Learning-Based Compositional Model Checking of Behavioral UML Systems

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

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

Abstract. This work presents a novel approach for applying compositional model checking of behavioral UML models, based on learning. 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. However, its applicability is often impeded by its high time and memory requirements. A successful approach to tackle this limitation is *compositional model checking*. Recently, great advancements have been made in this direction via automatic learning-based Assume-Guarantee reasoning.

In this work we propose a framework for automatic Assume-Guarantee reasoning for behavioral UML systems. We apply an off-the-shelf learning algorithm for incrementally generating environment assumptions 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.

1 Introduction

This work presents a novel approach for learning-based compositional model checking of behavioral UML systems. Our work focuses on systems that rely on *UML state machines*, a standard graphical language for modeling the behavior of event-driven software components. The *Unified Modeling Language* (UML) [3] is becoming the dominant modeling language for specifying and constructing embedded and safety critical systems. As such, the correct behavior of systems represented as UML models is crucial and model checking techniques applicable to such models are required.

Model checking [7] 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 such as a state transition graph, where nodes represent the current state of the system and edges represent transitions of the

An extended version including full proofs is published as a technical report in [22].

C. Braga and P.C. Ölveczky (Eds.): FACS 2015, LNCS 9539, pp. 275–293, 2016.
DOI: 10.1007/978-3-319-28934-2_15

system from one state to another. The specification is usually given as a temporal logic formula. The model checking algorithm traverses *all* of the system behaviors (i.e., paths in the state transition graph), and either concludes that all system behaviors are correct w.r.t. to the checked property, or provides a *counterexample* that demonstrates an erroneous behavior.

Model checking is widely recognized as an important approach to increase the reliability of hardware and software systems and is vastly used in industry. Unfortunately, its applicability is often impeded by its high time and memory requirements. One of the most appealing approaches to fighting these problems is *compositional model checking*, where parts of the system are verified separately. The construction of the entire system is avoided and consequently the model checking cost is reduced. 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 [14,17,27] 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 become a major technique to construct assumptions for the **AG** paradigm automatically. An automated *learning-based* ***AG*** *framework* was first introduced in [9]. 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 (e.g. [4,6,11,16,24,25]).

In this paper we propose a framework for automated learning-based **AG** reasoning *for UML state machines*. Our framework is similar to the one presented in [9], 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 [9], where the system's components and the learned assumptions are all presented as Labeled Transition Systems (LTSs), which are a form of low-level state transition graphs. To the best of our knowledge, this is the first work that applies learning-based assume guarantee reasoning in the context of behavioral UML systems.

A naive implementation of our framework might translate a given behavioral UML system into LTSs and apply the algorithm from [9] 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*

(e.g. [1,5,8,10,15,19,20,23,29]). 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 state transition graphs).

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 in any environment has a representative in A. Together they guarantee that $M_1||M_2$ satisfies φ in any environment.

$$\textbf{Rule AG-UML} \qquad \frac{(\mathit{Step}\ 1)\ \ [A]\ M_1\ \langle\varphi\rangle \qquad (\mathit{Step}\ 2)\ \ \langle \mathit{true}\rangle\ M_2\ [A]}{\langle \mathit{true}\rangle\ M_1||M_2\ \langle\varphi\rangle}$$

We assume φ is a safety property, and use the learning algorithm L^* [2,28] 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 $M_1||M_2 \not\models \varphi$.

UML state machines communicate via *asynchronous events* using thread-local event queues. When a state machine receives an event, it makes a *run-to-completion (RTC)* step, in which it processes the event and continues execution until it cannot continue anymore. During its execution, the state machine may send events to other state machines. 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 paper is organized as follows. Some background on UML and **AG** reasoning is given in Sect. 2. UML computations, executions, words and their relations are defined in Sect. 3. In Sect. 4 we present our framework, implementing **Rule AG-UML** for UML systems. We conclude in Sect. 5.

2 Preliminaries

2.1 UML Behavioral Systems

We present here a brief overview of behavioral UML systems, and in particular, UML state machines. We refer the interested reader to the UML specification [13].

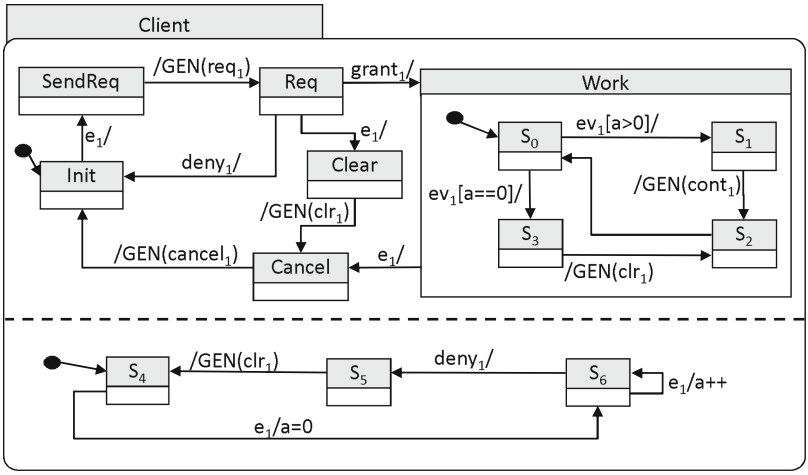

Fig. 1. Example State Machine of Class *client*

Behavioral UML systems include objects (instances of classes) that process events. Event processing is performed by state machines, which include complex features such as hierarchy, concurrency and communication. UML objects communicate by sending each other events (asynchronous messages) that are kept in *event queues* (EQs). Every object is associated with a single EQ, and several objects can be associated with the same EQ. In a multi-threaded system there are several EQs, one for each thread. Each thread executes a loop, taking an event from its EQ, and dispatching it to the target object, which then makes an RTC step. Only when the target object finishes its RTC step, the thread dispatches the next event available in its EQ. RTC steps of different threads are interleaved.

Figure 1 describes the state machine of class *client*. UML state machines include hierarchical states (states $Work$ and $Client$ in Fig. 1), a single initial state in each hierarchical state (e.g., state s_0 in $Work$), and transitions between states. Each transition is labeled with $t[g]/a$, where t, g and a are *trigger*, *guard*, and *action*, respectively. Each of them is independently optional. A trigger is an event name, a guard is a Boolean expression over local and global variables, and an action is a piece of code in the underlying language used by the model. Actions can include statements generating event e and sending it to the relevant EQ. We represent such statements as "GEN(e)". An event e includes the name of the event and the state machine to which the event is sent. The set of events of a system includes events sent by a state machine in the system, and events sent by the "environment" of the system (to be formally defined later).

A transition from state s is *enabled* if s is part of the current (possibly hierarchical) active state, the trigger (if there is one) matches the current event dispatched, and the guard holds (an empty guard is equivalent to *true*). Further, all transitions contained in s are disabled. For example, in Fig. 1, the transition

from $Work$ to $Cancel$ is enabled only if $Work$ is active, the event dispatched is e_1, and the transitions from s_0, s_1, s_2 and s_3 are disabled. When a transition is taken, the action labeling it is executed, and the state machine moves to the target state. An object executes an RTC step by traversing on enabled transitions, until it cannot continue anymore.

A state can include multiple orthogonal regions, separated by a dashed line, which corresponds to the parallel execution of the state machines contained in them (e.g., state *Client* has two orthogonal regions). When an event is dispatched to a state machine, and it has no enabled transitions, then the event is *discarded* and the RTC step terminates immediately. Otherwise, if there exists an enabled transition, we say that the event is *consumed*. In each RTC step only the first transition may consume an event. An exception is the case of orthogonal regions that share the same trigger. 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.

A *computation* of a system is defined as a sequence of system configurations. A *system configuration* includes information about the current state of each state machine in the system, the contents of all the EQs, and the value of all variables in the system. The initial configuration in a computation matches the initial state of the system, and the system moves from configuration c to configuration c' by executing an enabled transition or by receiving an event from the environment. A formal definition of computations can be found in [21].

2.2 Assume Guarantee Reasoning and Compositional Verification

[9] 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 [9] uses a learning algorithm for generating assumptions A_i and a model checker for verifying the two steps in the rule.

2.3 The L^* Algorithm

The learning algorithm used in [9] was developed by [2], and later improved by [28]. 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 $w \in U \setminus L(C)$.

3 Representing Executions as Words

A behavioral UML system with n state machines is denoted by $Sys = M_1||...||M_n$. 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). For simplicity of presentation, we assume the following restrictions: (a) Transitions with triggers do not generate events, and each transition may generate at most one event, (b) A state machine does not generate events to itself, (c) An event e cannot be generated by more than one state machine, and (d) 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. The events of a system include events sent by a state machine in the system denoted $ESys$, and events sent by the "environment" of the system denoted $EEnv$. For a system Sys, $ESys(Sys) = Gen(M_1) \cup ... \cup Gen(M_n)$, and $EEnv(Sys) = \{Con(M_1) \cup ... \cup Con(M_n)\} \setminus \{Gen(M_1) \cup ... \cup Gen(M_n)\}$. We denote $EV(Sys) = ESys(Sys) \cup EEnv(Sys)$. 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.

Recall that a computation of Sys is a series of configurations. Based on the above assumptions on Sys, each move from configuration c to configuration c' in a computation is labeled by at most one of $tr(e)$ and $gen(e)$, where $tr(e)$ denotes that when moving from c to c' event e was dispatched to the target state machine, and $gen(e)$ denotes that event e was either generated by a state machine in Sys (if $e \in ESys(Sys)$) or sent by the environment of Sys (if $e \in EEnv(Sys)$). Note that it is possible that a move is denoted with neither (labeled with ϵ).

[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.

Note that events are always generated before they are dispatched. UML2 places no restrictions on the implementation of the EQs, and neither do we. However, a specific implementation implies restrictions on the possible order of events. For example, if 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. 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 Sys by $L_{ex}(Sys)$. Note that every computation matches a single execution. However, different computations may match the same execution.

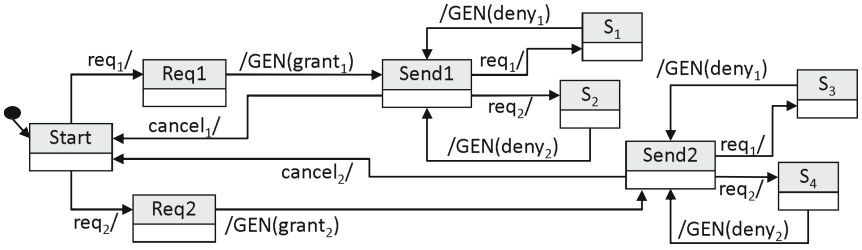

Fig. 2. Example State Machine for Class *server*

Example. *Consider the system $Sys = server||client$ where client and server are presented in Figs. 1 and 2, respectively. Then $gen(e_1), tr(e_1), gen(req_1), tr(req_1), gen(grant_1) \in L_{ex}(Sys)$[2]. However, $gen(e_1), tr(e_1), gen(cancel_1) \notin L_{ex}(Sys)$, 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, and ex be an execution over EV. The *projection of ex w.r.t. EV'*, denoted $ex \downarrow_{EV'}$, is the projection of ex on $\{tr(e), gen(e) | e \in EV'\}$. The following theorem is a result of the fact that state machines communicate only through events.

Theorem 1. *Let $Sys = M_1||...||M_n$, and let ex be an execution over $EV(Sys)$. Then, $ex \in L_{ex}(Sys)$ iff for every $i \in \{1, ..., n\}$, $ex \downarrow_{EV(M_i)} \in L_{ex}(M_i)$.*

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

Definition 2. *Let M be a state machine. $\sigma = (t, (e_1, .., e_n))$ is in* the alphabet of M, *$\Sigma(M)$, 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 $e_1, ..., e_n$.*

[2] In the examples throughout the paper we assume EQs are implemented as FIFOs.

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. *Let M = client (Fig. 1). Then $\Sigma(M) = \{(e_1, (req_1)), (deny_1, \epsilon), (e_1, (clr_1, cancel_1)), (e_1, \epsilon), (deny_1, (clr_1)), (grant_1, \epsilon), (ev_1, (clr_1)), (ev_1, (cont_1)), (ev_1, \epsilon)$. 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 3. *Let Sys be a system that includes state machine M, let $ex = f_1, f_2, ... \in L_{ex}(Sys)$, 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 $ex \triangleright w$, 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. *Consider execution $ex = gen(e_1)$, **$tr(e_1)$**, **$gen(req_1)$** $tr(req_1)$, $gen(grant_1)$, $gen(ev_1)$, **$tr(ev_1)$** $\in L_{ex}(server||client)$. We denote with **bold** the parts of the execution that represent behavior of 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 $ex' = gen(e_1), gen(ev_1)$, **$tr(e_1)$**, **$gen(req_1)$**, $tr(req_1)$, **$tr(ev_1)$**, $gen(grant_1)$, $ex' \triangleright w$.*

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) [26] without the next-time operator. Model checking safety properties can be reduced to handling properties of the form $\forall Gp$ for a state formula p[3] [18], which means that along every execution path, p globally holds (every execution path satisfies Gp). That is, every reachable configuration satisfies p. We therefore assume $\varphi = \forall Gp$. 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.

[3] In LTL, the syntax of this property is AGp. We choose to denote it by $\forall Gp$ in order to differentiate the property from **AG** (which stands for Assume-Guarantee).

Theorem 4. *Let ex be an execution over EV and let p be a property over events $EV' \subseteq EV$. Then $ex \models Gp$ iff $ex \downarrow_{EV'} \models Gp$.*

4 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 Sect. 1) holds for UML state machines, and present a framework for applying **Rule AG-UML** for UML state machines (Sect. 4.1). We give a detailed description of the framework in Sects. 4.2 and 4.3, discuss its correctness in Sect. 4.4, and present a performance analysis in Section 4.5.

4.1 A Framework for Employing Rule AG-UML and Its Correctness

First, we formally define the meaning of the two premises in **Rule AG-UML**: $[A]M\langle\forall Gp\rangle$ 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 $ex \in L_{ex}(M)$, $ex \downarrow_{EV(A)} \in L_{ex}(A)$.

Theorem 5. *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 $\varphi = \forall Gp$. Then* **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 2) 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. *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 $t \in EEnv(M_1||M_2)$ and $n = 0$: add (t, ϵ) to Σ only if either t is part of φ or there exists $\sigma_1 = (t, (e'_1, ..., e'_k)) \in \Sigma$ s.t. $k > 0$.*

Example. *Let $Sys = server||client$ where $server$ is M_1 and $client$ is M_2, and let $\varphi = \forall G(\neg(InQ(grant_1) \wedge InQ(deny_1))$. 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 7. *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 $ex \triangleright w$ and $ex \downarrow_{EV(M_1)} \in L_{ex}(M_1)$, 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. $ex \triangleright w$. 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 8. *$\langle true \rangle M_1 || M_2 \langle \varphi \rangle$ holds iff for every execution $ex \in L_{ex}(M_2)$, there exists $w \in A_w$ s.t. $ex \triangleright w$, where A_w is the weakest assumption w.r.t. M_1 and φ.*

Proof Sketch. *The proof of direction $\Leftarrow$ is based on the definitions of executions (full proof available in [22]). For the proof of direction $\Rightarrow$, assume there exists an execution $ex_1 \in L_{ex}(M_2)$ and no word $w \in A_w$ s.t. $ex_1 \triangleright w$. Thus, there exists a word $w \in \Sigma^* \setminus A_w$* s.t. *$ex_1 \triangleright w$. We show that $\langle true \rangle M_1 || M_2 \langle \varphi \rangle$ does not hold. If $w \notin A_w$, then there exists an execution ex_2* over *$EV(\Sigma) \cup EV(M_1)$ s.t. $ex_2 \downarrow_{EV(M_1)} \in L_{ex}(M_1)$, $ex_2 \triangleright w$,* and *$ex_2 \not\models Gp$. We then construct an execution ex by combining ex_1* and *ex_2. Our construction ensures that $ex \downarrow_{EV(M_i)} \in L_{ex}(M_i)$ for $i \in \{1, 2\}$. We conclude that $ex \in L_{ex}(M_1 || M_2)$, and show that $ex \not\models Gp$ as well. Note that the construction of ex is not straightforward; ex_1 and ex_2 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_2 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.* □

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 9. *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 $\forall Gp$. 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.

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 system Sys where some of its state machines include err state, $L_{ex}(Sys)$ represents only the executions that do not reach err state on any of its state machines.

4.2 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 $ex \downarrow_{EV(M_1)} \in L_{ex}(M_1)$ and $ex \triangleright w$. If this holds, then (by the definition of A_w in Definition 7) $w \in A_w$ 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 $\sigma_i = (t_i, (e_1^i, e_2^i, ..., e_{k_i}^i))$, for $i \in \{1, ..., m\}$. The state machine $M(w)$ is presented in Fig. 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.

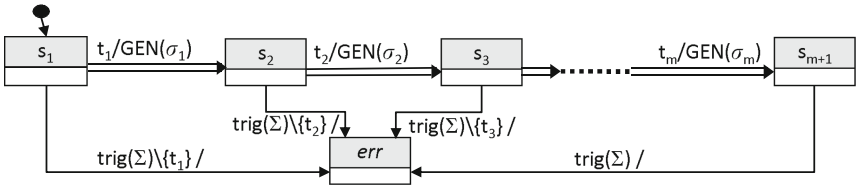

Fig. 3. $M(w)$ constructed for w

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 10. *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. $ex \triangleright w'$.*

Once $M(w)$ is constructed, the Teacher model checks $M(w)||M_1 \models \forall G(p \vee IsIn(err))$, where $IsIn(s)$ denotes that s is 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[4]. 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 11. *$M(w)||M_1 \models \forall G(p \vee IsIn(err))$ iff $w \in A_w$.*

4.3 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.

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. $ex \triangleright w$.

Definition 12 ($A(C)$ 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:*

[4] It is ok to require p on a prefix leading to state err, since A_w is prefix closed for safety properties.

(1) For every $q \in Q$ and $\sigma = (t, (e_1, .., e_n)) \in \Sigma$ where $\delta(q, \sigma) = q'$ add a compound transition labeled with $t[qs = q]/qs := q'; GEN(\sigma)$ from init to end (if $q' \neq q_{err}$) or to err (if $q' = q_{err}$).
(2) Add a transition with no trigger, guard or action from end to init.

Example. *For $Sys = server||client$ and $\varphi = \forall G(\neg(InQ(grant_1) \wedge InQ(deny_1))$, the conjecture DFA C returned from the L^* algorithm, and state machine $A(C)$ representing $L(C)$, are presented in Fig. 4.*

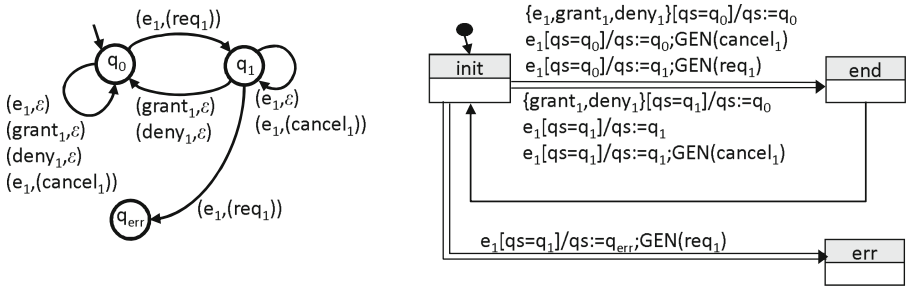

Fig. 4. The conjecture DFA C (left) and the state machine $A(C)$ (right)

The construction ensures that for every $t \in trig(\Sigma)$ and for every $q \in Q$ there exists a transition 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 13. *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. $ex \triangleright w$.*

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

Check $[\boldsymbol{A(C)}]\boldsymbol{M_1}\langle\boldsymbol{\varphi}\rangle$: Oracle 1 performs *Step* 1 in the compositional rule by model checking $A(C)||M_1 \models \forall G(p \vee IsIn(err))$. 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 $[A(C)]M_1\langle\varphi\rangle$ holds, then the Teacher forwards $A(C)$ to Oracle 2.

Check $\langle\boldsymbol{true}\rangle\boldsymbol{M_2}[\boldsymbol{A(C)}]$: Oracle 2 preforms *Step* 2 in the compositional rule. That is, check that for every execution $ex \in L_{ex}(M_2)$, $ex \downarrow_{EV(A(C))} \in L_{ex}(A(C))$. 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, $\mathcal{M}$, that is composed from modifications of M_2 and $A(C)$ as two orthogonal regions. $\mathcal{M}$ is constructed so that the behavior of M_2 is *monitored* by $A(C)$ after every RTC step. $\mathcal{M}$ 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 $\mathcal{M}$ moves to an error state, indicating that $\langle true \rangle M_2[A(C)]$ does not hold. The general structure of $\mathcal{M}$ is presented in Fig. 5.

From here on, we denote M_2 and $A(C)$ that are regions in $\mathcal{M}$ as $\hat{M}_2$ and $\hat{A}(C)$, respectively. We add a local queue, IQ, and two local variables, rtc and tr, to $\mathcal{M}$. tr "records" the event e dispatched to $\mathcal{M}$, if $e \in trig(\Sigma)$. IQ "records" events generated by $\hat{M}_2$ which are from $evnts(\Sigma)$. Whenever $\hat{M}_2$ generates an event from $evnts(\Sigma)$, it also pushes the event to IQ. $\hat{A}(C)$ 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 $\mathcal{M}$. 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 $\mathcal{M}$ is 0. $rtc = 3$ indicates that $\hat{M}_2$ is executing an RTC step that should be monitored. $rtc = 2$ indicates that $\hat{M}_2$ finished its execution, and $\hat{A}(C)$ can monitor the behavior. $rtc = 1$ indicates that the monitoring step of $\hat{A}(C)$ was successful, i.e., $\hat{A}(C)$ has a behavior that matches $\hat{M}_2$. If the monitoring of $\hat{A}(C)$ failed, then rtc at the end of the RTC step is 2, indicating an error.

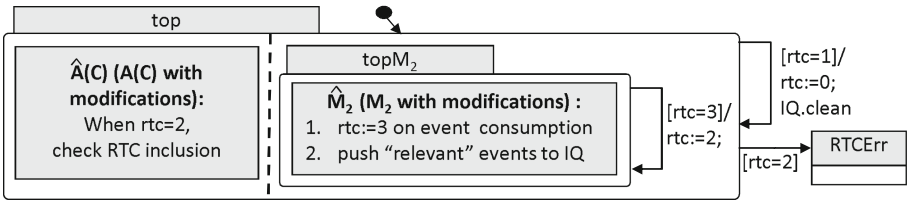

Fig. 5. General scheme for $\mathcal{M}$ created from $A(C)$ and M_2

The following modifications are applied to M_2 for constructing $\hat{M}_2$: 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 12) for constructing $\hat{A}(C)$:

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 $\hat{A}(C)$ is at state *step* and $rtc = 0$ holds, then $\hat{M}_2$ discarded the event in the current RTC step. $\hat{A}(C)$ has a matching behavior if it has a behavior that consumes an event and does not generate events. The transitions described in (3) and (5) monitor RTC steps of $\hat{M}_2$ 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 $\hat{A}(C)$, and transitions in (4) and (5) monitor RTC steps that are not legal in $\hat{A}(C)$.
The correctness of our construction is captured in the following theorem.

Theorem 14. *For every* $ex \in L_{ex}(\mathcal{M})$*: ex reaches state RTCErr iff* $ex \downarrow_{EV(M_2)} \in L_{ex}(M_2)$ *and* $ex \downarrow_{EV(A(C))} \notin L_{ex}(A(C))$.

After constructing $\mathcal{M}$, Oracle 2 model checks $\mathcal{M} \models \forall G(\neg IsIn\ (RTCErr))$. 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 $\hat{M}_2$ 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 Sect. 4.2). 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 $cex \downarrow_{EV(M_2)} \in L_{ex}(M_2)$, $cex \downarrow_{EV(M_2)} \triangleright w$ and $w \notin A_w$ (Theorem 8). The Teacher then returns *false*.

Example. *Consider the system server||client and the assumption A(C) (Fig. 4). When checking* $\langle true \rangle client[A(C)]$*, the model checker may return a counterexample cex, represented by the word* $w = (e_1, (req_1)), (e_1, (cancel_1)), (e_1, (req_1))$ $(cex \triangleright w)$. $cex \downarrow_{EV(M_2)} \in L_{ex}(client)$, $cex \downarrow_{EV(M_2)} \triangleright w$ *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* $\forall G(\neg(InQ(grant_1) \wedge InQ(deny_1)))$. *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* $server||client \not\models \varphi$.

4.4 Correctness

We first argue correctness of our approach, and then the fact that it terminates.

Theorem 15. *Given state machines* M_1 *and* M_2, *and a property* $\forall Gp$, *our framework returns true if* $M_1||M_2 \models \forall Gp$ *and* $false$ *otherwise.*

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 [12]. 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 8, either *true* or a counterexample.

4.5 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 LTSs 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 [9,28], 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. *We re-visit the example presented throughout Sect. 4.* $\varphi = \forall G(\neg(InQ(grant_1) \wedge InQ(deny_1)))$, *and* $Sys = server||client$ *where server is* M_1, *client* is M_2. *The final DFA learned when using event sequences is presented in Fig. 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 over single event occurrence, 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* $\{tr(e_1), tr(grant_1), tr(deny_1), gen(req_1), gen(cancel_1)\}$.

5 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* 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. Our framework is presented for $Sys = M_1||M_2$ where both M_1 and M_2 are state machines. However, M_1 and M_2 can both be systems that 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 $\widetilde{M_2}$ where each M_i^2 is an orthogonal region in $\widetilde{M_2}$. The executions of $\widetilde{M_2}$ are equivalent to those of M_2. We can then apply our framework on $M_1||\widetilde{M_2}$.

In the future we plan to investigate other assume-guarantee rules in the context of behavioral UML system. For example, we would like to define a framework for checking $[A_1]M[A_2]$. Such a framework will enable us to apply recursive invocation of the **AG** rule, where M_2 includes several state machines.

References

1. Majzik, I., Darvas, A., Beny, B.: Verification of UML statechart models of embedded systems. In: Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS 2002), pp. 70–77. IEEE (2002)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Booch, G., Rumbaugh, J.E., Jacobson, I.: The unified modeling language user guide. J. Database Manag. **10**(4), 51–52 (1999)
4. Strichman, O., Chaki, S.: Optimized L*-based assume-guarantee reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007)
5. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model checking large software specifications. IEEE Trans. Softw. Eng. **24**(7), 498–520 (1998)
6. Chen, Y.-F., Tsay, Y.-K., Clarke, E.M., Farzan, A., Wang, B.-Y.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT press, Cambridge (1999)
8. Clarke, E.M., Heinle, W.: Modular translation of statecharts to SMV. Technical report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science (2000)
9. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
10. Dubrovin, J., Junttila, T.A.: Symbolic model checking of hierarchical UML state machines. In: Application of Concurrency to System Design (ACSD 2008), pp. 108–117. IEEE (2008)
11. Farzan, A., Tsay, Y.-K., Chen, Y.-F., Wang, B.-Y., Clarke, E.M.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
12. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Autom. Softw.Eng. **12**(3), 297–320 (2005)
13. Object Management Group. OMG Unified Modeling Language (UML) Superstructure, version 2.4.1. formal/2011-08-06 (2011)
14. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Trans. Program. Lang. Syst. **16**(3), 843–871 (1994)

15. Meller, Y., Yorav, K., Grumberg, O.: Applying software model checking techniques for behavioral UML models. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 277–292. Springer, Heidelberg (2012)
16. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Form. Methods Syst. Des. **32**(3), 285–301 (2008)
17. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
18. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Form. Methods Syst. Des. **19**(3), 291–314 (2001)
19. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the spin model-checker. Formal Asp. Comput. **11**(6), 637–664 (1999)
20. Madhukar, K., Metta, R., Singh, P., Venkatesh, R.: Reachability verification of rhapsody statecharts. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013), pp. 96–101. IEEE (2013)
21. Grumberg, O., Meller, Y., Yorav, K.: Verifying behavioral UML systems via CEGAR. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 139–154. Springer, Heidelberg (2014)
22. Meller, Y., Grumberg, O., Yorav, K.: Learning-based compositional model checking of behavioral UML systems. Technical report CS-2015-05, Department of Computer Science, Technion - Israel Institute of Technology (2015)
23. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in PROMELA/SPIN. In: Workshop on Industrial-Strength Formal Specification Techniques (WIFT 1998), pp. 90–101. IEEE (1998)
24. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. Form. Methods Syst. Des. **32**(3), 207–234 (2008)
25. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Form. Methods Syst. Des. **32**(3), 175–205 (2008)
26. Pnueli, A.: The temporal logic of programs. In: Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science (FOCS 1977) (1977)
27. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Formal Models of Concurrent Systems, pp. 123–144. Springer-Verlag, Berlin (1985)
28. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. In: Symposium on Theory of Computing (STOC 1989), pp. 411–420. ACM (1989)
29. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The rhapsody UML verification environment. In: Software Engineering and Formal Methods (SEFM 2004), pp. 174–183. IEEE (2004)