Symbolic Execution for Realizability-Checking of Scenario-based Specifications

Joel Greenyer
Software Engineering Group,
Leibniz Universität Hannover, Hannover, Germany
greenyer@inf.uni-hannover.de

Timo Gutjahr
Zühlke Engineering GmbH
Hannover, Germany
Timo.Gutjahr@zuehlke.com

***Abstract*—Scenario-based specification with the Scenario Modeling Language (SML) is an intuitive approach for formally specifying the behavior of reactive systems. SML is close to how humans conceive and communicate requirements, yet SML is executable and simulation and formal realizability checking can find specification flaws early. The realizability checking complexity is, however, exponential in the number of scenarios and variables. Therefore algorithms relying on explicit-state exploration do not scale and, especially when specifications have message parameters and variables over large domains, fail to unfold their potential. In this paper, we present a technique for the symbolic execution of SML specifications that interprets integer message parameters and variables symbolically. It can be used for symbolic realizability checking and interactive symbolic simulation. We implemented the technique in SCENARIOTOOLS. Evaluation shows drastic performance improvements over the explicit-state approach for a range of examples. Moreover, symbolic checking produces more concise counter examples, which eases the comprehension of specification flaws.**

I. INTRODUCTION

Many software-intensive systems, especially *cyber-physical systems*, consist of reactive components that interact with each other and the environment in order to realize complex and often safety-critical functionality.

During the early design of such systems, it is natural to conceive and communicate their behavior in the form of scenarios that describe how the system components may, must, or must not interact in reaction to environment events. The formal modeling and analysis of such scenarios is desirable in order to find specification flaws early, thereby avoid costly iterations, and to have a solid basis for further development,

Live Sequence Charts (LSCs) [1], [2], and a textual variant, the Scenario Modeling Language (SML), support modeling scenarios formally. SML extends LSCs with concepts for specifying environment assumptions and dynamic topologies [3].

LSC/SML scenarios are executable via the *play-out* algorithm [4], [2], [5]. The algorithm can be used to simulate the interplay of the scenarios. Violations encountered during simulation runs hint at possible specification flaws.

Simulation alone, however, cannot prove the absence of flaws. Therefore, there exist approaches to formally check the *realizability* of such specifications [6], [7], [8], [9], [10], [11], [12], [13]. *Realizability checking* means checking whether an implementation for a specification exists [14]; if a specification is *unrealizable*, it means that it contains inconsistencies and that the environment can always force the system to violate the specification. Realizability checking can be done through *controller synthesis*, which is the automatic construction of implementations (usually state-based controllers) from a specification. This requires an exploration of the state space induced by the specification, which grows exponentially with the number of scenarios and variables (state space explosion).

One way to address this issue is by mapping the realizability checking problem to BDD-based model checkers or game solvers [15], [16]. However, it is difficult to map *rich scenario language concepts*, like parameterized messages, dynamic polymorphic bindings [17], or dynamic topologies [3], to the lower-level languages used by these tools.

Therefore, we investigate the alternative: to adapt the concept of *symbolic execution* [18], [19] to LSC/SML specifications, as it could be more easily integrated into existing tools that already support the play-out of LSC or SML specifications with rich language concepts.

Symbolic execution is a technique for executing programs with symbolic instead of concrete values for inputs. It can deduce for which constraints on the inputs it is possible to arrive at a particular part of a program that is of interest, e.g., the violation of an assertion. Symbolic execution was also extended for reactive systems, which periodically receive inputs and provide outputs, [20], [21], [22]. For LSC/SML scenario specification, symbolic execution would mean performing play-out execution with symbolic values for environment event parameters and initial component state attributes.

Such an approach could improve the scalability of realizability checking *for specifications that have message parameters and variables over large domains*. Symbolic execution, however, is generally known not to scale for larger programs, because the number of program paths that must be checked grows exponentially with the program size. For the same reason, the symbolic execution of scenarios may not necessarily improve the scalability with an increasing number of scenarios.

We address the following research questions in this paper:

RQ1: How can the SML/LSC play-out algorithm be extended for the symbolic interpretation of input values?

RQ2: How can this extended algorithm be employed for realizability checking?

RQ3: What is the performance of the symbolic interpretation approach compared to the explicit state approach?

In answer, to these research questions, our contributions are:

***RQ1*:** We developed an extension to the SML/LSC play-out algorithm to support the symbolic interpretation of message parameters and component variables. The challenge, compared to the symbolic execution of sequential programs, is that during play-out execution, multiple scenarios can be active and formulate conditions over message parameter- and variable values. One step in symbolic play-out could thus result in branching into as many paths as required to cover all possible combinations of branchings implied by each active scenario.

***RQ2*:** We developed a symbolic realizability checking technique that, based on symbolic play-out, proves whether or not a specification is *play-out executable*. This conditions says that for any sequence of environment events, a play-out execution *(a)* never causes a violation in any guarantee scenario, and *(b)* always eventually listens for the next environment event.

In order to prove play-out executability, we must first build a finite graph of all symbolic play-out executions. On this graph, we search for violating states and cycles of only systems steps.

Building such a graph includes deciding, on the exploration of a new transition (representing a play-out step), whether this transition should lead to a new symbolic state or to an already existing one that *matches* the new symbolic state.

We consider two symbolic state matching techniques:

a. state equivalence: Symbolic states are merged when they represent the same set of concrete states. With this technique, the paths in the symbolic play-out graph represent exactly the concrete executions, which allows us to derive the correctness and completeness of the checking algorithms. However, it may also result in large graphs, since many symbolic states could be created with intersections of concrete states that they represent.

b. state subsumption: A newly explored state is merged into an existing state when the concrete states represented by the existing state is a superset of the concrete states represented by the newly explored state, i.e., the former *subsumes* the latter. This leads to an *over-approximation*: the resulting graphs can be much smaller than the ones created with state equivalence, while it still ensures that all reachable symbolic states represent exactly the reachable concrete states. However, not all paths in the symbolic play-out graph are possible concrete execution paths. This means that found cycles of only system steps may be spurious—but additional tests can verify this.

The work in this paper has the limitation that the play-out executability condition is stronger than more general notions of realizability (cf. [14], [23], [7], [16]). Checking these notions of realizability usually requires solving games, whereas we only consider reachability properties and cycle detection. Nevertheless, this work presented here is still useful and can be extended to support more general notions of realizability.

We implemented the symbolic realizability checking techniques within SCENARIOTOOLS. For constraint solving, we integrated the SMT solver Z3 [24]. In particular, SCENARIOTOOLS supports the symbolic execution of feature-rich specifications, including dynamic polymorphic bindings [17], and systems with dynamic component topologies [3]. Symbolic execution paths can also be explored interactively.

***RQ3*:** We compared the performance of realizability checking based on symbolic and concrete play-out graphs for a range of examples. The technique based on state equivalence only performed well in some cases, while the technique based on state subsumption scaled well for all examples.

Structure: We introduce an example in Sect. II, basic concepts in Sect. III, and SML in Sect. IV. We give further definitions in Sect. V, explain symbolic play-out and realizability checking in Sect. VI, and show evaluation results in Sect. VII. Related work appears in Sect. VIII and we conclude in Sect. IX

Artifact download: http://scenariotools.org/models-2017/

II. EXAMPLE

As an example, we consider an *oven temperature controller* that regulates the temperature of an oven to a set-point temperature. The set-point temperature is stored in a variable. As inputs, the controller can receive temperature measurements from a sensor and new set-point temperature settings from a user. As outputs, the controller can turn a heater on or off, as well as a light that indicates whether the heater is on or off.

We consider the following guarantees:

- ***G1:*** When the temperature controller (`ctr`) receives a measurement from the temperature sensor with a value greater or equal to the set-point temperature, `ctr` must turn the heater off; if the temperature is smaller than the set-point temperature, `ctr` must turn the heater on.
- ***G2:*** When `ctr` receives a new set-point temperature value, it must assign it to its set-point temperature variable.
- ***G3:*** When `ctr` receives a temperature measurement with a temperature smaller than the set-point temperature, `ctr` must turn the light on.
- ***G4:*** When `ctr` receives a temperature measurement with a temperature greater of equal to the set-point temperature, `ctr` must turn the light off.

III. SCENARIO-BASED MODELING – BASIC DEFINITIONS

A specification defines how objects in an *object model* must interact by sending messages. The objects represent system components and environment entities, and are partitioned into *system* and *environment* objects. Objects are instances of classes in a *class model* and carry values for attributes defined by their class. We give basic definitions. Some concepts are simplified, e.g., we do not define associations between classes.

Definition III.1 (Class model)**.** A *class model* is a tuple $CL = (Cl, D, Attr, Op)$, where Cl is a set of *classes*, D is a set of *data types*, $Attr \subseteq Cl \times Name \times D$ is a set of named attributes that are typed over data types D. $Name$ is a set of names. $Op \subseteq Cl \times Name \times Parameter^*$ with $Parameter = (Name \times D)$ is a set of *operations* that have a name and a possibly empty list of parameters typed over data types. $Op(cl)$ are the operations of a class $cl \in Cl$.

Definition III.2 (Object model)**.** An *object model* $\mathcal{O} = (O, O_c, O_u, CL, InstanceOf, Values, AttrValues)$ is a tuple where O is a set of objects, which is partitioned into *controllable* objects O_c (also called *system* objects) and *uncontrollable* objects O_u (also called *environment* objects). $CL =$

$(Cl, D, Attr, Op)$ is a class model and $InstanceOf : O \rightarrow Cl$ defines the class for each object. $Values$ is a set of data values, and $AttrValues : O \times Attr \rightarrow Values$ defines the values of each object's attributes (matching the data types).

Definition III.3 (Message Event)**.** We consider *synchronous* communication where the sending and receiving of a message is a single event, called *message event* or only *event*. In an object model $\mathcal{O} = (O, O_c, O_u, CL, InstanceOf, Values, AttrValues)$, a message event $\sigma = (o_s, op, (val_0, ...), o_r) \in O \times Op \times Values^* \times O$, is a tuple where o_s and o_r are the sending resp. receiving objects, $op \in Op(InstanceOf(o_r))$ is an operation of the target object's class, and $(val_0, ...) \in Values^*$ is a possibly empty list of values that match the parameters of *op*. A message event is *controllable* if $o_s \in O_c$ and *uncontrollable* if $o_s \in O_u$. A controllable message event is also called *system event*; an uncontrollable message event is also called *environment event*. Σ_O is the set of all message events in object model O. Σ_{O_u} is the set of all uncontrollable message events; Σ_{O_c} is the set of all controllable message events.

Message events can change attribute values of objects. By convention, message events referring to operations named $\mathsf{set}\mathit{Attr}(\mathsf{p}{:}\mathsf{D}_{\mathsf{Attr}})$, where *Attr* is an attribute of the receiving object's class with type $\mathsf{D}_{\mathsf{Attr}}$, will change the receiving object's value for *Attr* to the value carried by the message event. We do not consider object creation or destruction, but the above definitions could be extended to reference properties (pointers), and many-valued properties.

Definition III.4 (Run)**.** A *run* of a system $\pi = \mathcal{O}_0, \sigma_0, \mathcal{O}_1, \sigma_1, ...$ is an infinite sequence of object models and message events where all $\mathcal{O}_i, i \geq 0$ only differ in $AttrValues$.

IV. Scenario Modeling Language (SML)

SML is a textual scenario specification language. There are two interpretations of an SML specification: the *declarative* and the *operational* interpretation. In the declarative interpretation, we think of the specification as a description of the valid runs of a system. The operational interpretation, via the play-out algorithm, is a description of how the controllable objects react to uncontrollable events. In this paper, we base our notion of realizability on the operational interpretation, and thus focus the following explanations on this interpretation.

Listing 1 shows the SML specification for the oven temperature controller. An SML specification references a *domain* class model, called `oven` here. The class and object models are not defined in SML, but instead SML uses MOF [25]. In ScenarioTools, which is based on the Eclipse Modeling Framework (EMF), the class and object models are Ecore models resp. their instances. Ecore implements EMOF [25].

To allow for a flexible combination of an SML specification with different initial object models, an SML specification represents objects by *roles*. The mapping between the roles in an SML specification and a particular initial object model is defined by a *run configuration*, which we omit for brevity.

An SML specification partitions controllable and uncontrollable objects by listing classes of controllable objects as in line 7. Here the controller object is controllable. Instances of other classes are uncontrollable.

The possible *ranges of parameter values* for message events can be defined as shown in lines 9-12.

The scenarios are contained in *collaborations*, which define *roles* that represent objects in the object model. The messages in the scenarios have a sending and a receiving role. Here the roles are *static*, which means that the run configuration defines fixed one-to-one mappings between a role and an object.

```
1  import "../model/oven.ecore"
2
3  specification OvenSpecification {
4
5  domain oven // class model
6
7  controllable{ Controller }
8
9  parameter ranges {
10  Controller.measuredTemp(tmp = [0..300]),
11  Controller.modifySetPointTemp(setPointTmp = [50..300])
12 }
13
14 collaboration OvenCollaboration {
15
16  static role Controller ctr
17  static role TemperatureSensor ts
18  static role Heater heater
19  static role Panel panel
20
21  guarantee scenario OvenRegulation {
22   var EInt temp
23   ts->ctr.measuredTemp(bind temp)
24   alternative [temp >= ctr.setPoint]{
25    strict requested ctr->heater.turnOff()
26   } or [temp < ctr.setPoint]{
27    strict requested ctr->heater.turnOn()
28   }
29  }
30
31  guarantee scenario ModifySetPointTemperature {
32   var EInt setPointTemp
33   panel->ctr.modifySetPointTemp(bind setPointTemp)
34   strict requested ctr->ctr.setSetPointTemp(setPointTemp)
35  }
36
37  guarantee scenario PreheatLightOn {
38   var EInt temp
39   ts->ctr.measuredTemp(bind temp)
40   // error: should be >=, to be detected.
41   interrupt [temp > ctr.setPoint]
42   strict requested ctr->panel.preheatingLight(Status:ON)
43  }
44
45  guarantee scenario PreheatLightOff {
46   var EInt temp
47   ts->ctr.measuredTemp(bind temp)
48   interrupt [temp < ctr.setPoint]
49   strict requested ctr->panel.preheatingLight(Status:OFF)
50  }
51  }
52 }
```

Listing 1. Specification of oven temperature controller

The scenarios shown in List. 1 model the guarantees ***G1-G4***. The operational (play-out) interpretation of the scenarios is as follows. First, the system waits for environment message events to occur. When an event occurs that matches the first message of a scenario, an *active copy* of that scenario is created. If the scenario message has a parameter binding expression `bind` $\langle$`var`$\rangle$, e.g., l. 23 in List. 1, the corresponding scenario variable is bound to the value carried by the message event. In the scenario OvenRegulation, the measured temperature is stored. Then the active scenario progresses. Conditions like alternatives and interrupts are evaluated immediately. If an interrupt condition evaluates to false, the active scenario is terminated. Otherwise, the scenario progresses, until the next message is reached, which is then *enabled*.

Scenarios messages can be *strict* or *non-strict*. When a strict message is enabled, it means that no message event must occur that matches another message in the same scenario that is not currently enabled. Otherwise, this is a *safety-violation* of the scenario. When a strict parameterized message is enabled where the scenario specifies a concrete parameter value, like `Status:ON` (l. 42), then also no message event is allowed to occur that carries another parameter value. Non-strict messages are allowed to be violated in this way; then the active scenario terminates as a result. This case does not occur here, since only the first messages are non-strict.

Scenario messages can also be *requested*. When there is a set of active scenarios with enabled requested system messages, the system non-deterministically executes a message event that matches one of these messages, but does not lead to any safety violation of another active scenario. We also say that an event can be *blocked* by another active scenario. As a result of the message event execution, the active scenarios progress further, terminate when they reach their end, and other scenarios may be activated as a result. As long as enabled requested system messages remain, the system must keep executing them. Otherwise, the system again waits for the next environment event and the process repeats.

Infinite sequences of system message executions is forbidden, since then the system would never again listen to environment events. Moreover, if there are enabled requested system messages, but all are blocked, this creates a forbidden deadlock state. The latter situation can occur in the example: when a temperature is measured that is equal to the set-point temperature, the scenarios PreheatLightOn and PreheatLightOff request conflicting parameter values—to turn the preheating light off and on. The flaw is in line 41, where the greater-than operator should be a greater-or-equal-to.

V. Play-out Graph, Play-out Executability

In the following, we define the concepts of scenario and specification more formally. We also define the *Play-Out Graph* (POG), a structure that captures all the possible play-out executions of a specification with a particular initial object model. We then define *play-out executability*, the notion of realizability that we want to check.

We formalize a scenario as a special form of transition system. SML scenarios can be mapped to this form.

Definition V.1 (Scenario)**.** A *scenario* $sc = (L, Var, Msg, Msg_{init}, l_0, l_{end}, l_{sv}, L_{exp}, \Delta, \Delta_{req}, \Delta_{sv})$ is a tuple with

- a finite set of *locations* L.
- a *start location* $l_0 \in L$.
- an *end location* $l_{end} \in L$.
- a finite set of *scenario variables* Var.
- a set of *scenario messages* $Msg \subset O \times Op \times ParamExp^* \times O$. A scenario message $msg = (o_s, op, (paramexp_0, ...), o_r) \in Msg$ represents one or several message events; $o_s, o_r \in O$ are the sending resp. receiving object, $op \in Op$ is an operation, and $(paramexp_0, ...)$ are zero or more *parameter expressions*, matching the parameters defined by op. $paramexp_i$ can be *(i)* a literal value, *(ii)* a wildcard (*) *(iii)* a value of a variable in Var that matches the type of the ith parameter of op, or *(iv)* a *binding expression* for a variable in Var of compatible type (a binding expression can be used to assign a message event's parameter value to a scenario variable).
- a set of *initializing* scenario messages $Msg_{init} \subseteq Msg$.
- a *safety violation location* $l_{sv} \in L$.
- a set of *progress-expected* locations $L_{exp} \subset L$, where $l_0 \notin L_{exp}$ and $l_{sv} \in L_{exp}$.
- a transition relation $\Delta : L \times Msg \times Guard \times L$. A transition $\delta = (l_s, msg, guard, l_t) \in \Delta$ has a source and a target location l_s and l_t, msg is a scenario message, and $guard$ is a predicate expression over object model attributes and scenario variables. We also require that Δ is condition/event deterministic, i.e. for two transitions $\delta_1 = (l_s, msg, guard_1, l_{t1})$ and $\delta_2 = (l_s, msg, guard_2, l_{t2})$ it must be that $guard_1 \Leftrightarrow guard_2$ is a contradiction.
- For each initializing scenario message there exists a transition leaving the initial location: for all $msg_{init} \in Msg_{init}$ it must be that $(l_0, msg_{init}, guard_{init}, l) \in \Delta$ for some $l \in L$ and $guard_{init} \in Guard$ that must only refers to object model attributes and not to scenario variables.
- $\Delta_{req} \subset \Delta$ is a set of *requested* transitions; source locations of requested transitions are progress-expected, i.e., for all $(l, msg, guard, l') \in \Delta_{req}$ it must be that $l \in L_{exp}$. Also, transitions in Δ_{req} are labeled with scenario messages where the sender is a system object.
- $\Delta_{sv} \subset \Delta$ is a set of *forbidden* transitions, which lead to the safety-violation location, i.e., iff $(l, msg, guard, l') \in \Delta_{sv}, guard \in Guard, l \in L$ then $l = l_{sv}$.
- Δ defines no transitions that leave l_{sv} or l_{end}.

Figure 1 shows the transition-system scenarios for the SML scenarios in List. 1. Locations where the scenarios are active are labeled with the lines in List. 1 that they correspond to.

In OvenRegulation, the two transitions leaving l_1 to l_{end} correspond to the messages appearing in the alternative fragment in l. 25/27 of List. 1. l_1 is progress-expected and the transitions leaving l_1 are requested, represented by the thick border and arrow lines. The transition leading to l_{sv} represents safety violation that can occur due to the strictness of the messages. In the PreheatLightOn/-Off scenarios, the transitions from l_0 to l_{end} represent the interrupt conditions.

Definition V.2 (Specification)**.** A *specification* $Spec = (Sc_G, \mathcal{O}_0)$ consists of an initial object model $\mathcal{O}_0$ and a set of *guarantee* scenarios $Sc_G = \{sc_{G1}, ..., sc_{Gk}\}$.

Before explaining the POG, we define when a message event matches a scenario message.

Definition V.3 (Signature)**.** Given a message event $\sigma = (o_s, op, (val_0, ...), o_r)$, we call $sig(\sigma) = (o_s, op, o_r)$ its *signature*. Likewise, given a scenario message, $msg =$

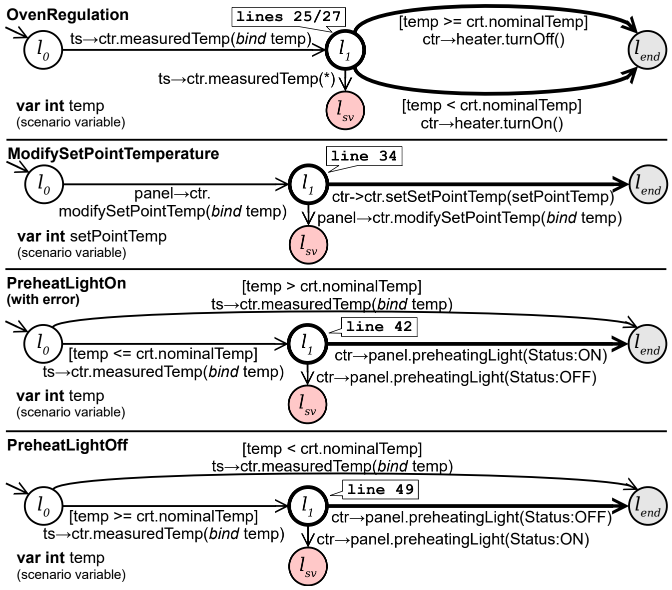

Figure 1. The transition-system scenarios correspond to the SML scenarios

$(o_s, op, (paramexp_0, ...), o_r)$, its *signature* is $sig(msg) = (o_s, op, o_r)$. $Sig(O)$ is the set of all signatures where an object $o \in O$ is the sender. ($Sig(O)$ will be needed in Def. VI.4.)

Definition V.4 (Message event matches scenario message)**.** A message event σ *matches* a scenario message msg if $sig(\sigma) = sig(msg)$ and if for parameter expressions $paramexp_i$ of msg and parameter values val_i of σ:

(i) if $paramexp_i$ is a wildcard or binding expression.
(ii) if $paramexp_i$ specifies a literal value, then it equals val_i.
(iii) if $paramexp_i$ specifies the value of a scenario variable var, then the value of var equals val_i.

Definition V.5 (Play-out graph (POG))**.** Given a specification $Spec = (Sc_G, \mathcal{O}_0)$, the play-out graph $POG(Spec) = (S, s_0, \Sigma_O, T)$ is a tuple where S is a set of states, s_0 is the start state, Σ_O is a set of message events among objects O of $\mathcal{O}_0$, and $T \subseteq S \times \Sigma_O \times S$ is a transition relation. A state $s \in S$ is a tuple $s = (\mathcal{O}, AS)$ of an object model $\mathcal{O}$ and a set of active scenarios $AS = \{as_0, ...as_n\}$. An active scenario $as = (sc, l, VarValues_{sc}) \in AS$ is a tuple where $sc \in Sc_G$ is a scenario, l is the current location, and $VarValues_{sc} : Var_{sc} \to Values$ is a mapping from scenario variables of sc to values. We write $\mathcal{O}(s)$ for the object model of s, and $AS(s)$ for the active scenarios of s. s_0 is defined as $s_0 = (\mathcal{O}_0, \emptyset)$. T is the smallest relation satisfying the following conditions:

1) *environment steps:* for all states $s = (\mathcal{O}, AS) \in S$ where AS contains no active scenarios with enabled requested transitions, it must be that $(s, \sigma_u, s') \in T$ for each environment event $\sigma_u \in \Sigma_{O_u}$.
2) *system steps:* for all states $s = (\mathcal{O}, AS) \in S$ where AS contains at least one active scenario with an enabled requested transitions, it must be that $(s, \sigma_c, s') \in T$ for each system event $\sigma_c \in \Sigma_{O_c}$ that matches a scenario message labeling an enabled requested transition of an active scenario, unless (*blocking:*) σ_c matches the message labeling *(a)* an enabled forbidden transition of any active scenario or *(b)* an initializing forbidden scenario message where $gurad_{init}$ evaluates to true w.r.t. $\mathcal{O}(s)$.
3) *change of object model:* if $(s, \sigma, s') \in T$ and $\sigma = (o_s, setA, (val), o_r)$ is a set-event for an attribute a and carries the value val as the parameter value, then $\mathcal{O}(s')$ is the same as $\mathcal{O}(s)$, except that $AttrValue(o_r, a) = val$.
4) *active scenario initialization and progress:* if $(s, \sigma, s') \in T$, then $AS(s')$ is as follows:
   - *active scenario initialization:* if σ is a message event that matches an initializing scenario message msg_{init} of a scenario sc where it labels a transition $(l_0, msg_{init}, guard_{init}, l')$ and where l' is not the end location of sc and $guard_{init}$ evaluates to true w.r.t. $\mathcal{O}(s)$, then $AS(s')$ contains an active scenario $as = (sc, l', VarValues'_{sc})$. $VarValues'_{sc}$ assigns all scenario variables with their default value, unless msg_{init} specifies parameter binding expressions for scenario variables, in which case these variables are assigned the respective parameter values carried by σ.
   - *active scenario progress:* if σ is a message event that matches a scenario message msg that labels an enabled transition $(l, msg, guard, l')$ of an active scenario $as = (sc, l, VarValues_{sc}) \in AS(s)$ then, if l' is not an end location of sc, $AS(s')$ contains the active scenario $as = (sc, l', VarValues'_{sc})$ where $VarValues'_{sc}$ assigns all scenario variables the same value as $VarValues_{sc}$, unless msg specifies parameter binding expressions for scenario variables, in which case these variables are assigned the respective parameter values carried by σ.
   - *event ignored by active scenario:* if σ is a message event that does not match any scenario message msg that labels an enabled transition of an active scenario $as = (sc, l, VarValues_{sc}) \in AS(s)$ then $as \in AS(s')$.
   - *active scenario termination:* Active scenarios reaching their end location will not be contained in $AS(s')$.

Figure 2 shows the POG for the oven specification and an initial object model where the controller's initial set-point temperature is 200. The POG is only shown in parts as it has ~700.000 states. This is due to the parameter

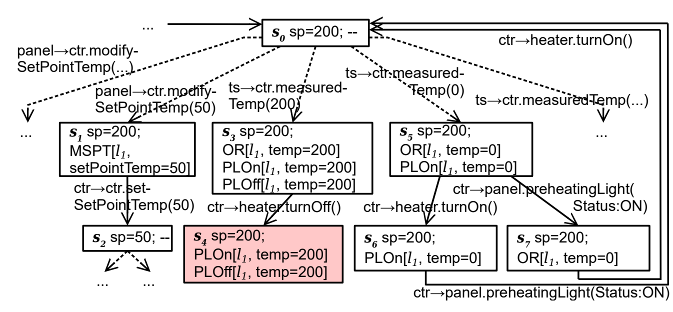

Figure 2. Part of the POG for the oven temperature regulation specification and an initial object model where the set-point temperature is set to 200.

ranges. s_0 alone has 502 successor states. The state labels show the stored set-point temperature sp=... as the only changing attribute of the object model. Active scenarios are encoded as MSPT = ModifySetPointTemperature, OR = OvenRegulation, PLOn/PLOff = PreheatLightOn/-Off, then in brackets follow the current locations of the active scenarios, as shown in Fig. 1, and the values of scenario variables. Transitions that are labeled with environment message events are dashed, solid otherwise. In s_0 there are 301 outgoing transitions for `ts->ctr.measuredTemp(`x`)` with $x \in [0..300]$ and 251 outgoing transitions for `ts->ctr.modifySetPointTemp(`y`)` with $y \in [50..300]$. In s_5, s_6, s_7, we see how the active scenarios OR and PLOn create a non-deterministic choice of whether to first turn on the heater and then the preheating light or vice versa. s_4 is a state where all transitions lead to a safety violation, due to the contradiction of whether to turn the preheating light on or off.

We define *play-out executability* as a notion of realizability.

Definition V.6 (Play-out executability)**.** A specification $Spec$ is *play-out executable* if $POG(Spec)$ contains

- no deadlock states (happens when at least one next system step is requested, but all of them are blocked, as in Fig. 2).
- no *safety-violating states*, i.e., state where at least one active scenario is in a safety-violating location. (Such violations can be caused by environment events).
- no cycles of only system events.

More general notions of realizability exist (cf. [14], [23], [7], [16]), which are defined via the existence of an implementation that satisfies a specification. Without laying out the details, play-out executability is a stronger condition; it implies the existence of an implementation, but if an implementation exists, the specification may not be play-out executable. For instance, it could be that a POG contains deadlock states, but they could be avoided by a system that makes smart choices in states where multiple system steps are possible. In this case, the specification is realizable but not play-out executable.

The POG is finite if the specification is *finite*, which means that the number of scenarios, objects, variables, and attributes are finite, and all domains are finite. In this case, checking play-out executability is decidable: Deadlock states and safety-violating states can be found by using search algorithms, e.g. DFS. Cycles of only system events can be found via nested-DFS or Tarjan's algorithm for finding strongly connected components (SCCs). The run-time complexities of these algorithms are linear w.r.t. to the size of the graph.

VI. Symbolic Play-Out and Realizability Checking

To counteract the problem of POG explosion due to message parameters, scenario variables, and object attributes, we turn to a symbolic interpretation of their values. By considering object attributes as symbolic inputs, we can even analyze specifications with multiple initial object models at once.

We define a *symbolic play-out graph* (SPOG) in Sect. VI-A, then describe how to check play-out executability based on this graph in Sect. VI-B, and describe approximation techniques in Sect. VI-C.

A. Symbolic Play-Out Graph

A *symbolic play-out graph* (SPOG) is a play-out graph where each *symbolic state* represents a set of concrete POG states, and a path represents a set of possible play-out runs. Each symbolic state may store *symbolic values* for variables and attributes and transitions are labeled with *symbolic message events*, which carry symbolic values for their parameters. Moreover, each symbolic state has a *path constraint* (PC), which is a formula over symbolic values; The PC constrains the possible concrete values of the symbolic values in a symbolic state, and it is formed as follows.

Each time that a parameterized event occurs in a symbolic state $\tilde{s}$, a new symbolic value is introduced for each parameter and, typically, some constraint for that new symbolic value is added, so that the PC of a target symbolic state $\tilde{s}'$ is the conjunction of that constraint and the PC of $\tilde{s}$. For example, consider the symbolic event `ts->ctr.measuredTemp(t0)`, where $t0$ is a symbolic value. Since a range `[0..300]` is defined for this parameter (see List. 1), the constraint $0 \leq t0 \leq 300$ is added to the PC of the target state.

Where the symbolic message event matches a scenario message with a binding expression for a parameter, as for example in the scenario OvenRegulation, the symbolic value carried by the symbolic message event is also assigned to the corresponding scenario variable, i.e., a *symbolic active scenario* stores `temp =` $t0$. Via set-messages, as in the scenario ModifySetPointTemperature, the new symbolic value can then also be assigned to an object attribute,

Moreover, it may be that, depending on the concrete values assumed for symbolic values, the active scenarios progress differently. For example, in the scenario OvenRegulation there is an alternative condition over the scenario variable `temp` and the object attribute `ctr.setPointTemp`, which means that the progress depends on the relationship of the symbolic values stored for that scenario variable and object attribute, i.e., $t0 \geq sp0$ or $t0 < sp0$, where $sp0$ is the symbolic value stored for `ctr.setPointTemp`. In such a case, a symbolic message event may lead to two or more target states with different path constraints and different progress of the active scenarios. We call this a *split*.

In this example the scenario OvenRegulation is not the only scenario that implies a split, but the scenarios PreheatLightOn/-Off also progress differently, depending on the relationship of the symbolic values carried by `ctr.setPointTemp` and their scenario variables, which are also bound to $t0$. Effectively, there is a split into three states:

1) $t0 > sp0$: OvenRegulation takes first alternative and PreheatLightOff progresses beyond the interrupt
2) $t0 < sp$: OvenRegulation takes second alternative and PreheatLightOn progresses beyond the interrupt
3) $t0 = sp0$: OvenRegulation takes first alternative and PreheatLightOn and PreheatLightOff progress beyond the interrupt (here we again have the deadlock)

Definition VI.1 (Symbolic object model)**.** A *symbolic object model* $\tilde{\mathcal{O}} = (O, O_c, O_u, CL, InstanceOf, SymAttrValues)$ is an object model as in Def. III.2, except $SymAttrValues$ maps attributes of objects to concrete or *symbolic values*: $SymAttrValues : O \times Attr \rightarrow Values \cup SymValues$.

Definition VI.2 (Symbolic specification)**.** A *symbolic specification* $\widetilde{Spec} = (Sc_G, \tilde{\mathcal{O}}_0)$ is a specification that refers to a symbolic initial object model $\tilde{\mathcal{O}}_0$ instead of a concrete one.

Definition VI.3 (Symbolic message event)**.** A *symbolic message event* $\tilde{\sigma} = (o_s, op, (sym_0, ...), o_r) \in O \times Op \times SymValues^* \times O$, is a message event as in Def. III.3 except that instead of carrying concrete values for parameters, a symbolic message event carries symbolic values $(sym_0, ...)$.

Definition VI.4 (Symbolic play-out graph (SPOG))**.** Given a symbolic specification $\widetilde{Spec} = (Sc_G, \tilde{\mathcal{O}}_0)$, with O being the objects of $\tilde{\mathcal{O}}_0$, the *symbolic play-out graph* $SPOG(\widetilde{Spec}) = (\tilde{S}, \tilde{s}_0, \tilde{\Sigma}_O, \tilde{T})$ is a tuple where $\tilde{S}$ is a set of symbolic states $\tilde{s}_0 \in \tilde{S}$ is a symbolic start state, $\tilde{\Sigma}_O$ is a set of symbolic message events among objects O. $\tilde{T} \subseteq \tilde{S} \times \tilde{\Sigma}_O \times \tilde{S}$ is a transition relation. A state $\tilde{s} \in \tilde{S}$ is a tuple $\tilde{s} = (\tilde{\mathcal{O}}, \widetilde{AS}, PC)$ that consists of a symbolic object model $\tilde{\mathcal{O}}$, a set of *symbolic active scenarios* $\widetilde{AS} = \{\widetilde{as}_0, ...\widetilde{as}_n\}$, and a *path constraint* PC. A symbolic active scenario $\widetilde{as} = (sc, l, SymVarValues_{sc}) \in \widetilde{AS}$ is an active scenario where instead of mapping scenario variables to concrete values, $SymVarValues_{sc} : Var_{sc} \rightarrow Values \cup SymValues$ is a mapping from scenario variables of sc to concrete or symbolic values. A path constraint PC is a formula of a decidable logic over symbolic values. We write $PC(\tilde{s})$ for the path constraint of a symbolic state $\tilde{s}$, $\tilde{\mathcal{O}}(\tilde{s})$ is the symbolic object model of $\tilde{s}$, and $\widetilde{AS}(\tilde{s})$ are the symbolic active scenarios of $\tilde{s}$. $\tilde{s}_0$ is defined as $\tilde{s} = (\tilde{\mathcal{O}}_0, \emptyset, true)$. $\tilde{T}$ is the smallest relation satisfying the following conditions:

1) *enabled environment events:* for all states $\tilde{s} = (\tilde{\mathcal{O}}, \widetilde{AS}, PC) \in \tilde{S}$ where $\widetilde{AS}$ contains no active scenarios with enabled requested transitions, for all signatures $sig \in Sig(O_u)$, the symbolic message event σ_u with (1) $sig(\tilde{\sigma}_u) = sig$ and (2) σ_u carries new symbolic values $(p_0, ...)$ for each parameter, is an *enabled event* in $\tilde{s}$ (If an event is enabled in $\tilde{s}$, it means that there is a set of zero or more transitions $(\tilde{s}, \tilde{\sigma}_u, \tilde{s}')$ in $\tilde{T}$, as detailed below.)
2) *enabled system events:* for all states $\tilde{s} = (\tilde{\mathcal{O}}, \widetilde{AS}, PC) \in \tilde{S}$ where $\widetilde{AS}$ contains at least one active scenarios with at least one enabled requested transitions, and for each scenario message msg that labels such a transition, the symbolic message event $\tilde{\sigma}_c$ with (1) $sig(\tilde{\sigma}_c) = sig(msg)$ and (2) σ_c carries new symbolic values $(p_0, ...)$ for each parameter, is *enabled* in $\tilde{s}$.
3) *change of object model:* if $(\tilde{s}, \tilde{\sigma}, \tilde{s}') \in \tilde{T}$ and $\tilde{\sigma} = (o_s, setA, (symVal), o_r)$ is a set-event for an attribute a and carries the value $symVal$ as the parameter value, then $\tilde{\mathcal{O}}(s')$ is the same as $\tilde{\mathcal{O}}(s)$, except that its $SymAttrValue(o_r, a) = symVal$.
4) *split / active scenarios initialization and progress:* if $\tilde{\sigma}$ is enabled in $\tilde{s}$ (as in 1) and 2) above), then $\tilde{T}$ contains the transitions defined as follows (we define conditions on which we have different combinations of active scenario progresses and initializations, which then become part of the target state's PC): Let $\Delta_1^{\tilde{\sigma}}, ..., \Delta_m^{\tilde{\sigma}}$ be the initializing transitions of scenarios in Sc_G labeled with a scenario message msg where $sig(msg) = sig(\tilde{\sigma})$. Furthermore, let $\Delta_{m+1}^{\tilde{\sigma}}, ..., \Delta_n^{\tilde{\sigma}}$ be the enabled transitions of the active scenarios in $\tilde{s}$ labeled with a scenario message msg where $sig(msg) = sig(\tilde{\sigma})$. If $\delta = (l_s, msg, guard, l_t)$ is a transition in $\Delta_i^{\tilde{\sigma}}$, then $pr_\delta^{\tilde{\sigma}}$ is the *progress condition for* δ *on* $\tilde{\sigma}$; it consists of $guard$ conjoined with $p_j = val$ for each parameter j of msg where msg specifies a literal value val and $p_k = SymVarValues(var)$ for each parameter k where msg specifies a variable value var. (*blocking:*) If $\tilde{\sigma}$ is a system event and l_t is a safety-violating state, then $pr_\delta^{\tilde{\sigma}}$ is $false$ (as we will see below, this inhibits splitting into safety-violations). We define $Pr_i^{\tilde{\sigma}}$ as the set of all progress conditions of transitions in $\Delta_i^{\tilde{\sigma}}$, plus the condition that results from the negation of their disjunction, i.e., $Pr_i^{\tilde{\sigma}}$ is the set of all conditions where *one* active scenario is activated / progresses differently, or is not activated / does not progress at all, on $\tilde{\sigma}$. Let then $\Phi^{\tilde{\sigma}}$ be the set of all satisfiable conjunctions of conditions, one from each $Pr_i^{\tilde{\sigma}}$, i.e., the conditions on which we have *a different combination of active scenario progresses and initializations* on $\tilde{\sigma}$. For each $\varphi^{\tilde{\sigma}} \in \Phi^{\tilde{\sigma}}$, if $PC(\tilde{s}) \wedge \varphi^{\tilde{\sigma}}$ is satisfiable, i.e., $\varphi^{\tilde{\sigma}}$ represents a possible progress under the current path constraint, we then have a transition $(\tilde{s}, \tilde{\sigma}, \tilde{s}'^{\varphi}) \in \tilde{T}$ where $PC(\tilde{s}'^{\varphi}) = PC(\tilde{s}) \wedge \varphi^{\tilde{\sigma}}$ and $\widetilde{AS}(\tilde{s}'^{\varphi})$ is defined as follows:
   - *active scenario initialization:* if $\delta = (l_0, msg_{init}, guard_{init}, l')$ is a transition in scenario sc, l' is not the end location of sc, $sig(\tilde{\sigma}) = sig(msg_{init})$, and $pr_\delta^{\tilde{\sigma}}$ is (syntactically) part of φ, then $\widetilde{AS}(\tilde{s}'^{\varphi})$ contains a symbolic active scenario $\widetilde{as} = (sc, l', SymVarValues_{sc})$. $SymVarValues_{sc}$ assigns all scenario variables with their default value, except where msg specifies a parameter binding expression for a scenario variable, this variable is assigned the corresponding symbolic value carried by $\tilde{\sigma}$.
   - *active scenario progress:* if $\delta = (l, msg, guard, l')$ is an enabled transition in the active symbolic scenario $\widetilde{as} = (sc, l, SymVarValues_{sc})$, l' is not the end location, $sig(\tilde{\sigma}) = sig(msg)$, and $pr_\delta^{\tilde{\sigma}}$ is (syntactically) part of φ, then $\widetilde{AS}(\tilde{s}'^{\varphi})$ contains a symbolic active scenario $\widetilde{as}' = (sc, l', SymVarValues'_{sc})$ where $SymVarValues'_{sc}$ has the same value assignments as $SymVarValues_{sc}$, except that where msg specifies a parameter binding expression for a scenario variable, this variable is assigned the corresponding symbolic value carried by $\tilde{\sigma}$.
   - *event ignored by active scenario:* if an active scenario $\widetilde{as} = (sc, l, SymVarValues_{sc})$ has no enabled transitions labeled with a message msg such that $sig(\tilde{\sigma}) = sig(msg)$, then $\widetilde{as} \in \widetilde{AS}(\tilde{s}')$.
   - *(active scenario termination)* Symbolic active scenarios reaching their end location will not be in $\widetilde{AS}(\tilde{s}')$.

Figure 3 shows the SPOG for the oven temperature regulation specification. It contains two extensions that, for brevity, we do not include in our formal definitions: (1) constraints that result from message parameter ranges, and (2) a constraint for the initial range of the controller's set-point temperature, so that the PC for the initial state is not $true$ but $50 \leq n0 \leq 300$. Note that the PCs deliberately show redundant subformulas as they result from the definition of the φs in Def. VI.4.

Compared with the POG (Fig. 2) we see that the initial state now only has *four* outgoing transitions: one represents the setting of a new set-point temperature. The other three transitions (leading to s_3, s_5, s_9) represent the split for `ts→ctr.measuredTemp(...)` events as explained above. The PCs in s_3, s_5, s_9 are the PC of s_0 conjoined with the different satisfiable combinations of the guard conditions of the initializing transitions for `ts→ctr.measuredTemp(...)` events in all specification scenarios. The symbolic active scenarios are the scenarios that are initialized under the respective conditions.

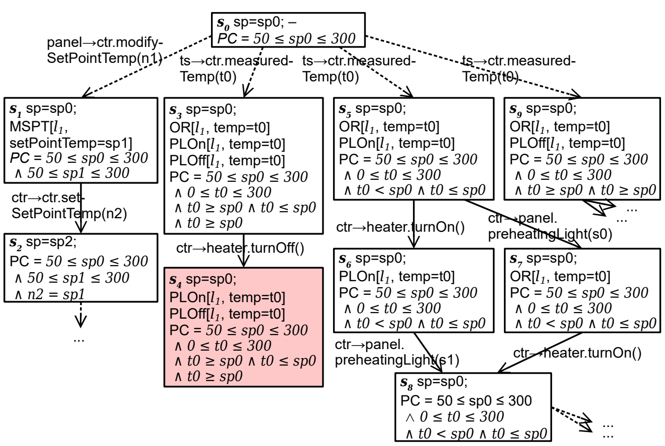

Figure 3. Part of the SPOG for the oven temperature regulation specification.

The SPOG as defined in Def. VI.4 may not be finite, since message events repeatedly introduce new symbolic values. Thus the resulting PCs are likely to be different syntactically even if the symbolic states represent the same set of concrete states. From s_4 in Fig. 3 for example we have two paths that join again in s_8, since the new symbolic values $s0$, $s1$ are not stored. But the SPOG is still infinite.

In order to be able to check play-out executability on the SPOG, we merge symbolic states when they are *semantically equivalent*, i.e., they represent the same set of concrete states. The state equivalence is defined as follows.

Definition VI.5 (symbolic state equivalence, SPOG_{EQ})**.** Let s and s' be two symbolic states that are equal up to the names of symbolic values and their PC; we also say they are *structurally equal*. Let $\alpha_0, ..., \alpha_n$ / $\alpha'_0, ..., \alpha'_n$ be the symbolic values bound to object attributes or scenario variables of s / s', with α_i bound to the same scenario variable or object attribute as α'_i, and let $\beta_0, ..., \beta_{m_1}$ / $\beta'_0, ..., \beta'_{m_2}$ be the other symbolic values appearing in $PC(s)$ / $PC(s')$. If name collisions between $\alpha_i/\alpha'_i/\beta_i/\beta'_i$ occur, they are resolved by consistent renaming. s and s' are *equivalent* if the PCs are equivalent under the assumption that the symbolic values for the same scenario variables and object attributes take the same values, but without any constraints on the other symbolic values, i.e., if the following formula is satisfiable:

$$\forall \alpha_0, ..., \alpha_n, \alpha'_0, ..., \alpha'_n : (\alpha_0 = \alpha'_0 \land ... \land \alpha_n = \alpha'_n)$$
$$\Rightarrow ((\exists \beta_0, ..., \beta_{m_1} : PC(s)) \Leftrightarrow (\exists \beta'_0, ..., \beta'_{m_2} : PC(s')))$$

$SPOG_{EQ}(\widetilde{Spec})$ is a symbolic play-out graph for a specification $\widetilde{Spec}$ where two states are merged if they are equivalent.

Example: states s_0 and s_2 in Fig. 3 are equivalent because

$$\forall sp0, sp2 : (sp0 = sp2)$$
$$\Rightarrow ((50 \leq sp0 \leq 300) \Leftrightarrow (\exists sp1, sp0_2 : 50 \leq sp0_2 \leq 300$$
$$\land 50 \leq sp1 \leq 300 \land sp1 = sp2))$$

is satisfiable. ($sp0_2$ is renamed from $sp0$ for state $s2$ due to a name collision.) Also s_8 is equal to s_0 (and s_2). In fact, the SPOG_{EQ} for the example specification only has 10 states: all states shown in Fig. 3 minus s_2 and s_8 (which are merged into s_0), plus the two successors of s_9, which again have s_0 as their successor. This is a drastic reduction compared to the ~700.000 states for one (explicit) POG.

We establish the following properties for a SPOG_{EQ}:

Lemma VI.1 (Equivalence of SPOG_{EQ} and POGs)**.** *Let $\widetilde{Spec} = (Sc_G, \tilde{\mathcal{O}}_0)$ be a symbolic specification that is finite (as in Sect. III). This means that also $\tilde{\mathcal{O}}_0$ represents finitely many explicit object models $\mathcal{O}_0^1, ..., \mathcal{O}_0^n$. Then $SPOG_{EQ}(\widetilde{Spec})$ represents $POG(Spec^1), ..., POG(Spec^n)$, $Spec^i = (Sc_G, \mathcal{O}_0^i)$, in the sense that:*

1) *each state reachable from the initial state in any $POG(Spec^i)$ is represented by at least one symbolic state in $SPOG_{EQ}(\widetilde{Spec})$ that is reachable from the initial symbolic state.*
2) *Each path in a $POG(Spec^i)$ is represented by a path in $SPOG_{EQ}(\widetilde{Spec})$ and each path in $SPOG_{EQ}(\widetilde{Spec})$ represents to a least one path in at least one $POG(Spec^i)$.*

1) holds since the initial symbolic state of $SPOG_{EQ}(\widetilde{Spec})$ represents all initial states of $POG(Spec^1), ..., POG(Spec^n)$, and because the split condition ensures that if a concrete state is represented by a symbolic state, each successor or the concrete state is represented by a successor of the symbolic state, i.e., no states are lost. 2) holds since a transition between two symbolic states $\tilde{s}$ and $\tilde{s}'$ in $SPOG_{EQ}(\widetilde{Spec})$ exists if and only if for all concrete states represented by $\tilde{s}$, some $POG(Spec^i)$ has a corresponding outgoing transition to a concrete state represented by $\tilde{s}'$.

B. Checking play-out executability

Play-out executability checking can be done on a SPOG_{EQ} using the same search and cycle detection algorithms as explained for the (explicit state) POG in Sect. III.

Correctness/Completeness: From Lemma VI.1 follows that deadlock resp. safety violating states and cycles with only system events exist in the SPOG_{EQ} if and only if they exist in one of the POGs that it represents.

Termination: the checking terminates because the SPOG_{EQ} is finite for finite specifications, which follows from the definition of symbolic states in Def. VI.4, and Def. VI.5.

C. Approximation: state subsumption

Although the SPOG_{EQ} for the above example is very small compared to the concrete POGs, we may not see such drastic reductions in all cases. Even worse, the SPOG_{EQ} could have more symbolic states than there are concrete states. Suppose that there are n structurally equal concrete states in a POG, these could be represented by up to 2^n different symbolic state in a SPOG_{EQ} (each representing a different subset of states).

If we extend our oven example by the assumption that temperature measurements only increase or decrease in steps of one degree, we observe the case that after the first temperature measurement, the temperature is in [0..300], as indicated by the ranges (see List. 1). After the next measurement it is either in [1..300] or [0..299], and so on. This means that a POG could have a set of 301 structurally equal states that in the SPOG_{EQ} would be represented by 301*302/2 different symbolic states (because there are $n*(n+1)/2$ different sub-ranges of [0..n]).

To counteract this problem, we turn to a different condition for merging symbolic states, namely merging a newly explored symbolic state with an existing one if the existing one represents a superset of the states represented by the new one, i.e., if the new symbolic state is *subsumed* by the existing one.

The *subsumption condition* is similar to the equivalence condition in Def. VI.5, only that the equivalence operator in the formula is replaced by an implication ($\Rightarrow$, i.e., left to right), which makes it an antisymmetric relation where $\tilde{s}'$ subsumes $\tilde{s}$. $SPOG_{SUM}(\widetilde{Spec})$ is a symbolic play-out graph for a specification $\widetilde{Spec}$ where, during exploration, a newly explored symbolic state $\tilde{s}_{new}$ is merged into an existing symbolic state $\tilde{s}_{old}$ if $\tilde{s}_{old}$ subsumes $\tilde{s}_{new}$. With different orders of exploration of states there may be different graphs for the same specification.

The play-out executability checking based on a SPOG_{SUM} is performed as before, by searching for deadlock / safety violating states and checking for cycles of only system steps. It terminates for finite specifications, since SPOG_{SUM} is finite in this case. Also, the checking is complete, but to achieve correctness, we require an additional step, as explained below.

Lemma VI.1.1) holds for SPOG_{SUM}. However, Lemma VI.1.2) holds only in one direction, namely each path in a $POG(Spec^i)$ is represented by a path in $SPOG_{SUM}(\widetilde{Spec})$, but there may be paths in $SPOG_{SUM}(\widetilde{Spec})$ which correspond to no path in any $POG(Spec^i)$. This is sufficient to argue that if there is a reachable deadlock or safety-violating state in $SPOG_{SUM}(\widetilde{Spec})$, it must also exist in some $POG(Spec^i)$ and vice versa. I.e., detection of deadlock / safety violating states is correct and complete.

If a $POG(Spec^i)$ contains a cycle of only system steps, it will also exist in $SPOG_{SUM}(\widetilde{Spec})$. This establishes completeness for the detection of such cycles, but not correctness, since if $SPOG_{SUM}(\widetilde{Spec})$ contains a cycle of only system steps, we cannot imply that it occurs in any $POG(Spec^i)$. Thus, if a cycle of only system steps is found in $SPOG_{SUM}(\widetilde{Spec})$, we need to check subsequently whether there exists a sequence of concrete message events that exercises such a cycle. The found symbolic cycle can help in guiding such a check, i.e., it can help in selecting the right message events and concrete parameter values. Similarly, we require such a check if we want to obtain concrete paths to deadlock / safety-violating states.

VII. Implementation and Evaluation

We implemented the symbolic execution and realizability checking procedures in ScenarioTools, which is an Eclipse-based tool suite for modeling and analyzing SML specifications with rich scenario language features, such as dynamic polymorphic bindings [17], assumption scenarios, and dynamic component topologies [3]. The execution and analysis in ScenarioTools is based on an execution engine that interprets the scenarios and is able to build play-out graphs. We extended this engine to build symbolic play-out graphs, which can be used for realizability checking as explained above, or for interactive step-by-step simulation. For constraint-solving, we integrated the Z3 SMT solver [24].

In the following, we present evaluation results based on a range of examples: (1) the above oven temperature regulation example and (2) an extended variant that includes humidity regulation, an on/off attribute, and it includes assumptions that measured temperatures will increase/decrease only in one degree steps. The third example (3) is the specification of a tunnel controller that coordinates the passage of cars in a narrow tunnel. See the appendix of [26] for details. All examples are tested with different ranges for parameters and different checking techniques: based on explicit-state POGs (EXP), SPOG_{EQ}s (EQ), and SPOG_{SUM}s (SUM). We time-out the measurements if no result is reported after 600 seconds.

Table I shows the results of checking the oven temperature regulation specification as in List. 1, including its specification flaw, with different parameter ranges. The left column shows the ranges for the set-point temperature (**sp**) (includes initial attribute value as well as the event parameter) and the measured temperature (**mt**). For the explicit state cases, the initial set-point temperature is 0. The example is a best-case example—the size of the explicit-state POG almost doubles when parameter ranges increase by 10, while the sizes of the symbolic play-out graphs remain constant.

For the extended oven temperature regulation (Table II), we have an additional relative humidity (**rh**) parameter. The state graphs are larger. Most interestingly, the symbolic checking with state equivalence does not scale and times out even before the explicit-state checking. This has to do with an explosions of the symbolic states as explained above. The times for symbolic checking with state subsumption remains constant.

Table I
OVEN REGULATION EXAMPLE (EXP/EQ/SUB)

ranges (sp mt)	#states	#transitions	time(s)
[0..10] [0..10]	484/10/10	1056/16/16	6/1/1
[0..20] [0..20]	1764/10/10	3906/16/16	25/1/1
[0..30] [0..30]	3844/10/10	8556/16/16	63/1/1
[0..40] [0..40]	6724/10/10	15006/16/16	121/1/1
[0..50] [0..50]	10404/10/10	23256/16/16	205/1/1
[0..60] [0..60]	14884/10/10	33306/16/16	312/1/1
[0..70] [0..70]	20164/10/10	45156/16/16	479/1/1
[0..300] [0..300]	-/10/10	-/16/16	>600/1/1

Table II
EXTENDED OVEN REGULATION EXAMPLE (EXP/EQ/SUB)

ranges (mt sp rh)	#states	#transitions	time(s)
[0..2] [0..2] [0..1]	513/453/184	2496/1509/599	39/117/33
[0..4] [0..2] [0..1]	944/656/172	6577/2240/559	106/225/30
[0..6] [0..2] [0..1]	1514/846/172	13435/2898/559	193/397/31
[0..8] [0..2] [0..1]	2208/-/172	23430/-/559	410/>600/31
[0..10] [0..2] [0..1]	3014/-/172	38455/-/559	565/>600/32
[0..20] [0..2] [0..1]	-/-/172	-/-/559	>600/>600/32
[0..40] [0..2] [0..1]	-/-/172	-/-/559	>600/>600/31
[0..80] [0..2] [0..1]	-/-/172	-/-/559	>600/>600/33
[0..300] [0..300] [0..100]	-/-/200	-/-/631	>600/>600/35

The tunnel controller coordinates the safe passage of cars that can enter a narrow tunnel from two sides. If there are cars in the tunnel, cars coming from the other direction must wait until all cars have left the tunnel. The specification consists of 9 scenarios and the number of cars in the tunnel is represented by an integer parameter. Table III shows the results for checking play-out executability. We consider different maximum numbers of cars that can be in the tunnel. We see that the explicit-state POGs grow with the number of cars in the tunnel. Again, the symbolic checking with state equivalence shows even larger graphs than the explicit-state POGs. The SPOGs with state subsumption again have a constant size.

Table III
TUNNEL CONTROLLER (EXP/EQ/SUB)

Max #cars	#states	#transitions	time(s)
1	74/258/44	264/847/168	2/36/4
2	102/578/44	362/1817/168	2/125/4
10	326/-/44	1146/-/168	9/>600/4
100	2846/-/44	9966/-/168	356/>600/4

The examples show that the symbolic checking with state subsumption scales well with increasing parameter ranges, while symbolic checking with state equivalence only scales in some cases and can even be worse than explicit-state analysis. Evaluating further examples is necessary to study the effects in more detail and is planned for future work.

VIII. Related Work

Wang et al. describe the symbolic play-out of LSC specifications by a mapping to a constraint logic program [27]. They support the symbolic interpretation of scenarios variables and lifeline bindings with unboundedly many components. We do not consider symbolic values for role/lifeline bindings, whereas they do not consider changing object properties.

Roychoudhury et al. describe the analysis of Symbolic Message Sequence Charts (SMSCs) [28], which can also represent the interaction between unboundedly many components. They consider that MSCs can be executed sequentially as specified by a High-level Message Sequence Chart (HMSC). The interplay of multiple concurrently active and progressing scenarios as in LSCs/SML is not considered.

Harel et al. describe a symbolic analysis and composition approach of Behavioral Programs (BPs) [30], which share some of the core concepts of LSCs/SML [31]. In their work not the entire program is executed symbolically, but rather a stepwise compositional verification and composition process is described, which is supported by an SMT-Solver.

Cimatti et al. describe a feasibility checking method for Message Sequence Charts (MSCs) of hybrid systems based on SMT solving and k-induction [29]. They, however, only check single MSCs, whereas our approach considers the interplay of multiple scenarios.

Zurowska and Dingel describe the symbolic execution of UML-RT state machines [22] for analyzing invariants and reachability properties. It is based on mapping the UML-RT state machines to functional finite state machines, for which a symbolic execution procedure is described. Similar to our approach, also subsumption checking of states is suggested for backtracking in the symbolic state exploration.

The symbolic execution of state machines is also described by Thums et al. [32]. Rapin describes the symbolic analysis of Input-Output Transition Systems [20].

IX. Conclusion

We described a technique for the symbolic execution of scenario-based specifications in SML that interprets integer message parameters and variables symbolically and show how it can be used for symbolic realizability checking. The evaluations show that in combination with state subsumption, the technique scales well with growing parameter ranges.

The approach could be integrated into an existing tool that is capable of building play-out graphs for SML specifications with rich language features, thus achieving better scalability w.r.t. integer parameters and attributes over large ranges while not impairing support for other language features.

In future work, we will consider using the technique for the analysis of timed specifications and study further approximation techniques (some ideas appear in [26]). Another goal is extending the work for controller synthesis and checking other notions of realizability. Also, this approach can be combined with previous work [33] for synthesizing test cases.

Acknowledgment

Funded by the German Israeli Foundation for Research and Development (GIF), grant No. 1258, and the Deutsche Forschungsgemeinschaft (DFG), project EFFISYNTH.

References

[1] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," in *Formal Methods in System Design*, vol. 19, 2001, pp. 45–80.

[2] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[3] J. Greenyer, D. Gritzner, G. Katz, and A. Marron, "Scenario-based modeling and synthesis for reactive systems with dynamic system structure in scenariotools," in *Proceedings of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016)*, J. de Lara, P. J. Clarke, and M. Sabetzadeh, Eds., vol. 1725. CEUR, 2016, pp. 16–32.

[4] D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach," *SoSyM*, vol. 2, pp. 82–107, 2003.

[5] C. Brenner, J. Greenyer, and V. Panzica La Manna, "The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions," in *Proc. 12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, vol. 58. EASST, 2013.

[6] D. Harel and H. Kugler, "Synthesizing state-based object systems from LSC specifications," *Foundations of Computer Science*, vol. 13:1, pp. 5–51, 2002.

[7] Y. Bontemps and P. Heymans, "From Live Sequence Charts to State Machines and Back: A Guided Tour," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 999–1014, 2005.

[8] D. Harel, H. Kugler, and A. Pnueli, "Synthesis revisited: Generating statechart models from scenario-based requirements," in *Formal Methods in Software and Systems Modeling*, vol. 3393. Springer, 2005, pp. 309–324.

[9] J. Sun and J. S. Dong, *Synthesis of Distributed Processes from Scenario-Based Specifications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 415–431. [Online]. Available: https://doi.org/10.1007/11526841_28

[10] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of Partial Behavior Models from Properties and Scenarios," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 384–406, 2009.

[11] K. G. Larsen, S. Li, B. Nielsen, and S. Pusinskas, "Scenario-based analysis and synthesis of real-time systems using Uppaal," in *Proc 13th Conf. on Design, Automation, and Test in Europe (DATE'10)*, March 2010.

[12] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi, "Incrementally synthesizing controllers from scenario-based product line specifications," in *Proc. 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2013*, 2013.

[13] C. Brenner, J. Greenyer, and W. Schäfer, "On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications," in *Fundamental Approaches to Software Engineering (FASE 2015)*, ser. Lecture Notes in Computer Science, A. Egyed and I. Schaefer, Eds. Springer, 2015, vol. 9033, pp. 51–65.

[14] M. Abadi, L. Lamport, and P. Wolper, "Realizable and unrealizable specifications of reactive systems," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, G. Ausiello, M. Dezani-Ciancaglini, and S. Della Rocca, Eds. Springer Berlin Heidelberg, 1989, vol. 372, pp. 1–17. [Online]. Available: http://dx.doi.org/10.1007/BFb0035748

[15] M. Cordy, J. Greenyer, P. Heymans, and A. M. Sharifloo, "Efficient consistency checking of scenario-based product line specifications," in *Proceedings of the 20th International Requirements Engineering Conference (RE 2012)*, M. P. E. Heimdahl and P. Sawyer, Eds. IEEE, 2012, pp. 161–170. [Online]. Available: http://jgreen.de/wp-content/documents/2012/ConsistencyCheckingScenarioBasedProductLineSpecifications.pdf

[16] S. Maoz and Y. Sa'ar, "Assume-guarantee scenarios: Semantics and synthesis," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 335–351. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33666-9_22

[17] S. Maoz, "Polymorphic scenario-based specification models: semantics and applications," *Software and System Modeling*, vol. 11, no. 3, pp. 327–345, 2012.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[19] L. a. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, 1976. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702368

[20] N. Rapin, "Symbolic execution based model checking of open systems with unbounded variables," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5668 LNCS, pp. 137–152, 2009.

[21] C. S. Pasareanu and D. Balasubramanian, "Statechart analysis with symbolic pathfinder," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 772–772.

[22] K. Zurowska and J. Dingel, "Symbolic execution of UML-RT State Machines," *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, p. 1292, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2245276.2231981

[23] J. Greenyer and E. Kindler, "Compositional synthesis of controllers from scenario-based assume-guarantee specifications," in *Model-Driven Engineering Languages and Systems: Proceedings of the ACM/IEEE 16th International Conference (MODELS 2013)*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds. Springer Berlin Heidelberg, 2013, vol. 8107, pp. 774–789.

[24] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[25] O. M. G. (OMG), "OMG meta object facility (MOF) core specification," November 2016, OMG document `formal/2016-11-01`. [Online]. Available: http://www.omg.org/spec/MOF/2.5.1/PDF

[26] T. Gutjahr, "Symbolische Ausführung für die Analyse von szenariobasierten Spezifikationen," Master's Thesis, Leibniz Universität Hannover, 2016. [Online]. Available: http://jgreen.de/wp-content/documents/msc-theses/2016/Gutjahr2016.pdf

[27] T. Wang, A. Roychoudhury, R. H. C. Yap, and S. C. Choudhary, "Symbolic execution of behavioral requirements," in *Practical Aspects of Declarative Languages: 6th International Symposium, PADL 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 178–192. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24836-1_13

[28] A. Roychoudhury, A. Goel, and B. Sengupta, "Symbolic message sequence charts," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 2, pp. 12:1–12:44, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2089116.2089122

[29] A. Cimatti, S. Mover, and S. Tonetta, "Proving and explaining the unfeasibility of message sequence charts for hybrid systems," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, TX: FMCAD Inc, 2011, pp. 54–62. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157654.2157666

[30] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss, "On composing and proving the correctness of reactive behavior," *2013 Proceedings of the International Conference on Embedded Software, EMSOFT 2013*, 2013.

[31] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Comm. ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[32] A. Thums, G. Schellhorn, F. Ortmeier, and W. Reif, *Interactive Verification of Statecharts*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 355–373. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27863-4_20

[33] V. P. L. Manna, I. Segall, and J. Greenyer, "Synthesizing tests for combinatorial coverage of modal scenario specifications," in *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, Sept 2015, pp. 126–135.