a transition from $step$ to err, which (by construction) sets rtc to 2, or from $step$ to end, which (by construction) sets rtc to 1, and the RTC step on $\widehat{A(C)}$ then executes the null transition from end to $init$.
      If $\widehat{A(C)}$ reached state err, then rtc is 2, and transition τ_3 becomes enabled, which causes $\mathcal{M}(M_2, A(C))$ to move to state $RTCErr$, and the RTC step terminates with $rtc = 2$.
      If $\widehat{A(C)}$ reaches state $init$ with $rtc = 1$, then transition τ_2 becomes enabled, setting rtc to 0, and the RTC step terminates.
  - If $\widehat{M_2}$ discards e, then by construction of Σ, $(e, \epsilon) \in \Sigma$. Thus, $\widehat{A(C)}$ consumes e and moves from state $init$ to state $step$. Let $qs = q$. Since C is complete, then there exists a transition from q (in C) denoted with (e, ϵ). Based on the construction of $A(C)$ and $\widehat{A(C)}$, there exists a transition from $step$ that is now enabled (to either err or end). Similarly to the cases above, the property holds when the RTC step terminates.

We conclude that at the end of the RTC step of $\mathcal{M}(M_2, A(C))$, the variable *rtc* is 0 iff the execution is not at state $RTCErr$ and $\widehat{A(C)}$ is at state *init*. We return to the main theorem: ex reaches state $RTCErr$ iff $ex' \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$ and $ex' \downharpoonleft_{EV(A(C))} \notin L_{ex}(A(C))$.

$\Longleftarrow$: Assume ex does not reach state $RTCErr$ (that is, $ex' = ex$). Then since only $\widehat{M_2}$ generates events in $\mathcal{M}(M_2, A(C))$, clearly $ex \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$. By the construction of $\widehat{A(C)}$ it is clear that if $\widehat{A(C)}$ reaches state *err*, then *rtc* is set to 2, which causes $\mathcal{M}(M_2, A(C))$ to move to state $RTCErr$. We can therefore conclude that if ex does not reach state $RTCErr$ then $\widehat{A(C)}$ does not reach *err* state. By the construction of $\widehat{A(C)}$, it also holds that $ex \downharpoonleft_{EV(A(C))} \in L_{ex}(A(C))$.

$\Longrightarrow$: Assume ex reaches state $RTCErr$. Consider the prefix on ex' without the last RTC step (that reaches state $RTCErr$, denoted ex''. Since ex'' does not reach $RTCErr$, from the proof of $\Longleftarrow$ we know that $ex'' \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$ and $ex'' \downharpoonleft_{EV(A(C))} \in L_{ex}(A(C))$, and the RTC step of ex'' terminated when $\widehat{A(C)}$ at state *init*. This means that during the last RTC of ex', *rtc* started with value 0 and was set to 2. Similarly to the induction proof, we can show that this means that during the last RTC step $\widehat{A(C)}$ traversed from *init* to *err*. Thus $ex' \downharpoonleft_{EV(A(C))} \notin L_{ex}(A(C))$. Clearly, $ex' \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$.

$\square$

After constructing $\mathcal{M}(M_2, A(C))$, Oracle 2 model checks $\mathcal{M}(M_2, A(C)) \models AG(\neg IsIn\ (RTCErr))$. If the model checker returns *true*, then the Teacher returns *true* and our framework terminates the verification, because according to **Rule AG-UML**, φ has been proved on $M_1||M_2$. Otherwise, if the model checker returns $false$ with a counterexample execution *cex*, then *cex* is analyzed as follows.

Counterexample Analysis:

Note that only $\widehat{M_2}$ generates events. Thus, by projecting the execution *cex* on $\{tr(e)|e \in trig(\Sigma)\} \cup \{gen(e)|e \in evnts(\Sigma)\}$ we can obtain $w \in \Sigma^*$ s.t. $cex \triangleright w$. The Teacher executes a membership query on w, for checking whether w is in A_w (as presented in Section 6.3.2). If the membership query succeeds (i.e, $w \in A_w$), the Teacher informs L^* that the conjecture is incorrect, and gives it w to witness this fact (since $w \in A_w$ but $w \notin L(C)$). If the membership query fails then the Teacher concludes that $\langle true \rangle M_1||M_2 \langle \varphi \rangle$

does not hold, since $cex \downharpoonleft_{EV(M_2)} \in L_{ex}(M_2)$, $cex \downharpoonleft_{EV(M_2)} \rhd w$ and $w \notin A_w$ (Theorem 6.14). The Teacher then returns $false$.

Example 6.22 *Consider the system $\Gamma = server||client$ and the assumption $A(C)$ from Figure 6.4. When checking $\langle true \rangle client[A(C)]$, the model checker may return a counterexample cex, represented by the word $w = (e_1, (req_1)), (e_1, (cancel_1)), (e_1, (req_1))$ ($cex \rhd w$). $cex \downharpoonleft_{EV(M_2)} \in L_{ex}(client)$, $cex \downharpoonleft_{EV(M_2)} \rhd w$ and $w \notin L(C)$.*

During counterexample analysis, the Teacher performs a membership query on w. This check fails, since there exists an execution of $M(w)||server$ that violates the property $AG(\neg(InQ(grant_1) \wedge InQ(deny_1)))$. Note that the property is violated even though server receives the event $cancel_1$ before it receives the second req_1. However, there exists a behavior of the environment of *$M(w)||server$ that causes violation of the property: if server receives event req_2 after $cancel_1$, then when it receives the second req_1 it will send $deny_1$. Note that since every state machine runs on a different thread, it is possible that the event $grant_1$, previously sent to client, was not yet dispatched. Thus, when $deny_1$ is added to the EQ of client, the property is violated. Since the membership query fails, we conclude that $server||client \not\models \varphi$.*

6.3.4 Correctness

We first show soundnessof our approach, and then show that it terminates.

Theorem 6.23 *Given state machines M_1 and M_2, and a property AGp, our framework returns true if $M_1||M_2 \models AGp$ and $false$ otherwise.*

Proof: The Teacher in our framework uses the two steps of the **Rule AG-UML** to answer conjecture queries. Our framework returns $false$ if it detects an execution on M_2 whose projection on Σ is not in A_w. By Theorem 6.14 this implies that $M_1||M_2 \not\models AGp$.

Our framework returns *true* only when both steps of the **Rule AG-UML** return *true*. That is, it learned $L(C)$ s.t. the state machine $A(C)$ satisfies both steps of the rule. By the construction of Σ, and since $\Sigma(A(C)) = \Sigma$ then it holds that $EV(A(C)) \subseteq EV(M_2)$. Thus, based on Theorem 6.10, it holds that $M_1||M_2 \models AGp$. □

Termination: Assuming the number of configurations of $M_1||M_2$ is finite, the weakest assumption w.r.t. M_1 and φ, A_w, is a regular language. To prove this, we construct an accepting automaton for A_w similarly to the construction in [24]. Since A_w is a regular language, then by correctness of the L^* algorithm, we are guaranteed that if it keeps receiving counterexamples, it will eventually produce A_w. The Teacher will then apply *Step* 2, which will return, based on Theorem 6.14, either *true* or a counterexample.

6.3.5 Performance Analysis

Our framework for automated learning-based **AG** reasoning is applied directly at the state machine level. That is, the system's components and the learned assumptions are state machines. However, the learning is done by applying an off-the-shelf L^* algorithm, whose conjectures are DFAs and its membership queries are words. Thus we need to translate DFAs and words into state machines. On the other hand we never need to translate from state machines back to low level representation (such as LTSs or DFAs). It is important to emphasize that, as shown above, the translation from DFAs and words to UML state machines is simple and straightforward, since the state machines created do not include complex features (such as hierarchy or orthogonality). On the other hand, a translation from UML state machines to DFAs may result in an exponential blowup, since the hierarchy and orthogonal structure should be flattened. Moreover, the event queues need to be represented explicitly, causing another blowup. Note that applying such a translation to LTSs does not influence the number of the membership or conjecture queries, as the learned assumption remains the same. However, it complicates the model checking used to answer these queries, since the system is much larger.

Our framework learns assumptions over an alphabet consisting of *sequences of events* representing RTC steps of M_2. We refer to this alphabet as *RTC alphabet.* Note that it is also possible to apply the framework (with minor modifications) over an alphabet consisting of single event occurrences (called *event alphabet*) rather then over the RTC alphabet, while still keeping the learning at the UML level. However, learning over the RTC alphabet is often better, as discussed below.

The complexity of the L^* algorithm can be represented by the number of membership and conjecture queries it needs in order to learn an unknown language U. As shown in [50, 15], the number of membership queries of L^* is $O(n^2 \cdot k + n \cdot log(m))$ and the number of conjecture queries is at most $n-1$, where n represents the number of states in the learned DFA, k is the size of the alphabet, and m is the size of the longest counterexample returned by the Teacher. This results from the characteristics of L^*, which learns the minimal automaton for U, and from the fact that each conjecture is smaller than the next one.

In theory, the size of the RTC alphabet might be much larger than the size of the event alphabet. This happens when every possible sequence of events is a possible RTC step of M_2. However, in practice typical state machines exhibit only a much smaller number of different RTC steps. Moreover, the number of states in the DFA Q_{RTC} learned over the RTC alphabet may be much smaller than the number of states in the DFA Q_{evnt} over the event alphabet. This is because a single transition in Q_{RTC} might be replaced by a sequence of transitions in Q_{evnt}, one for each of the events in the RTC. The above observations are demonstrated in the following example.

Example 6.24 *We re-visit the example presented throughout section 6.3.* $\Gamma = server||client$ *where server is* M_1, *client is* M_2, *and* $\varphi = \forall G(\neg(InQ(grant_1) \wedge InQ(deny_1)))$. *The final DFA learned when using sequences over RTC alphabet is presented in Figure 6.4(a). The total number of membership queries is* $O(3^2 \cdot 5 + 3 \cdot log2)$ *and there are* 2 *conjecture queries.*

If we apply learning of sequences over single event occurrences, then there are $O(4^2 \cdot 5 + 4 \cdot log3)$ *membership queries and* 3 *conjecture queries, since the resulting DFA has* 4 *states and the alphabet is* $\{tr(e_1), tr(grant_1), tr(deny_1), gen(req_1), gen(cancel_1)\}$.

6.4 AG for Systems with Multiple State Machines

In the previous section we introduced a framework for applying **AG** reasoning on UML systems of type $M_1||M_2$, where M_1 and M_2 are state machines. In this section we present extension of the framework for systems with multiple state machines.

The correctness of the framework presented in Section 6.3, which is based on learning the weakest assumption A_w (Corollary 6.15), assumes that A_w is defined over the assumption alphabet of a *single state machine* M_2. Moreover, the meaning of words and the relation between executions and words are defined under the assumption that words represent the behavior of a single state machine. When matching an execution to a word (Definition 6.7), for system Γ that includes a state machine M: An execution $ex \in L_{ex}(\Gamma)$ matches a word $w \in \Sigma(M)^*$ if the behavior of M on ex matches w. Assume now that we replace M with $M'||M''$ (M', M'' are two state machines), where M' and M'' are executed on two different threads. This means that executions of M' and M'' might be interleaved. Thus, there might be executions $ex \in L_{ex}(\Gamma)$ that do not match *any* word $w \in (\Sigma(M') \cup \Sigma(M''))^*$.

It is important to note that M_1 in the framework presented in Section 6.3 can be a system that includes *several state machines*. Moreover, M_2 can also include several state machines, as long as the state machines of M_2 run on a single thread. If M_2 includes multiple state machines $M_1^2||...||M_k^2$ that run on a single thread, then we can construct a single state machine $\tilde{M}_2$ where each M_i^2 is an orthogonal region in $\tilde{M}_2$. Since every RTC step starts with a consumption of an event, and the events sent to each M_i^2 are unique, then an RTC step on $\tilde{M}_2$ executes the RTC step in a single region. The executions of $\tilde{M}_2$ are equivalent to those of M_2, and we can then apply our framework on $M_1||\tilde{M}_2$.

In this section we propose a framework for applying **AG** reasoning where M_2 includes several state machines *each on a different thread*, for the case of *star-type systems*. These are systems that include a server, MS, and multiple clients, MC_i, s.t. the clients communicate only with the server (and not with each other). We rely on the unique structure of star-type systems for proposing an implementation for **Rule AG-UML**, where $M_1 = MS$ and $M_2 = MC_1||...||MC_n$. We also show why the framework is not correct in the general case where the second component includes several state machines that possibly communicate with each other. Note that as in the previous section, MS can include several state machines.

We start with defining the alphabet of a system, and define the relation between executions and words in such alphabet. We extend Definition 6.5 and define the alphabet of a *system* $\Gamma = M_1||...||M_n$ as $\Sigma(\Gamma) = \Sigma(M_1) \cup ... \cup \Sigma(M_n)$. Let w be a word in $\Sigma(\Gamma)^*$, the projection of w on $\Sigma(M_i)$ is denoted as $w\downharpoonright_{\Sigma(M_i)}$. We extend Definition 6.7 and match an execution to a word over *system alphabet*.

Definition 6.25 *Let $\Gamma = M_1||...||M_n$ and let $ex \in L_{ex}(\Gamma)$. Let $\hat{\Sigma} = \Sigma(M_1) \cup ... \cup \Sigma(M_m)$ for $m \leq n$, and let $w = \sigma_1, ..., \sigma_j$ be a word in $\hat{\Sigma}^*$. ex matches w, denoted $ex \rhd w$ if the following holds.*

1. *For every $i \in \{1, ..., m\}$: $ex \rhd w \downharpoonright_{\Sigma(M_i)}$, and*
2. *Let $\sigma_i = (t_i, (e^i_1, ..., e^i_{k_i}))$, and let $\xi_1 = tr(t_1), ..., tr(t_j)$. Also, let $\xi_2 = tr(e'_1), tr(e'_2), ...$ be the projection of ex on $\{tr(e) | e \in trig(\hat{\Sigma})\}$. Then $\xi_1 = \xi_2$.*

Intuitively, a word w represents a *collection of executions* where the behavior of each state machine matches the relevant projection on w (requirement 1), and the executions agree with w on the *order of the dispatched events* (requirement 2). Note that the order of generation of events ($gen(e)$) on different threads does not have to match w. Thus, every word w matches a collection of executions that represent different interleavings of the state machines in Γ, and every execution $ex \in L_{ex}(\Gamma)$ matches a *single word* $w \in \hat{\Sigma}^*$.

In order to apply the L^* algorithm, we define the assumption alphabet Σ as follows. Let $\Gamma = MS||MC_1||...||MC_n$, and for every $i \in \{1, ..., n\}$, let Σ_i be the assumption alphabet of MC_i w.r.t. MS and φ. Then $\Sigma = \Sigma_1 \cup ... \cup \Sigma_n$. The definition of weakest assumption (Definition 6.13) is directly extended to the case of multiple state machines.

The correctness of our framework is based on Theorem 6.14, which can be extended for star-type systems. However, it cannot be extended for the general case of multiple state machines in M_2. The main challenge in the proof of Theorem 6.14 is constructing an execution ex by combining ex_1 over $EV(\Sigma) \cup EV(M_1)$ s.t. $ex_1 \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$ and $ex_2 \in L_{ex}(M_2)$, where both $ex_1 \rhd w$ and $ex_2 \rhd w$. This construction exploits the fact that if there exists an execution $ex_2 \in L_{ex}(M_2)$ that matches w, then every execution ex'_2, where M_2 behaves as ex_2 with a different interleaving of M_2 and the environment, ex'_2 is a possible execution of M_2 as well. We exploit the fact that in star-type systems the clients do not communicate with each other, and make the following observation:

Corollary 6.26 *Let $\Gamma = MS||MC_1||...||MC_n$ be a star-type system, φ be a safety property, let Σ be the assumption alphabet of $MC_1||...||MC_n$ w.r.t. MS and φ, and let $w \in \Sigma^*$.*

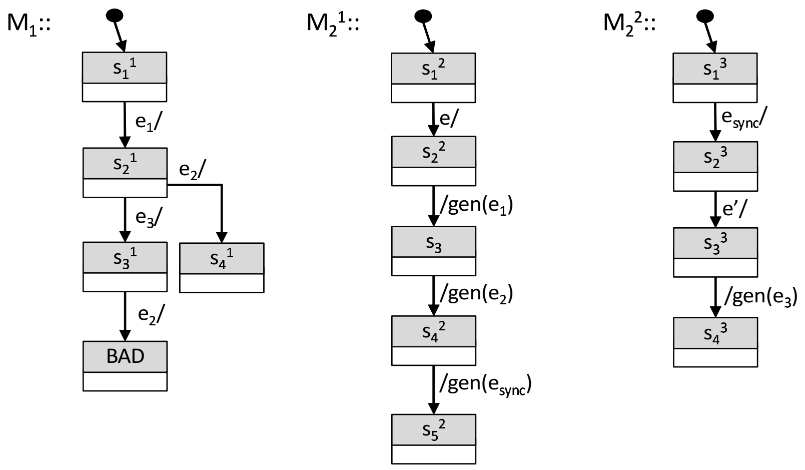


Figure 6.6: Non star-type system example

If there exists an execution $ex \in L_{ex}(MC_1||...||MC_n)$ s.t. $ex \triangleright w$ then the following holds. For every execution $\hat{ex}$ over $EV(\Sigma)$ s.t. $\hat{ex} \triangleright w$, there exists an execution $ex' \in L_{ex}(MC_1||...||MC_n)$ s.t. $ex' \downharpoonright_{EV(\Sigma)} = \hat{ex}$.

The above corollary states that if there exists an execution of $MC_1||...||MC_n$ that matches a word $w \in \Sigma^*$, then there exists an execution of $MC_1||...||MC_n$ for every possible interleaving that matches w. This holds for systems where the state machines do not communicate with each other. However, clearly, this observation does not hold for systems where the state machines communicate with each other, as exemplified in the following.

Example 6.27 *Let $\Gamma = M_1||M_1^2||M_2^2$ be the system presented in Figure 6.6, and assume we want to check that Γ does not reach BAD state by applying the* **AG** *reasoning. Note that M_1 reaches BAD state only if it receives events e_1, followed by e_3, followed by e_2. Note also that since M_2^1 generates event e_{sync}, which is consumed by M_2^2, then in Γ, e_3 is always generated* after *e_1 and e_2 were generated. Therefore, Γ does not violate φ.*

Assume now $M_2 = M_1^2||M_2^2$. The interface alphabet, Σ, is $\{(e,(e_1,e_2)), (e',(e_3))\}$. According to the definition of weakest assumption, the word $w = (e,(e_1,e_2)),(e',(e_3)) \notin A_w$. Since for the execution: $ex_1 = gen(e), gen(e')$,

$tr(e), gen(e_1), tr(e'), gen(e_3), gen(e_2), tr(e_1), tr(e_2), tr(e_3) \in EV(\Sigma) \cup EV(M_1)$ the following holds: $ex_1 \triangleright w$, $ex_1 \downharpoonright_{EV(M_1)} \in L_{ex}(M_1)$, and $ex_1 \not\models \varphi$.

When considering $M_2 = M_1^2 || M_2^2$, there exists an execution $ex_2 = gen(e), tr(e), gen(e_1), gen(e_2), gen(e_{sync}), gen(e'), tr(e_{sync}), tr(e'), gen(e_3)$ s.t. $ex_2 \triangleright w$. Thus, there exists an execution $ex_2 \in L_{ex}(M_2)$ and a word $w \in \Sigma^* \setminus A_w$ s.t. $ex_2 \triangleright w$. Therefore, the above example shows that Theorem 6.14 does not hold.

Theorem 6.14 does not hold since the interleaving of M_2 where e_3 is generated by M_2^2 before *e_2 is generated by M_2^1, although it is a possible interleaving represented by w, it is not a possible behavior of M_2 (due to internal dependencies). Thus, we cannot construct an execution that follows an execution of M_1 and the environment, violates φ, and also describes a legal behavior of M_2.*

Note that it is possible to include as part of the interface alphabet the events that joined by M_1^2 and M_2^2 (i.e., e_{sync}). However, this will result in an alphabet Σ that might be very large.

Since Theorem 6.14 holds for star-type systems, then in order to apply the **AG** framework on such systems, we need to provide a Teacher that answers membership and conjecture queries for star-type systems.

6.4.1 Membership Queries

It is important to notice that in order to answer a membership query for $w \in \Sigma^*$, it is not enough for the Teacher to construct a *single* state machine whose behavior matches w. Executions over Σ are interleaving executions of several threads, and thus cannot be represented in a single state machine. The Teacher therefore creates a *collection of state machines*, $M_1(w), ..., M_n(w)$ s.t. it guarantees that for every ex over $EV(\Sigma) \cup EV(MS)$: $ex \in L_{ex}(M_1(w)||...||M_n(w)||MS)$ iff $ex \downharpoonright_{EV(MS)} \in L_{ex}(MS)$ and $ex \triangleright w$.

$M_i(w)$ is constructed based on $w \downharpoonright_{\Sigma_i}$, similarly to the construction of $M(w)$ (Section 6.3.2). We add a global variable *cnt*, initialized to 1, that synchronizes the consumption of the events dispatched to $M_1(w), ..., M_n(w)$. Let $w = \sigma_1, ..., \sigma_k$, and let $w \downharpoonright_{\Sigma_i} = \sigma_{i_1}, \sigma_{i_2}, ..., \sigma_{i_m}$, $M_i(w)$ is presented in Figure 6.7.

During execution, the variable *cnt* keeps track of the number of triggers consumed from w, and ensures that $M_1(w)||...||M_n(w)$ conforms with

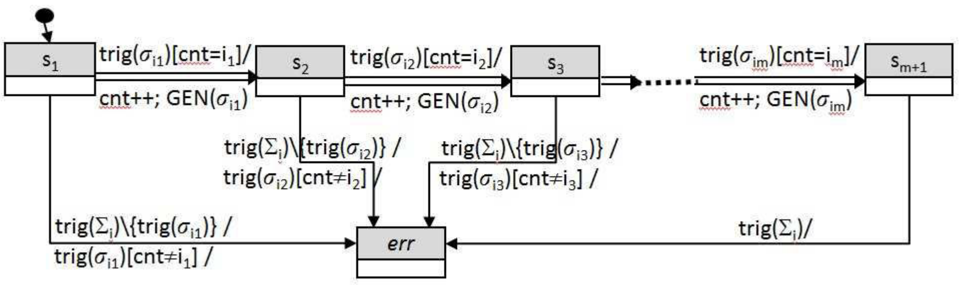


Figure 6.7: $M_i(w)$ representing $w\downharpoonright_{\Sigma_i}$

requirement 2 of Definition 6.25. The synchronization is done *only on the consumption of the events*, and therefore all possible executions over $EV(\Sigma)$ that match w are executions of $M_1(w)||...||M_n(w)$. Theorems 6.16 and 6.17 then hold for star-type systems, where $M(w) = M_1(w)||...||M_n(w)$.

Example 6.28 *Let $\Gamma = server||client_1||client_2$, where the state machine of server is presented in Figure 6.1, and the state machine of both client$_1$ and client$_2$ is presented in Figure 6.8 (i is 1 for client$_1$ and 2 for client$_2$).*

Let $\varphi = AG(\neg(InQ(grant_1) \wedge InQ(grant_2)))$. That is, φ states that it is not possible for both grant$_1$ and grant$_2$ to be in the event queues. Σ, the alphabet of the assumption of client$_1$||client$_2$ w.r.t. server is $\{(e_i, \epsilon), (e_i, (req_i)), (e_i, (cancel_i)), (grant_i, \epsilon), (deny_i, \epsilon)\}$ for $i \in \{1, 2\}$.

For checking the word $w = (e_2, (cancel_2)), (e_1, (req_1)) \in \Sigma^*$, the Teacher creates the two state machines $M_1(w)$ and $M_2(w)$ presented in Figure 6.9. Notice that by construction of $M_1(w)$ and $M_2(w)$, for every execution $ex \in L_{ex}(server||M_1(w)||M_2(w))$, $ex\downharpoonright_{\Sigma(client_1)}$ matches a prefix of $(e_1, (req_1))$, $ex\downharpoonright_{\Sigma(client_2)}$ matches a prefix of $(e_2, (cancel_2))$, and e_2 is consumed before e_1. Thus, $ex \rhd w'$ for $w' \in \Sigma^*$ a prefix of w.

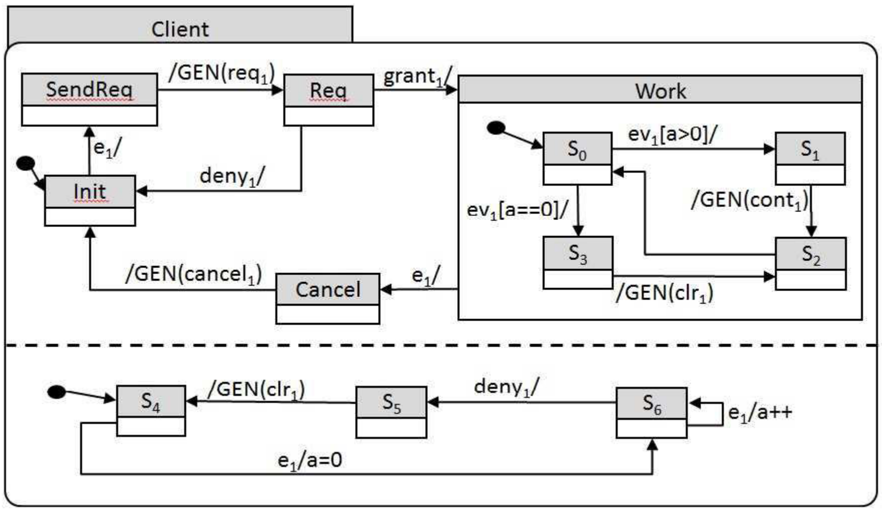


Figure 6.8: Example State Machine for Class $client_i$

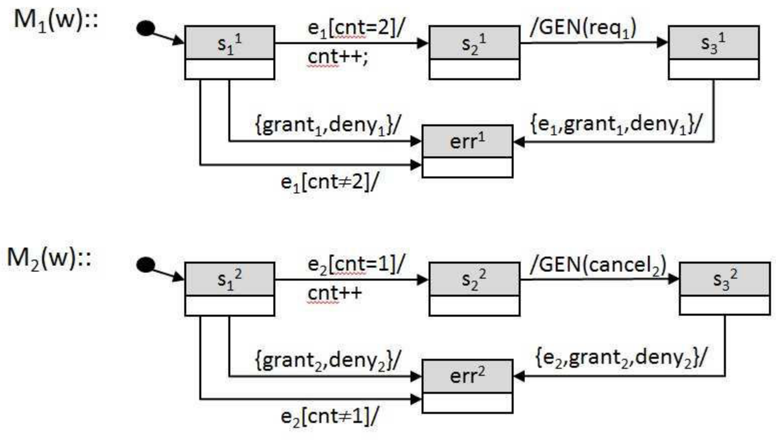


Figure 6.9: Example for $M_1(w)$ and $M_2(w)$

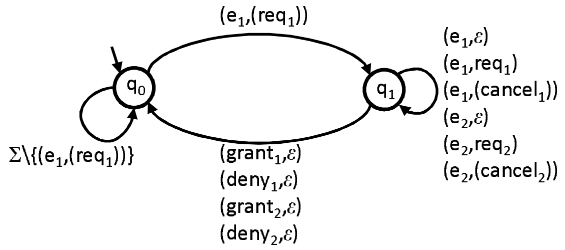


Figure 6.10: Conjecture DFA C for multiple clients

6.4.2 Conjecture Queries

Constructing State Machines From a DFA:

Let $C = (Q, \Sigma, \delta, q_0, Q \setminus \{q_{err}\})$ be the conjecture of the L^* algorithm. The Teacher constructs *a collection of state machines* $A_1(C), ..., A_n(C)$ from C, where each $A_i(C)$ is constructed from the projection of C on Σ_i. Let $C_i = (Q, \Sigma_i, \delta_i, q_0, Q \setminus \{q_{err}\})$ be a DFA where $\delta_i = \delta \cap (Q \times \Sigma_i \times Q)$. $A_i(C) \equiv A(C_i)$, where $A(C_i)$ is the state machine created for C_i as described in Section 6.3.3 (Definition 6.18). Notice that the variable qs is a joint variable of $A_1(C), ..., A_n(C)$. Theorem 6.20 then holds for star-type systems, where $A(C) = A_1(C)||...||A_n(C)$.
From now on we denote $\mathcal{A}(C) = A_1(C)||...||A_n(C)$ and $\mathcal{M} = MC_1||...||MC_n$.

Check $[\mathcal{A}(C)]MS\langle AGp\rangle$:

Oracle 1 performs *Step* 1 in the compositional rule by model checking $\mathcal{A}(C)||MS \models AG(p \vee IsIn(err))$. If the model checker returns $false$ with a counterexample execution cex, the Teacher informs L^* that the conjecture is incorrect, and gives it the word $w \in \Sigma^*$ s.t. $cex \triangleright w$ to witness this fact ($w \in L(C)$ and $w \notin A_w$). If the model checker returns $true$, indicating that $[\mathcal{A}(C)]MS\langle AGp\rangle$ holds, then the Teacher forwards $\mathcal{A}(C)$ to Oracle 2.

Example 6.29 *Let $\Gamma = server||client_1||client_2$, as presented in the previous example, and let $\varphi = AG(\neg(InQ(grant_1) \wedge InQ(grant_2)))$. Figure 6.10 presents a possible conjecture DFA returned by the L^* algorithm. Figure 6.11 presents the state machines $A_1(C)$ and $A_2(C)$ constructed from C. Note that there is no q_{err} state in C and thus there is no err state in $A_i(C)$ as well.*

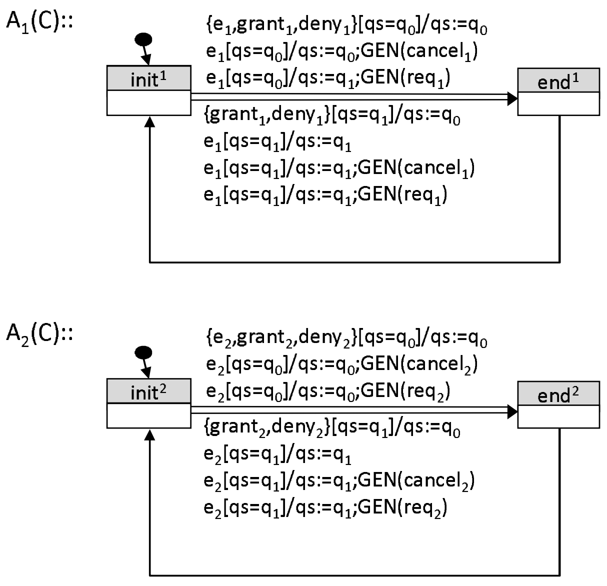

A1(C)::
{e1,grant1,deny1}[qs=q0]/qs:=q0
e1[qs=q0]/qs:=q0;GEN(cancel1)
e1[qs=q0]/qs:=q1;GEN(req1)
init1
end1
{grant1,deny1}[qs=q1]/qs:=q0
e1[qs=q1]/qs:=q1
e1[qs=q1]/qs:=q1;GEN(cancel1)
e1[qs=q1]/qs:=q1;GEN(req1)
A2(C)::
{e2,grant2,deny2}[qs=q0]/qs:=q0
e2[qs=q0]/qs:=q0;GEN(cancel2)
e2[qs=q0]/qs:=q0;GEN(req2)
init2
end2
{grant2,deny2}[qs=q1]/qs:=q0
e2[qs=q1]/qs:=q1
e2[qs=q1]/qs:=q1;GEN(cancel2)
e2[qs=q1]/qs:=q1;GEN(req2)

Figure 6.11: State machines $A_1(C)$ and $A_2(C)$

When verifying $\mathcal{A}(C)||MS \models AG(p \vee IsIn(err))$, the model checker returns $false$ with a counterexample. A possible counterexample might be $cex = gen(e_1), gen(e_1), gen(e_2), tr(e_1), gen(req_1), tr(e_1), gen(cancel_1), tr(e_2),$ $gen(req_2), tr(req_1), gen(grant_1), tr(cancel_1), tr(req_2), gen(grant_2)$. This execution matches the word $w = (e_1, (req_1)), (e_1, (cancel_1)), (e_2, (cancel_2))$ over Σ. Since $cex \rhd w$, then $w \notin A_w$, and the Teacher informs L^* that the conjecture is incorrect.

Check $\langle true\rangle\mathcal{M}[\mathcal{A}(C)]$:

Oracle 2 preforms *Step* 2 in the compositional rule. That is, it checks that for every execution $ex \in L_{ex}(\mathcal{M})$, $ex\downharpoonright_{EV(\mathcal{A}(C))} \in L_{ex}(\mathcal{A}(C))$. In section 6.3.3 this check was done by constructing a single state machine, $\mathcal{M}(M_2, A(C))$. However, executions of both $\mathcal{M}$ and $\mathcal{A}(C)$ are interleaving executions of *several threads*, and thus cannot be represented by a single state machine.

Let ex be an execution of a system Γ, ex is referred to as an *atomic RTC execution* if the RTC steps of the state machines in Γ do not interleave. We exploit the fact that the clients do not communicate with each other, and make the following observation: For every execution $ex \in L_{ex}(\mathcal{M})$ and $w \in \Sigma(\mathcal{M})^*$ s.t. $ex \triangleright w$, there exists an atomic RTC execution ex' over $EV(\mathcal{M})$ s.t. $ex' \in L_{ex}(\mathcal{M})$ and $ex' \triangleright w$. The result of this observation is that in order to check execution inclusion on star-type systems, it is enough to check atomic RTC executions. This is captured in the following theorem.

Theorem 6.30 *$L_{ex}(\mathcal{M}) \downharpoonleft_{EV(\mathcal{A}(C))} \subseteq L_{ex}(\mathcal{A}(C))$ iff for every atomic RTC execution $ex \in L_{ex}(\mathcal{M})$, $ex \downharpoonleft_{EV(\mathcal{A}(C))} \in L_{ex}(\mathcal{A}(C))$.*

Oracle 2 checks execution inclusion by considering only atomic RTC executions of both $\mathcal{M}$ and $\mathcal{A}(C)$. Notice that if we consider only atomic executions of a system $\Gamma = M_1||...||M_n$, then we can construct a *single state machine* with n orthogonal regions, each including a single M_i. Oracle 2 constructs two *state machines* MC and $A(C)$, representing only atomic RTC executions of $\mathcal{M}$ and $\mathcal{A}(C)$ respectively. *Step* 2 is then done based on MC and $A(C)$, as defined in Section 6.3.3.
Following, we formally define how to create a single state machine with orthogonal regions from a system with state machines that do not communicate.

Definition 6.31 *Let $\Gamma = M_1||...||M_n$ s.t. for every $i \neq j$, M_i and M_j do not communicate. The* orthogonal joint state machine *of Γ, denoted $\Delta(\Gamma)$, is a state machine that includes n orthogonal regions, where each region $i \in \{1, ..., n\}$ includes M_i with the following modifications. Each variable v of M_i is replaced with variable v_i: for every transition of M_i, if v is part of the guard or the action (or both), replace v with v_i.*
For every transition of M_i labeled with trigger t, replace t with t_i.

The renaming of the variables ensures that in $\Delta(\Gamma)$, variables are local to their region. The renaming of the triggers ensures uniqueness of the triggers between the different regions. Note that since for every $i \neq j$, M_i and M_j do not communicate, then the target of events generated on actions of $\Delta(\Gamma)$ is not a state machine in Γ, and thus no need to modify the generation of the events in $\Delta(\Gamma)$.

We define the operator $\widetilde{ex}$ as follows.

Definition 6.32 *Let* $\Gamma = M_1||...||M_n$ *be a system, and let* $ex = f_1, f_2, ...$ *be an execution in* $L_{ex}(\Gamma)$*.* $\widetilde{ex} = f'_1, f'_2, ...$ *where for every* $i \geq 1$ *the following holds:*

- *If* $f_i = tr((e, M_j))$ *where* $j \in \{1, ..., n\}$ *then* $f'_i = tr((e_j, \Delta(\Gamma)))$*.*
- *If* $f_i = gen((e, M_j))$ *where* $j \in \{1, ..., n\}$ *then* $f'_i = gen((e_j, \Delta(\Gamma)))$*.*
- *Otherwise,* $f'_i = f_i$*,*

The following theorem captures the relation between the executions of Γ and $\Delta(\Gamma)$.

Theorem 6.33 *Let* $\Gamma = M_1||...||M_n$ *s.t. for every* $i \neq j$*,* M_i *and* M_j *do not communicate. Let ex be an atomic RTC execution. Then ex is in* $L_{ex}(\Gamma)$ *iff* $\widetilde{ex} \in L_{ex}(\Delta(\Gamma))$*.*

Following, we define how to construct a single state machine from $\mathcal{A}(C)$.

Definition 6.34 *Let* $\mathcal{A}(C) = A_1(C)||...||A_n(C)$ *be the state machines constructed from DFA C. Assume that for every* $i \in \{1, .., n\}$*, the states of* $A_i(C)$ *are* $init_i$*,* end_i *and* err_i*. The* joint state machine *of* $\mathcal{A}(C)$*, denoted* $\Omega(\mathcal{A}(C))$*, includes states init, end and err, where init is the initial state.* $\Omega(\mathcal{A}(C))$ *has a single variable, qs initialized to* q_0 *(the same joint variable that exists in the different* $A_i(C)$ *state machines). For* $i \in \{1, ..., n\}$*, and for every transition in* $A_i(C)$ *from* $init_i$ *to* end_i *(or to* err_i*) labeled with* $t[g]/a$*, add a transition in* $\Omega(\mathcal{A}(C))$ *from init to end (or to err) labeled with* $t_i[g]/a$*. Also add a null transition from end to init (as in* $A_i(C)$*).*

The following theorem captures the relation between the executions of $\mathcal{A}(C)$ and $\Omega(\mathcal{A}(C))$.

Theorem 6.35 *An atomic RTC execution ex is in* $L_{ex}\mathcal{A}(C)$ *iff* $\widetilde{ex} \in L_{ex}(\Omega(\mathcal{A}(C)))$*.*

Oracle 2 Checks execution inclusion on atomic RTC executions by constructing $\Delta(\mathcal{M})$ and $\Omega(\mathcal{A}(C))$, and checking *Step* 2 as defined in Section 6.3.3.

Example 6.36 *Let $\Gamma = server||client_1||client_2$, as presented in the previous example, and let $\varphi = AG(\neg(InQ(grant_1) \wedge InQ(grant_2)))$.*

The conjecture DFA C returned from the L^* algorithm, for which Step 1 holds, is presented in Figure 6.12. q_{err} state and transitions to q_{err} are denoted with dashed lines. The multiple q_{err} states are for readability. Note that $L(C) \neq A_w$. For example, the word $w = (e_2, (req_2)), (grant_2, \epsilon), (e_1, (cancel_1)) \notin L(C)$, although $w \in A_w$. Note that φ is violated, for example, in the following scenario. $client_1$ sends a req_1 followed by a $cancel_1$. Following, $client_2$ sends a req_2. The server will, in turn, send $grant_1$ to $client_1$ followed by $grant_2$ to $client_2$. If $client_1$ sent $cancel_1$ before it received $grant_1$, then it is possible that both $grant_1$ and $grant_2$ are in the event queues, thus violating the property.

For checking Step 2, oracle 2 constructs a single state machine MC from $client_1$ and $client_2$ in two orthogonal regions, and constructs $A(C)$ from $\mathcal{A}(C)$. The check of Step 2 returns true, stating that every atomic RTC execution of $client_1||client_2$ has a representative execution in $A(C)$. We can then conclude that $\Gamma \models \varphi$.

Let $\Gamma' = server||client'_1||client'_2$, where the state machine of server is presented in Figure 6.1, the state machine of $client'_1$ is presented in Figure 6.2, and the state machine of $client'_2$ is presented in Figure 6.8 (where $i = 2$). Note that Σ is the same as in the previous case, and thus the conjecture DFA C returned from the L^* algorithm, for which Step 1 holds, is the same as before (Figure 6.12).

When checking Step 2, Oracle 2 returns $false$, with a counterexample. A possible counterexample is the word $w = (e_1, (req_1)), (e_1, (clr_1, cancel_1)), (e_2, (req_2))$. For this word, $w \in L_{ex}(client'_1||client'_2)$, however $w\downharpoonright_\Sigma \notin L(A)$.

During the counterexample analysis, the Teacher executes a membership query on $w\downharpoonright_\Sigma = (e_1, (req_1)), (e_1, (cancel_1)), (e_2, (req_2))$. This query returns $false$, indicating that $w\downharpoonright_\Sigma \notin A_w$. We can then conclude that $\Gamma' \not\models \varphi$.

6.5 Applying Assume-Guarantee Reasoning Recursively

In the previous section we introduced a framework for applying **AG** reasoning for star-type systems. In this section we extend the framework presented

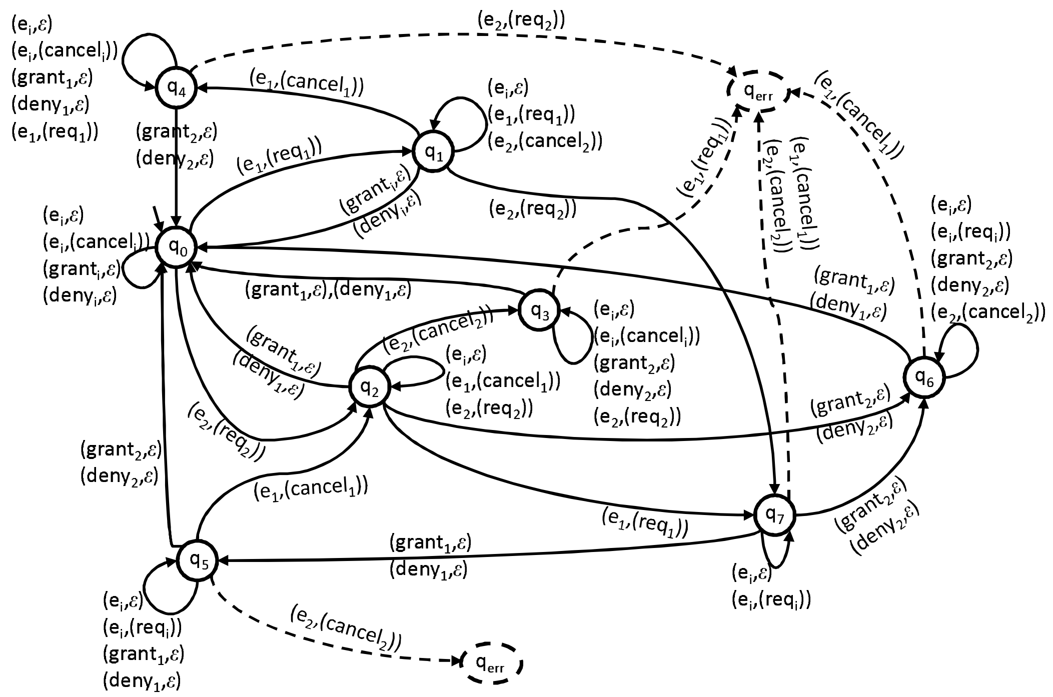


Figure 6.12: The conjecture DFA C

previously, and present a framework for applying recursive reasoning. That is, we present how to apply the following compositional rule for star-type systems:

Rule AG-UML-Mult

$$\begin{array}{ll}
(\mathit{Step}\ 1) & [A_1]MS\langle\varphi\rangle \\
(\mathit{Step}\ 2) & [A_2]MC_1[A_1] \\
 & \vdots \\
(\mathit{Step}\ n) & [A_n]MC_{n-1}[A_{n-1}] \\
(\mathit{Step}\ n+1) & \langle \mathit{true}\rangle MC_n[A_n] \\
\hline
 & \langle \mathit{true}\rangle MS||MC_1||...||MC_n\langle\varphi\rangle
\end{array}$$

We start by formally defining the meaning of $[A']M[A]$. Intuitively, $[A']M[A]$ holds iff every execution of $A'||M$ has a representative in A. That is, $[A']M[A]$ holds iff $EV(A) \subseteq EV(A') \cup EV(M)$ and for every $ex \in L_{ex}(A'||M)$, $ex\downharpoonright_{EV(A)} \in L_{ex}(A)$.

Theorem 6.37 *Let MS be a state machine, and for $i \in \{1,...,n\}$ let MC_i and A_i be state machines s.t. for every $j \in \{2,...,n\}$: $EV(A_{j-1}) \subseteq EV(A_j) \cup EV(MC_{j-1})$. Let p be a property over events $EV' \subseteq (EV(A_1) \cup EV(MS))$, and let $\varphi = AGp$. Then* **Rule AG-UML-Mult** *is sound.*

Proof: Assume by means of negation that *Step* 1 to *Step* $n+1$ hold, however $\langle \mathit{true}\rangle MS||MC_1||...||MC_n\langle AGp\rangle$ does not hold.
This means that there exists an execution $ex \in L_{ex}(MS||MC_1||...||MC_n)$ s.t. $ex \not\models Gp$. By Lemma 6.3, $ex\downharpoonright_{EV(MC_n)} \in L_{ex}(MC_n)$. Thus, since *Step* $n+1$ holds, $ex\downharpoonright_{EV(A_n)} \in L_{ex}(A_n)$.
It also holds (by Lemma 6.3) that $ex\downharpoonright_{EV(MC_{n-1})} \in L_{ex}(MC_{n-1})$.
Since $ex\downharpoonright_{EV(MC_{n-1})} \in L_{ex}(MC_{n-1})$ and $ex\downharpoonright_{EV(A_n)} \in L_{ex}(A_n)$, then by Lemma 6.3 $ex\downharpoonright_{EV(A_n)\cup EV(MC_{n-1})} \in L_{ex}(A_n||MC_{n-1})$, and since *Step* n holds, we can conclude that $ex\downharpoonright_{EV(A_{n-1})} \in L_{ex}(A_{n-1})$. Similarly, it can be shown that $ex\downharpoonright_{EV(A_1)} \in L_{ex}(A_1)$. Since $ex\downharpoonright_{EV(MS)} \in L_{ex}(MS)$ and $ex\downharpoonright_{EV(A_1)} \in L_{ex}(A_1)$, then by Lemma 6.3 $ex\downharpoonright_{EV(A_1)\cup EV(MS)} \in L_{ex}(A_1||MS)$. Since *Step* 1 holds, we can conclude that $ex\downharpoonright_{EV(A_1)\cup EV(MS)} \models Gp$. Based on Theorem 6.9, $ex \models Gp$ as well. A contradiction. We then conclude that $\langle \mathit{true}\rangle MS||MC_1||...||MC_n\langle AGp\rangle$ holds, which means that **Rule AG-UML-Mult** is sound. □

At each *Step* i, we use L^* to iteratively construct assumption A_i, until either all premises of **Rule AG-UML-Mult** hold, or until a real counterexample is found. For $i \in \{1, ..., n\}$, let Σ_i be the assumption alphabet of MC_i w.r.t. MS and φ. Then Σ^i, the alphabet of A_i, is defined as $\Sigma^i = \Sigma_i \cup ... \cup \Sigma_n$.

The notion of *weakest assumption* is extended for state machines.

Definition 6.38 *Let M and A be two state machines, and let Σ_w be an alphabet such that $EV(A) \subseteq EV(\Sigma_w) \cup EV(M)$. A language $A_w \subseteq \Sigma_w^*$ is* the weakest assumption *w.r.t. M and A if the following holds: $w \in A_w$ iff for every execution ex over $EV(\Sigma_w) \cup EV(M)$, if $ex \triangleright w$ and $ex \downharpoonright_{EV(M)} \in L_{ex}(M)$ then $ex \downharpoonright_{EV(A)} \in L_{ex}(A)$.*

Assume we could construct a set of state machines $M_1^{A_w}, ..., M_m^{A_w}$ that represent A_w. That is, for every execution ex over $EV(\Sigma_w)$, $ex \in L_{ex}(M_1^{A_w} || ... || M_m^{A_w})$ iff there exists $w \in A_w$ s.t. $ex \triangleright w$. Then $M_1^{A_w} || ... || M_m^{A_w}$ describes exactly those executions over Σ_w that when executed with M have a representative in A.

For $i \in \{1, ..., n-1\}$, and for state machines MC_i,MC_{i+1},...,MC_n, it is possible to extend Theorem 6.14 and prove the following.

Theorem 6.39 *Let A_i be a state machine over Σ^i, and let $A_w^{i+1} \subseteq \Sigma^{i+1}$ be the weakest assumption w.r.t. MC_i and A_i. Then $\langle true \rangle MC_i || ... || MC_n [A_i]$ iff for every execution $ex \in L_{ex}(MC_{i+1} || ... || MC_n)$, there exists $w \in A_w^{i+1}$ s.t. $ex \triangleright w$.*

The proof of the above theorem exploits the fact that the state machines MC_i,MC_{i+1},..., MC_n do not communicate with each other. From the above theorem we can conclude that in order to prove **Rule AG-UML-Mult**, the goal of L^* is to learn A_w^i (for $i \in \{1, ..., n\}$).

Step 1 is done as described in Section 6.4. Following, we present how to recursively verify $MC_1 || ... || MC_n$ w.r.t. A_1. We define when a system is *fully interleaved.* Intuitively, a system if fully interleaved if for every execution ex of the system, every possible interleaving of ex is also an execution of the system.

Definition 6.40 *Let $\Gamma = M_1 || ... || M_n$ be a system, let $ex \in L_{ex}(\Gamma)$ be an execution and let $w \in \Sigma(\Gamma)^*$ be a word s.t $ex \triangleright w$. We say that Γ is* fully interleaved *if for every ex' over $\Sigma(\Gamma)$ where $ex' \triangleright w$, $ex' \in L_{ex}(\Gamma)$.*

The correctness of our framework is based on the following observation, which exploits the characteristics of fully interleaved systems. Let Γ and Γ' be two systems where Γ' is fully interleaved and $EV(\Gamma') \subseteq EV(\Gamma)$. In order to check execution inclusion (i.e., check that $L_{ex}(\Gamma) \downharpoonright_{EV(\Gamma')} \subseteq L_{ex}(\Gamma')$), it is enough to check that every word w over $\Sigma(\Gamma)$, if there exists *some execution* in $L_{ex}(\Gamma)$ that matches w, then this execution has a representative in $L_{ex}(\Gamma')$. This is captured in the following theorem.

Theorem 6.41 *Let Γ and Γ' be two systems where Γ' is fully interleaved, and $EV(\Gamma') \subseteq EV(\Gamma)$. $L_{ex}(\Gamma) \downharpoonright_{EV(\Gamma')} \subseteq L_{ex}(\Gamma')$ iff for every word $w \in \Sigma(\Gamma)^*$, if there exists $ex \in L_{ex}(\Gamma)$ s.t. $ex \triangleright w$ then $ex \downharpoonright_{EV(\Gamma')} \in L_{ex}(\Gamma')$.*

Note that A_1 is fully interleaved, based on the correctness of Theorem 6.20 for star-type systems. For $i \in \{2, ..., n\}$, when proving *Step i*, we assume A_{i-1} is fully interleaved, and provide a Teacher that uses the L^* algorithm for learning assumption A_i. Our construction ensures that A_i is fully interleaved, and that A_i does not communicate with MC_{i-1}. We can therefore conclude that for every execution $ex \in L_{ex}(A_i||MC_{i-1})$ there exists an atomic RTC execution $ex' \in L_{ex}(A_i||MC_{i-1})$ that is represented by the same word. We say that ex' is the *atomic RTC representative* of ex. Thus, based on Theorem 6.41, in order to check *Step i*, it is enough to ensure that every *atomic RTC execution* of $A_i||MC_{i-1}$ has a representative in A_{i-1}.

6.5.1 Membership Queries

Let A_i be a fully interleaved system over $\Sigma^i = \Sigma_i \cup ... \cup \Sigma_n$. To answer a membership query for $w \in (\Sigma^{i+1})^*$, the Teacher must return *true* iff $w \in A_w^{i+1}$. The Teacher creates a collection of state machines $M_{i+1}(w), ..., M_n(w)$ such that for every $j \in \{i+1, ..., n\}$, $\Sigma(M_j(w)) \subseteq \Sigma_j$. Assume we construct $M_{i+1}(w), ..., M_n(w)$ as presented in Section 6.4.1. By construction, $M_{i+1}(w)||...||M_n(w)$ is fully interleaved. Moreover, $M_{i+1}(w)||...||M_n(w)$ do not communicate with MC_i, thus every execution ex of $M_{i+1}(w)||...||M_n(w)$ $||MC_i$ has an atomic RTC representative. We conclude that based on Theorem 6.41, $[M_{i+1}(w)||...||M_n(w)]MC_i[A_i]$ holds iff for every atomic RTC execution $ex \in L_{ex}(M_{i+1}(w)||...||M_n(w)||MC_i)$, $ex \downharpoonright_{EV(A_i)} \in L_{ex}(A_i)$. This is captured in the following theorem.

Theorem 6.42 *$L_{ex}(M_{i+1}(w)||...||M_n(w)||MC_i) \downharpoonright_{EV(A_i)} \subseteq L_{ex}(A_i)$ iff for every atomic RTC execution $ex \in L_{ex}(M_{i+1}(w)||...||M_n(w)||MC_i)$, $ex \downharpoonright_{EV(A_i)} \in L_{ex}(A_i)$.*

We construct a *single state machine* $M^{i+1}(w)$ whose executions are *exactly* the atomic RTC executions of $M_{i+1}(w)||...||M_n(w)$. $M^{i+1}(w)$ is therefore constructed as described in section 6.3.2. Once $M^{i+1}(w)$ is obtained, Oracle 1 constructs a *single state machine* whose executions ar exactly the atomic RTC executions of $M^{i+1}(w)||MC_i$. This is done by constructing $\Delta(M^{i+1}(w)||MC_i)$ (from Definition 6.31). We denote $\Delta(M^{i+1}(w)||MC_i)$ as $\Delta_i(w)$. Similarly, in order to consider only atomic RTC executions of A_i, Oracle 1 constructs $\Omega(A_i)$ (from Definition 6.34).

Execution inclusion between atomic RTC executions of $M_{i+1}(w)||...||$ $M_n(w)||MC_i$ and A_i is then done by checking execution inclusion between $\Delta_i(w)$ and $\Omega(A_i)$. This check is similar to the check of *Step* 2 in Section 6.3.3. I.e., construct a new state machine, $\mathcal{M}(\Delta_i(w), \Omega(A_i))$, and model check $\mathcal{M}(\Delta_i(w), \Omega(A_i)) \models AG(\neg IsIn(RTCErr))$. Recall that in the construction of $\mathcal{M}(\Delta_i(w), \Omega(A_i))$, *every behavior* of $\Delta_i(w)$ is monitored. However, in this case we want to monitor only behaviors of $\Delta_i(w)$ that *follow* w. That is, executions of $\Delta_i(w)$ that do not reach *err* state (on $M^{i+1}(w)$). We therefore modify $\mathcal{M}(\Delta_i(w), \Omega(A_i))$ in order to ensure that only behaviors that do not reach *err* state on $\Delta_i(w)$ are monitored. We denote the modified state machine as $\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i))$

Recall that the variable *rtc* is used for fixing the order of execution along an RTC step of $\mathcal{M}(\Delta_i(w), \Omega(A_i))$. On $\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i))$, $rtc = 4$ indicates that the execution of $\Delta_i(w)$ does not match w, and thus there is no need to ensure execution inclusion on the rest of this execution. The general structure of the modified $\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i))$ is presented in Figure 6.13. We modify $\Delta_i(w)$ as follows. For every transition in the region of $M^{i+1}(w)$ to state *err*, replace $rtc := 3$ on the action with $rtc := 4$.

The correctness of our construction is captured in the following theorem:

Theorem 6.43 *For every $ex \in L_{ex}(\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i)))$ there exists $w' \in (\Sigma^{i+1})^*$ s.t. $ex \triangleright w'$ and the following holds:*

- *ex reaches state RTCOk iff w' is* not *a prefix of w, and*

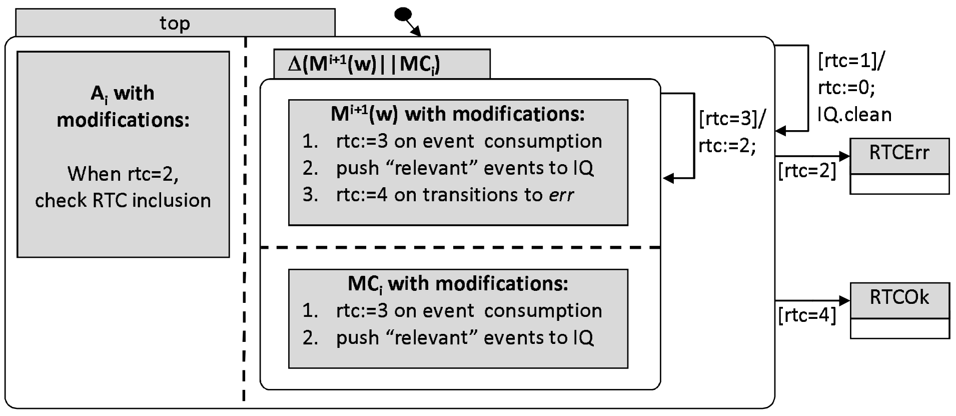

top
A_i with modifications:
When rtc=2, check RTC inclusion
$\Delta(M^{i+1}(w)||MC_i)$
$M^{i+1}(w)$ with modifications:
1. rtc:=3 on event consumption
2. push "relevant" events to IQ
3. rtc:=4 on transitions to *err*
MC_i with modifications:
1. rtc:=3 on event consumption
2. push "relevant" events to IQ
[rtc=3]/ rtc:=2;
[rtc=1]/ rtc:=0; IQ.clean
[rtc=2]
RTCErr
[rtc=4]
RTCOk

Figure 6.13: General scheme for $\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i))$

- *ex reaches state RTCErr iff w' is a prefix of w, $ex \downharpoonright_{EV(\Delta_i(w))} \in L_{ex}(\Delta_i(w))$, and $ex \downharpoonright_{EV(\Omega(A_i))} \notin L_{ex}(\Omega(A_i))$).*

Once $\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i))$ is constructed, the Teacher model checks $\hat{\mathcal{M}}(\Delta_i(w), \Omega(A_i)) \models AG(\neg IsIn(RTCErr))$. The Teacher returns *true*, indicating that $w \in A_w^{i+1}$ iff the model checker returns *true*.

6.5.2 Conjecture Queries

Constructing A State Machine From a DFA:

Let $C = (Q, \Sigma^{i+1}, \delta, q_0, Q \setminus \{q_{err}\})$ be the conjecture of the L^* algorithm. Assume we construct $A_{i+1}(C), ..., A_n(C)$ as presented in Section 6.4.2. By construction, $A^{i+1}(C) = A_{i+1}(C)||...||A_n(C)$ is fully interleaved. Moreover, $A^{i+1}(C)$ do not communicate with MC_i, thus every execution ex of $A^{i+1}(C)||MC_i$ has an atomic RTC representative. We conclude that based on Theorem 6.41, $[A^{i+1}(C)]MC_i[A_i]$ holds iff for every atomic RTC execution $ex \in L_{ex}(A^{i+1}(C)||MC_i)$, $ex \downharpoonright_{EV(A_i)} \in L_{ex}(A_i)$. This is captured in the following theorem.

Theorem 6.44 *$L_{ex}(A^{i+1}(C)||MC_i) \downharpoonright_{EV(A_i)} \subseteq L_{ex}(A_i)$ iff for every atomic RTC execution $ex \in L_{ex}(A^{i+1}(C)||MC_i)$, $ex \downharpoonright_{EV(A_i)} \in L_{ex}(A_i)$.*

We construct a *single* state machine from C whose executions are *exactly* the atomic RTC executions of $A^{i+1}(C)$. The construction is done as described in Definition 6.34. We denote the resulting state machine as $\Omega(A^{i+1}(C))$.

Check $[A^{i+1}(C)]MC_i[A_i]$**:**

Oracle 2 constructs a *single state machine* whose executions are exactly the atomic RTC executions of $A^{i+1}(C)||MC_i$. This is done by constructing $\Delta(\Omega(A^{i+1}(C))||MC_i)$ (from Definition 6.31). We denote $\Delta(\Omega(A^{i+1}(C))||MC_i)$ as Δ_i. In order to consider only atomic RTC executions of A_i, Oracle 2 constructs $\Omega(A_i)$ (from Definition 6.34). Similar to the case of membership query in Section 6.5.1, execution inclusion is done by constructing a new state machine, $\mathcal{M}(\Delta_i, \Omega(A_i))$. Since we want to monitor only behaviors of Δ_i that match a legal execution of $A^{i+1}(C)$, we modify $\mathcal{M}(\Delta_i, \Omega(A_i))$ accordingly. We denote the modified state machine as $\hat{\mathcal{M}}(\Delta_i, \Omega(A_i))$

Again, on $\hat{\mathcal{M}}(\Delta_i, \Omega(A_i))$, $rtc = 4$ indicates that the execution of Δ_i does not match a legal execution of $\Omega(A^{i+1}(C))$, and thus no need to ensure execution inclusion on the rest of this execution. We modify Δ_i as follows. For every transition in the region of $\Omega(A_{i+1}(C))$ to state err, replace $rtc := 3$ on the action with $rtc := 4$.

The correctness of our construction is captured in the following theorem:

Theorem 6.45 *For every* $ex \in L_{ex}(\hat{\mathcal{M}}(\Delta_i, \Omega(A_i)))$ *there exists* $w' \in (\Sigma^{i+1})^*$ *s.t.* $ex \triangleright w'$ *and the following holds:*

- *ex reaches state RTCOk iff* $ex\downharpoonright_{EV(A^{i+1})(C)} \notin L_{ex}(A^{i+1}(C))$
- *ex reaches state RTCErr iff* $ex\downharpoonright_{EV(\Delta_i)} \in L_{ex}(\Delta_i)$, *and* $ex\downharpoonright_{EV(\Omega(A_i))} \notin L_{ex}(\Omega(A_i))$.

Once $\hat{\mathcal{M}}(\Delta_i, \Omega(A_i))$ is constructed, the Teacher model checks $\hat{\mathcal{M}}(\Delta_i, \Omega(A_i)) \models AG(\neg IsIn(RTCErr))$. If the model checker returns $false$ with a counterexample execution cex, the Teacher informs L^* that the conjecture is incorrect, and gives it the word $w \in (\Sigma^{i+1})^*$ s.t. $cex \triangleright w$ to witness this fact ($w \in L(C)$ and $w \notin A_w^{i+1}$). If the model checker returns $true$, then the Teacher returns $true$, indicating that $[A^{i+1}(C)]MC_i[A_i]$ holds, then the Teacher forwards $A^{i+1}(C)$ to *Step* $i+1$.

6.6 Conclusion

We presented a framework for applying learning-based compositional verification of behavioral UML systems. Note that our framework is completely automatic; we use an off-the-shelf L^* algorithm. However, our Teacher works at the UML level. In particular, the assumptions generated throughout the learning process are state machines. From the regular automaton learned by the L^* algorithm, we construct a *state machine* (or several state machines) which is a conjecture on M_2. Also, the Teacher answers membership and conjecture queries by "translating" them to model checking queries on state machines.

We have extended the basic framework for **AG** reasoning to apply for recursive **AG** reasoning on star-type systems. In the future we plan to investigate other assume-guarantee rules in the context of behavioral UML systems. Another interesting extension of this work is developing a framework for applying the **AG** rule, where M_2 includes several state machines in systems which are not star-type. One straightforward way to handle systems which are not star-type is by learning words over *single occurrences of generation or consumption of events.* That is, do not define words as sequences of events representing RTC steps. However, such a definition loses the advantage of learning equivalence classes of executions.

Chapter 7

Conclusions

We presented three methods that aim at improving model checking of behavioral UML systems. The first method exploits software model checking for the verification of behavioral UML systems. Our translation to *verifiable* C preserves the high-level structure of the system and significantly eases the workload of the model checker. The second method provides an automatic CEGAR-like framework for abstraction and refinement of behavioral UML systems. Our abstraction and refinement are both at the UML level: the abstract model is a behavioral UML system that includes *abstract* state machines.The last method applies automatic learning-based compositional verification of behavioral UML systems. We use an off-the-shelf L^* algorithm. However, our Teacher works at the UML level. In particular, the assumptions generated throughout the learning process are state machines.

We believe there is more to be done in order to make model checking of behavioral UML systems more efficient. Our different methods provide a first step at different directions, and each of them can be further extended.

Bibliography

[1] István Majzik Adám Darvas and Balzs Beny. Verification of UML statechart models of embedded systems. In *In Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS 2002), IEEE Computer Society TTTC*, pages 70–77, 2002.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[3] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69–83, 2009.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207, Amsterdam, The Netherlands, March 1999. Springer.

[5] Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification (CAV'08)*, volume 5123 of *LNCS*, Princeton, NJ, USA, July 2008. Springer.

[6] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. The unified modeling language user guide. *J. Database Manag.*, 10(4):51–52, 1999.

[7] Sagar Chaki and Ofer Strichman. Optimized L*-based assume-guarantee reasoning. In *Tools and Algorithms for the Construc-*

tion and Analysis of Systems (TACAS'07), volume 4424 of *LNCS*, Braga, Portugal, March 2007. Springer.

[8] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon Damon Reese. Model checking large software specifications. *IEEE Trans. Software Eng.*, 24(7):498–520, 1998.

[9] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating DFA's for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, York, UK, March 2009.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. *Journal of the ACM*, 50(5):752–794, 2003.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT press, December 1999.

[12] Edmund M. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science, 2000.

[13] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 168–176, Barcelona, Spain, March 2004. Springer.

[14] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology*, 17(2), 2008.

[15] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*,

TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] Lucas Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Automated Software Engineering (ASE'09)*, pages 137–148, Auckland, New Zealand, November 2009. IEEE Computer Society.

[17] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. Viatra - visual automated transformations for formal verification and validation of UML models. In *Automated Software Engineering (ASE'02)*, pages 267–270, Edinburgh, Scotland, UK, September 2002. IEEE Computer Society.

[18] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Formal Methods for Components and Objects (FMCO'02)*, volume 2852 of *LNCS*, pages 71–98, Leiden, The Netherlands, November 2002. Springer.

[19] Jori Dubrovin and Tommi A. Junttila. Symbolic model checking of hierarchical uml state machines. In *Application of Concurrency to System Design (ACSD'08)*, pages 108–117, Xi'an, China, June 2008. IEEE.

[20] Azadeh Farzan, Yu-Fang Chen, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Extending automated compositional verification to the full class of omega-regular languages. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, Budapest, Hungary, March 2008.

[21] Harald Fecher, Michael Huth, Heiko Schmidt, and Jens Schönborn. Refinement sensitive formal semantics of state machines with persistent choice. *Electron. Notes Theor. Comput. Sci.*, 250(1):71–86, September 2009.

[22] Harald Fecher and Jens Schönborn. UML 2.0 state machines: Complete formal semantics via core state machines. In *Formal*

Methods: Applications and Technology (FMICS'06/PDMC'06), LNCS, pages 244–260, Berlin, Heidelberg, 2007. Springer.

[23] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, Braga, Portugal, March 2007.

[24] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Component verification with automatically generated assumptions. *Automated Software Engg.*, 12(3):297–320, July 2005.

[25] Object Management Group. OMG Unified Modeling Language (UML) Superstructure, version 2.4.1. formal/2011-08-06, 2011.

[26] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.

[27] Orna Grumberg, Yael Meller, and Karen Yorav. Applying software model checking techniques for behavioral uml models. In *Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 277–292, Paris, France, August 2012. Springer.

[28] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301, 2008.

[29] Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 212–226, Vienna, Austria, April 2006.

[30] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[31] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala Latvala, and Ivan Porres. Model checking dynamic and hierarchical UML state machines. In *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa 2006)*, 2006.

[32] Nima Kaveh. Using model checking to detect deadlocks in distributed object systems. In *Engineering Distributed Objects (EDO'00)*, volume 1999 of *LNCS*, pages 116–128, Davis, CA, USA, November 2000. Springer.

[33] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001.

[34] Diego Latella, István Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the spin model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[35] Shuang Liu, Yang Liu, Etienne Andre, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and JinSong Dong. A formal semantics for complete UML state machines with communications. In *integrated Formal Methods (iFM'13)*, volume 7940 of *LNCS*, pages 331–346. Springer, June 2013.

[36] Yael Meller, Orna Grumberg, and Karen Yorav. Verifying behavioral UML systems via CEGAR. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, volume 8739 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2014.

[37] Yael Meller, Orna Grumberg, and Karen Yorav. Applying software model checking techniques for behavioral UML systems. In *12th International Conference on Formal Aspects of Component Software (FACS'15)*, Niteroi, Rio de Janeiro, Brazil, October 2015.

[38] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in promela/spin. In *Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, pages 90–101, Boca Raton, FL, USA, October 1998. IEEE Computer Society.

[39] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Formal Methods in System Design*, 32(3):207–234, 2008.

[40] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *STTT*, 8(2):128–145, 2006.

[41] IST-2001-33522 OMEGA. http://www-omega.imag.fr, 2001.

[42] Ivan Paltor and Johan Lilius. Formalising UML state machines for model checking. In *The Unified Modeling Language - Beyond the Standard (UML'99)*, volume 1723 of *LNCS*, pages 430–445, Fort Collins, CO, USA, October 1999. Springer.

[43] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3), 2008.

[44] A. Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[45] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science (FOCS'77)*, 1977.

[46] C M Prashanth, K Chandrashekhar Shet, and Janees Elamkulam. An efficient event based approach for verification of UML statechart model for reactive systems. *International Conference on Advanced Computing and Communications (ADCOM'08)*, pages 357–362, 2008.

[47] Christian Prehofer. Behavioral refinement and compatibility of statechart extensions. *Electron. Notes Theor. Comput. Sci.*, 295:65–78, May 2013.

[48] Greg Reeve and Steve Reeves. Logic and refinement for charts. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, pages 13–23, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[49] Rhapsody. http://www-01.ibm.com/software/awdtools/rhapsody.

[50] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 411–420, New York, NY, USA, 1989. ACM.

[51] RuleBasePE. http://www.haifa.ibm.com/projects/verification/RB_Homepage/.

[52] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The rhapsody UML verification environment. In *Software Engineering and Formal Methods (SEFM'04)*, pages 174–183, Beijing, China, September 2004. IEEE Computer Society.

[53] Peter Scholz. Incremental design of statechart specifications. *Sci. Comput. Program.*, 40(1):119–145, May 2001.

[54] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, March 1995.

[55] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 108–125, Austin, Texas, USA, November 2000. Springer.

[56] A. J. H. Simons, M. P. Stannett, K. E. Bogdanov, and W. M. L. Holcombe. Plug and play safely: Rules for behavioural compatibility. In *International Conference on Software Engineering and Applications (IASTED'02)*, pages 263–268, Cambridge, MA, USA, 2002.

[57] Nishant Sinha and Edmund M. Clarke. SAT-based compositional verification using lazy learning. In *Computer Aided Verification*

(CAV'07), volume 4590 of *LNCS*, Berlin, Germany, July 2007. Springer.

בפרק 5 אנו מציעים גישה דמוית CEGAR לבדיקת מודל של מערכות מבוססות מודלים התנהגותיים של UML. אנו מציגים טרנספורמציה של מודל למודל אשר מייצרת *מודל אבסטרקטי* מתוך מודל קונקרטי נתון. הטרנספורמציה שלנו נעשית ברמת ה-UML, ולכן תוצאתה היא מערכת מבוססת מודלים התנהגותיים של UML חדשה, שהיא אבסטרקציה של המערכת המקורית. אנו מאמצים את שיטת CEGAR לגישתנו, ומבצעים עידון במקרה הצורך. גם העידון מוגדר כטרנספורמציה של מודל למודל ונעשה ברמת ה-UML.

אימות מודולרי עבור מודלים התנהגותיים של UML

דרך נוספת על מנת להתמודד עם בעיית התפוצצות המצבים היא בדיקת מודל מודולרית. בבדיקת מודל מודולרית הרכיבים השונים של המערכת נבדקים בנפרד זה מזה, במטרה להימנע במבנית המערכת השלמה ולצמצם את עלות בדיקת המודל. במרבית המקרים אי אפשר לאמת את רכיבי המערכת בניתוק מוחלט מיתר המערכת. זאת מכיוון שההתנהגות התקינה של כל רכיב תלויה באינטראקציה שלו עם הסביבה (או שאר המרכיבים). על כן, הוצעה בספרות סכמת ה-Assume-Guarantee, או בקיצור AG. סכמה זו מציעה לאמת רכיב, תחת הנחות על התנהגות הסביבה. לאחר מכן, הסביבה נבדקת על מנת להבטיח שהיא מספקת את ההנחות.

רבות מהעבודות על אימות מודולרי מבוססות על סכמת ה-AG ועל למידה לצורך ייצור ההנחות. הרעיון המרכזי בסכמת AG אוטומטית מבוססת למידה הוא לבצע הנחות AG באופן איטרטיבי, כאשר בכל איטרציה בונים הנחה ובודקים האם היא מתאימה. בדיקת ההתאמה מתבססת על למידה ועל אלגוריתם בדיקת מודל. עבודות רבות מציעות אופטימיזציות לסכמה הבסיסית הזו, וכן מיישמות את הסכמה בהקשר של חוקי AG אחרים.

בפרק 6 אנו מציגים סכמת AG אוטומטית מבוססת למידה עבור מערכות מבוססות מודלים התנהגותיים של UML. המאפיין המרכזי בסכמה שלנו הוא שאנו נשארים ברמת ה-UML. זאת אומרת, הרכיבים השונים של המערכת, כמו גם ההנחות הנלמדות הם כולם רכיבי UML.

כעת נסקור חלק מהאתגרים בבדיקת מודל עבור מודלים התנהגותיים של UML. כמו כן, נסקור את הדרך שלנו להתמודדות עם אתגרים אלו.

בדיקת מודל עבור מודלים התנהגותיים של UML

כלים לבדיקת מודל מניחים שהמערכת הנבדקת מתוארת בשפת מפרט מתאימה. עבודות קודמות בהקשר של בדיקת מודל עבור מודלים התנהגותיים של UML תרגמו את המודלים לשפות כגון SMV או VIS, שתיהן מתאימות במיוחד עבור חומרה, ל- PROMELA (שפת הקלט של הכלי SPIN), שמתאימה בעיקר לפרוטוקולי תקשורת, או ל-IF, שמיועד למערכות זמן אמת.

אנו מאמינים כי מודלים התנהגותיים של UML מזכירים בעיקר מערכות תוכנה. אי לכך אנו בוחרים לתרגם מודלים אלו לשפת C, ולאמץ שיטות *בדיקת מודל מבוססות תוכנה* עבורם. התרגום שלנו משמר את המבנה העקרוני של המודל מבוסס UML: אובייקטים מונחי אירועים (event driven objects) אשר מתקשרים דרך תור אירועים. חישוב של המערכת מכיל סדרה של צעדי ריצה-עד-סיום (Run To Completion), או בקיצור צעדי RTC. כל צעד RTC מתחיל בכך שתור האירועים שולח אירוע לאובייקט היעד, אשר בתורו מבצע סידרה מקסימלית של צעדים.

בפרק 4 אנו מציגם את הגישה שלנו לבדיקת מודל של מודלים התנהגותיים של UML, המיישמת שיטות בדיקת מודל מבוססות תוכנה.

אבסטרקציה ועידון עבור מודלים התנהגותיים של UML

אבסטרקציות מסתירות חלק מפרטי המערכת, על מנת לקבל מערכת חדשה אשר מכילה *יותר התנהגויות ופחות מצבים* מאשר המערכת הקונקרטית (המקורית). המערכת האבסטרקטית המתקבלת מאופיינת בכך שאם התכונה מתקיימת במערכת האבסטרקטית, אז היא גם מתקיימת במערכת הקונקרטית. לעומת זאת, אם התכונה לא מתקיימת במערכת האבסטרקטית, לא ניתן להסיק דבר עבור המערכת הקונקרטית. גישת *אבסטרקציה-עידון מונחית דוגמה נגדית* (*CounterExample-Guided Abstraction* Refinement), או בקיצור CEGAR, מספקת שיטה אוטומטית ואיטרטיבית לביצוע אבסטרקציה ועידון של המערכת, כשהעידון מתבסס על הדוגמה הנגדית שהתקבלה מבדיקת המערכת האבסטרקטית. כאשר בדיקת המודל מחזירה דוגמה נגדית אבסטרקטית, מתבצע חיפוש אחר דוגמה נגדית קונקרטית מתאימה. אם קיימת דוגמה נגדית קונקרטית כזו, אזי זוהי דוגמה נגדית אמתית במערכת הקונקרטית. אחרת, אם לא קיימת דוגמה נגדית קונקרטית המתאימה לדוגמה הנגדית האבסטרקטית, אזי הדוגמה הנגדית אינה אמתית (spurious), ויש צורך ב*עידון*. עידון הוא תהליך הפוך לאבסטרקציה, שבו מוסיפים פרטים למודל האבסטרקטי, במטרה להסיר את הדוגמה הנגדית שאינה אמתית.

תקציר מורחב

מערכות ממוחשבות חולשות על כמעט כל תחום של חיינו ופעולתם הנכונה היא הכרחית. *בדיקת מודל* הינה תהליך אוטומטי שבהינתן מערכת בודק קיום של תכונה מסוימת ביחס למערכת זו. המערכת מתוארת בדרך כלל כמכונת מצבים בצורת גרף מעברים. התכונה נתונה כנוסחה בלוגיקה טמפורלית (Temporal Logic). התהליך של בדיקת מודל משמעו מעבר על *כל* ההתנהגויות של המערכת. תוצאת המעבר יכולה להיות או הוכחה שהמערכת נכונה ביחס לתכונה הנבדקת, או *דוגמה נגדית* המתארת התנהגות שגויה של המערכת.

בדיקת מודל יושמה ומיושמת במערכות חומרה ותוכנה, ונעשה בה שימוש נרחב בתעשייה. אך המגבלה העיקרית של השיטה היא *בעיית התפוצצות המצבים*. בעיה זו מקורה במספר המצבים במערכות אמיתיות. בדיקת המודל דורשת זיכרון רב וזמן חישוב רב. אי לכך, עבור מודלים גדולים, בדיקת מודל עלולה להיות בלתי ישימה. חלק נכבד מהמחקר בתחום מוקדש להתמודדות עם בעיה זו.

שפת המידול המאוחדת (*The Unified Modeling Language*), או בקיצור UML, הינה שפת מידול מקובלת, המשמשת לאפיון, הדמיה, ובניה של מערכות. שפה זו נותנת כלים על מנת לייצג את המערכת כאוסף של אובייקטים ולתאר את המבנה הפנימי של המערכת, וכן את התנהגותה. UML הוכרזה כשפת מידול מונחת עצמים סטנדרטית על ידי ה-Object Management Group (OMG). שפה זו הופכת לשפת המידול השולטת במערכות משובצות מחשב (embedded systems). אי לכך, התנהגות נכונה של מערכות המיוצגות כמערכות UML הינה קריטית, ונדרשות שיטות בדיקת מודל עבור מערכות מסוג זה.

בעבודה זו אנו מציגים שיטות חדשות לשיפור בדיקת המודל עבור מודלים התנהגותיים של UML. מטרתנו העיקרית היא להשאיר את תהליך בדיקת המודל *ברמת ה-UML*. זאת אומרת, במקום לתרגם את המודלים ההתנהגותיים של UML לייצוג נמוך כלשהו (למשל למבנה קריפקה) ולהפעיל אופטימיזציות על הייצוג הנמוך, מטרתנו היא להפעיל את האופטימיזציות על מודל ה-UML ישירות. גישה זו מאפשרת לנו לנצל באופטימיזציות שלנו מידע ברמה הגבוהה, הנובע מהמבנה וההתנהגות הייחודיים של מודלים אלו, מידע אשר נעלם במעבר לייצוג נמוך. חשוב לציין שההישארות ברמת מודל ה-UML מועילה ביותר גם עבור המשתמש, מכיוון שהתכונה, האופטימיזציות וגם הדוגמה הנגדית ניתנות ברמת ה-UML ולכן הינן בעלות משמעות.

ישנם שני אתגרים מקבילים להתמודדות כאשר מטפלים בבדיקת מודל עבור מודלים התנהגותיים של UML. הראשון הוא כיצד להפעיל כלים קיימים לבדיקת מודל על מודלים אלו. האתגר השני הוא כיצד להתמודד עם בעיית התפוצצות המצבים בהקשר של מודלים התנהגותיים של UML. שתיים מהגישות המבטיחות להתמודד עם בעיית התפוצצות המצבים הן *אבסטרקציה* ו*בדיקת מודל מודולרית*. אנו מציעים ליישם את שתי השיטות הללו בהקשר של מודלים התנהגותיים של UML.

המחקר נעשה בהנחיית פרופ' ארנה גרימברג ודר' קרן יורב בפקולטה למדעי המחשב .

בראש ובראשונה אני אסירת תודה למנחה שלי, ארנה גרימברג, על הנחייתה המעולה ותמיכתה לאורך שנות לימודי. תודה רבה לך על שלימדת אותי להיות חוקרת, והראית לי את הכיף והיופי שבמחקר. תמיד סיפקת לי השראה ועידוד, וידעת תמיד לעשות זאת בחיוך. יותר מכל תודה לך על חברותך. על הפסקות הקפה, השיחות, המסיבות והריקודים. כל אלו הפכו את הלימודים למהנים כל כך. אני מרגישה בת-מזל על שהיית המנחה שלי.

אני רוצה להודות למנחה הנוספת שלי, קרן יורב. תודה לך על שהכרת לי את העולם של UML. תודה על שתמיד ידעת למצוא את הזמן להדריך ולעזור לי, ולחלוק איתי את הרעיונות שלך. אני מרגישה אסירת תודה על הזכות לעבוד איתך וללמוד ממך.

אני רוצה להודות להורי, קובי ותמי קלקה, על אהבתכם ותמיכתכם בכל ההיבטים של חיי ולימודי, מבית ספר יסודי ועד לימודי הדוקטורט. אני מודה לחמי וחמותי, יצחק ודניאלה מלר, שעשו כל שביכולתם לסייע לי במהלך לימודי.

לבסוף, אני מודה לבעלי נמרוד, ולבנותי עדי, מאיה ונגה, על אהבתכם, על שאתם עושים אותי מאושרת, ועל כך שאתם מראים לי בכל יום מהם הדברים החשובים באמת בחיי. נמרוד, תודה על חברותך ועל תמיכתך האינסופית במהלך הלימודים שלי. לא יכולתי לעשות זאת בלעדיך.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

שיטות בדיקת מודל עבור מודלים התנהגותיים של UML

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

דוקטור לפילוסופיה

יעל מלר

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

שבט תשע"ו   חיפה   ינואר 2016

שיטות בדיקת מודל עבור מודלים התנהגותיים של UML

יעל מלר