Bounded Invariant Checking for Stateflow*

Predrag Filipovikj
Scania CV AB
Södertälje, Sweden
KTH Royal Institute of Technology
Stockholm, Sweden
predrag.filipovikj@scania.com

Gustav Ung
Scania CV AB
Södertälje, Sweden
gustav.ung@scania.com

Dilian Gurov
KTH Royal Institute of Technology
Stockholm, Sweden
dilian@kth.se

Mattias Nyberg
Scania CV AB
Södertälje, Sweden
KTH Royal Institute of Technology
Stockholm, Sweden
mattias.nyberg@scania.com

Stateflow models are complex software models, often used as part of industrial safety-critical software solutions designed with Matlab Simulink. Being part of safety-critical solutions, these models require the application of rigorous verification techniques for assuring their correctness. In this paper, we propose a refutation-based formal verification approach for analyzing Stateflow models against invariant properties, based on bounded model checking (BMC). The crux of our technique is: i) a representation of the state space of Stateflow models as a symbolic transition system (STS) over the symbolic configurations of the model, and ii) application of incremental BMC, to generate verification results after each unrolling of the next-state relation of the transition system. To this end, we develop a symbolic structural operational semantics (SSOS) for Stateflow, starting from an existing structural operational semantics (SOS), and show the preservation of invariant properties between the two. We define bounded invariant checking for STS over symbolic configurations as a satisfiability problem. We develop an automated procedure for generating the initial and next-state predicates of the STS, and a prototype implementation of the technique in the form of a tool utilising standard, off-the-shelf satisfiability solvers. Finally, we present preliminary performance results by applying our tool on an illustrative example and two industrial models.

1 Introduction

Stateflow [30] is a proprietary graphical modelling language developed and maintained by Mathworks. It is an extension of a formalism for modelling complex systems through hierarchical state machines called Statecharts [21]. The rich graphical formalism and the variety of supporting tools in the Matlab Simulink environment enable the development of highly complex software models, which in many instances are classified as *safety-critical*. The correctness of safety critical systems is regulated by domain-specific safety standards (e.g., ISO26262 [22] in the automotive domain), which require correct operation of such systems at all times with strongly regulated error margins.

One way of enabling a high level of quality-assurance for safety-critical systems is to employ rigorous mathematics-based verification methods popularly known as *formal verification techniques*. The main challenges of applying formal techniques for verification of Stateflow models stem from two main factors: i) tractability of the verification process due to the high-complexity of the Stateflow models, and ii)

*Work partially funded by the FFI Programme of the Swedish Governmental Agency for Innovation Systems (VINNOVA) as the AVerT2 project 2021-02519.

M. Luckcuck and M. Farrell; M. Autili, L. Berardinelli,
A. Bucaioni and C. Pompilio (Eds.):
Formal Methods for Autonomous Systems and
Automated and verifiable Software sYstem DEvelopment
EPTCS 371, 2022, pp. 38–52, doi:10.4204/EPTCS.371.3

the lack of formal semantics for the Stateflow language publicly disclosed by Mathworks. The problem of formal verification of Stateflow models has been addressed in a number of research endeavours, which have focused either on defining a *de-facto* formal semantics for the language [6, 17, 19, 20], or proposing a model-to-model transformation schemes for converting Stateflow models into some formalism of interest [2, 23, 31]. The former group of approaches often resort to exhaustive verification techniques which are likely not to scale for industrial-size models. The main limitation of the latter group of approaches is that their analysis models are not provably correct against the original Stateflow model. At present, industry relies mainly on the proprietary SLDV tool [18] by Mathworks for the formal verification of their models. Although the tool provides a completely automated workflow for refutation-based and induction-based verification [13], as it is proprietary, it is neither open-source nor transparent about its exact formal underpinnings and internal workings. On top of the information scarcity, the SLDV tool is distributed under a license that explicitly forbids benchmarking or any other form of direct comparison with another approach or tool, be it commercial or of purely academic nature.

In this work, we are tackling the aforementioned challenges for formal analysis of Stateflow models by presenting a technique that applies *bounded model checking* (BMC) [5] over *symbolic executions* [25] of Stateflow models. We adopt BMC as the underlying technique for verification for two main reasons: first, to leverage the power of SAT/SMT-based model checking [4], and second, to alleviate the state-space explosion by incrementally exploring all system executions of bounded length [4], until the problem becomes intractable or a property violation is detected. In this paper, we focus on checking *invariant* properties, which are state properties that hold in all reachable states of a given program. Even though invariant properties represent just one class of properties, based on our previous and current experiences in collaboration with industrial partners, it is often considered to be the most important one for safety-critical systems.

Contributions Our verification technique consists of the following ingredients. First, we derive a set of symbolic structural operational semantics rules (SSOS). The SSOS rules are obtained by uniformly translating into symbolic counterparts the rules of an already existing third-party SOS for Stateflow [19]. We build on top of this particular set of SOS rules, because it is the only available operational semantics for Stateflow that is suitable for our needs, and because the correctness of the rules has already been validated against the *simulation semantics* of Stateflow (see [19]). The SSOS is needed for deriving a *symbolic transition system* (STS) at a suitably high level of granularity of the execution steps (which we choose to be the level of Stateflow program statements), abstracting from the intricate many-layered transitions of the original SOS. As our second contribution, we present two theorems that show that the SOS and SSOS *simulate* each other. This result is crucial for the correctness of our technique. Our third contribution is a translation, using the SSOS, of Stateflow programs into STS over symbolic configurations, and the encoding of this STS and the given invariant property into a set of constraints in the SMT-LIB format [3]. This set of constraints can then be used as input to most of the modern SMT solvers. In our work, we use the Z3 SMT solver [10] from Microsoft Research. Finally, as our fourth and final contribution we give preliminary evidence for the practical usefulness of our approach by applying it on an illustrative Stateflow model. Even though initially we planned to compare our approach against the SLDV tool, in the end it was not possible due to the strict licensing constraints imposed by Mathworks.

Related work A significant portion of existing approaches for verification of Stateflow rely on different transformation rules and schemes for the basic Stateflow modeling constructs into some existing formal framework, as presented in [9, 23, 26, 31]. The main limitation of these approaches is the lack of means

for proving the correctness of their transformation schemes, which in turn hinders the provability of the correctness of their formal models.

Another class of approaches includes the ones that build on top of the existing Stateflow semantics. Miyazawa et al. [27] provide a formalization of Stateflow in a refinement language called Circus. The authors provide semantics characteristic to the specific refinement language, whereas in our case the semantics are defined in generalized SOS-style. The CoCoSim framework [6, 7] is perhaps one of the most comprehensive bodies of work on the topic of formal verification of Simulink/Stateflow models. The framework builds on top of a denotational semantics for the Stateflow language [17]. For analysis, the framework compiles the Stateflow models into Lustre models, which is the core difference to our work as we start from an SOS style semantics of the Stateflow language. Finally, there are number of approaches that treat Stateflow models as either hybrid or stochastic models, and apply corresponding modelling and analysis techniques and tools for verification [1, 12, 24, 32]. The core difference to our approach is that these approaches treat the Stateflow model as either linear hybrid or Bayesian models, and resort to simulation-based techniques for the formal analysis of the model.

Structure Our paper is organised as follows. In Section 2, we outline the required background concepts that we use throughout the paper. Next, in Section 3, we present the SSOS for Stateflow programs, followed by the characterization of the relationship between the concrete and symbolic semantics in Section 4. Then, in Section 5, we show how an STS over symbolic configurations can be constructed using the SSOS rules (Section 5.1), followed by an informal encoding procedure into an SMT-LIB script (Section 5.2). Next, we show a preliminary evaluation of our approach, based on the running example (Section 6). Finally, in Section 7, we present our conclusions and outline directions for future work.

2 Background

In this section, we present an overview of the concepts on which we build our work. First, in Section 2.1 we give a succinct overview of the Stateflow modeling language. Next, in Section 2.2 we give a brief overview of the existing Stateflow imperative language and its SOS. In Section 2.3 we recall the general concept of Satisfiability Modulo Theories (SMT) and the Z3 tool, and finally, in Section 2.4 we give an overview of Bounded Model Checking (BMC).

2.1 Stateflow

Stateflow [30] is a graphical modeling language developed by Mathworks, integrated into the Matlab Simulink [29] modelling environment.

A Simulink Stateflow model can be broadly divided into two parts: *control* and *data*. The control part is modeled through the concepts of *Stateflow state*, *connective junction*, and *transition*, whereas the data part is modelled through a set of *data variables* and *events*. The control of the Stateflow diagram in Figure 1 consists of 6 Stateflow states, 4 connective junctions and 13 transitions. Each Stateflow state is decorated with a set of *state actions*, which includes: *entry (en)*, *duration (du)* and *exit (ex)*. Each action represents an atomic routine. A Stateflow state is either *atomic* or *composite*. Composite Stateflow states contain other states (called *substates*) in their internal structure. A composite state is an Or-composition if only one of its substates can be active at any point in time, or an And-composition if there can be more than one simultaneously active states. The parallelism in the context of And-compositions only means concurrent activation of its substates; the execution, however, is strictly sequential and assigned

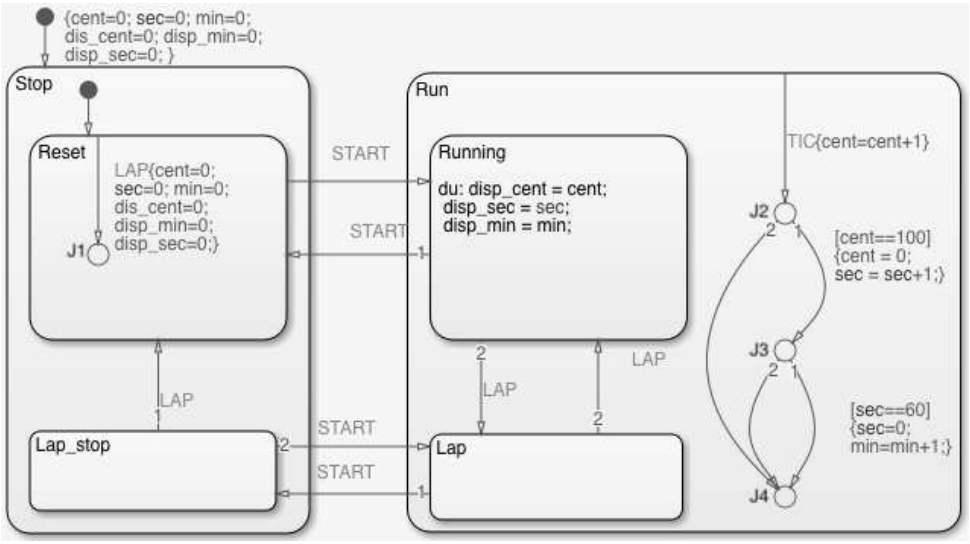

Figure 1: Simple Stateflow diagram - timer example [19].

by the developer. The junctions are used for modelling different branches of execution when a Stateflow diagram moves from one control point into another.

The dynamics of the control flow of a Stateflow diagram is modelled through a set of *transitions* of the following format: $s \xrightarrow{e,c,ca,ta} s'$, where s and s' are the *source* and the *destination* state or junction, respectively, e is the *transition event* that triggers the execution of the transition, which is enabled by the condition (c); ca and ta are transition actions which are executed when c evaluates to true and destination is reached, respectively.

The informal execution semantics of Stateflow models is very intricate and has been explained in detail in the Stateflow user guide published by Mathworks [30]. Due to space limitations, we omit here the details of the informal execution semantics, but give in the following section an overview of a *de-facto* formal one.

2.2 Stateflow Imperative Language: Formal Syntax and Structural Operational Semantics

In order to formalize Stateflow, Hamon and Rushby propose in [19, 20] an imperative language that is a strict subset of the Stateflow graphical language. In the following, we give a brief overview of the language and its operational semantics.

The imperative language is based on the following syntactic categories: state (s), junction (j), event (e), action (a) and condition (c). A transition $t = (e_t, c, a_c, a_t, d)$ is composed of a transition event e_t, condition c, condition and transition actions a_c, a_t, respectively, and a destination d to which it fires. Transitions are grouped into transition lists, which ensure their sequential execution based on a predefined order. A junction definition list J associates a list of transitions with junctions. A state definition list SD associates each state variable (s) with a state definition $sd = ((a,a,a), C, T_i, T_o, J)$. Each sd contains 3 actions, a composition C, lists of internal and outgoing transitions T_i and T_o, respectively, and a junction definition list J. Finally, the composition C can be of type $Or(s_a, p, T, SD)$, where s_a is the active state, p is the path, T is a transition list, and SD is a list of state definitions; or of type $And(b, SD)$, which has a Boolean value b signifying whether the component is active or not, a path p, and a state definition

$$\frac{(e = e_0) \vee (e_0 = \emptyset) \quad e \vdash (c, D_1) \to \top \quad e \vdash (ca, D_1) \hookrightarrow D_2}{e \vdash ((e_0, c, ca, ta, d), D_1) \to D_2, Fire(d, ta)}$$

(a) $[\text{t-FIRE}]_{\text{SOS}}$ rule

$$\frac{(tv = No) \vee (tv = End) \quad \forall i \in [0, \ldots, n]\ e, D_i, J \vdash sd_i \to sd_i', D_{i+1}, No}{e, D_0, J, tv \vdash And\{s_0 : sd_0 \cdots s_n : sd_n\} \to And\{s_0 : sd_0' \cdots s_n : sd_n'\}, D_{n+1}, No}$$

(b) $[\text{AND}]_{\text{SOS}}$ rule

Figure 2: Illustrative sample of SOS rules.

list *SD*. In the reminder of the manuscript, we will use the term *Stateflow program* regardless if it is modeled using the original Stateflow graphical language or the imperative language as we are going to be handling only models that can be rewritten in the imperative language.

The execution of a Stateflow program consists of processing an input event through a sequence of discrete steps. The operational semantics is formalised by a set of 27 layered rules, which precisely prescribe the sequence of actions involved in the processing of an event through the elements of the imperative language [19]. In our work, we refer to executions derivable using the SOS rules as *concrete*. The based form of event processing in Stateflow programs expressed in the imperative language is given as:

$$e \vdash (P, D) \to (P', D'), tv$$

which reads as follows: processing an event e in an environment D through a program component P produces a new environment D', a new program component P', and a transition value tv. An environment $D : Var \to Val$ is a mapping from variables to values. Env denotes the set of all possible environments; P is an element of the Stateflow imperative language, whereas $tv \in \{Fire(d,a) \mid No \mid End\}$ is a transition value which indicates whether a transition has fired ($Fire(d,a)$) or not ($No \mid End$). All of the rules in the SOS extend and slightly differ from this general form [20].

Figure 2a illustrates the $[\text{t-Fire}]_{\text{SOS}}$ SOS rule. The rule describes how a Stateflow transition fires, and intuitively captures the following: in the concrete execution, if the evaluation of a condition evaluates to true ($\top$), and the execution of the condition action *ca* modifies the environment, then a Stateflow program performs a transition, and raises a *Fire* transition value. In a similar way, the Figure 2b showcases the $[\text{AND}]_{\text{SOS}}$ SOS rule, which describes how an And-composition is executed by sequentially executing its substates. For the complete set of SOS rules, we refer the interested reader to the original work by Hamon and Rushby [19, 20].

2.3 Satisfiability Modulo Theories and Z3

The problem of determining whether a Boolean formula can be made true by assigning truth values to the constituent Boolean variables is known as the *Boolean satisfiability problem* (SAT). A decision procedure for SAT is a procedure that generates a (satisfying) assignment for the variables for which a given formula is true, whenever the formula is satisfiable. *Satisfiability Modulo Theories* (SMT) represents an extension of SAT, where some of the logic symbols are interpreted by a background theory [4]. Examples of such background modulo-theories are the theory of equality, the theory of integer numbers, and the theory of real numbers.

Z3 [10] is a state-of-the-art SMT solver and theorem prover developed by Microsoft Research. The input is a model specified in a text-based assertion language that follows the SMT-LIB standard [3]. Z3 provides a number of APIs for different programming languages, including C and Python, which enables

the integration of the Z3 solver with other applications. The input model consists of a set of variables of specific types (also called sorts), and a set of assertions that express constraints over the variables. If the set of constraints (assertions) is satsifiable, the Z3 solver returns result `sat`, accompanied with the interpretation of the variables. In the opposite case, Z3 returns the result `unsat` and a minimal set of unsatisfiable assertions. Finally, if the model is intractable, the solver returns `unknown`.

2.4 Bounded Model Checking

Bounded Model Checking (BMC) is a refutation-based verification technique for checking properties over finite-state transition systems. For checking invariant properties of the type *"something bad never happens"*, BMC unrolls the transition relation until one of the following becomes true: i) a "bad" state has been reached, or ii) a predefined number of unrolling steps has been reached. The number (k) of unrolling steps is called *bound*, while the set of all executions of length k is called *reachability diameter*. Having a reachability diameter of limited size, BMC alleviates the state-space explosion problem at the expense of completeness of the procedure.

Definition 1 (Symbolic Transition System) *A* symbolic transition system *is a pair $S = (I,R)$, where the unary predicate $I(\cdot)$ is a first-order logic (FOL) formula over the components of configurations representing the initial set of configurations, and the binary predicate $R(\cdot,\cdot)$ is a formula representing the "next-state" transition relation, satisfying the equivalences:*

$$I(c) \Leftrightarrow c \in C_0$$
$$R(c,c') \Leftrightarrow (c,c') \in \rightarrow$$

Every initialized path in S of length k can be characterized by the formula:

$$path(c_0,c_1,\ldots,c_k) \triangleq I(c_0) \wedge \bigwedge_{i=0}^{k-1} R(c_i,c_{i+1}), \tag{1}$$

and then, the existence of an initialized path of length k is equivalent to the satisfiability of the formula $path(x_0,x_1,\ldots,x_k)$, where x_i are variables over configurations.

Let φ be a unary predicate over configurations, i.e., a property. We define the corresponding k-bounded invariant property, denoted φ^k, as the formula:

$$\forall c_0,c_1,\ldots,c_k \,.\, (path(c_0,c_1,\ldots,c_k) \Rightarrow \bigwedge_{i=0}^{k} \varphi(c_i)) \tag{2}$$

A path that contains a configuration in which φ^k does not hold is called a *counter-example*, and is characterized by the negation of the above formula, i.e.:

$$\exists c_0,c_1,\ldots,c_k. \, (path(c_0,c_1,\ldots c_k) \wedge \bigvee_{i=0}^{k} \neg\varphi(c_i)) \tag{3}$$

Given that the predicates I, R, and φ can be expressed as FOL formulas, it should be obvious how the refutation of k-bounded invariant properties can be reduced to an SMT problem.

$$\frac{e \vdash (t, \langle \Delta_1, pc_1 \rangle) \rightarrow \langle \Delta_2, pc_2 \rangle, Fire(d,ta)}{e, J \vdash (t.T, \langle \Delta_1, pc_1 \rangle) \rightarrow \langle \Delta_2, pc_2 \rangle, Fire(d,ta)}$$

(a) [T-FIRE]$_{\text{SSOS}}$ rule

$$\frac{(tv = No) \vee (tv = End) \quad \forall i \in [0,\ldots,n] \quad e, \langle \Delta_i, pc_i \rangle, J \vdash sd_i \rightarrow (sd_i', \langle \Delta_{i+1}, pc_{i+1} \rangle, No)}{e, \langle \Delta_0, pc_0 \rangle, J, tv \vdash And\{s_0 : sd_0 \cdots s_n : sd_n\} \rightarrow (And\{s_0 : sd_0' \cdots s_n : sd_n'\}, \langle \Delta_{n+1}, pc_{n+1} \rangle, No)}$$

(b) [AND]$_{\text{SSOS}}$ rule

Figure 3: Illustrative sample of SSOS rules.

3 Symbolic Structural Operational Semantics

In this section, we present our SSOS semantics for the Stateflow imperative language, which we use as a basis for constructing an STS $\widehat{S}$ for a given Stateflow program. We start from the existing *de-facto* SOS semantics as in [19, 20], and transform each of the SOS rules uniformly into a corresponding symbolic counterpart.

In the original formalization, the sets of variables (*Var*) and values (*Val*), as well as the sets of actions (*Act*) and conditions (*Cond*) are considered to be a part of the action language which is distinct from the Stateflow language itself. The details for the actions and conditions are abstracted away; however, it is assumed that the semantics of the executing actions and the evaluating conditions is available via judgments of the form:

$$\text{(i) } e \vdash (a, D) \hookrightarrow D' \text{ and (ii) } e \vdash (c, D) \rightarrow \top \mid \bot$$

which are read as follows: (i) evaluating an action (a) in a current environment (D) produces a new environment (D'), and (ii) evaluating a condition (c) in an environment (D) produces either true or false Boolean value.

The set of SSOS rules is created by uniformly transforming each of the SOS rules into a corresponding symbolic rule, by: i) replacing each valuation of the program variables, called *environment* (D), with a symbolic representation (Δ), and ii) adding a path condition (pc). Consequently, we update the action execution and condition evaluation, which evaluate over the symbolic environment and path condition, respectively. Following the basic principles of symbolic execution [25], in the set of SSOS rules we treat the data component of the language in a symbolic way, whereas the control-flow remains concrete.

We define a *symbolic configuration* $sc \in SC$ as a structure $(P, \langle \Delta, pc \rangle)$, where P is any component from the Stateflow imperative language. We introduce a new set of *symbolic variables* (symbols), denoted *Sym*, and a bijection $g : Var \rightarrow Sym$ between the program variables and the symbols. The path condition pc is simply a Boolean expression over the set of symbols, whereas the *symbolic environment* $\Delta \in SEnv$ is a mapping Δ: $Var \rightarrow Expr_{Sym}$ from program variables to (arithmetic) expressions over symbols. Finally, we assume that symbolic action execution and symbolic condition evaluation are provided via semantic functions of type $\mathscr{SA} : Act \rightarrow (SEnv \rightarrow SEnv)$ and $\mathscr{SB} : Cond \rightarrow (SEnv \rightarrow BExpr_{Sym})$, respectively.

We can now define the axioms for action execution and condition evaluation, for symbolic execution of Stateflow programs, as follows:

$$\begin{aligned} &e \vdash (a, \langle \Delta_1, pc_1 \rangle) \hookrightarrow \langle \Delta_2, pc_1 \rangle \text{ if } \Delta_2 = \mathscr{SA}[\![a]\!](\Delta_1) \\ &e \vdash (c, \langle \Delta_1, pc_1 \rangle) \rightarrow \langle \Delta_1, pc_2 \rangle \text{ if } pc_2 = pc_1 \wedge \mathscr{SB}[\![c]\!](\Delta_1) \end{aligned} \tag{4}$$

The initial symbolic configuration is $(P, \langle \Delta_0, pc_0 \rangle)$, where P is a component of the Stateflow imperative language, $\Delta_0 = g$, and $pc_0 = \top$.

The set of SOS rules can now be uniformly translated into a corresponding SSOS counter-part. Due to space constraints, in Figure 3, we show two instances of the SSOS rules, which are the symbolic counter-part of the SOS rules from Figure 2. For the complete set of SSOS rules, we refer the reader to the accompanying technical report [15]. The [t-FIRE] rule in Figure 3a describes how a Stateflow transition (t) fires by appending the symbolic evaluation of the condition $t.c$ to the current path condition and by symbolically executing the condition action $t.ca$ over the current symbolic environment Δ. When a transition fires, a transition event $Fire(t.d, t.ta)$ is generated. Similarly, the [AND] rule in Figure 3b describes the how the And-composition is processed symbolically.

Since we are overloading the transition relation symbol "$\rightarrow$" in the SOS and SSOS rules, further in the paper we shall use "$\xrightarrow{SOS}$" for transitions derivable with the SOS rules, and "$\xrightarrow{SSOS}$" for transitions derivable with the SSOS rules.

4 Characterization of the SSOS

Our SSOS semantics is essentially an operational semantics for symbolic execution of Stateflow programs. It opens up the opportunity for application of a broader spectrum of verification techniques, such as: *testing* (purely symbolic, or as a combination of symbolic and concrete (concolic) testing [16]) or *bounded model checking* [5]. To be able to reason symbolically over Stateflow programs, however, one must first provide a formal characterization of the relationship between its concrete and symbolic execution. In this section, we prove two results that characterize this relationship. In Theorem 4 we show that for each derivable SSOS transition there exists a corresponding derivable SOS transition. Conversely, in Theorem 4 we show that for each derivable SOS transition there exists a derivable SSOS transition. The connection is established in both cases by means of an interpretation of the symbolic values for which the Boolean expression added to the path condition holds.

First, we introduce some additional notation. Let $\beta : SEnv \times Env \rightarrow Env$ be a function that transforms a symbolic environment Δ into a concrete one $\beta(\Delta, D)$ with the help of an environment D that serves as an interpretation of the symbolic values; for any $v \in Var$, let $\beta(\Delta, D)(v)$ be defined as the value of the expression $\Delta(v)$ in the (renamed) environment $D \circ g^{-1}$. Similarly, let $\mathscr{B} : BExpr_{Sym} \rightarrow (Env \rightarrow Bool)$ be a function that evaluates path conditions in concrete environments, so that $\mathscr{B}[\![pc]\!](D)$ is the Boolean value of the path condition pc in $D \circ g^{-1}$. Finally, observing that the transitions derived by the SSOS rules only (potentially) add a conjunct to the current path condition pc_k to obtain a new path condition pc^{k+1}, let pc_k^{k+1} denote this added conjunct (or $\top$, if no conjunct is added).

Theorem 1. *If* $(P_1, \langle \Delta_1, pc_1 \rangle) \xrightarrow{SSOS} (P_2, \langle \Delta_2, pc_2 \rangle, tv)$, *then for all* $D_0 \in Env$ *s.t.* $\mathscr{B}[\![pc_1^2]\!](\beta(\Delta_1, D_0)) = \top$, *we have* $(P_1, \beta(\Delta_1, D_0)) \xrightarrow{SOS} (P_2, \beta(\Delta_2, D_0))$.

Our next result establishes the reverse direction.

Theorem 2. *If* $(P_1, D_1) \xrightarrow{SOS} (P_2, D_2)$, *then for all* $pc_1 \in BExpr_{Sym}$, $\Delta_1 \in SEnv$ and $D_0 \in Env$ *such that* $\beta(\Delta_1, D_0) = D_1$, *there exist* pc_2, $pc_1^2 \in BExpr_{Sym}$ *and* $\Delta_2 \in SEnv$ *such that* $pc_2 = pc_1 \wedge pc_1^2$, $\mathscr{B}[\![pc_1^2]\!](\beta(\Delta_1, D_0)) = \top$, $\beta(\Delta_2, D_0) = D_2$ *and* $(P_1, \langle \Delta_1, pc_1 \rangle) \xrightarrow{SSOS} (P_2, \langle \Delta_2, pc_2 \rangle)$.

For the proofs of Theorems 4 and 4, we refer the reader to the accompanying technical report [15].

There are two important corollaries of the above two results, which, for reasons of space limitations, will only be stated informally here. First, both results lift naturally to *executions*, i.e., to sequences of transitions. Note in particular how in Theorem 4 the "for all pc_1 ... there exists pc_2" part allows the sequential composition of transitions. Second, when starting from a true path condition, as one does in symbolic execution, the *satisfying assignments* for the path condition at the end of any symbolic path, viewed as interpreting environments, define precisely the concrete paths that follow the symbolic one.

Furthermore, the executions in SOS and SSOS can be shown to *simulate* each other with respect to processing external events. It is well-known that invariant properties are preserved by simulation, and thus, can be checked by symbolically executing the given Stateflow program. Even if limited, this class of properties is important in industrial contexts, as our collaboration with Scania on formally verifying safety-critical embedded code generated from Simulink models has shown.

5 From Stateflow Programs to SMT Solving

In our work, we focus on checking invariant properties over symbolic representation of Stateflow programs, by means of BMC. In Section 3 we developed an SSOS for Stateflow, and exhibited in Section 4 a simulation relation between executions derived in SOS and SSOS, which is sufficient for the preservation of invariant properties. In the following, we show how we use the SSOS to relate Stateflow programs to STS over symbolic configurations. We define the *k-bounded invariant checking* problem for the latter representation (Section 5.1), and show how this problem can be encoded as an SMT problem (Section 5.2).

5.1 Bounded Invariant Checking for Stateflow Programs

In this section, we define a version of STS that encode the *symbolic* behaviors of Stateflow programs, and then adapt the BMC problem to such transition systems.

Definition 2 (STS over Symbolic Configurations) *A symbolic transition system over the symbolic configurations of a given Stateflow program is an STS $\widehat{S} = (\widehat{I}, \widehat{R})$, in the sense of Definition 2.4, but over the symbolic configurations and transitions of the program as induced by the SSOS rules.*
$\widehat{I}(\cdot)$ and $\widehat{R}(\cdot,\cdot)$ are thus a unary "initialization" predicate and a binary "next-state" predicate over the symbolic configurations of the program, respectively.

The formal relationship between an STS over symbolic configurations $\widehat{S}$ and an ordinary STS S of a Stateflow program is given by the following result.

Proposition 1. *Let SF be a Stateflow program, $S = (I,R)$ be an STS over its concrete configurations as induced by the SOS rules, and $\widehat{S} = (\widehat{I},\widehat{R})$ be an STS over its symbolic configurations as induced by the SSOS rules. Then, the following equivalences hold:*

$$(1)\ \widehat{I}(P,\langle\Delta,pc\rangle) \Leftrightarrow \exists D_0 \in Env.\ I(P,D_0) \wedge I(P,\beta(\Delta,D_0)) \wedge \mathscr{B}[\![pc]\!](\beta(\Delta,D_0))$$

$$(2)\ \widehat{R}((P,\langle\Delta_1,pc_1\rangle),(P',\langle\Delta_2,pc_2\rangle)) \Leftrightarrow \begin{array}{l}\exists D_0 \in Env.\ \mathscr{B}[\![pc_1]\!](\beta(\Delta,D_0)) \wedge \mathscr{B}[\![pc_2]\!](\beta(\Delta',D_0)) \\ \wedge\ R((P,\beta(\Delta_1,D_0)),(P',\beta(\Delta_2,D_0))))\end{array}$$

Proof. Follows from Definition 2.4 (see Section 2.4), and from Definition 5.1, Theorem 4 and Theorem 4 (see Section 4). The complete proof can be found in the full version of the manuscript [15]. □

Now, let φ be a predicate over the concrete configurations of a Stateflow program. Predicate φ induces a corresponding predicate $\widehat{\varphi}(sc) \triangleq \varphi(sc[g^{-1}])$ over the symbolic configurations $sc = (P, \langle \Delta, pc \rangle)$, where g is the bijection from Section 3. Assuming an interpretation for the *path* and *k-bounded invariant property* formulas for executions over symbolic configurations, the counter-example path formula (3) for symbolic executions can be rewritten as follows:

$$\exists sc_0, \ldots, sc_k.\ (path(sc_0, \ldots, sc_k) \wedge \bigvee_{i=0}^{k} \neg \widehat{\varphi}(sc_i)) \tag{5}$$

Based on formula (5), we derive the following.

Theorem 3. *Let SF be a Stateflow program, $\widehat{S} = (\widehat{I}, \widehat{R})$ be an STS over its symbolic configurations, and φ^k be a k-bounded invariant property. Then, the following two statements are equivalent:*

1. *SF satisfies the k-bounded invariant property φ^k.*
2. *The formula $path(sc_0, \ldots sc_k) \wedge \bigvee_{i=0}^{k} \neg \widehat{\varphi}(sc_i)$ is UNSAT.*

Proof. (By contradiction.) Assume that a given Stateflow program does not satisfy the k-bounded invariant property φ^k, and that statement (2) holds. By the definition of a k-bounded invariant property, such a property fails if there exists a path in which the last configuration violates φ. By Definition 5, such a path exists if the formula given in (2) is satisfiable (SAT), which contradicts the initial assumption. The other direction is shown analogously. □

Now that we have formally defined BMC invariant checking for STS over symbolic configurations, in the next section we show how to construct such STS for a given Stateflow program.

5.2 From Stateflow Programs to SMT Scripts

In this section, we present a succinct version of a procedure for deriving an STS from a given Stateflow program using the set of SSOS rules, and the transformation of the STS predicates into quantifier-free FOL formulas that can be used for *k-bounded invariant checking* over symbolic configurations, as defined in Theorem 5.1.

The procedure presented in this section demonstrates the derivation of an STS in which the transitions between configurations correspond to transitions at the top `Or`-composition level, which corresponds to our Stopwatch running example (see Figure 1). Due to the layered structure of the imperative language, each such transition consists of a series of transitions corresponding to the various constituent syntactic components of a given Or-composition. Our approach to the derivation of the top-level transitions is to use our SSOS to perform *symbolic execution* between any possible pair of consecutive control points of the program, for arbitrary data values. One should note that in general case, the derivation of the STS is not strictly bound to the top-level component, as it can be done against any syntactic class of the Stateflow imperative language.

As a result of our adopted modeling principle, the configurations for the induced STS are of the following type: $(Or, \langle \Delta, pc \rangle)$. Even though the program component during execution remains the same (the top-level Or-component), it can be the case that its internal configuration changes. The internal

configuration of an Or-component is characterized by the set of active substates. Consequently, the *program control points* correspond to the possible internal configurations at the top Or-composition level.

We model the Stateflow program control points using a set of Boolean variables, denoted as Var_C. For every *Or* control point, Var_C can be partitioned into two subsets: the set $Var_{C^+} = \{v \mid v \in Var_C, v = \top\}$ corresponding to the active states of *Or*, and $Var_{C^-} = Var_C \setminus Var_{C^+}$. Thus, a control point *Or* is characterized by the formula:

$$\Phi_{Or} \triangleq \bigwedge_{v \in Var_{C^+}} v \wedge \bigwedge_{v \in Var_{C^-}} \neg v \tag{6}$$

The path condition *pc* is a quantifier-free Boolean expression over symbols, and as such can be viewed as a quantifier-free FOL formula Φ_{pc}. Based on Δ, one can construct a quantifier-free FOL formula modulo theory of arithmetic for Φ_Δ, over the set of data variables $Var_D = Var \setminus Var_C$ as follows:

$$\Phi_\Delta \triangleq \bigwedge_{v \in Var_D} v' = \Delta(v) \tag{7}$$

Now that we have defined the construction of quantifier-free FOL formulas for each of the components of the symbolic configurations of an STS, we can construct, for every transition T_i between symbolic configurations, a quantifier-free FOL formula (Φ_{T_i}) modulo theory of arithmetic, as follows:

$$\Phi_{T_i} \triangleq \Phi_{Or} \wedge \Phi_{pc_1^2} \Rightarrow \Phi_{Or'} \wedge \Phi_{\Delta'} \tag{8}$$

Intuitively, the formula (8) can be interpreted as follows: when the program is at control point *Or*, the path condition pc_1^2 gives the condition for the program to move to the control point Or', upon which the data will change according to Δ'.

Finally, based on the formula (8) and Proposition 1, we encode the predicates $\widehat{I}$ and $\widehat{R}$ as the following quantifier-free FOL modulo theory of arithmetic formulas:

$$\begin{aligned} \widehat{I} &\triangleq \Phi_{Or_\emptyset} \wedge \Phi_{\Delta_0} \\ \widehat{R} &\triangleq \bigwedge_{T_i \in T} \Phi_{T_i} \end{aligned} \tag{9}$$

where T is the set of all derivable SSOS transitions from the initial top-level composition, which can be computed using any search algorithm starting from the initial program control point ($Or_\emptyset$).

The final step in the process of generating an SMT model is the encoding of the FOL-formulas into corresponding SMT assertions. For details on how the FOL formulas are encoded into SMT assertions, we refer the readers to the full version of this manuscript [15].

6 Implementation and Experimental Comparison

In this section, we first present an implementation of our approach, henceforth referred to as the SESF tool. Even though the most natural way to assess the applicability and the practical usefulness of our approach is to benchmark it against the SLDV tool on a wider set of use cases, in the end it was not possible due to the licensing constraints described in Section 1. Therefore, we proceed with benchmarks only with our own tool. As benchmarks, we use three Stateflow models, including the Stopwatch running

example from Section 2.1, and two industrial models provided by Scania. The main purpose of the experimental comparison is to assess how SESF performs w.r.t. execution time and scalability.

Implementation The SESF tool is composed of two main components: i) a symbolic execution engine that generates the STS in terms of its constituent predicates $\widehat{I}$ and $\widehat{R}$, and ii) an SMT-based model-checking engine which translates the $\widehat{I}$ and $\widehat{R}$ predicates into corresponding SMT formulas, and performs their unrolling alongside the user-provided invariant property (incremental or fixed), either until a predefined bound is reached, or until the problem becomes intractable. The tool is written in Python, and part of the source code is already publicly available [14].

In order to be able to analyze the model, we need to transform it into a formal counter-part which is suitable for analysis. For this purpose, we first measure the preprocessing time of the tool, i.e., the time required for deriving an analysis model from the Stopwatch Stateflow model. Our SESF tool requires 5 seconds for generating the STS, translating it into an SMT script, and unrolling it for 200 execution steps, without performing syntax and consistency check. Note that SESF performs the unrolling step as part of the model generation. Therefore, the model construction time is dependent on the unrolling bound. Another approach would be to perform unrolling during verification step on an as needed basis. This ensures that no unnecessary unrolling is performed.

In our first benchmark, we apply SESF on the Stopwatch model with the following parametric invariant property: *"The value of cent is always between 0 and X"*, for $X \in \{25, 50, 75, 98\}$. By inspecting the Stopwatch model one can see that the variable *cent* gets values in the interval $[0, 99]$, thus for all values of X there is a counter-example. In this case SESF was able to find a counter-example for each instance of the property, within the following time in seconds for each value of $X(cent)$: SESF = {7, 9, 17, 31}.

Next, we analyze the Stopwatch model against the following invariant property: *"The value of sec is always between 0 and 1"*, given a reachability diameter of [100, 125, 150, 175, 200] execution steps. For the aforementioned set of reachability diameters, the SESF terminates in [47, 99, 181, 319, 487] seconds verification time, and negligible model construction time in all cases. A counterexample is found for the last reachability diameter.

The first industrial Stateflow model is a part of a larger vehicle feature that performs identification of a new driver. The model is composed of 6 innermost states, and 29 transitions. It differs from the Stopwatch model in that it is driven by input variables, and not by events. Therefore, the input variables were considered to be free variables in each time step. SESF could analyze a true invariant property for this model with a diameter of 50 in roughly 4.5s.

The second industrial model is used to set program variables based on engine states, and is mostly composed of junctions and transitions between them. To make the analysis easier, we developed a synthetic model with $n = 5$ junctions and 2^n transitions. On this particular model, SESF did not terminate within 2 hours. Our hypothesis for such a poor performance is that SESF can only check the property against a model derived after a complete execution step at the top level instead of after each syntactic element is processed. Fortunately, this limitation of SESF is of a purely implementation nature that will be fixed in future releases, and does not affect the formal underpinning of our approach. To better understand the importance of this limitation, we inspected 72 industrial models, and we discovered that only 2 of them use junction-based sub-parts.

7 Conclusion

We presented a technique for provably correct symbolic analysis of Stateflow programs with respect to invariant properties using BMC. To this end, we developed a symbolic structural operational semantics (SSOS) for the Stateflow language based on the previous work by Hamon and Rushby [19, 20]. We characterized the relationship between the two semantics by exhibiting a simulation relation between them. Next, we defined the bounded invariant checking problem for STS over symbolic configurations, as induced for a given Stateflow program by the set of the SSOS operational rules, and presented informally a procedure for deriving the initial and next state predicates of the STS. Finally, we showed how to generate, from the STS, a set of quantifier-free FOL assertions in SMT-LIB format suitable for analysis using state-of-the-art SMT solvers. The main benefit of our work is that it lays down the foundations for the development of tools for the scalable verification of complex industrial Stateflow models by means of existing symbolic techniques, which we demonstrated with bounded invariant checking on several use-case models. Even though we initially planned to compare our approach against the state-of-the-practice SLDV tool, we had to withdraw from our idea once we discovered the license constraints imposed by Mathworks.

Our work can be extended in several directions. First, our formal characterization of the SSOS can be strengthened by means of a stronger equivalence between the concrete and symbolic representations of Stateflow programs, to formally underpin the symbolic verification of a wider class of properties than invariant properties, such as LTL properties. Second, one can explore the possibility of extending our BMC approach from refutation-based to a verification one, by adding induction [11]. Along this line of research, one could include the option of converting the generated STS into an input format for tools that implement more sophisticated model checking algorithms, such as Lustre [28] models for the Kind2 model checker [8]. Finally, one can extend our experimental evaluation in terms of the number of models, and include other tools in our comparison, such as the CoCoSim framework.

References

[1] Rajeev Alur, Aditya Kanade, S Ramesh & KC Shashidhar (2008): *Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models*. In: *Proceedings of the 8th ACM International Conference on Embedded Software*, pp. 89–98, doi:10.1145/1450058.1450071.

[2] Chonlawit Banphawatthanarak, Bruce H Krogh & Ken Butts (1999): *Symbolic Verification of Executable Control Specifications*. In: *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, IEEE, pp. 581–586, doi:10.1109/CACSD.1999.808712.

[3] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2015): *The SMT-LIB Standard: Version 2.5*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[4] Clark Barrett & Cesare Tinelli (2018): *Satisfiability Modulo Theories*. Springer International Publishing, Cham, doi:10.1007/978-3-319-10575-8˙11.

[5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu et al. (2003): *Bounded Model Checking*. *Advances in Computers* 58(11), pp. 117–148, doi:10.1016/S0065-2458(03)58003-2.

[6] Hamza Bourbouh, Pierre-Loic Garoche, Christophe Garion, Arie Gurfinkel, Kahsai Temesghen & Xavier Thirioux (2017): *Automated Analysis of Stateflow Models*. doi:10.29007/b8gq.

[7] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard & Claire Pagetti (2020): *CoCoSim, a Code Generation Framework for Control/Command Applications: An Overview of CoCoSim for Multi-periodic Discrete Simulink Models*. In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*.

[8] Adrien Champion, Alain Mebsout, Christoph Sticksel & Cesare Tinelli (2016): *The Kind 2 Model Checker*. In: *International Conference on Computer Aided Verification*, Springer, pp. 510–517, doi:10.1007/978-3-319-41540-6˙29.

[9] C. Chen (2010): *Formal Analysis for Stateflow Diagrams*. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, pp. 102–109, doi:10.1109/SSIRI-C.2010.29.

[10] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3˙24. Available at `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[11] Leonardo De Moura, Harald Rueß & Maria Sorea (2003): *Bounded Model Checking and Induction: From Refutation to Verification*. In: *International Conference on Computer Aided Verification*, Springer, pp. 14–26, doi:10.1007/978-3-540-45069-6˙2.

[12] Parasara Sridhar Duggirala, Sayan Mitra, Mahesh Viswanathan & Matthew Potok (2015): *C2E2: A Verification Tool for Stateflow Models*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 68–82, doi:10.1007/978-3-662-46681-0˙5.

[13] J-F Etienne, S Fechter & E Juppeaux (2010): *Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain*. In: *Complex Systems Design & Management*, Springer, pp. 61–72, doi:10.1007/978-3-642-15654-0˙4.

[14] Predrag Filipovikj (2021): *SESf tool*. `https://github.com/predragf/sesf`. [Online; accessed: September 28, 2022].

[15] Predrag Filipovikj, Dilian Gurov & Mattias Nyberg (2021): *Bounded Invariant Checking for Stateflow Programs*. *CoRR* abs/2103.06248. arXiv:2103.06248.

[16] Patrice Godefroid, Nils Klarlund & Koushik Sen (2005): *DART: Directed Automated Random Testing*. *SIGPLAN Not.* 40(6), p. 213–223, doi:10.1145/1064978.1065036.

[17] Grégoire Hamon (2005): *A Denotational Semantics for Stateflow*. In: *Proceedings of the 5th ACM international conference on Embedded software*, pp. 164–172, doi:10.1145/1086228.1086260.

[18] Grégoire Hamon (2008): *Simulink Design Verifier - Applying Automated Formal Methods to Simulink and Stateflow*. In: *Third Workshop on Automated Formal Methods*. Invited paper.

[19] Grégoire Hamon & John Rushby (2004): *An Operational Semantics for Stateflow*. In: *International Conference on Fundamental Approaches to Software Engineering*, Springer, pp. 229–243, doi:10.1007/978-3-540-24721-0˙17.

[20] Grégoire Hamon & John Rushby (2007): *An Operational Semantics for Stateflow*. *International Journal on Software Tools for Technology Transfer* 9(5-6), pp. 447–456, doi:10.1007/s10009-007-0049-7.

[21] David Harel (1987): *Statecharts: A Visual Formalism for Complex Systems*. *Science of Computer Programming* 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.

[22] ISO (2011): *Road vehicles – Functional safety*.

[23] Yu Jiang, Houbing Song, Yixiao Yang, Han Liu, Ming Gu, Yong Guan, Jiaguang Sun & Lui Sha (2019): *Dependable Model-driven Development of CPS: From Stateflow Simulation to Verified Implementation*. *ACM Transactions on Cyber-Physical Systems* 3(1), p. 12, doi:10.1145/3078623.

[24] Stefan Kaalen, Anton Hampus, Mattias Nyberg & Olle Mattsson (2022): *A Stochastic Extension of Stateflow*. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, Association for Computing Machinery, New York, NY, USA, p. 211–222, doi:10.1145/3489525.3511679.

[25] James C King (1976): *Symbolic Execution and Program Testing*. *Communications of the ACM* 19(7), pp. 385–394, doi:10.1145/360248.360252.

[26] B Meenakshi, Abhishek Bhatnagar & Sudeepa Roy (2006): *Tool for Translating Simulink Models into Input Language of a Model Checker*. In: *International Conference on Formal Engineering Methods*, Springer, pp. 606–620, doi:10.1007/11901433˙33.

[27] Alvaro Miyazawa & Ana Cavalcanti (2012): *Refinement-oriented Models of Stateflow Charts*. *Science of Computer Programming* 77(10-11), pp. 1151–1177, doi:10.1016/j.scico.2011.07.007.

[28] Daniel Pilaud, N Halbwachs & JA Plaice (1987): *LUSTRE: A Declarative Language for Programming Synchronous Systems*. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, 178, p. 188, doi:10.1145/41625.41641.

[29] Inc. The MathWorks (2020): *Matlab&Simulink - Simulink User's Guide*. `https://www.mathworks.com/help/pdf_doc/simulink/simulink_ug.pdf`. [Online; accessed: September 28, 2022].

[30] Inc. The MathWorks (2020): *Matlab&Simulink - Stateflow User's Guide*. `https://www.mathworks.com/help/pdf_doc/stateflow/stateflow_ug.pdf`. [Online; accessed: September 28, 2022].

[31] Yixiao Yang, Yu Jiang, Ming Gu & Jiaguang Sun (2016): *Verifying Simulink Stateflow Model: Timed Automata Approach*. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, ACM, New York, NY, USA, pp. 852–857, doi:10.1145/2970276.2970293.

[32] Paolo Zuliani, André Platzer & Edmund M Clarke (2010): *Bayesian Statistical Model Checking with Application to Simulink/Stateflow Verification*. In: *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pp. 243–252, doi:10.1145/1755952.1755987.