Bounded Verification of State Machine Models

Nafiseh Kahani
University of Ottawa
nkahani@uottawa.ca

James R. Cordy
Queen's University
cordy@cs.queensu.ca

Abstract

In this work, we propose a bounded verification approach for state machine models that is independent of any model checking tools. This independence is achieved by encoding the execution semantics of state machine models as SMT formulas that reduce the verification of a state machine to the satisfiability problem for its corresponding formula. More specifically, our approach takes as input a state machine model, a depth bound, and the system properties (as invariants), and then automatically verifies models of systems in a three-phase process: (1) First it generates all possible execution paths of the model to the specified bound, and encodes each of the execution paths as SMT formulas; (2) It then augments the SMT formulas with the negation of the given invariants; and (3) Finally, it uses an SMT solver to check the satisfiability of the instrumented formula. We have applied our approach in the context of UML-RT (the UML profile for modeling real-time embedded systems) and assessed the applicability, performance, and scalability of our approach using several case studies extracted from the literature.

ACM Reference Format:
Nafiseh Kahani and James R. Cordy. 2020. Bounded Verification of State Machine Models. In *Proceedings of ACM Conference (Conference'20).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Model-Driven Development (MDD) has been successfully applied in the development of Real-Time Embedded (RTE) systems, which are supported by industrial-strength tools such as IBM Rational Software Architect (IBM RSA RTE). State Machines (SMs) and their variants (e.g., UML-RT state machines [1]) are one of the main notations supported by tools to model the behaviour of RTE systems. While many aspects of software development can potentially be automated using SMs, thus far automation is mainly limited to code generation, interpretation, and documentation. Other important software development activities, such as testing and verification of SMs, have yet to be fully automated [2].

The majority of the existing work on automatic verification and analysis of SMs leverages existing model checking tools such as SPIN [3] by translating the SM to the input language of the model checker. This "translational" approach can take advantage of the advanced optimizations and features of the existing model checkers. However, model checkers have typically been designed for specific domains, for example, SPIN was developed to verify network protocols [4]. This leads to a significant semantic mismatch, as many features of modeling languages (e.g., the hierarchical structure of SMs) are not directly supported by the input languages of the model checkers. This complicates the translation [5], which often involves abstraction, simplification and complex transformations, verification of which is required to ensure that the original intent of the SMs is preserved. In addition, the output of model checkers (e.g., counterexamples) is expressed in terms of the model checkers' concepts rather than the SM. Thus, utilizing model checkers also involves a reverse translation of their output back to the modeling language, which again must be verified. These reverse translations are critical, because without them the user is forced to match model checker concepts to the original models by hand, which can be confusing and error prone [4].

Satisfiability Modulo Theories (SMT) extends Boolean satisfiability (SAT) by supporting useful first-order logical theories (e.g., arithmetic and bit-vectors) in addition to Boolean logic. SMT solvers such as Z3 [6], whose performance has improved significantly in the last two decades, are the key enabling technologies for various software verification and analysis applications. Some practical examples are (but are not limited to) verification of device drivers [7], bounded model checking [8], and program synthesis [9]. With the exception of one or two other attempts [10, 11], the use of SMT solvers for the verification and analysis of models is limited to static models (e.g., class diagrams) and OCL constraints.

Inspired by existing program verification and analysis techniques, this work relies on SMT solvers to address the bounded verification of SM models, specified using UML-RT [12], the UML profile for development of RTE systems. More specifically, our approach accepts as input the system properties (as invariants), a bound for the verification, and a behaviour model of the system. It then: (1) analyzes the model and extracts all possible execution paths of the model to the specified bound, (2) encodes each of the execution paths and instruments them with the given invariants (as a path formula and instrumented path formula) and (3) uses SMT solvers to verify the models by solving the instrumented path formulas. We have evaluated our approach using a number of different case studies whose state-space varies from small to large. The evaluation shows that the performance is reasonable, and the approach can successfully verify the specified properties in all of our cases.

This work complements previous work in the area of SM verification by providing a direct bounded verification approach for SMs that is independent of existing model checkers. The most important contributions of this work are: (1) A systematic approach and relevant formalization to encode the execution of SMs (more specifically UML-RT state machines) as SMT-formulas. The encoding fully respects the semantics of the SMs, without abstracting, simplifying or ignoring any features. (2) A method for leveraging

Conference'20, October 2020, Montreal, Canada

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

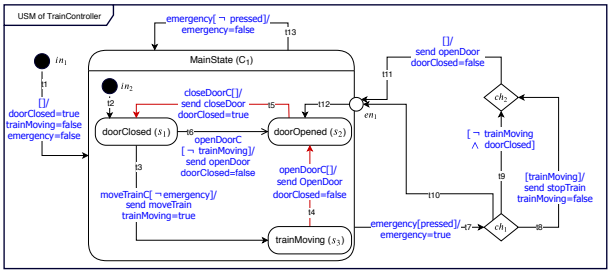

Figure 1: Behaviour of the train controller *CTR*

SMT solvers for the practical bounded verification of SMs. Similarly to model checking techniques [13] that use binary decision diagrams (BDD) [14], bounded verification techniques can suffer from scalability issues (the path explosion problem). However, they allow users to limit the explosion by setting a path depth bound, trading completeness for practicality and scalability. Fortunately, the majority of errors in practical systems can be found using verification to a reasonable bound [15].

The remainder of this paper is organized as follows. In Section 2, we provide background on the specification and modeling formalisms used in our solution, and introduce a running example. In Section 3, we provide a detailed description of the individual phases of our bounded verification approach. Section 4 evaluates our approach by analyzing its applicability and performance in a number of case studies. Finally, we overview related work in Section 5, and conclude in Section 6.

2 Background

In this section, we first introduce an illustrative example. Then, we describe the terms and notations we use to model a system, the execution semantics of the models, and expressing system properties using invariants.

2.1 An Illustrative Example

We use a simplified controller of a train system [16] to illustrate important concepts throughout the paper. The system is composed of five components: a train *Controller (CTR)*, an *Environment (ENV)*, a train *Sensor (SEN)*, a *Door*, and an *Engine*. We assume that users' inputs are handled by *ENV*. The *SEN* component reads all sensors, e.g., detecting if a passenger presses the emergency key. The *CTR* component receives input from the *SEN* and *ENV* components, and controls actuators for the *Engine* and *Door*. Component *CTR* can receive input messages *openDoorC, closeDoorC, moveTrainC, stopTrainC,* and *emergency(pressed)*, where *pressed* is the message payload of type Boolean that indicates whether the passenger has pressed the emergency key. It also can produce output messages *openDoor, closeDoor, moveTrain,* and *stopTrain*. In addition, *CTR* has three Boolean variables *emergency*, *trainMoving*, and *doorClosed* that encodes the train's emergency status, whether the train is presently in motion, and whether the door is closed.

The behaviour of component *CTR* is shown in Fig. 1 using SM (Def. 4), where each transition is annotated with '*trigger[guard]/actions*'. Note that the SM intentionally contains faults (red transitions) to illustrate our approach. In this paper, we focus on two safety properties that the behaviour of component *CTR* must meet. (*P1*) The *door* can not be closed when the *emergency* key is pressed. (*P2*) The *train* can only move when the door is closed.

2.2 UML Profile for Real-Time Systems (UML-RT)

UML-RT [12] is a language that specifically designed for Real-Time Embedded (RTE) systems, with soft real-time constraints. Over the past two decades, it has been used successfully in industry to develop several large-scale industrial projects (e.g., [17]), and has a long, successful track record of application and tool support, via, e.g., IBM RSA-RTE [18], and Papyrus-RT [19]. In this paper we use UML-RT to evaluate and illustrate our approach. In the following, we present a concise formalization of UML-RT which will be required to understand our approach. A more detailed discussion of UML-RT can be found in [12].

Definition 1. (Read function) Let tp be a tuple $\langle r_1 \dots r_n \rangle$ where $r_1 \dots r_n$ refer to the names of the tuple entries. We use $tp.r_i$ to denote reading the value of entry r_i. For example, we use *person.name* to read the value of entry *name* of tuple $person\langle name, family \rangle$.

Definition 2. (Model Structure of an RTE System) We model a system as a set of communicating components. A component c is defined as a tuple $\langle I, O, V, B \rangle$, where I denotes the input messages that the component can receive, O denotes the output messages that the component can produce, V is a set of variables of types *Integer* and *Boolean*, and B refers to the behaviour of the component, which is defined using a UML-RT State Machine (USM) (Def. 4).

Definition 3. (Action Language) We assume the existence of an *action language* that supports the primitive operations: (1) accessing/updating variables, (2) arithmetic/Boolean expressions, and (3) sending messages. Since our work is focused at the model-level, we assume a concise and simple action language without control flow constructs such as *while* loop. Encoding other programming languages' constructs as SMT formulas has been addressed extensively in other work [20].

Definition 4. (UML-RT State Machine (USM)) We specify the behaviour of a component c using a *USM*, defined as a tuple $\langle S, T, C \rangle$. $S = S_b \cup S_c \cup S_p$ is a set of states, T is a set of transitions, and $C \subseteq S_c \times (S \cup T)$ denotes an acyclic containment relationship. States can be basic (S_b), composite (S_c), or pseudo-states (S_p). Basic states are primitive states that the execution stays in until an outgoing transition is triggered. Composite states encapsulate a sub-state machine. Pseudo-states are transient control-flow states. There are six kinds of pseudo-states, including *initial*, *choice-point*, *history*, *junction-point*, *entry-point*, and *exit-point*, (i.e., $S_p = S_{in} \cup S_{ch} \cup S_h \cup S_j \cup S_{en} \cup S_{ex}$). Composite and basic states can have entry and exit actions expressed using the action language.

Definition 5. (Transition) A transition t is a 5-tuple $(src, guard, trig, act, des)$, where $src, des \in S$ refer to non-empty source and destination states of the transition respectively, *guard* is a logical expression coded using the action language, *trig* is a set of messages that trigger the transition, and *act* is the transition's action, also expressed using the action language.

Definition 6. (Execution of a USM) We use a Labeled Transitional System (LTS), consisting of a tuple $\langle \Gamma, \mathcal{A}, \gamma_0, H, R \rangle$ to define

the execution semantics of a *USM*, where Γ is a set of configurations, $\mathcal{A}$ is the set of actions (i.e., entry, exit, and transition actions defined in the *USM*), $\gamma_0 \in \Gamma$ is the initial configuration, H is a mapping function from composite states to their last visited sub-states (if any), and R is a transition relation (to avoid confusion with the syntax of *USM*s, we use the term 'execution step' instead of 'transition' in the rest of this paper). A configuration $\gamma \in \Gamma$ is defined as a tuple $\langle \sigma, E \rangle$ where $\sigma \in \mathcal{S}$ refers to the execution state of the configuration, and E contains values of the component variables at the configuration.

The execution of a *USM* begins at γ_0 where $\gamma_0.\sigma$ is the initial state of the *USM*, and $\gamma_0.E$ is created based on the default values of the variables (Integer and Boolean variables, set to 0 and false respectively). Execution continues if a new configuration exists that has an execution step relationship with the current configuration.

Figure 2 defines four rules in the form of operational semantic rules [21] that specify when two configurations, current configuration, $\gamma = \langle \sigma, E \rangle$, and new configuration, $\gamma' = \langle \sigma', E' \rangle$, have the execution step relationship (i.e., $(\gamma, \gamma') \in R$). The presentation of the rules makes use of the definitions in Table 1. The rules are defined based on the execution semantics of UML-RT described in [12, 22]. Details of the rules are as follows:

Rule 1: This rule is applicable to configurations whose execution state is one of the pseudo-states, except for history and choice-point. According to Rule 1, an execution step is taken from γ to γ' ($(\gamma, \gamma') \in R$), if there is an outgoing transition from the execution state of the current configuration ($\gamma.\sigma$) that executes the related actions and moves the execution to a new configuration (γ').

Rule 2: This rule is applicable to configurations whose execution state is a basic state. If a transition can be enabled (Table 1) from the execution state of configuration γ, an execution step is taken from γ to γ' that executes the related actions and moves the execution to a new configuration (γ').

Rule 3: This rule is applicable to configurations whose current state is a composite state (implicit history state). If function $next\_s(\sigma, \mathcal{H})$ (Table 1) returns a state, then an execution step is taken from γ to γ' that executes the related actions and moves the execution to a new configuration (γ').

Rule 4: This rule is applicable to configurations whose current state is a choice-point. Guards of the outgoing transitions from the execution state of γ are evaluated, and the first transition whose guard evaluates to *true* is selected. The result is an execution step from γ to γ' that executes the related actions and moves the execution to a new configuration (γ').

Definition 7. (System Execution and Run-to-Completion Semantics) The execution of an RTE system can be defined as a collection of its components' *USM* executions, which interact with each other by passing messages. We do not describe the details of the composition here, and we assume that the RTE system execution is managed by a controller. The controller is responsible for scheduling and message-passing between components, and guarantees that an incoming message will be fully processed before the processing of the next message begins (run-to-completion semantics).

Definition 8. (Execution Path) In the context of the execution of a *USM*, an execution path π with length n is defined as a sequence of configurations $\gamma_0\gamma_1...\gamma_n$ for $n \in N$, where $\forall i < n$, $(\gamma_i, \gamma_{i+1}) \in R$ and γ_0 is the initial configuration of the *USM*. The *length* of a path is equal to the number of configurations in the path.

$$\frac{\sigma \in S_p \setminus (S_h \cup S_{ch}),\ t = out\_t(\sigma)}{(\langle \sigma, E \rangle, \langle t.des, E' = apply(E, acts(\sigma, t)) \rangle) \in R} \quad (1)$$

$$\frac{\sigma \in S_b\,, t = enabled(\sigma)}{(\langle \sigma, E \rangle, \langle t.des, E' = apply(E, acts(\sigma, t)) \rangle) \in R} \quad (2)$$

$$\frac{\sigma \in S_c, s = next\_s(\sigma, H)}{(\langle \sigma, E \rangle, \langle s, E' = apply(E, acts(\sigma, s)) \rangle) \in R} \quad (3)$$

$$\frac{\sigma \in S_{ch}, t \in out\_t(\sigma) \wedge eval(E, t.guard)}{(\langle \sigma, E \rangle, \langle t.des, E' = apply(E, acts(\sigma, t)) \rangle) \in R} \quad (4)$$

Figure 2: Membership rules of execution step relationship of configurations $\gamma = \langle \sigma, E \rangle$ and $\gamma' = \langle \sigma', E' \rangle$

Table 1: Helper functions

Function	Description
out_t(s)	returns outgoing transitions from state *s*.
parent(s)	returns the first-level container state of state *s*.
parents(s)	returns all container states of state *s*.
$next\_s(s, \mathcal{H})$	(1) returns the last visited state in state *s* from history $\mathcal{H}$, (2) if (1) is unsuccessful (i.e., the state is active for the first time), returns the initial state inside *s*, and (3) if (1) and (2) are unsuccessful, returns $\emptyset$.
enabled(s)	checks state *s* and its ancestors in bottom-up order, and returns the first (i.e., most deeply nested) outgoing transition, which can be triggered by the received message. It returns $\emptyset$ if no transition can be triggered.
eval(E, g)	evaluates guard *g* based on the values in *E* and returns the result.
$msg_v(m)$	accepts a message *m* and returns a variable msg_v_m.
payload(m)	accepts message *m* and returns its payloads.
$apply(E, a_1 \ldots a_n)$	executes a sequence of actions $a_1 \ldots a_n$ based on the values in *E* and returns the updated *E*.

Definition 9. (Well-formedness of a *USM*) Following [12, 18, 19], we define the well-formedness constraints of a *USM* as follows: (1) There are no *AND-state*s (orthogonal regions), and no UML *fork*, *join*, *shallow history*, or *final states*. (2) Any transition to a composite state is assumed to end in an implicit history state inside the composite state. (3) Triggers of transitions starting from the same basic or composite state must be disjoint.

Definition 10. (System Properties) System properties (S_{inv}) capture the system's required properties in the form of OCL-like invariants. S_{inv} denotes a set of invariants defined as quantifier-free first-order logic formulas, all of which must hold during the entire execution of the system. E.g., in the context of the illustrative example, the safety properties *P1* and *P2* can be captured using:

```
Invariant P1 {emergency ⟹ ¬ closed}   Invariant P2 {moving ⟹ closed}
```

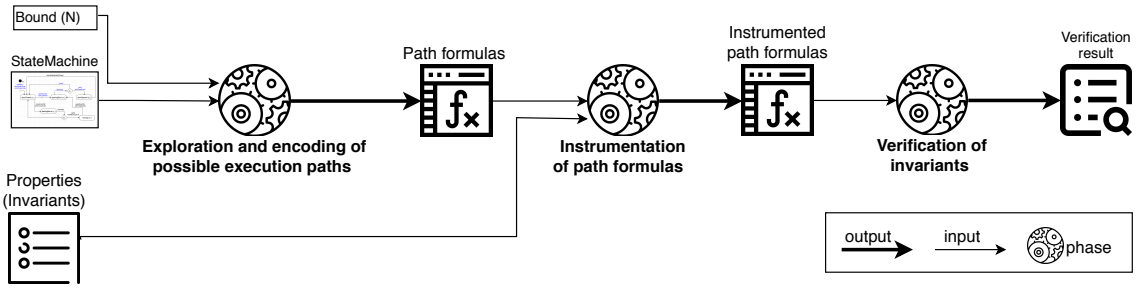

Figure 3: Overview of our approach

3 Approach

Figure 3 presents an overview of our approach for bounded verification of invariants of a system's behavior. We assume as inputs the behavior of the system in the form of a USM, a depth bound, and the safety properties in the form of system invariants expressed in our OCL-like notation (Def. 10). Table 1 lists a set of helper functions that are used in the remainder of this paper. Our approach consists of three phases, as follows.

Phase 1. Bounded exploration and encoding of execution paths. This phase takes a USM and a depth bound as input, and explores all possible execution paths of the USM, based on its execution semantics (Def. 6) until the given bound is reached. During the exploration, each path is encoded as an SMT formula. By encoding the execution paths as SMT formulas, the execution of the path is reduced to solving the path formula. Thus, we check the satisfiability of the path formula to see whether or not the path is a possible path (reachable) during execution. Unsatisfiable paths are discarded during the exploration, since they are not reachable.

To encode a path, variables' values in each configuration (E) of the path are encoded based on the variables' values in its previous configuration. To do that, the relevant actions are converted to single static assignment (SSA) programs [23].

Phase 2. Instrumentation of path formulas. In addition to encoding the execution paths, this phase encodes the defined invariants as SMT formulas, to allow their verification. To verify if an invariant holds in an execution path π, we first create a negation of the invariant, and then instrument the path formulas to capture the negation of the invariant in all configurations of π. In this way, the verification of the invariant is reduced to checking the (non-) satisfiability of the instrumented path formulas.

Phase 3. Verification of invariants. Finally, to verify the invariants, this phase checks the satisfiability of the instrumented path formulas. Since the invariants are negated in Phase 2, satisfiability of the instrumented formula of a path π implies that the invariant is violated in the path. In this case, based on the satisfiability models (assignment of variables at each configuration of the execution path) and the execution path, a counterexample is provided. Otherwise, if none of the instrumented formulas is satisfiable, we conclude that the invariant holds to the given bound.

Algorithm 1: Bounded Exploration of Execution Paths

```
1  Input A component c = ⟨inp, out, vars, usm⟩, a bound
2  Output A set of execution paths (paths)
3  Let γ0 be the initial configuration of usm as
       defined in Def. 6
4  Let h be a map (from composite state to state)
       that keeps history
5  Let paths be an empty set
6  Let encodedPaths be an empty set
7  explorePath(<γ0>,bound,h)
8
9  Function explorePath(Sequence  path, Integer bound
       , Map h)
10   if (bound=0)
11     return
12   Let f be the encoding of path as an SMT formula
13   if (f is satisfiable)
14       Add a clone of path to paths
15       add f to encodedPaths
16   else
17       return
18   Let γ denote the last element of path
19   if (γ.σ ∈ Sb)
20     for (msg ∈ c.I) # all input messages of c
21       if (enabled(γ.σ, msg))
22         t=enabled(γ.σ, msg)
23         h[parent(t.des)]=t.des
24         append configuration ⟨t.des, {}⟩ to path
25   else if (γ.σ ∈ Sin ∪ Sex ∪ Sj ∪ Sen ∪ Sch)
26     Let trans be out_t(γ.σ)
27       for (t ∈ trans)
28       append configuration ⟨t.des, {}⟩ to path
29       if (path is satisfiable)
30           explorePath(path, bound − 1, h)
31   else if (γ.σ ∈ Sc)
32     Let s be next_s(γ.σ, h)
33     append configuration ⟨s, {}⟩ to path
34     explorePath(path, bound − 1, h)
```

3.1 Bounded Exploration and Encoding of Execution Paths

3.1.1 Bounded Exploration of Execution Paths. Algorithm 1 presents our method for the exploration and encoding of possible execution paths of a USM to a specific bound. It accepts a component c and a bound as input, and produces sets of possible execution paths (*paths*) and their encoding as SMT formulas (*encodedPaths*). The

algorithm first creates the initial configuration of the component's *USM* and calls the recursive function *explorerPath* that traverses the USM's states based on the execution semantics of the USM (Def. 6) to populate the execution paths. Function *explorerPath* encodes the path as an SMT formula (Sec.3.1.2), and checks its satisfiability. Upon unsatisfiability of the formula, or reaching the bound, it terminates. Otherwise it branches based on the execution state of the last configuration in the explored path, as follows:

Basic state: When the execution reaches a basic state s_b (i.e., the execution state of the last configuration is a basic state), it can progress based on the received messages and outgoing transitions of the execution state and its ancestors. Thus, to explore all possible execution paths after the execution reaches s_b, the function iterates on all possible input messages of the component, and checks if their reception can branch the execution. For all possible execution branches from s_b, separate execution paths are saved whose configurations are the same until reaching s_b.

Pseudo state: When the execution reaches a pseudo state s_p (i.e., the execution state of the last configuration is a pseudo state), it can branch to any of the outgoing transitions from s_p. Thus, the function explores all outgoing transitions from s_p and saves them as separate execution paths whose configurations are the same until reaching s_p.

Composite state: As discussed above, any transition to a composite state is assumed to be ended at a history state. Thus, the execution treats a composite state as a history state. When the execution reaches a history state s_c, it can only progress on a single path based on the value of the history. Thus, the function calculates the next execution state ($s = next\_s(s_c, H)$), and appends it to the path with configuration $\langle s, \{\}\rangle$.

Note that, the algorithm sets the variables' values in the configuration to {}. The variables' values will be calculated symbolically as SMT formulas based on the relevant actions.

As an example, there are 7 execution paths with bound 5 (paths with length 5 or smaller) in the context of the train controller's USM, and the following is an example path with bound 5.

$\langle \langle in_1, \{\}\rangle \langle C_1, \{\}\rangle \langle in_2, \{\}\rangle \langle s_1, \{\}\rangle \langle s_2, \{\}\rangle \rangle$

3.1.2 Encoding the Execution Paths as SMT Formulas. To capture an execution path as an SMT formula (path formula), first the set of variables is defined, based on which all configurations of the path can be captured. Second, variables' values in each configuration are captured as a conjunction of equality checks in which each variable value is formulated based on its value in the previous configuration. In the following, we discuss the details of these three steps.

Definition of Variables. We assume that V^c is a set that contains all the variables of component c. The set of variables of the path π with length n of component c is then defined as:

$$V^{\pi} \leftarrow V^{c}_{0\ldots n} \cup V^{msg}_{0\ldots n} \cup V^{pay}_{0\ldots n}$$

where

$$V^{c}_{0\ldots n} \leftarrow \bigcup_{\forall\, i\in 0\ldots n} \; \bigcup_{\forall v\in V^c} v_i$$

includes all variables' values in all configurations of the path, where i denotes the identifier of each configuration in the path. Next,

$$V^{msg}_{0\ldots n} \leftarrow \bigcup_{\forall\, i\in 0\ldots n} \; \bigcup_{\forall\, m\in c.I\cup c.O} msg_v(m)_i$$

Algorithm 2: Capture a Configuration as an SMT Formula

```
1  Input: Configuration γ_i, γ_{i-1}, Sequence actions between γ_i
       and γ_{i-1}, Component c
2  Output: a formula f
3  Let f be a formula and set it to true
4  ssaactions ← Convert2SSA(actions, c)
5  for (v in variables of c)
6    Let v_i refer to variable v in configuration γ_i
7    create an equality check statement eq and set its LHS to
         v_i
8    Set RHS of eq with calcV(v,ssaactions, γ_i,γ_{i-1})
9    Let f be f ∧ eq
10 Append the f based on the values of message variables
       resulting from Convert2SSA
11
12 Function Convert2SSA(Sequence actions, Component c)
13   Let V^c refer to the variables of c
14   Let V^msg be the message variables of component c (i.e.,
         ⋃_{m∈c.I} msg_v(m)) that are set to false
15   Let assignCounts be a map from V^c to their assignment
       count that is 0 at the beginning
16   for (act in actions)
17     if (act is assignment)
18       replace v in LHS of act with an auxiliary variable
             v_{assignCounts[v]+1}
19       replace v in RHS of act with an auxiliary variable
             v_{assignCounts[v]}
20       increase assignCounts[v] by one for v in LHS of act
21     else (act is send action)
22       Let m be the messages which are sent by act
23       Set msg_v(m) in V^msg to true
24       Set the payload of m if any
25     return transformed actions
26 Function calcV(Variable v_l, SSAActions acts,
       Configuration γ_i, γ_{i-1})
27     set v_l from its value in γ_{i-1} (i.e., v_{l i} = v_{l i-1})
28     Let act be the last assignment of v_l
29     for (v_r in RHS of act)
30       if (index of v_r is larger than zero)
31         replace v_r in act with calcV(v_r,acts,γ_i, γ_{i-1})
32     replace the variables in RHS with index zero with
           variables from γ_{i-1}
33   return the RHS of the updated act
```

consists of *Boolean* variables to capture reception of input messages or generation of output messages at each configuration. Finally, we have

$$V^{pay}_{0\ldots n} \leftarrow \bigcup_{\forall\, i\in 0\ldots n} \; \bigcup_{\forall\, m\in c.I\cup c.O} payload(m)_i$$

which consists of variables to capture the payload of messages at each configuration.

For example, V^{π}, the variables of the path for a path with length 2, in the context of the running example, would be:

$V^c_0 = \{trainMoving_0, emergency_0, doorClosed_0\}$,
$V^c_1 = \{trainMoving_1, emergency_1, doorClosed_1\}$,
$V^{mqs}_0 = \{msg_v\_moveTrainC_0, msg_v\_openDoorC_0, ...\}$,
$V^{mqs}_1 = \{msg_v\_moveTrainC_1, msg_v\_openDoorC_1, ...\}$,
$V^{pay}_0 = \{pressed_0\}$, $V^{pay}_1 = \{pressed_1\}$

Capture the Execution Path as an SMT Formula. Let π be an execution path $\gamma_0, \gamma_1 \ldots \gamma_n$. To create a path formula for π, first we capture all of its configurations as formulas, and then create a path formula, which is a conjunction consisting of formulas for all of its configurations. To capture a configuration γ_i as a formula, first the relevant actions between γ_i and its previous configuration γ_{i-1} are transformed into static single assignment (SSA) form [23] in which

each variable is assigned only once. Second, we calculate the values of each variable based on the values in the previous configuration. Finally, we convert the assignments to equality checks, and create a formula that is a conjunction of the equality checks of all variables. The related guard is considered in conjunction with the formula.

Algorithm 2 presents the details of our method for capturing a configuration as an SMT formula. It accepts a configuration (γ_i), the prior configuration in the path (γ_{i-1}), the relevant actions, extracted according to the definition of the execution step relationship (Figure 2), and a component c. It calls the function *convert2SSA* which converts the actions to SSA form, then iterates on all variables of the component, and calculates the variables values in configuration γ_i based on γ_{i-1}. To do that, it calls *calcV* which calculates the variables values based on the last assignment of the variables, by replacing all intermediate variables in the RHS (the right hand side of the statement) with the initial variable (i.e., the variable with index zero). Intuitively, the initial variables refer to the variables values in the previous configuration. Thus, after calculations based on the initial variables, they are replaced with variables of configuration γ_{i-1}. The algorithm converts the assignments to equality check expressions, and creates a formula that is the conjunction of all equality checks. If a variable is not assigned by actions, its value is set to the variable value from γ_{i-1}, and the message and payload variables are also set based on the results of function *convert2SSA*.

Transformation of Actions into SSA Form. Function *convert2SSA* in Algorithm 2 accepts a sequence of actions and converts it to SSA form. It first creates a map *assignCounts* from variables to an integer that counts the number of assignments of each variable. At the beginning, the count of assignments of each variable is set to zero. The function then iterates over all actions, and replaces the referenced variables by the actions with an auxiliary variable created by subscripting the original variable. The subscript of each auxiliary variable is set based on the number of assignments of the variable prior to the current action, that is, (a) the index of the variable v in the LHS (the left hand side of the statement) is set to $assignCounts[v] + 1$, and (b) the variable v in the RHS is set to $assignCounts[v]$. The first step in Figure 4 is shown as an example of how a sequence of actions is translated to SSA form. The function also processes send message actions by setting the variables of the relevant messages and their payload. This allows users to define invariants related to message sending.

Figure 4 shows how a configuration γ_i with three variables x, y, z of a path π is captured as a formula, assuming the left box of the figure denotes the relevant actions between γ_{i-1} and γ_i.

3.2 Instrumentation of Path Formulas

As discussed, the invariants capture the system properties that must hold during the entire execution of a component. To verify whether an invariant holds for an execution path, we need to verify that it holds in all of its configurations. In bounded verification techniques, proof by contradiction is used rather than checking and proving that the invariant holds in all possible situation. To do that, a negation of the invariant is created, whose satisfiability is checked with the help of the SMT solver. If the negation of the invariant is satisfied, then the invariant is violated; otherwise, we can conclude that the invariant holds until the defined bound of the path is reached. To check the satisfiability of an invariant for an execution path, each configuration of the path is instrumented (by adding an assertion) based on the negation of the invariant. The instrumenting of a configuration (e.g., γ) is performed by replacing the used variables in the invariant with the variables of γ.

According to the run-to-completion semantics, the execution of a USM is not interrupted between moving from a configuration whose execution state is a basic state and reaching another configuration whose execution state is a basic state. Therefore, the instrumentation of a configuration whose execution state is not a basic state is unnecessary, and may cause spurious results for the verification. To prevent that, the instrumentation is applied only in configurations whose execution state is a basic state, as well as in the initial configuration of the path.

3.3 Verification of Invariants

Generation of a Counter-Example. When the instrumented formula of a path π is satisfied, the assignments of variables relevant to satisfying models, along with the path, provide enough information to construct an instance of the execution path (a trace) that violates the invariant. This is simply performed by iterating on all configurations of the path, and setting its variables by reading them from satisfying models. It is also possible to reconstruct the violating execution by execution of the actions along the path, which require an interpreter or simulator. Our work presents the violating execution based on the variables' values from the satisfying model.

Verification of the USM's Execution and Managing Bound. Our previous discussion is mainly focused on the verification of an invariant along a path. It is worth mentioning that to verify an invariant in the context of the USM to a certain bound, the invariant is checked against all possible paths to the specified bound. In addition, when a bound n is specified for the verification, the verification process is started from bound 1 and is incrementally applied to all bounds between 1..n. This is helpful for the efficiency of the verification and generation of violating execution paths with minimum length, which are easier to understand.

Figure 5 shows an execution path of length 4 instrumented with invariant *P1* (Def. 10). Note that only configurations whose executions state are basic state are instrumented.

3.4 Limitations

Our current implementation of the approach has the following limitations, all of which can be addressed in future work.

(1) We check UML-RT state machines in isolation. In UML-RT, components can only communicate with each other by sending of messages, and in our approach we consider all the possible input messages of each component. Therefore, our approach does not make any assumption concerning the behaviour of the relevant components (compositions) and verifies the state machine for the worst-case scenario. This can lead to counterexamples that are not valid when the state machine is verified in composition with other components. We leave handling composition to future work.

(2) Our approach only supports the encoding of loop and branch free actions. The encoding of loop and branch statements has been addressed extensively in the context of programming languages [15, 24], and importing it into our approach is left for future work.

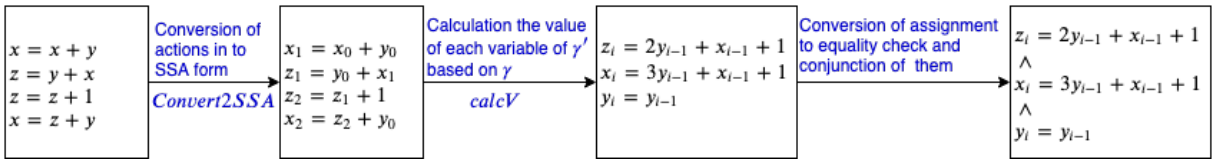

Figure 4: An example of encoding configurations γ_i and γ_{i-1}, assuming that the leftmost box contains the relevant actions.

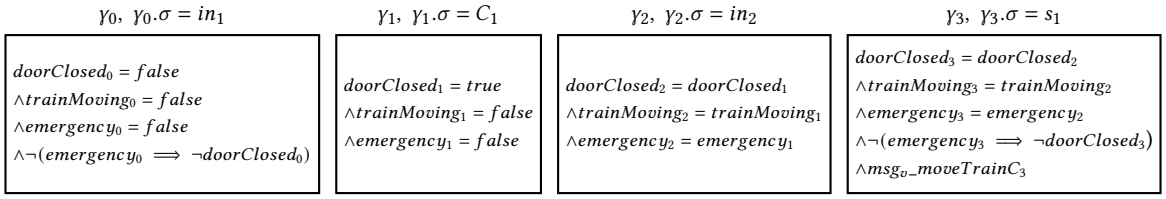

Figure 5: The instrumented path formula of path $\langle \langle in_1, \{\}\rangle \langle C_1, \{\}\rangle \langle in_2, \{\}\rangle \langle s_1, \{\}\rangle \langle s_3, \{\}\rangle \rangle$

Note: Only the first four configurations are shown. Message variables with $false$ value are emitted and the label above each box denotes the configuration id and its execution state.

4 Validation

We have created a prototype that embodies our approach. Our prototype uses Z3 [6] as the SMT solver, the Epsilon Object Language [25] to implement the exploration of execution paths, Xtext [26] to capture and validate system properties, and the Eclipse Modeling Framework (EMF) to handle execution paths. The prototype is a command line tool which provides simple functionalities to capture and verify system properties, and provides a counter example when the given invariant is violated. The development of our tool is still in progress. However, the core verification functionality for USMs is already available.

We consider the following two research questions, to discuss the applicability, performance, and scalability of our approach.

RQ1 (Applicability): Can our approach handle the verification of practical behavioral models of RTE systems?

RQ2 (Performance and Scalability): What is the performance and scalability of our approach?

4.1 Case Studies

To answer these questions, we used five case studies, including a *Car Door Central Lock system*, a *Parcel Router*, a *Rover*, a *FailOver System*, and a *Debuggable FailOver* system. As shown in Table 2, these models have different complexities that range from simple models with only 8 states to large models with up to 350 states.

The Car Door Central Lock system [27] is a control system for locking and unlocking car doors. The Parcel Router [28, 29] is an automatic system where tagged parcels are routed through successive chutes and switchers to a corresponding bin. The system is time-sensitive, and jams can appear due to variations in the time required by a parcel to transit through the different chutes. It checks for potential parcel jams, and prevents parcels from being transferred from one chute to another until the next chute is empty. The simplified version ignores jams.

The Rover system model [30] allows an autonomous robot to move in different directions. The rover is equipped with three wheels, driven by two engines. Additionally, it is equipped with several sensors, such as temperature and humidity sensors.

The FailOver system [31, 32] is an implementation of a computer server fail-over mechanism. It involves a set of servers processing client requests. To provide high availability, the system supports two replication modes, passive and active [33].

The Debuggable FailOver system is a debuggable version of the FailOver system, which is generated using MDebugger [27]. The complexity of this model is high, and allows us to check that the refinement and analysis time do not skyrocket when the model size grows exponentially.

4.2 Experiments

EXP1. Applicability. To answer RQ1, first we applied our prototype to the verification of the invariants of the running example of this paper (Section 2.1) with bounds 5 and 15. We then configured our prototype for the verification of "`invariant T true`" on the largest USM (i.e., the SM with the largest number of states and transitions) of the set of case studies listed in Table 2. Intuitively, invariant T always holds, which implies that our prototype must check all of the possible execution paths. This allows measurement of the worst-case verification time for a certain bound. We ran this verification with bounds 5 and 15, and recorded the number of explored execution paths and the time required for exploration, encoding, and checking the satisfiability of the paths, in each case.

To measure the effectiveness of the specified bounds, we measured the *coverage* of the explored paths as a percentage of the USM's states included at least once in an explored path. Coverage gives us a good estimate of whether the specified bound is enough to explore all of the possible behaviors of the USM.

Our current implementation does not yet support encoding of loop and conditional statements in actions automatically. None of the USMs in our case studies contains a loop. However, some of them contained conditional statements (*if-else*), which we transformed manually.

EXP2. Performance and Scalability. To answer RQ2, we have configured our prototype for the verification of "`invariant T true`" on our largest case study, the *Debuggable FailOver*. We ran the verification with a sequence of bounds from 5 to 24, until all of the

Table 2: Model complexity of case studies, exploration/encoding of execution paths time, and worst case verification time of an invariant

Model	USM Size		# of the paths		Cov. (%).		Exp. time (ms)		Enc. time (ms)		SAT. time (ms)		Over. (Sec)	
	S	T	B-5	B-15	B-5	B-15	B-5	B-15	B-5	B-15	B-5	B-15	B-5	B-15
Car Door Central Lock	8	10	7	114	55%	100%	206	441	704	1002	282	3037	2	5
Parcel Router	14	25	22	16844	29%	100%	357	7,648	854	1279	757	310,492	2	320
Rover	16	21	8	182	62%	100%	310	2326	801	1254	294	4502	2	9
FailOver	31	43	10	6126	38%	100%	276	4500	716	1685	326	163,285	2	170
Debuggable FailOver	350	620	7	539	9%	70%	803	5147	600	4001	267	14050	2	24

S: State, *T*: Transition, *B-5*: Bound 5, *B-15*: Bound 15, *Enc.*: Encoding, *Ver.*: Verification, *Exp.*: Exploration, *SAT.*: Checking Satisfiability, *Over.*: Overall time, *Cov.*: Coverage

states of the system were covered in the explored paths. Similarly to *EXP1*, we recorded the coverage, explored paths, and relevant computation times for each bound.

Execution Environment. We used a 2.7 GHz Intel Core *i*5 computer with 8*GB* of memory for all experiments, which is a typical development PC rather than particularly powerful hardware.

4.3 Results

4.3.1 RQ1: Applicability Verification of the Invariants of the Running Example. The prototype successfully verified all of the invariants of the running example with bounds 5 and 15. With bound 5, 7 execution paths (with length 5 or less) were explored, encoded, and verified in less than two seconds. The verification results showed that both invariants (*P1* and *P2*) hold. However, with verification bound 15, which took less than 7 seconds, 1,226 execution paths were explored, and the results of the verification showed that both invariants can be violated. Counter examples (shortest violating execution paths) were generated, as follows. (The configurations are shown based only on their execution states.)

P1 counter-example:

based on the configurations: $\langle in_1, C_1, in_2, s_1, s_2, ch_1, en_1, s_2, s_1 \rangle$

P2 counter-example:

based on the configurations: $\langle in_1, C_1, in_2, s_1, s_3, s_2 \rangle$

Updating the guard of the transition from state s_2 to state s_1 to '$\neg emergency$', and removing the transition between state s_3 and state s_2 fixed both problems.

Results of the Verification on Case Studies. We have successfully verified the USMs for all of the case studies. Table 2 summarizes the complexity of the case studies, and the results of the verifications. The size of the case studies ranges between 8 and 350 states. *Debuggable Failover* has 350 states. The *# of the paths* column presents the number of explored paths with bound 5 (paths with length 5 and smaller) and 15 (paths with length 15 and smaller) for each case study. The *Parcel Router* and *Car Door Central Lock* have the highest and lowest number of execution paths respectively. The column *Coverage* shows the percentage of the USMs' states that are included in at least one of the paths. The exploration with bound 15 provides full coverage of all models except *Debuggable Failover*. Columns *Exp. time* and *Enc. time* show the computing times for the exploration, and encoding and instrumentation of the execution paths respectively, and , and *SAT. time* the time for checking the satisfiability of the paths.

Finally, the last column *Overall Time* shows the overall verification times for each of the case studies with bounds 5 and 15, which range between 5 and 320 seconds for bound 15. These times seem acceptable since: (1) they account for the worst-case computation time, by verifying "`invariant T true`", forcing the satisfiability of all paths to be checked. (2) a large number of paths are explored with bound 15, and almost all of the USMs' states are covered.

Since the prototype can handle the verification of all of the case studies, including one of size comparable to industrial models in a reasonable time, we can safely conclude that the applicability of our approach is acceptable. Moreover, our implementation is still a prototype, and there remains the potential for improvement using parallel verification and exploration of paths, since the verification of each path is independent of the others.

4.3.2 RQ2: Scalability and Performance Figure 6 shows the coverage of explored execution paths (the left sub-figure), the number of explored paths (the middle sub-figure), and the computation time for exploration, encoding, and checking of the satisfiability (SAT) for the paths of *Debuggable Failover*. Full coverage of states is reached with bound 23, in which 54,516 paths of length 23 or less are explored in 220 seconds (about 6 minutes). The overall verification time with bound 23 is 1,483 seconds (about 25 minutes), 85% (i.e., 21 minutes) of which is related to checking the satisfiability of the paths (i.e., instrumented path formulas). Most of the rest of the time is consumed in exploration, and the computing time for encoding paths is negligible.

The right sub-figure of Fig. 6 shows the trend of computation times for exploration, encoding, and SAT time of the paths as the bound increases. As the bound increases, the number of explored paths increases (middle sub-figure), and as a result, the exploration and SAT time are also increased. The SAT time increases drastically at bound 21 (i.e., when about 1,000 paths are explored). This suggests that even verification using SMT-Solvers does not entirely mitigate scalability issues. However, it allows us to manage and apply it up to a reasonable bound (e.g., until full coverage is reached). Moreover, as mentioned, our implementation is still a prototype, and there remains the potential for improvement using parallel verification and exploration of paths. Exploration time also increases after bound 21, but is still manageable and relatively small compared to SAT time. Finally, the encoding time is constant and negligible. This is due to the fact that actions are only transformed to *SSA* form, and since the number of actions in a model is finite,

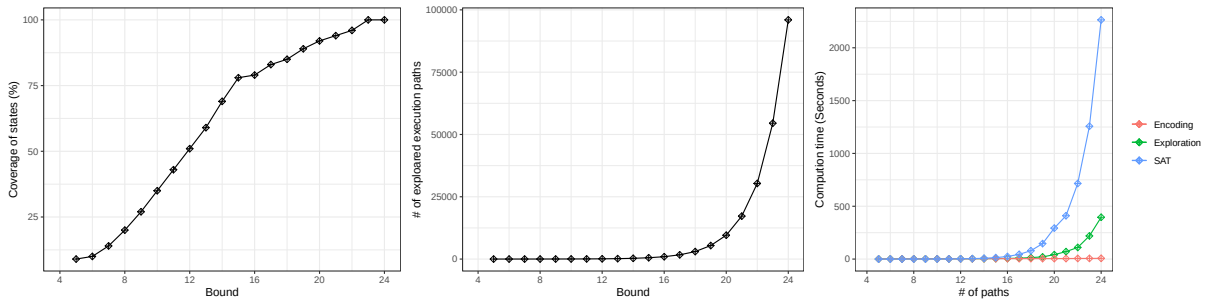

Figure 6: Coverage, number of explored paths, and verification time of *Debuggable Failover* with bounds *5..24*

the encoding time does not increase after actions are transformed the first time.

5 Related work

In this section, we focus on methods developed for verification of UML behavioral models (e.g., SMs and statechart diagrams). We review these methods based on the classification of Crane [34]: mathematical models, and translation approaches.

Mathematical Models. These approaches use verification techniques that rely on mathematical concepts and notations, such as Petri Nets [35, 36], Transition Systems [34], Abstract State Machines (ASM) [37, 38], and Rewriting Systems [39, 40].

Using a mathematical notation encourages precision and attention to detail, which makes it more likely to have complete and unambiguous resulting semantics [34]. However, most of the mathematically-based approaches do not provide a high level of abstraction that can be easily understood by users [34, 39].

Translation Approaches These approaches perform verification by translating a UML behavioral model into a formal language, such as a specification language, or the input language of a model checker (e.g., [41]).

Model Checking Languages. The aim is to reuse existing model checkers, and to translate a model to the input language of a model checker. However, the translation of a sufficiently large model is complex and difficult to test, and the analysis results are not directly traceable back to the original model. There are several works on UML behavioral model verification ([42, 43]) proposing translation to Promela. Latellat et al. [43, 44] proposed an approach that translates the models into Promela code, and then verifies them by the model checker SPIN. Beeck [45] extended Latellat's work by supporting more UML statechart features, such as the history mechanism as well as entry and exit actions. Lilius and Paltor [42] have proposed a tool, called vUML that uses the information contained in the class diagrams, statecharts and collaboration diagrams of a model to generate a Promela specification and then invokes the SPIN model checker. Eshuis and Wieringa [46] proposed a verification tool for UML activity diagrams, where an activity diagram is translated into an input format for verifying by the model checker SMV. Similarly, Knapp and Merz [47] exploit model checking to verify the UML state machines. Beato et al. [48] translated models specified using both UML state machines and activity diagrams to the input language of SMV.

Specification Languages. Meng et al. [49] proposed a formalization of the UML statechart diagrams based on the formal specification language RSL (RAISE Specification Language), which is a language for specifying and designing software systems. Kim et al. [50] have formalized UML using OBJECT-Z. They encapsulate the abstract syntax and the static and dynamic semantics for each individual model constructed as a single Object-Z class.

The translation of the behavioural models into a formal language is a popular approach. However, as discussed in Section 1, due to the semantics mismatch between SMs and model checking languages, advanced features of SMs (such as hierarchical structure) are not directly supported by model checkers. This complicates the translation [5] that often is achieved by simplification and complex transformations, verification of which is required to ensure that the original intent of the SMs is preserved.

Methods applicable directly to the UML-RT language are less common [12, 51–54]. Leue et al. [53] proposed a translation to the AsmL language used in SpecExlorer. Posse and Dingel [12] performed a translation to Kiltera, which is an extension of classical process algebras.

SMT solvers have been used in the verification and analysis of UML/OCL models. Soeken et al. [55] presented an approach to verify the dynamic view of a UML class diagram including operations with pre- and post-conditions. In another approach [56] an SMT-based approach was proposed for model finding to increase confidence in the correctness of a UML/OCL model. Clarisó et al. [57] proposed an approach to aid in the bounded verification of UML/OCL models. The approach operates by translating the UML/OCL model into a constraint satisfaction problem. Dania and Clavel [58] proposed a mapping from OCL to many-sorted first-order logic.

6 Conclusion

In this paper, we have proposed a novel technique for verifying UML behavioral models as hierarchical state machines using SMT solvers. We have described our approach in detail and analyzed its applicability, performance, and scalability in a number of different case studies. In contrast with existing work, our approach does not require translation to other formalisms, and is not dependent on program verification tools. Instead, it leverages bounded verification and takes as input system properties specified as invariants that are relatively easy to express. The results of the evaluation provide evidence that our approach performs very well when verifying models for finite-state systems.

References

[1] B. Selic, "Using UML for Modeling Complex Real-Time Systems," in *Languages, Compilers, and Tools for Embedded Systems*. Springer, 1998, pp. 250–260.

[2] B. Selic, "What will it take? a view on adoption of model-based methods in practice," *Software & Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[3] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[4] W. Visser, M. B. Dwyer, and M. Whalen, "The hidden models of model checking," *Software & Systems Modeling*, vol. 11, no. 4, pp. 541–555, 2012.

[5] K. Zurowska and J. Dingel, "Language-specific model checking of UML-RT models," *Software & Systems Modeling*, vol. 16, no. 2, pp. 393–415, 2017.

[6] "Z3," https://github.com/Z3Prover/z3, 2019, retrieved October 14, 2019.

[7] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *International Conference on Computer Aided Verification*. Springer, 2001, pp. 260–264.

[8] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[9] S. J. A. T. Gulwani, Sumit and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.

[10] B. Czipó, A. Hajdu, T. Tóth, and I. Majzik, "Exploiting hierarchy in the abstraction-based verification of statecharts using SMT solvers," *FESCA ETAPS*, pp. 31–45, 2017.

[11] L. Baresi, G. Blohm, D. S. Kolovos, N. Matragkas, A. Motta, R. F. Paige, A. Radjenovic, and M. Rossi, "Formal verification and validation of embedded systems: the UML-based MADES approach," *Software & Systems Modeling*, vol. 14, no. 1, pp. 343–363, 2015.

[12] E. Posse and J. Dingel, "An executable formal semantics for UML-RT," *Software & Systems Modeling*, vol. 15, no. 1, pp. 179–217, 2016.

[13] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*, 2011, pp. 1–30.

[14] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Trans. on*, vol. 100, no. 8, pp. 677–691, 1986.

[15] D. Kroening and M. Tautschnig, "CBMC-C bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[16] C. Damas, B. Lambeau, P. Dupont, and A. V. Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE Trans. on Software Engineering*, vol. 31, no. 12, pp. 1056–1073, 2005.

[17] L. Dohmen and L. J. Somers, "Experiences and lessons learned using UML-RT to develop embedded printer software," in *Product Focused Software Process Improvement*. Springer Berlin Heidelberg, 2002, pp. 475–484.

[18] IBM, "IBM RSARTE," https://www.ibm.com/developerworks/downloads-/r/architect/index.html, 2019, retrieved October 14, 2019.

[19] Eclipse, "Eclipse Papyrus for Real Time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, 2019, retrieved March 19, 2019.

[20] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176.

[21] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[22] M. von der Beeck, "A formal semantics of UML-RT," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 768–782.

[23] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1988, pp. 12–27.

[24] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI-C programs using SAT," *Formal Methods in System Design*, vol. 25, no. 2, pp. 105–127, Sep 2004.

[25] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

[26] Xtext, "Xtext," http://www.eclipse.org/Xtext, 2019, retrieved October 14, 2019.

[27] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 419–430.

[28] W. Swartout and R. Balzer, "On the inevitable intertwining of specification and implementation," *Communications of the ACM*, vol. 25, no. 7, pp. 438–440, 1982.

[29] J. Magee and J. Kramer, *State Models and Java Programs*. Wiley, 1999.

[30] M. Bagherzadeh, "Model-level debugging in the context of the model-driven development," Ph.D. dissertation, 2019.

[31] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive failover for real-time middleware with passive replication," in *15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, 2009, pp. 118–127.

[32] N. Kahani, N. Hili, J. R. Cordy, and J. Dingel, "Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems," in *Proceedings of the 9th International Workshop on Modelling in Software Engineering*. IEEE Press, 2017, pp. 12–18.

[33] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[34] M. L. Crane and J. Dingel, "On the semantics of UML state machines: Categorization and comparision," in *In Technical Report 2005-501, School of Computing, Queen's*. Citeseer, 2005.

[35] J. Lian, Z. Hu, and S. M. Shatz, "Simulation-based analysis of UML statechart diagrams: methods and case studies," *Software Quality Journal*, vol. 16, no. 1, pp. 45–78, 2008.

[36] Y. Zhao, Y. Fan, X. Bai, Y. Wang, H. Cai, and W. Ding, "Towards formal verification of UML diagrams based on graph transformation," in *IEEE International Conference on E-Commerce Technology for Dynamic E-Business*. IEEE, 2004, pp. 180–187.

[37] K. Compton, J. Huggins, and W. Shen, "A semantic model for the state machine in the unified modeling language," in *Proceeding of Dynamic Behavior in UML Models: Semantic Questions*, 2000, p. 25–31.

[38] Y. Jin, R. Esser, and J. W. Janneck, "Describing the syntax and semantics of UML statecharts in a heterogeneous modelling environment," in *International Conference on Theory and Application of Diagrams*. Springer, 2002, pp. 320–334.

[39] D. Varró, "A formal semantics of UML statecharts by model transition systems," in *International Conference on Graph Transformation*. Springer, 2002, pp. 378–392.

[40] J. Kong, K. Zhang, J. Dong, and D. Xu, "Specifying behavioral semantics of UML diagrams through graph transformations," *Journal of Systems and Software*, vol. 82, no. 2, pp. 292–306, 2009.

[41] Z. Pap, I. Majzik, A. Pataricza, and A. Szegi, "Completeness and consistency analysis of UML statechart specifications," in *Proceedings IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, 2001, pp. 83–90.

[42] J. Lilius and I. P. Paltor, "vUML: A tool for verifying UML models," in *14th IEEE International Conference on Automated Software Engineering*, 1999, pp. 255–258.

[43] D. Latella, I. Majzik, and M. Massink, "Towards a formal operational semantics of UML statechart diagrams," in *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 1999, pp. 331–347.

[44] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker," *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999.

[45] M. von der Beeck, "A structured operational semantics for UML-statecharts," *Software and Systems Modeling*, vol. 1, no. 2, pp. 130–141, 2002.

[46] R. Eshuis and R. Wieringa, "Tool support for verifying UML activity diagrams," *IEEE Trans. on Software Engineering*, vol. 30, no. 7, pp. 437–447, 2004.

[47] A. Knapp and S. Merz, "Model checking and code generation for UML state machines and collaborations," *Proceedings 5th Wsh. Tools for System Design and Verification*, pp. 59–64, 2002.

[48] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente, "UML automatic verification tool with formal methods," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 4, pp. 3–16, 2005.

[49] S. Meng, Z. Naixiao, and B. K. Aichernig, "The formal foundations in RSL for UML statechart diagrams," 2004.

[50] S.-K. Kim and D. Carrington, "A formal model of the UML metamodel: The UML state machine and its integrity constraints," in *International Conference of B and Z Users*, 2002, pp. 497–516.

[51] K. Zurowska and J. Dingel, "Model checking of UML-RT models using lazy composition," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2013, pp. 304–319.

[52] M. Saaltink and I. Meisels, "Using SPIN to analyse RoseRT models," Technical Report, ORA Canada, Tech. Rep., 1999.

[53] S. Leue, A. Ştefănescu, and W. Wei, "An AsmL semantics for dynamic structures and run time schedulability in UML-RT," in *International Conference on Objects, Components, Models and Patterns*. Springer, 2008, pp. 238–257.

[54] R. Ramos, A. Sampaio, and A. Mota, "A semantics for UML-RT active classes via mapping into circus," in *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 2005, pp. 99–114.

[55] M. Soeken, R. Wille, and R. Drechsler, "Verifying dynamic aspects of UML models," in *2011 Design, Automation & Test in Europe*, 2011, pp. 1–6.

[56] N. Przigoda, R. Wille, and R. Drechsler, "Ground setting properties for an efficient translation of OCL in SMT-based model finding," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pp. 261–271.

[57] R. Clarisó, C. A. González, and J. Cabot, "Smart bound selection for the verification of UML/OCL class diagrams," *IEEE Trans. on Software Engineering*, vol. 45, no. 4, pp. 412–426, 2017.

[58] C. Dania and M. Clavel, "OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pp. 65–75.