Synthesis of State Machine Models

Nafiseh Kahani
Queen's University, Canada
kahani@cs.queensu.ca

Mojtaba Bagherzadeh
University of Ottawa, Canada
m.bagherzadeh@uottawa.ca

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

Abstract

The automated synthesis of behavioural models in the form of state machines (SMs) from higher-level specifications has a high potential impact on the efficiency and accuracy of software development using models. In this paper, inspired by program synthesis techniques, we propose a model synthesis approach that takes as input a structural model of a system and its desired system properties, and automatically synthesizes executable SMs for its components. To this end, we first generate a synthesis formula for each component, consistent with the system properties, and then perform a State Space Exploration (SSE) of each component, based on its synthesis formula. The result of the SSE is saved in a Labeled Transition System (LTS), for which we then synthesize detailed actions for each of its transitions. Finally, we transform the LTSs into UML-RT (UML real-time profile) SMs, and integrate them with the original structural models. We assess the applicability, performance, and scalability of our approach using several different use cases extracted from the literature.

CCS Concepts

• **Software and its engineering → Model-driven software engineering**;

Keywords

MDD, MDE, State Machine, Model Synthesis

ACM Reference Format:
Nafiseh Kahani, Mojtaba Bagherzadeh, and James R. Cordy. 2020. Synthesis of State Machine Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410936

1 Introduction

Model-driven development (MDD) advocates the use of models during the entire software development process. By leveraging abstraction and automation, MDD techniques can simplify communication and design activities, increase productivity and compatibility between systems, and boost development efficiency [1–4]. Despite the promised benefits, application of MDD in practice has not yet reached its full potential [5]. One of the important reasons is related to the complex and time-consuming process of designing and creating the models, which often requires special expertise and experience [6]. To deal with this problem, several approaches have been proposed to automate various parts of the modelling process, including the synthesis of behavioural models in the form of state machines (SMs) from higher-level specifications [7–10].

The main techniques for automatically synthesizing SMs focus on: (1) deriving behavioural models from system-level scenarios (scenario-based synthesis), and (2) identifying models from high-level properties, such as Linear Temporal Logic (LTL) specifications (correct-by-construction synthesis). The first of these, scenario-based synthesis (e.g., [11, 12]), leverages positive/negative scenarios (specified in a variety of ways, such as UML sequence diagrams) to derive behaviour models. Inductive learning techniques (e.g., grammar induction) are used to generate SMs compatible with the scenarios. Scenarios are focused and straightforward; however, they are just partial examples of the desired or undesired behaviour. A complete set of scenarios needs to be truly representative of the desired system. Such scenarios are often unavailable, difficult to prepare, or difficult to check for correctness and completeness.

The second technique, correct-by-construction synthesis (e.g., [13–15]), synthesizes a correct-by-construction design for a reactive system (if it exists), based on a temporal logic specification of the system created by users. The synthesized system is guaranteed to meet the system specification. However, the high computational complexity of the search algorithm can make it impractical for large systems. To deal with the scaling issue, some work limits the temporal specification to fragments of LTL, for example, General Reactivity of Rank 1 (GR(1)) [16], in order to improve efficiency.

Satisfiability-Modulo-Theory (SMT) solvers have been shown to have some of the key functionality necessary for program synthesis, without the need for the design tool developer to implement a solver or a custom design-space search algorithm to find a solution [17]. SMT solvers have been successfully applied in several practical applications [18], such as generating optimal code sequences [19], general-purpose peephole optimizers [20], automating repetitive programming tasks, and completion of a program sketch [18].

Inspired by the SMT solver-based program synthesis techniques, in this work we transform system properties and a structural model of a Real-time Embedded (RTE) system into a set of quantifier-free first-order logical formulas, thus reducing the synthesis of state models to the solution of the formulas. More specifically, our approach accepts a structural model of an RTE system, and a set of system properties defining system invariants and pre/post conditions using OCL-like expressions. It then synthesizes an executable SM for each component of the system, in four phases: (1) It generates a synthesis formula for each component, which is consistent with system properties and structural models, and respects the execution semantics of the component. (2) It performs a State Space Exploration (SSE) by enumeration of all possible input messages of the component. Each step of the SSE solves the synthesis formula

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS '20, October 18–23, 2020, Virtual Event, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410936

in a different context, based on input messages and the current execution state. The result of the *SSE* is saved as a Labeled Transition System (LTS). (3) The *LTS* produced in phase 2 does not include actions (i.e., action code). A backtracking algorithm is used to search over the set of possible actions to find a sequence of actions for each transition. During the search, each action is executed symbolically and checked based on the synthesis formula, to ensure its consistency with the system properties. (4) Finally, we apply state minimization techniques inspired by existing work to minimize the number of states in the LTSs, transform them into UML-RT SMs [21], and integrate them with the structural models.

This work complements the state of the art in the area of model synthesis, by providing an end-to-end solution using SMT-solvers that is directly integrated with a production MDD tool (Papyrus-RT [22, 23]). The most important contributions of the work are: (1) A systematic approach and relevant formalization to leverage SMT solvers for the synthesis of SM models, and a publicly-available implementation [24] for follow-up research. (2) An automated method for the synthesis of well-formed UML-RT SM models with detailed specifications, including actions on transitions, which yields synthesized SMs that are ready to be executed. Existing design-by-construction techniques often synthesize Büchi-automata or LTS without actions, which can not be directly executed or used by MDD tools. At best, existing scenario-based work preserves actions that are already explicitly specified in input scenarios.

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 model synthesis approach. Section 4 evaluates our approach by analyzing its applicability and performance. We overview related work in Section 5, and conclude in Section 6.

2 Background

In this section, we describe the terms and notations we use to specify the structural model of a system, the execution semantics of the system, and system properties. We also introduce the example system we use as a running example in this paper.

Definition 1. Read function (Projection). Let *tp* be a tuple of attributes $\langle a_1, \ldots, a_n \rangle$, where $a_1 \ldots a_n$ refer to attributes' names. We use $tp.a_i$ to denote reading the value of attribute a_i. For example, we use *person.name* to read the value of attribute *name* of tuple *person* = ⟨*name, family*⟩.

Definition 2. Structural Model of a Real-time Embedded (RTE) System. We define a *protocol/interface* as a set of pairs (m, d), where $m \in \mathcal{M}$ (i.e., a universal set of messages) is a message, and $d \in \{input, output\}$ specifies whether a message is consumed (*input* message) or produced (*output* message). A message can have a *payload*, which is the set of values conveyed by the message. We define a *component* as a tuple $\langle \mathcal{P}, \mathcal{V}, \beta \rangle$, where $\mathcal{P} \subseteq \boldsymbol{\mathcal{P}}$ (i.e., the universal set of ports) is a set of ports, $\mathcal{V}$ is a set of variables, and β refers to the specification of the component's *behaviour*, which is defined using a SM. A *port* is defined as a pair $(t, conjugated)$, where $t \in \mathcal{I}$ refers to the type of the port (a protocol), and $conjugated \in$ *{true, false}* specifies whether or not the port is conjugated. The direction of messages of conjugated ports is reversed. Finally, the *structural model* of a system is defined as a tuple $\langle C, \mathcal{I}, con, in \rangle$, where C is a set of components, $\mathcal{I}$ is a set of protocols, *con* is a connectivity relationship $\subseteq \boldsymbol{\mathcal{P}} \times \boldsymbol{\mathcal{P}}$, and *in* is an acyclic containment relationship $\subseteq C \times C$. Whenever two ports $p1, p2$ are connected by *con* (i.e., $(p1, p2) \in con$) then both have the same type (i.e., $p1.t = p2.t$) and exactly one of them must be conjugated. This condition ensures that connected ports are 'compatible'.

Definition 3. Timed Behaviours. RTE systems often have timed behaviour that needs to be specified. To support time in this work, we assume that an RTE system contains a timing interface *(startTimer, input), (timeout, output)*, and a component RTS (run-time service) with a port of type *timing*. Components requiring timing services then use a connection with the RTS component to treat timing events as messages.

Definition 4. Action Language. We assume the existence of an *action language* that supports primitive operations such as accessing/updating variables, arithmetic/boolean expressions, and sending messages, but do not define a particular syntax for it.

Definition 5. State Machines (SMs). A SM is defined as a tuple $\langle \mathcal{S}, \mathcal{T}, in \rangle$. Where $\mathcal{S} = \mathcal{S}_b \cup \mathcal{S}_c \cup \mathcal{S}_p$ is a set of states, $\mathcal{T}$ is a set of transitions, and $in \subseteq \mathcal{S}_c \times (\mathcal{S} \cup \mathcal{T})$ denotes an acyclic containment relationship. States can be *basic* ($\mathcal{S}_b$), *composite* ($\mathcal{S}_c$), or *pseudo-states* ($\mathcal{S}_p$). Basic states are primitive states whose execution remains 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*. Composite and basic states can have entry and exit actions that are coded using an action language. Note that, with the exception of the initial state, our approach synthesizes state machines without composite or pseudo-states.

Definition 6. Transition. Let $inp(c)$ refer to the messages that can be received by a component c owning a SM. A *transition t* is a 5-tuple ⟨*src, guard, trig, act, des*⟩, where $src, des \in S$ refer to non-empty source and destination of the transition respectively, *guard* is a logical expression coded using the action language, $trig \subseteq inp(c)$ is a set of messages that trigger the transition, and *act* is the transition's action coded using the action language.

Definition 7. Execution Semantics of a SM. We use a labeled transition system (LTS), which is a tuple $\langle \Sigma, \mathcal{A}, \sigma_0, \longrightarrow \rangle$ to capture the execution of the SM of component c of system s. Σ is a set of execution states, $\mathcal{A}$ is a set of actions, $\sigma_0 \in \Sigma$ is the initial execution state, and $\longrightarrow$ is a transition relation (execution step). An *execution state* has a snapshot (e) of a map, $\mathcal{E}$, from component variables to values. Two execution states σ, σ' are considered equal when their snapshots are the same ($\sigma.e = \sigma'.e$). The initial state $\sigma_0.e$ is set to default (initial) values of the variables, which are 0 for *Integer* and *false* for *Boolean* variables.

An *execution step* is captured as a tuple $\langle \sigma, act, m, g, \sigma' \rangle$ (i.e., $\sigma \xrightarrow[act]{m[g]} \sigma'$) that, upon receiving message m (triggering message), evaluates the logical expression g (guard). If the guard holds, the execution moves from execution state σ (current execution state) to execution state σ' (next execution state) while executing action $act \in \mathcal{A}$ that may update variables ($\mathcal{E}$) and produce outputs. Note

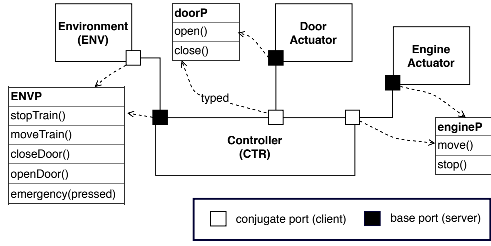

Figure 1: A Simplified Train Controller System

that often only actions and relevant execution states are captured by an *LTS*, but we also capture the relevant triggering message and the guard. This extension helps us to capture all relevant information of the execution that is necessary for the generation of SMs.

The execution of a system can be defined as a collection of its components' executions, which interact with each other by passing messages. We assume that system execution is managed by a run-time service (*RTS*), which 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 starts (run-to-completion semantics). It also sends the message *initialize* to all components to begin their execution, which results in an execution step from σ_0 to σ'. The message *initialize* is the first message processed by all components.

Definition 8. System Properties. System properties (S_{prop}) capture the system's (non-)functional properties in the form of OCL-like constraints. S_{prop} is defined as a tuple $\langle S_{inv}, M_{cond}\rangle$, where S_{inv} denotes a set of invariants defined as a quantifier-free first-order logic formula, which must hold during the entire execution of the system. M_{cond} denotes a non-empty set of message conditions. A message condition, *mc*, of message *m* is defined as a set of pairs $\langle m_{pre}, m_{post}\rangle$, where m_{pre}, m_{post} are quantifier-free first-order logic formulas specified based on two consecutive execution states σ and σ', defining how receiving message *m* changes σ into σ'. If the m_{pre} (pre-condition) holds based on a snapshot of σ, then the reception of *m* will cause an execution step from σ to σ', in which the m_{post} (post-condition) must then hold. The m_{pre} pre-condition can only be defined based on the current values of the variables (in the snapshot of σ) and the payload of *m*, whereas the m_{post} post-condition can be defined based on the payload of message *m*, and the current and next values of the variables (in the snapshots of σ and σ' respectively). M_{cond} must always contain an entry for the *initialize* message that has no pre-condition.

2.1 A Running Example

We use the simplified train software system [8] shown in Fig. 1 to illustrate important concepts throughout the paper. The system is composed of four components: a train *Controller (CTR)*, an *Environment (ENV)*, a *Door*, and an *Engine*. We assume that users' inputs including pressing emergency key are handled by *ENV*. The *CTR* component receives input from the *ENV* components, and controls actuators for the *Engine* and *Door*. As shown in Fig. 1, there are three interfaces, *EnginP*, *DoorP*, and *ENVP* to manage communication between the components. One message of *ENVP* has payload (parameters): *pressed* is a Boolean value that indicates whether the passenger has pressed the emergency key. In addition, *CTR* has a Boolean variable *emergency* that encodes the train's emergency

```
1  Constraints TrainExample {
2    Invariant R2 {moving  ⟹  (closed ∧ ¬emergency)}
3    Invariant R3 {emergency  ⟹  (¬closed ∧ ¬moving)}
4    Message initialize() // MC_1
5       Post: ¬closed ∧ ¬moving ∧ ¬emergency
6    Message open() // MC_2
7       Post: ¬closed
8    Message close()  // MC_3
9       Post: closed
10   Message move()  // MC_4
11      Post: moving
12   Message stop() // MC_5
13      Post: ¬moving
14   Message emergency(pressed) // MC_6 R1
15      Pre: pressed
16      Post: emergency ∧ ¬moving ∧ ¬closed
17   Message emergency(pressed)  // MC_7 R1
18      Pre: ¬pressed
19      Post: ¬emergency
20   Message openDoor() // MC_8
21      Post: ¬closed
22   Message closeDoor()  // MC_9
23      Post: closed
24   Message moveTrain() // MC_10
25      Post: moving
26   Message stopTrain()  // MC_11
27      Post: ¬moving
28   }
```

Listing 1: System Constraints of the Running Example (R1-R5 are defined in Sec. 2.1)

status, *Engine* has a Boolean variable *moving* indicating whether the train is presently in motion, and *Door* has a Boolean variable *closed* that indicates whether the door is closed.

The train must fulfill the following requirements. R1: When the emergency key is pressed (i.e., an emergency message is received with payload *pressed=true*), the train must stop, and its door should be opened immediately. R2: The train can only move when the door is closed and the emergency key is not pressed. R3: When the train is in emergency status, the train must not move, and its doors must be kept open. R4: User commands that are sent to *CTR* by *ENV* (e.g., messages *openDoor* and *moveTrain*) must be handled appropriately.

Listing 1 shows a script of the system properties in our notation for the running example. `Invariant R2` and `R3` address the *R2* and *R3* requirements respectively. The remainder of the script specifies the conditions for the input messages of the components.

The scope of components' variables is global, and they can be used throughout the script. This allows the user to declare system properties based on overall system execution rather than individual components. Thus the user does not need to be concerned with the required interactions for fulfilling them. By default, references to a variable in pre-conditions and post-conditions refer to its current and next values respectively. To refer to the current value of a variable *var* in post-conditions, the notation *cur_var* is used. The payload of a message is indicated in parentheses following the message name (e.g., *emergency(pressed)*, lines 14-16).

3 Approach

Overview. Figure 2 presents an overview of our approach to automatically generating a SM for each component that satisfies all system properties and contains all the necessary details for execution (i.e., states, transitions, and actions). We assume as input a structural model of the system in the form of UML-RT structure

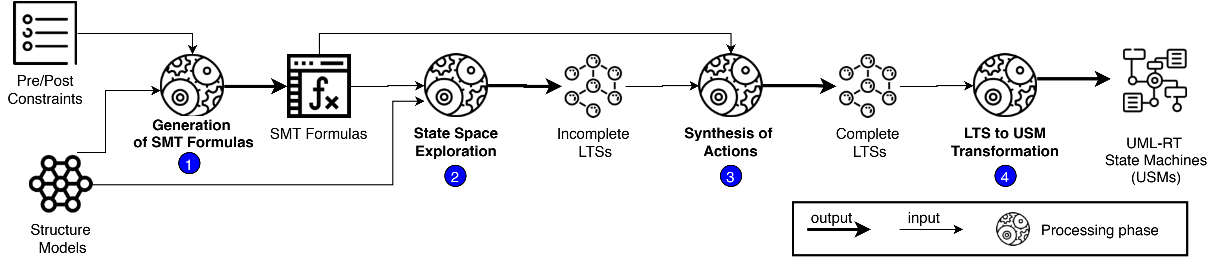

Figure 2: Overview of our approach

Table 1: Helper functions

Function	Description
$cur_V(v)$	returns a new variable of the same type as v prefixed with $cur_$
$nex_V(v)$	returns a new variable of the same type as v prefixed with $next_$
$inp(c)$	returns input messages of component c
$inp_V(m)$	accepts a message m and returns a *Boolean* variable inp_m
$ID(mc)$	creates unique Boolean variables MC_i where ($i \in \mathbb{N}$) for message condition mc
$ref(f)$	returns a set of variables updated in f formula
$cur_F(f)$	returns a new formula in which the system variables used by formula f are substituted by their corresponding ones from the current variables ($\in V_{cur}$)
$cond(m)$	returns message conditions of m message
$nex_F(f)$	returns a new formula in which the system variables used by formula f are substituted by their corresponding ones from the next variables ($\in V_{next}$)
$unref(f)$	returns system variables not updated by the formula f

models, and the system properties expressed in the OCL-like notation introduced in Def. 8. Table 1 lists helper functions that are used in the remainder of this section. Our approach consists of four phases, as follows.

Phase 1. Generation of Synthesis Formulas. Our approach reduces the execution of a component to solving an SMT formula (i.e., a first-order logic formula without quantifiers), referred to as a *synthesis formula*, which allows symbolic execution of the component. The generated synthesis formulas are consistent with the system properties, structural model, and the execution semantics of the component discussed in Def. 7.

Phase 2. State Space Exploration. This phase takes the synthesis formulas and the structure model of the system as inputs, performs a SSE for each component, and produces an output *LTS*. The execution steps and states in the exploration are restricted to those that satisfy the synthesis formula. The SSE traverses the reachable execution states based on the possible inputs of the components, and each step of the exploration is reduced to solving the synthesis formula. We assume that the synthesized models are executed by a run-time system that supports run-to-completion, in order to guarantee that the execution steps cannot be interrupted. We also assume no shared variables between components, and that components only communicate with each other through message passing. Both of these assumptions are reasonable and supported by MDD tools such as IBM RSA-RTE [25].

Phase 3. Synthesis of Actions. The *LTSs* generated in Phase 2 encode the execution states and the execution steps between them; however, the actions of the execution steps are as yet *unknown* (Def. 7). In this phase, for each incomplete LTS, we perform a backtracking search over the possible actions for each execution step, and synthesize a sequence of actions for the step. The actions are synthesized based on symbolic execution. Therefore, the synthesized actions do not account for the run-time exception. For example, the sending of a message to a component may fail during the run-time.

Phase 4. Transformation of *LTSs* to *State Machines*. Finally, taking advantage of existing state minimization techniques [8, 26], we minimize the number of states in the *LTSs*, transform them to executable *SMs* using model transformation, and integrate them with the structural model. The result of the synthesis is a UML-RT model compatible with existing MDD tools such as Papyrus-RT.

3.1 Phase 1: Generation of Synthesis Formulas

In the first phase, we generate a synthesis formula for each component, consistent with the system properties and execution semantics of the component. In the following, first we describe the variables on which synthesis formulas are defined, and then explain how a synthesis formula is generated for each component.

3.1.1 Variables of a Synthesis Formula We assume that V^s_{sys} is a set that contains all the variables of system s. The set of variables of the synthesis formula for component c is then defined as:

$V^c_{all} \leftarrow V^s_{cur} \cup V^s_{next} \cup V^c_{msg} \cup V^c_{pay}$

where

$V^s_{cur} \leftarrow \{\forall v \in V^s_{sys} : cur_V(v)\}$ and

$V^s_{next} \leftarrow \{\forall v \in V^s_{sys} : nex_V(v)\}$

include variables to capture snapshots of σ and σ' respectively,

$V^c_{msg} \leftarrow \{\forall m \in inp(c) : inp_v(m)\}$

consists of *Boolean* variables to capture the reception of the input messages that drive execution of components. V^c_{pay} denotes the set of all components' input messages' payloads. For example, V^{CTR}_{all}, the variables of the synthesis formula for the component CTR in the context of the running example include:

$V^{CTR}_{cur} = \{cur_moving, cur_emergency, cur_closed\}$,
$V^{CTR}_{pay} = \{pressed\}$,
$V^{CTR}_{msg} = \{in_moveTrain, \ldots, in_emergency\}$,
$V^{CTR}_{next} = \{next_moving, next_emergency, next_closed\}$

3.1.2 Synthesis Formula For component c in system s with properties S_{prop}, we generate a synthesis formula F^c_{synt}, defined as the conjunction of four formulas:

$$F^c_{synt} \leftarrow F^s_{inv} \wedge F^c_{msg} \wedge F^c_{sem} \wedge F^c_{con}$$

Where F^s_{inv} (the *invariant formula*) is a conjunction of formulas corresponding to the system invariants ($S_{prop}.S_{inv}$). For each invariant, two formulas are generated, to capture the invariant in

both σ and σ'. The invariant formula is same for all components, since they are defined at the system level. They assure that the invariant holds at any given execution state.

$$F^s_{inv} \leftarrow \bigwedge_{i\in S_{prob}.S_{inv}} cur_F(i) \wedge nex_F(i)$$

For example, the invariant formula `Invariant R2` (Listing 1) of the train system (F^{train}_{inv}), looks like this:

$(cur_moving \implies (cur_closed \wedge \neg cur_emergency))\wedge$
$(next_moving \implies (next_closed \wedge \neg next_emergency))$

F^c_{msg} (the *messages formula*) is generated based on the message conditions of the component's input. It captures the defined conditions, and allows simulation of the reception of messages, and application of the related conditions. The message formula of component c of system s is defined as follows.

$$F^c_{msg} \leftarrow \bigwedge_{m\in inp(c)} \underbrace{\left(\bigwedge_{f\in cond(m)} \left(\overbrace{inp_V(m)}^{F^2_{msg}} \wedge \overbrace{cur_F(f.m_{pre})}^{F^3_{msg}} \implies \underbrace{nex_F(f.m_{post})}_{F^4_{msg}} \wedge \underbrace{\left(\bigwedge_{v\in unref(f.m_{post})} cur_V(v) = nex_V(v)\right)}_{F^5_{msg}} \right)\right)}_{F^1_{msg}}$$

For every condition of a message, we define an implication (F^1_{msg}) consisting of five parts. Parts F^2_{msg} and F^3_{msg} capture the reception of the message and its pre-condition respectively. The conjunction of F^2_{msg} and F^3_{msg} assures that the post-condition only applies when the relevant message is received and its pre-condition holds in σ.

F^4_{msg} corresponds to the post-condition of the message condition itself, and F^5_{msg} assures that only the variables updated in the post-condition are affected by the execution, and non-updated variables remain intact, checked by the conjunction of equalities for all unused variables in the σ and σ' states.

For example, the message condition *MC_7* (lines 17-19 of Listing 1) is translated like this, where each part of the formula is annotated with the relevant label:

$(F^2_{msg} : in_emergency) \wedge (F^3_{msg} : \neg pressed) \implies$
$\wedge\, (F^4_{msg} : \neg next_emergency)\wedge(F^5_{msg} : cur_closed = next_closed$
$\wedge\, cur_moving = next_moving)$

F^c_{sem} (the *semantic formula*) is the conjunction of four formulas dealing with the execution semantics of the component. More specifically, it assures that the SSE for the synthesis formula of a component: (1) is deterministic (i.e., only one or no pre-condition holds for the reception of a message in any state), (2) respects the run-to-completion semantics, and (3) does not change the current state (variables) if none of the message formulas applies. The semantic formula for component c of system s is defined as follows.

$$F^c_{sem} \leftarrow \left(\left(\overbrace{\bigoplus_{m\in inp(c)} \bigoplus_{f\in cond(m)} inp_V(m) \wedge cur_F(f.m_{pre})}^{F^1_{sem}} \right) \underbrace{\wedge \bigoplus_{m\in inp(c)} inp_V(m)}_{F^2_{sem}} \right) \vee \underbrace{\left(\neg F^1_{sem} \implies \bigwedge_{v\in V^s_{sys}} cur_V(v) = nex_V(v) \right)}_{F^3_{sem}}$$

Algorithm 1: State Space Exploration of a Component

```
1  Input; a system (s), a component(c), its variables (V_all^c),
       its synthesis formula (F_synt^c)
2  Output; An LTS (lts)
3  Create initial state σ0 based on the default values of
       variables and add it to lts
4  Let candidates be a FIFO list and push lts.σ0 into it
5  Let visited be an empty set
6  Let unsafe be an execution state
7  inp ← {initialize} # assures initialize is processed first
8  while{candidates is not empty}
9    pop candidates into σ
10   add σ to visited
11   for {msg in inp}
12     # enum returns a dummy value for msg without payload
13     for(e in enum(msg.payload))
14       Set the context (F_con^c) based on σ, msg, and e
15       if (∃model ⊨ F_synt^c)
16         Create σ' based on V_next^s from model
17         if (σ ≠ σ') # no change in variables
18             add σ --msg{e}[MCpre] / unknown--> σ' to lts
19             if  (σ' ∉ visited)
20                 push σ' onto candidates
21       else
22         add σ --msg{e}[MCpre] / invariantViolated-->unsafe to lts
23   inp ← inp(c)\{initialize}
```

F^2_{sem} assures that only one message can be received and processed at any execution state (run-to-completion) by defining exclusive disjunction ($\oplus$) of the message input variables. Similarly, F^1_{sem} assures that only one message condition can be applied to the reception of a message (deterministic execution). F^3_{sem} assures that if none of the pre-conditions of the received message hold, no variable in the system is changed.

Finally, F^c_{con} (the *context formula*) is a dynamic part of the synthesis formula that allows appending the execution context to the synthesis formula, by encoding the known values of variables and using an SMT solver to find the unknown values of variables. In this work, we use the context formula to specify the variables' values of σ (V_{cur}) and incoming messages (V_{msg}) as the known parts. We then use the SMT solver to see if there is a possible execution step simply by checking the satisfiability of the synthesis formula (by solving for the unknown part). If an execution step exists, we use the satisfying model (i.e., an assignment to all variables in V_{all} that makes the synthesis formula true) for V_{next} (i.e., σ'). This method is the core of our SSE, discussed later.

As an example, the following shows a context formula used to find the next execution state when an emergency message is received (*emergency=true*) with a payload (*pressed=true*). The current execution state is such that the train is not moving, the emergency key is not pressed, and the door is closed.

$known = in_emergency \wedge pressed \wedge\neg cur_moving \wedge$
$\neg cur_emergency \wedge cur_closed$
$unknown = next_moving{=}?,\ next_emergency{=}?,\ next_closed{=}?$

3.2 Phase 2: State Space Exploration

Algorithm 1 presents our breadth-first SSE method, which accepts a system s, a component c, the relevant variables (V^c_{all}), and synthesis formula of c (F^c_{synt}), and returns an *LTS* for the component. The algorithm first initializes all variables to their default values and creates the initial state of the LTS, defines a first-in

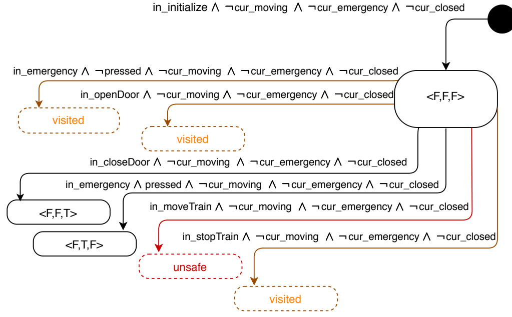

Figure 3: The first two iterations of the exploration of CTR. Edges are labeled with the related context formulas. States are labelled with tuples representing the variables <moving, emergency, closed>

first-out (FIFO) list (*candidates*) of candidate next states for the exploration, adds the initial state into *candidates*, and defines a set *visited* to keep track of explored states to ensure that each state is explored only once. For each candidate state (σ), the main loop of the exploration then injects all possible input messages of c by effective enumeration of possible values of their payload (Sec. 3.2.1). It then sets the context formula (Sec. 3.1.2) to reflect the message, its payload and the state being explored (σ). Assume that MC_{pre} refers to the pre-condition of the applied message condition, the algorithm calls the SMT solver for each context, and continues to one of the following cases:

(1) If F^c_{synt} *is satisfied and* $\sigma \neq \sigma'$, a new execution step of the exploration (corresponding to a positive scenario), whose action is *unknown* (i.e., its action needs to be synthesized based on the known parts of the step) is added to lts, then σ' is added to the candidate state if it has not been explored before (i.e., $\sigma' \notin visited$). The possible execution steps from σ' will be explored in a future iteration.

(2) If F^c_{synt} *is not satisfied* then at least one of the invariants has been violated. A new execution step is added, whose action is set to $invariantViolated$, which shows that processing the input message will violate the invariant. Normally an output LTS encodes only acceptable execution states. In this case, we also explicitly encode impossible states as unsafe. These negative execution scenarios will be essential to accurate synthesis of the final SM.

3.2.1 Efficient Enumeration of the Message Payload When the injected message has a payload with type *Integer*, the SSE should include all possible execution steps that can be caused by different values of the message payload. One method would be to simply enumerate all possible values of the payload. However, this can increase SSE time significantly. Instead, we use an execution path analysis technique to enumerate the minimum possible set of values of the payload necessary to cover all possible next execution steps.

For each payload p: (1) First, we create a dependency tree with the payload p as its root node. (2) For each dependent message condition (mc) of the payload (i.e., each mc whose pre/post conditions use the payload), we add a child to the root labeled with mc_{post} whose connecting edge is labeled with mc_{pre}. (3) For each leaf node n of the tree, we detect the dependent message conditions based on the variable(s) used in the label of n, and add them as children of n

Algorithm 2: Synthesis of Unknown Actions of an LTS

```
1  Input A system s, its variables (V^s_all) , A component(c),
       its lts)
2  Output A Completed LTS (lts)
3  Let model be a map that keeps the satisfying assignment (
       model) of a formula
4  for (stp in lts.→ such that step.act is unknown )
5    Let V_assign be the possible assignments of local
         variables of c
6    Let C_msg be message conditions of output messages of c
7    Let visited be an empty set
8    Let acts be a LIFO list
9    if{syntAct(stp.σ, stp.σ′, V_assign, C_msg, visited, acts)}
10     stp.act ← acts
11   else stp.act ← unsolvable
12
13 Function syntAct(State σ,σ′; Set V_assign, C_msg, visited; LIFO
        acts)
14   if (σ = σ′)
15     return true      #solution is complete
16   else
17     for (ac in (V_assign ∪ C_msg) \ visited)
18       if (ac in V_assign ∧ assign(ac,σ,F^c_synt,model)) ∨ (ac in
              C_msg∧ send(ac,σ,σ′,model))
19         read V^s_next from model saved by assign or send
               functions into σ1
20         Let Δ be the set of variables whose values are
               different in σ and σ′
21         Let Δ′ be the set of variables whose values are
               different in σ1 and σ′
22         if(Δ ⊆ Δ′) continue
23         push ac onto acts and add ac to visited
24         if(syntAct(σ1,σ′ V_assign, C_msg, visited, acts))
25           return true
26         else  pop ac from acts #backtrack
27     return false
```

labeled as in step 1. (4) We repeat step 3 until the leaf nodes no longer have any dependent message conditions. During payload enumeration, we create a path formula for each node in the tree which encodes the path from the root to the node. The goal is to find a value for the payload that makes the path formula satisfiable based on the current variable values V^s_{cur}. For efficiency, we begin the enumeration from the leaf nodes. The rationale is that if a value of the payload can satisfy the leaf path formula, there is no need to enumerate other nodes on the path.

Example. Figure 3 shows two iterations of the SSE for the *CTR* component of our train example.

3.3 Phase 3: Synthesis of Actions

This phase synthesizes the *unknown* actions of the generated LTSs from the previous phase. The action synthesis for an execution step focuses on finding a sequence (loop and branch free) of assignment and message sending actions to the relevant components that can change the execution state from σ to σ' without violating any system properties. To synthesize the actions of a step stp of component c, first the variables are divided into two groups, based on whether they belong to the component c (local variables of c) or not (external variables). Intuitively, component c can directly change (using assignment actions) its own local variables. However, to change external variables, it must send messages to the relevant other components.

Algorithm 2 shows our method for synthesizing *unknown* actions of an *LTS*. For each step with an *unknown* action, it first calculates all possible actions based on local and external variables, i.e., possible assignments of local variables based on the post-conditions

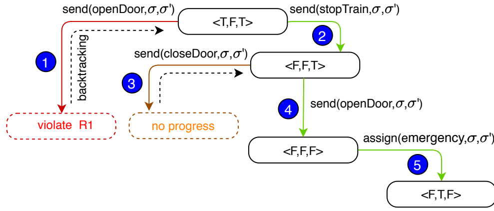

Figure 4: An example of the backtracking search to find a sequence of actions of an execution step between $\sigma = \langle T, F, T\rangle$ and $\sigma' = \langle F, T, F\rangle$ of *CTR*. States are labelled with tuples representing the variables <moving, emergency, closed>

of messages (V_{assign}) and output messages (C_{msg}). It then initializes the variables *visited* and *acts* and calls function *syntAct*.

syntAct explores all possible sequences of actions using a depth-first full backtracking search. It begins a search path by selection and application of one of the possible actions. If result of the action is satisfiable and changes at least one variable (i.e., makes progress towards reaching the target state), then it adds the action to *actions* and *visited* and continues the path by recursively calling *syntAct*. Otherwise, it backtracks one step (i.e., pops the last action) and tries another action. The search terminates when a solution is found, or when there are no more paths to explore.

The application of an action is achieved by calling the functions *assign* and *send*. The function *assign* whose code is omitted due to space limitation, takes an assignment expression, execution states σ, and a synthesis formula. It creates a context by setting V_{cur} based on σ and V_{next} based on σ and the assignment action. It then checks the satisfiability of synthesis formula based on the context. If the formula is satisfiable, it returns *true* and saves the satisfying models, which can be used by *syntAct*.

The function *send* whose code is omitted due to space limitation, accepts a synthesis formula belonging to the component that receives the message, and two execution states σ and σ' as input. If the message of *mc* has a payload, it creates a context formula (F_{con}) by setting the message variable of m (m_{in}), V_{cur} from σ, variables of V_{next} that can be affected by message condition *mc* from σ', and payload of the message. If the result is satisfiable, it returns *true* and saves the satisfying model. In addition to the value of the next variables, the model contains the value of the payload that the message should convey. Otherwise, if the message of *mc* has no payload, it resets F_{con} by setting the message variable of m (m_{in}), V_{cur} from σ, and variables of V_{next} that can be affected by message condition *mc* from σ'. It checks for the satisfiability again, saves the resulting model, and returns the results, if they are satisfiable. Otherwise, it returns false.

Figure 4 shows an example backtracking search in the context of component *CTR*. Variable *emergency* is a local variable of component *CTR* itself; however, *CTR* must send messages to components *Engine* and *Door* to close/open doors and stop/move the train.

3.4 Phase 4: Transformation of LTSs to SMs

An *LTS* captures the execution of a SM, possibly with a large number of states, some of which can be merged with each other. There is a strong body of work that addresses the problem of finding and merging the states of an *LTS* (state minimization), to create an *LTS* with a smaller number of states whose semantics are equivalent to the original. In this phase, inspired by these existing algorithms [8], we propose a state minimization algorithm for the extended *LTS*.

Algorithm 3: Minimization of an LTS

```
1  Let  allPairs be a list of all possible pairs (σ1,σ2) of
        execution states of lts
2  while p in allPairs
3      mergeStates(p.σ1, p.σ2)
4      remove dangling pairs because of the merge
5  remove the unsafe execution state and  unsafe steps
6
7  Function mergeStates(Execution State σ1 , σ2)
8    if (σ1 = σ2)
9      return true
10   for e1 in {e : e.σ = σ1  ∧ ¬ safe(e)}
11     if (∄ e2 : e2.σ = σ2 ∧ ¬disjoint(e2.g.e1.g) ∧ safe(e2))
12          continue
13     return false # unsafe merge
14   for  e1 in {e : e.σ = σ1  ∧ safe(e)}
15     if (∃e2 : e2.σ = σ2 ∧ ¬disjoint(e2.g.e1.g) ∧ ¬safe(e2))
16         return false # unsafe merge
17     for e2 in (e : e.σ = σ2 ∧ e.m = e1.m ∧ safe(e))
18         if (disjoint(e1.g, e2.g))
19             continue
20         else if (e1.act ≠ e2.act)
21             return false # non-deterministic
22         else if not (mergeStates(e1.σ', e2.σ'))
23             return false # non-determinism
24   Change the destination (σ') of execution steps that end
           at σ2 to σ1
25   remove σ2 and return true
```

Definition 9. Un(safe) Execution Step. An execution step is unsafe if it ends at the unsafe state (Sec. 3.2), and is safe otherwise. Based on this definition, we assume that function $safe(\sigma)$ returns *true* if σ is safe, and *false* otherwise.

Algorithm 3 presents our method for the minimization of an *LTS*. It begins by creating a list of all possible pairs (σ_1, σ_2) of execution states of the *LTS*. It then iterates over all of the pairs and tries to merge them. After each merge, pairs related to the merged execution states are removed from the list. Finally, it removes the unsafe execution state and all relevant steps, since execution must not reach them. (Note that they were added during SSE (Sec. 3.2) to help guide the merging process.)

Merging Execution States. Two execution states (σ_1, σ_2) are considered mergeable if their merge (1) does not cause un-resolvable non-determinism, and (2) does not lead to the merge of a safe execution state with an unsafe one (unsafe merge). Function *mergeStates* of Algorithm 3 presents our merge method, which accepts two execution states σ_1 and σ_2 and tries to merge σ_2 into σ_1. It first checks to see that none of the outgoing unsafe execution steps of σ_1 is matched with an outgoing safe execution step of σ_2 whose triggering messages are equal, and whose guards are not disjoint (line 10-12). Second, it performs a similar check for the outgoing safe execution steps (line 17-19). These two checks ensure that no unsafe execution state is merged with a safe execution state. Third, it checks if the merging of states would cause non-determinism, which can happen in two cases (line 20-23): (a) when two execution steps have the same trigger, non-disjoint guards, and different actions, and (b) when two execution steps have the same trigger, non-disjoint guards, and the same actions. In case (a), the merge is discarded. In case (b), we try to resolve the merge by merging

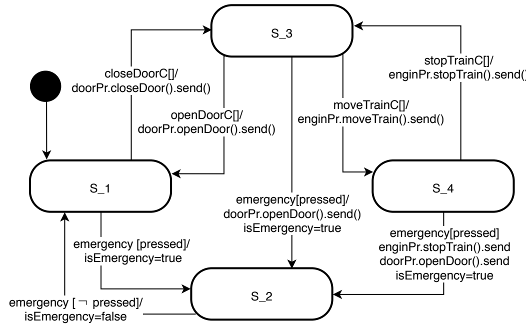

Figure 5: A final UML-RT state machine of the running example (for the CTR component). Transitions are labelled with trigger[guard]/actions.

subsequent execution state pairs recursively, to obtain a deterministic solution. If merging subsequent states is not possible, then the merge of σ, σ' is discarded, and the algorithm continues to the next candidate pair.

Integrating LTSs into the UML-RT Model as SMs. Finally, we use model transformations using to convert the transformed LTSs to SMs, and integrate them with the structural models. Note that since the result of exploration is saved into an EMF model, this phase is accomplished using a straightforward model transformation (whose details are not discussed here due to space limitations). The interested reader may refer to the source code of the transformation [24] (method `transfromLTS2UMLRT` in file `ComponentStateSpaceExplorer.java`). Figure 5 shows the final synthesized UML-RT SM of the running example (for the CTR component). Note that all transitions include their triggers, guards, and actions, and the model can be directly executed using existing *MDD* tools such as Papyrus-RT.

3.5 Soundness of the Proposed Approach

Each step in phase 2 (exploration in Sec. 3.2) and phase 3 (action synthesis in Sec. 3.3) of our approach is verified against the synthesis formulas as part of our process. Thus, the results are sound with respect to the original intentions of users (the system properties), if the synthesis formulas are correctly constructed in phase 1 (Sec. 3.1). The construction of synthesis formulas has been formally explained, and we follow a standard verification process, whose correctness can be easily checked using the details presented in this Section.

The minimization algorithm of phase 4 (Sec. 3.4) prevents nondeterminism, avoids the merging of safe and unsafe executions, and never removes any safe execution step. These three conditions ensure that the minimized *LTS* preserves all of the semantics of the original *LTS*. While we have not presented a formal proof of behaviour preservation here, such a proof can be constructed by (1) defining a bi-simulation relation [27] between the *LTSs* based on our extensions to execution steps (Def. 7), and (2) showing that the bi-simulation relation holds between the synthesized *LTS* and its minimized version. Note that the bi-simulation relation entails behaviour preservation [28].

4 Evaluation

We have created a prototype that embodies our approach. Our prototype uses Z3 [29] as the SMT solver, the Epsilon Object Language [30] to implement the model transformations, Xtext [31] to capture and validate system properties, and the Eclipse Modeling Framework (EMF) [32] to edit LTSs. The source code of the tool, along with experimental results, is publicly available online [24].

To assess the applicability, performance, and scalability of our approach, we consider the following two research questions, and compare with previous work where possible.

RQ1 (Applicability): Can our approach synthesize executable SM models from high-level specifications?

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

4.1 Case Studies

We undertook a study of the most relevant work in the synthesis of SM models in the context of MDD. When a sufficient description was available, we extracted the case study used by each approach. This resulted in the capture of four case studies: *Automatic Teller Machine (ATM) Controller* [11, 33–36], *Mine Pump Controller* [26], *Cache Server* [37], and *Train Controller* [8, 26]. A detailed discussion of the case studies can be found in [38].

We observed that the state space of each of the published case studies could be effectively minimized as part of SSE, and therefore their state space was not large enough to assess the scalability of our bounded SSE and LTS minimization approaches. (The largest case study's LTS contains only 128 states.) For this reason, we added the *Digital Watch*, described in [39], as a complementary case study. The behaviour specification of the digital watch requires integer variables, and yields a state space large enough to draw conclusions about the scalability and effectiveness of our bounded exploration and state minimization approaches.

4.2 Experiments

Specification. We specified the system properties of the case studies using the OCL-like language described in Def. 8. On average, each system took about one hour to specify in our notation. To ensure the quality of the specifications, the specified properties were checked by the other authors and improved upon if there were any issues. The published description of all of the case studies focuses only on the controller component, and the specification of the cache server is partial. We therefore evaluated our approach primarily based on the controller component of each case study. In all five case studies the complexity of the controller component is higher than any other component, and thus represents the most challenging synthesis problem.

Synthesis of Behavioural Models. We then used our prototype to synthesize the SM of each of the case studies from its system properties specification in our OCL-like language. The prototype was configured to separately output the time of each phase, the size and complexity of specifications, the size of the state space, and the generated LTSs. The maximum length for the sequence of generated actions was set to 5.

Bounded Exploration. Since the state space of the digital watch is large (more than 20 billion states), to test scalability we ran the SSE with depth bounds ranging from 1 – 60. To examine the effect of bounded exploration on the quality of the synthesized SMs, we repeated the synthesis of the ATM and mine pump case studies using depth bounds ranging from 3-8, and compared the resulting SMs.

Table 2: Results and computation time of synthesis of case studies

Case study	Spec. Size		LTS Size		SM Size			Synt. Time (Seconds.)						Cov. %
	V	MC	ES	ESt	S	T	Act. (LOC)	SMT G.	SSE	Act.	M	Tr	Over.	
Train System	3	13	5	12	5	13	11	.001	0.65	0.33	0.06	.34	1.04	100%
Cache Server	4	8	11	21	4	15	15	.001	1.21	0.11	.009	0.34	1.67	100%
ATM	6	16	8	10	3	10	19	.001	2.04	0.16	0.008	0.31	2.52	100%
Mine Pump	7	21	128	864	9	106	110	.001	50.51	11.40	0.17	0.38	62.46	100%
Digital Watch	6	6	3724	8915	1830	8915	1767	.001	471.42	8.93	18.14	7.48	505.97	60%

V: Variables, *MC*: Message Conditions, *ES*: Execution State, *ESt*: Execution Step, *S*: State, *T*: Transition, *Act.*: Action, *LOC*: Lines of Code

SMT G.: SMT Formula Generation, *SSE*: State Space Exploration, *M*: Minimization, *Tr*: Transformation, *Over.*: Overall, *Cov.*: Coverage,

Execution Environment. We used a computer equipped with a 3.5 GHz Intel Core *i*7 and 16 GB of memory for all experiments. The java heap size was set to 10 GB.

4.3 RQ1: Applicability

Our prototype successfully synthesized the behavioural models for all five case studies from their specification in our OCL-like language. Table 2 summarizes the complexity of the case studies in our notation, and the results of our automated synthesis of behavioural models for them. The *Spec. Size* column presents the complexity of the specification of each case study in terms of the number of variables *V* and message conditions *MC*. The mine pump specification requires the highest number of variables and message conditions.

State Space Exploration (SSE). As discussed in Sec. 3, the results of SSE are saved as LTSs. The *LTS Size* column shows the number of execution states *ES* and execution steps *ESt* for the controller component's LTS (i.e., the component with the largest LTS) in each case study. The LTSs for all of the case studies except the digital watch are relatively small, and easily explored by our tools. Bounded exploration to a maximum depth of 60 was used for the digital watch, which resulted in 3,724 states and 8,915 execution steps. The number of execution steps actually explored (EESt) and possible execution states (PES) is larger than the number of states and execution steps in the LTSs, since all unsafe states are merged into one unsafe state, and execution steps that do not change the state are not saved. All of the case studies (except the digital watch) come directly from the published evaluations of other state-of-the-art techniques, and our tool's ability to easily explore them clearly demonstrates its effectiveness compared to the other methods.

Synthesis of Actions. The *Cov.%* column of Table 2 shows the percentage of execution steps whose actions were successfully synthesized. The high level of coverage indicates that the action synthesis aspect of our approach can help automate much of the coding related to communication and assignments in SMs. The high level of coverage is not only due to our action synthesis algorithm – two assumptions of our method also contribute. First, the search space for action sequences in our method is limited to only sending messages and assigning local variables. Second, we limit our sequences of actions to those that are loop and branch free. The latter limitation explains the lower coverage of actions in the digital watch case study, which requires branching. Nevertheless, our generated actions are useful and practical, since writing actions related to the communication between components is challenging in multi-component systems, and the number of variables and messages in an embedded system is typically small and finite.

Effectiveness of Minimization of State and Final Synthesized State Machines. As discussed, the LTSs are minimized and then transformed into UML-RT SM models. The *SM Size* column shows the size of the final SMs for the controller component of each case study, in terms of the number of states *S*, transitions *T*, and lines of code in the synthesized actions *A (LOC)*. On average our state minimization reduces the size of the LTSs by factor of 4.5 in the case studies, the Train System by 1, Cache Server by 2.7, mine pump by 14.2, Digital Watch by 2, and ATM by 2.6. As a result of minimization, the final SMs for all four case studies extracted from the literature have fewer than 10 states, and can be easily understood by users.

While the LTS of the digital watch is minimized significantly, an SM with 1,830 states is clearly still not easily understood by users. Our investigation showed that many of the states were not merged due to actions that were not successfully synthesized (ref. Table 2). By manually adding these actions, the LTS of the digital watch is minimized to only two states and five transitions. Based on the points mentioned above, we can safely conclude that our state minimization techniques effectively reduce the size of the final state machines.

Arguably, with the exception of the digital watch, the state space of the case studies is not large. However, we should note that (1) none of the related work reports a large case study, and (2) synthesis techniques, even those in program synthesis, are still not mature enough to be applied at industrial scale. Nevertheless, our method and evaluation represents a significant step beyond the related work, specifically because (a) the state space of the digital watch (3,724 states) is more than 5 times larger than the state space of the largest published industrial case study, with 658 states reported [33], and (b) the synthesized SMs from our approach include executable actions, which no other approach addresses.

4.4 RQ2: Computation Time and Scalability

The *Synthesis Time* column of Table 2 shows the computation times (in seconds) for each phase of the synthesis for the case studies. As may be expected, SSE accounts for most of the computation time (93%), and generation of the synthesis formulas (*SMT G.*) takes very little time in all cases. The overall synthesis time (*Over.*) for the four case studies extracted from the literature ranges between 1 and 63 seconds (about a minute), which seems more than reasonable. The overall synthesis time of the much larger digital watch case study is 505 seconds (8.5 minutes), more than practical for the exploration and minimization of such a large state space (3,724 states).

To evaluate our method in the presence of a state space explosion, we explored state space exploration with depth bounds ranging

from 1 to 60 for the digital watch, and 3 to 8 for ATM and mine pump. The results show that in each of these cases the final synthesized SMs do not change after a relatively small bound (5 for digital watch, 4 for ATM, and 5 for mine pump). This suggests that bounded exploration can be an effective method to manage state space explosion in the context of our work, and that the LTSs generated from a bounded search can provide sufficient traces to synthesize an SM covering the majority of the expected behaviour. Nevertheless, this conclusion is based only on this particular set of case studies. While they cover a range of typical cases, they may not be representative of other systems.

In summary, given the reasonable performance of our approach for the synthesis of all of the case studies (505 seconds for the digital watch with 3,724 states, and 2 - 63 seconds for the others) and the effectiveness of bounded exploration in dealing with the state space explosion, we conclude that our approach can be practical for realistically sized embedded systems. Our current implementation is also only a prototype, and its performance can still be tuned to even greater efficiency in practice.

5 Related work

Synthesis of SMs to capture behaviour models of a system has long been an active area of research [6, 8, 9, 12, 26, 33–37, 40–45]. Existing work can be categorized into two groups: *scenario-based synthesis*, and *correct-by-construction synthesis*.

Scenario-based Synthesis. The main drawback of these methods is that defining a full set of scenarios to cover the entire description of system requirements is almost impossible, and often impractical [8]. Following we discuss some of them that are most close to our work. Harel et al. [6, 44] proposed a semi-automatic synthesis approach to infer Statechart models from scenario-based requirements specified using live sequence charts (LSCs). The work relies on a play-in/play-out approach that includes user interaction in the synthesis algorithm. Uchitel et al. [7, 33] presented an approach to generate LTSs for agents of a message sequence chart (MSC). Damas et al. [8] proposed an approach to synthesize an LTS from an MSC for the global system covering all positive scenarios and excluding all negative ones. The generalization process is guided by scenario questions asked of the end-user in an incremental synthesis process.

By contrast with these techniques, our approach does not require scenarios to be specified by users, and does not require user interaction during synthesis. Instead, it requires only user-provided OCL-like expressions to seed the synthesis process. Also, to make the SMs executable and more complete, we synthesize actions for the state transitions.

Whittle and Schumann [11] proposed an algorithm for automatically generating UML statecharts to synthesize the behaviour of component-based systems. The algorithm requires scenarios and pre/post conditions for scenario interactions. Similarly to our work, the pre/post conditions are expressed using OCL-like expressions based on global state variables. Similarly to Uchitel's work, the synthesis algorithm takes only positive scenarios into account. By contrast, our work does not require scenarios, and generates both positive and negative scenarios by state space exploration.

Correct-by-construction Synthesis. Work (e.g., [13–15]) in this group attempts to synthesize a correct implementation of the system based on a temporal logic specification of the system specified by users.

Game Theory-based Approaches. Several researchers [13, 15, 46, 47] have modelled the synthesis process, specified by the temporal logic formula, as a game between the environment and the program. A correct program is considered to be a winning strategy in this game. Kupferman et al. [48] proposed a compositional reactive synthesis algorithm that translates LTL formulas to nondeterministic generalized Buchi automata.

Multi-agent-based Approaches. A control synthesis algorithm for systems with multiple controllable agents has been studied by several previous authors [49–51]. Diaz et al. [49] proposed an approach for low-level control synthesis of multi-agent systems expressed as a finite state automaton. As a practical end-to-end synthesis method, our work differs from correct-by-construction work as follows. Our approach captures properties as message conditions, which we argue are much easier to specify than LTL formulas. Researchers in LTL-based synthesis (e.g., [15, 50]) identify specification and understanding of LTL formulas as a challenge and obstacle to use of their method. Also, to our knowledge, no correct-by-construction work synthesizes action code. Without action code, synthesized state machines cannot be executed. Also, LTL-based work generates LTS or Büchi automata, by contrast, we generate complete executable state machines.

Other Use of SMT Solvers. SMT solvers have been used in several other techniques [52–58], modelling the synthesis problem as generation of a program that satisfies the specification. We review these methods based on the classification of Gulwani [17].

Counter-example Guided (CEG) Inductive Synthesis. In this method the aim is to look for new program candidates with respect to the specification by iteratively generating counterexamples until a correct program is synthesized (e.g., [18]).

Sketching. Sketching is a synthesis approach that takes as input a program with holes, which are then automatically filled to satisfy a given specification (e.g., [53, 59, 60].

Super-optimizers. The goal here is to synthesize an optimal sequence of instructions that is functionally equivalent to a given piece of code (e.g., [17, 52]).

6 Conclusion

In this paper we have presented a novel technique for synthesizing behavioral models from high-level system specifications using SMT solvers in the context of MDD. We described our approach in detail and analyzed its applicability, performance, and scalability in a number of different use cases. By contrast with existing work, our approach is an end-to-end synthesis solution integrated with an MDD tool and synthesizes detailed actions for the transitions of the generated SMs.

To the best of our knowledge, our work is also the first work to leverage SMT-Solvers for the synthesis of the models. An implementation of our approach is publicly available [24], and we are hopeful that in follow-up research we can (1) improve the applicability and scalability of the model synthesis, and (2) extend our approach to support completion of incomplete (partial) SMs [61, 62].

Acknowledgment

This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3, 2003, pp. 1–17.
[2] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.
[3] N. Kahani and J. Cordy, "Comparison and evaluation of model transformation tools," in *Tech. Rep. 2015-627*. Queen's University, Dec. 2015, pp. 1–42.
[4] N. Kahani, M. Bagherzadeh, J. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software and Systems Modeling*, vol. 18, no. 4, p. 2361–2397, 2019.
[5] B. Selic, "What will it take? a view on adoption of model-based methods in practice," *Software and Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.
[6] D. Harel, H. Kugler, and A. Pnueli, "Synthesis revisited: Generating statechart models from scenario-based requirements," in *Formal Methods in Software and Systems Modeling*, 2005, pp. 309–324.
[7] S. Uchitel, G. Brunet, and M. Chechik, "Behavior model synthesis from properties and scenarios," in *Proc. 29th Intl. Conf. on Software Engineering*, 2007, pp. 34–43.
[8] 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.
[9] S. Uchitel and J. Kramer, "A workbench for synthesising behaviour models from scenarios," in *Proc. 23rd Intl. Conf. on Software engineering*, 2001, pp. 188–197.
[10] N. Kahani, "Automodel: a domain-specific language for automatic modeling of real-time embedded systems," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 515–517.
[11] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *Proc. 22nd Intl. Conf. on Software engineering*, 2000, pp. 314–323.
[12] J. Whittle and P. K. Jayaraman, "Synthesizing hierarchical state machines from expressive scenario descriptions," *ACM Trans. on Software Engineering and Methodology*, vol. 19, no. 3, pp. 1–45, 2010.
[13] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, 1992.
[14] E. Letier and W. Heaven, "Requirements modelling by synthesis of deontic input-output automata," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 592–601.
[15] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th Symposium on Principles of Programming Languages*, 1989, pp. 179–190.
[16] S. Maoz, J. O. Ringert, and R. Shalom, "Symbolic repairs for GR(1) specifications," in *Proc. 41st Intl. Conf. on Software Engineering*, 2019, pp. 1016–1026.
[17] S. Gulwani, O. Polozov, and R. Singh, "Foundations and trends in programming languages," *ACM Trans. Comput. Log.*, vol. 4, no. 1-2, pp. 1–119, 2017.
[18] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. 32nd Intl. Conf. on Software Engineering*, 2010, pp. 215–224.
[19] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," in *PLDI*, 2002, pp. 304–314.
[20] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *Proc. 8th USENIX Conf. on Operating Systems Design and Impl.*, 2008, pp. 177–192.
[21] B. Selic, "Using UML for modeling complex real-time systems," in *Languages, Compilers, and Tools for Embedded Systems*, 1998, pp. 250–260.
[22] Eclipse Foundation, "Eclipse Papyrus for Real Time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, 2019, retrieved March 19, 2019.
[23] N. Kahani, N. Hili, J. Cordy, and J. Dingel, "Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems," in *Proc. 9th Intl. Workshop on Modelling in Software Engineering*, 2017, pp. 12–18.
[24] N. Kahani, M. Bagherzadeh, and J. Cordy, "UMLRTSynthesizer," https://github.com/nafisehka/UMLRTSynthesizer, 2020.
[25] IBM, "IBM RSARTE," 2016, retrieved July 19, 2019. [Online]. Available: https://www.ibm.com/developerworks/downloads/r/architect/index.html
[26] C. Damas, B. Lambeau, and A. Van Lamsweerde, "Scenarios, goals, and state machines: a win-win partnership for model synthesis," in *Proc. 14th Intl. Symp. on Foundations of Software Engineering*, 2006, p. 197–207.
[27] D. Sangiorgi, *Introduction to bisimulation and coinduction.* Cambridge University Press, 2011.
[28] N. A. Lynch, *Distributed algorithms.* Elsevier, 1996.
[29] "Z3," https://github.com/Z3Prover/z3.
[30] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon transformation language," in *Intl. Conf. on Theory and Practice of Model Transformations*, 2008, pp. 46–60.
[31] Xtext. (2017) Xtext. [Online]. Available: http://www.eclipse.org/Xtext.
[32] EMF. (2017) Eclipse Modeling Framework (EMF). [Online]. Available: https://www.eclipse.org/modeling/emf.
[33] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Trans. on Software Engineering*, vol. 29, no. 2, p. 99–115, 2003.
[34] S. Vasilache and J. Tanaka, "Synthesis of state machines from multiple interrelated scenarios using dependency diagrams," in *In 8th World Multiconf. on systemics, cybernetics and informatics*, 2004, pp. 49–54.
[35] R. Hennicker and A. Knapp, "Activity-driven synthesis of state machines," in *Intl. Conf. on Fundamental Approaches to Software Engineering*, 2007, pp. 87–101.
[36] A. Ali, D. Jawawi, and M. A. Isa, "Scalable scenario specifications to synthesize component-centric behaviour models," *Intl. Journal of Software Engineering and Its Applications*, vol. 9, no. 9, pp. 79–106, 2015.
[37] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic, "Synthesizing partial component-level behavior models from system specifications," in *Proc. 7th Joint Meeting on Foundations of Software Engineering*, 2009, pp. 305–314.
[38] N. Kahani, "Synthesis and verification of models using satisfiability modulo theories," Ph.D. dissertation, 2020.
[39] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.
[40] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in *Proc. 36th Intl. Conf. on Software Engineering*, 2014, pp. 468–479.
[41] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Trans. on Industrial Informatics*, vol. 13, no. 4, pp. 1521–1530, 2017.
[42] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, 2016.
[43] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to statecharts," in *Intl. Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*, 1999, p. 61–71.
[44] D. Harel and H. Kugler, "Synthesizing state-based object systems from LSC specifications," *Intl. J. of Foundations of Computer Science*, vol. 13, no. 1, pp. 5–51, 2002.
[45] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, no. 2, pp. 86–95, 2017.
[46] D. Dill, "Trace theory for automatic hierarchical verification of speed independent circuits," *MA: MIT press*, vol. 24, pp. 1–180, 1989.
[47] R. Alur and S. L. Torre, "Deterministic generators and games for LTL fragments," *ACM Trans. Comput. Log.*, vol. 5, no. 1, pp. 1–25, 2004.
[48] N. P. O. Kupferman and M. Vardi, "Safraless compositional synthesis," in *Computer Aided Verification (CAV)*, 2006, pp. 31–44.
[49] Y. Diaz-Mercado, A. Jones, C. Belta, and M. Egerstedt, "Correct-by-construction control synthesis for multi-robot mixing," in *54th IEEE Conf. on Decision and Control (CDC)*, 2015, pp. 221–226.
[50] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.
[51] R. Alur, S. Moarref, and U. Topcu, "Compositional and symbolic synthesis of reactive controllers for multi-agent systems," in *Information and Computation*, 2018, pp. 616–633.
[52] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.
[53] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *ACM SIGPLAN Notices*, vol. 49, no. 11, pp. 404–415, 2006.
[54] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter, "Synthesis modulo recursive functions," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 407–426, 2013.
[55] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in SMT," in *Intl. Conf. on Computer Aided Verification*, 2015, pp. 198–216.
[56] Z. Huang, Y. Wang, S. Mitra, G. E. Dullerud, and S. Chaudhuri, "Controller synthesis with inductive proofs for piecewise linear systems: An SMT-based algorithm," in *54th IEEE Conf. on decision and control (CDC)*, 2015, pp. 7434–7439.
[57] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *ACM Sigplan Notices*, vol. 45, no. 1, 2010, pp. 313–326.
[58] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv, "A simple inductive synthesis methodology and its applications," in *ACM Sigplan Notices*, vol. 45, no. 10, 2010, pp. 36–46.
[59] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Intl. Conf. on computer aided verification*, 2013, pp. 934–950.
[60] A. Solar-Lezama, "The sketching approach to program synthesis," in *Asian Symp. on Programming Languages and Systems*, 2009, pp. 4–13.
[61] M. Bagherzadeh, N. Kahani, J. Karim, and J. Dingel, "PMExec: An execution engine of partial UML-RT models," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1178–1181.
[62] M. Bagherzadeh, N. Kahani, J. Karim, and J. Dingel, "Execution of partial state machine models," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–27, July 2020.