Representing Behavioural Models with Rich Control Structures in SMT-LIB

Nancy A. Day and Amirhossein Vakili
David R. Cheriton School of Computer Science
University of Waterloo

Technical Report 2015-14

September 1, 2015

Abstract

We motivate and present a proposal for how to represent extended finite state machine behavioural models with rich hierarchical states and compositional control structures (*e.g.,* the Statecharts family) in SMT-LIB. Our goal with such a representation is to facilitate automated deductive reasoning on such models, which can exploit the structure found in the control structures. We present a novel method that combines deep and shallow encoding techniques to describe models that have both rich control structures and rich datatypes. Our representation permits varying semantics to be chosen for the control structures recognizing the rich variety of semantics that exist for the family of extended finite state machine languages. We hope that discussion of these representation issues will facilitate model sharing for investigation of analysis techniques.

1 Introduction

The purpose of this paper is to initiate a discussion regarding the best way to represent behavioural models written in extended finite state machine-based languages with state hierarchy and rich composition structures (*e.g.,* the Statecharts family [24, 14]) in a common format for analysis. Currently, in most cases, the rich control structures found in these models are translated to more primitive structures for search-based analysis, such as model checking. In this translation, the structure found in the hierarchy and composition of states is either flattened and lost or represented in primitive variables and ignored in the analysis algorithm. With the growing capabilities of automated first-order provers (*e.g.,* SMT solvers [3]), which rely at least partly on deductive-based reasoning, we believe it will likely be useful to retain the structure and modularity found in these models to exploit it deductively in analysis (*e.g.,* with axioms to decompose a hierarchical state). There has been recent progress by us [22, 23] and others [6, 7] on using SMT solvers to do model checking of both finite and infinite state systems. During these investigations, in which we painfully wrote primitive Kripke structures in SMT-LIB, we realized that it would be an advantage to the model-driven engineering (MDE) [21] community to discuss a standard way to represent models with rich control structures. We hope that discussion of these representation issues will facilitate model sharing for investigation of analysis techniques.

In this paper, we motivate and present our proposal for how to represent the syntax of models with rich control structures in SMT-LIB [4][1]. We chose SMT-LIB as a base language because it is an existing, well-used, and well-accepted standard for representing satisfiability and validity problems in first-order logic. Many existing tools support SMT-LIB, including solvers (*e.g.,* [8, 2]) and APIs (*e.g.,* [17, 16, 10]). The

[1]Throughout this paper, when we say SMT-LIB, we mean the latest version of the standard, which is currently 2.5.

S-expressions of SMT-LIB are simple to parse and it is supported by a large community who are continuing to develop new decision procedures for rich datatypes.

There are existing standards for representing models with rich control structures. We created Composed Hierarchical Transition Systems (CHTSs) [18] to represent the syntax of models with rich control structures in XML. Hall and Zisman presented the OpenModel Modeling Language (OMML) [13], which was a similar attempt to create a standard format for the syntax of extended finite state machine models. The Object Management Group (OMG) has defined the Executable UML Foundation (fUML) [20] with its Action Language (Alf) [19]. Our goal in this paper differs from these approaches in that it is focused on representing the control structures within an existing logic for analysis, but without choosing a fixed semantics (as is the case in fUML). We are pursuing the goal of applying deductive reasoning to these models while recognizing the rich variety of semantics that modellers employ for very similar syntax [12]. We aim towards a useable representation of the syntax of the model that can be combined with separately formulated semantic functions that are independent of the specific model and can be varied depending on the language of the model.

A challenge in formalizing the syntax of the control structures of these models in first-order logic (FOL) is the lack of recursive datatypes. We present a novel representation of the syntax of the control structures that stays within the logic of uninterpreted functions [1], which is a decidable fragment of first-order logic. Thus, if the data and operations manipulated by a model are within a decidable fragment of FOL, our representation of the hierarchy of states and transitions keeps the representation of the model within a decidable fragment of logic.

One of the advantages of using SMT-LIB as a base language is that our models with complex control structures for behaviour can include operations on rich (and possibly undecidable) datatypes such as integers, lists, and uninterpreted types and functions. We expect that the ability to write and reason about behavioural models that are rich in both control and datatypes will be a significant advantage in MDE methodology. Integrating a representation of the syntax of complex control structures with variable semantics, and native SMT-LIB datatypes and operations required us to develop a middle point between deep and shallow embedding techniques [5] for representing one language in another.

We begin with an overview of our proposed approach in Section 2, then we present a simple example of our representation in Section 3. Because of the brevity of this paper, we assume the reader is familiar with Statecharts and that the reader is able to read SMT-LIB specifications. In Section 4, we discuss the design decisions we made in choosing the form of our representation. Finally, we conclude the paper with a discussion of future work.

2 Overview

At its most basic level, a behavioural model describes a set of traces of configurations. A *configuration* is a mapping from variables to values. This set of traces can often be concisely defined by a configuration relation. A *configuration relation* (sometimes called a next state relation or a transition relation) is a definition of the set of traces as a set of possible steps between two configurations. The source configuration of a step is called the *current configuration* and the destination configuration of a step is called the *next configuration*. Specifications that are control-oriented include a hierarchical set of states (sometimes called modes). Transitions originate and end at states and have labels. Generally speaking, a configuration relation for a model with states and transitions is the combined (not necessarily conjuncted) behaviour of the set of transitions in the form of implications: when the current configuration includes the source state of a transition and the guard on that transition's label is true of the current configuration, then the next configuration is similar to the current configuration except that the source state of the transition is replaced by its destination state and the action labelling the transition has taken effect.

In creating a representation in SMT-LIB of behavioural languages with rich control structures, we wanted to satisfy the following goals:

1. Represent the structure found in the syntactic description.

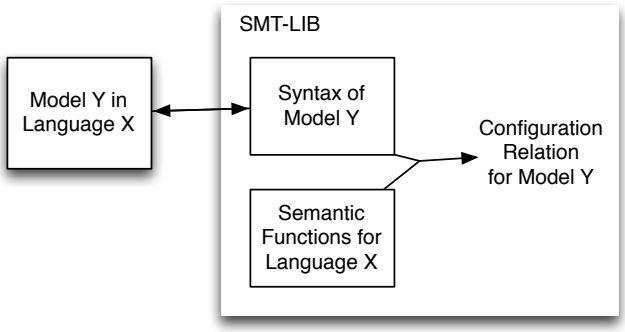


Figure 1: Overview

2. Be able to add semantic functions to the description to define a configuration relation for the model. These functions would depend on the semantics chosen for the language.

3. Integrate rich data and rich control descriptions.

4. Stay within a decidable subset of first-order logic in the formalization of the control aspects of the model.

We explain each goal in order below. Fig 1 illustrates our proposed approach.

Goal 1: In this paper, we present a method for representing the syntax of hierarchical control states and transitions in SMT-LIB (the "syntax" box of Fig 1). It should be possible to translate from a control-oriented language (Language X in Fig 1) into our representation easily. And the translation should be reversible, *i.e.,* there is a 1-1 relationship between the two representations. Some languages have syntactic sugar that we cannot include in our representation, but we want to ensure that the hierarchy and composition structure are completely represented in SMT-LIB.

Goal 2: The complicated nature of the semantics of control-oriented specifications comes in the multitude of ways modellers like to use the hierarchy of states to control the set of transitions relevant at a step [12]. Fig 1 shows how we plan to write semantic functions for the language of the model independently of the specific model. The result of combining the semantic functions with the representation of a specific model is a defined configuration relation for the model that can be used in model checking analysis in an SMT solver (or another kinds of analysis). Because these semantic functions are written separately from the syntactic representation, different meanings can be given to the syntax depending on the modeller's intent. In this paper, we only discuss our representation format for the syntax of models, however, we chose the elements in our syntactic representation format with the intent that they will be compatible with a set of first-order semantic functions very similar to those described in Esmaeilsabzali and Day [11]. These semantic functions will declaratively describe the meaning of the syntax and can be used by a deductive reasoning tool in analysis. For example, a likely semantic rule is that configurations that include an "and" state must include a child state from all of the components of the "and" state. In translations to analysis tools usually such a constraint is encoded operationally in the model. However, in deductive reasoning, such a rule can be used as the basis for a case split (beyond just case splits on variable values). Our purpose in directly representing the rich structure in these models is to allow deductive reasoning to exploit this structure.

Goal 3: Using SMT-LIB and SMT solvers opened the door to including rich data structures in our model beyond those usually found in control-oriented models. In MDE methodology, using such datatypes makes it possible to write a formal abstract model very early in the development life cycle and subject it to automated analysis. We faced the challenge of how to integrate the use of native SMT-LIB datatypes with the representation of the control syntax. We developed an approach that integrates deep and shallow

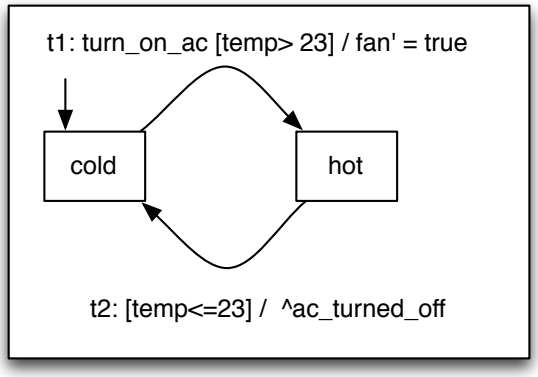


Figure 2: Simple Air Conditioning (AC) Model

methods of embedding of a language. Since the data and data operations are shallowly embedded (explained more in Sections 3 and 4), the syntactic representation requires some formalization of the elements of the configuration.

Goal 4: We challenged ourselves to ensure that at least the description of a model's control structure syntax stayed within a decidable fragment of first-order logic, namely the logic of uninterpreted functions [1]. This constraint meant that we avoided the use of any quantification in our description. Of course, if the modeller chooses data specifications that are not within a decidable fragment, then the overall description is not within this decidable fragment.

3 Example

We consider models that are hierarchical state machines with And, Or, and Basic states. Each transition has a source state, a destination state, a name, and optionally a guard, an action, and a generated event. Transitions are labelled in the form: **t1: event [guard] / action ^generated_event**. Fig 2 is an artificially simplified example of a model with control states. Because SMT-LIB is verbose, even small examples take many lines. Fig 3 shows the declarations of the types and constants that are required in every model in our format. We use the convention that sorts and functions required in our format begin with underscores. We follow the SMT-LIB convention that sorts begin with upper case letters. Fig 4 shows how we represent the state hierarchy through total function definitions. Using function definitions required us to introduce constants such as **_no_state** to handle partial functions.

The definitions for the transitions are shown in Fig 5, where each transition has a name and each part of the transition is defined separately using an if-then-else or disjunction based on the name of the transition. The guards of transitions are shallowly embedded in SMT-LIB, thus their definition must be dependent on a vector of the current values of variables of the configuration. The **_guard** function returns true if the guard of a transition is true in a configuration. Similarly, the actions of the transitions are shallowly embedded in SMT-LIB so their definition must be dependent on a vector of the current values of elements of the configuration and a vector of the next values of configuration elements. The **_action** function returns true if the action of a transition has taken effect in a step. However, we realized that this would not be enough information for the semantic functions to be able to assert that a configuration element that is not changed by a set of transitions taken in a step retains its previous value. To include this information, we introduced one function per configuration variable that returns a Boolean if a transition affects the value of that configuration variable. Since the set of transitions is finite, from this information (and the set of

```
(declare-sort _State 0)
(declare-fun _root () _State)
(declare-fun _no_state () _State)

(declare-sort _Kind 0)
(declare-fun _basic () _Kind)
(declare-fun _and () _Kind)
(declare-fun _or () _Kind)
(declare-fun _no_kind () _Kind)
(assert (distinct _basic _and _or _no_kind))

(declare-sort _Trans 0)

(declare-sort _Event 0)
(declare-fun _no_event () _Event)
```

Figure 3: Generic Parts of Representation in SMT-LIB

```
; declare every state name
(declare-fun cold () _State)
(declare-fun hot () _State)
(assert (distinct _root cold hot _no_state))

; represent the state hierarchy
(define-fun _kind (s _State) _Kind
  (ite (= s _root) _or
  (ite (= s cold) _basic
  (ite (= s hot) _basic
       (_no_kind)))))
(define-fun _parent (s _State) _State
  (ite (= s _root) _no_state
  (ite (= s cold) _root
  (ite (= s hot) _root
       (_no_state)))))
(define-fun _default (s _State) _State
  (ite (= s _root) cold
  (ite (= s cold) _no_state
  (ite (= s hot) _no_state
       (_no_state)))))
```

Figure 4: Representing State Hierarchy

configuration variables), the semantic functions can deduce when a configuration variable is not changed in a step.

Our format requires the following:

- Declarations of sorts: **_State**, **_Kind**, **_Trans**, **_Event**
- Declarations of constants: **_root**, **_no_state**, **_basic**, **_and**, **_or**, **_no_kind**, **_no_event**
- Declarations of constants for state names, transition names, and event names
- Definitions of functions: **_kind**, **_parent**, **_default**, **_src**, **_dest**, **_guard**, **_event**, **_action**, **_gen_event**, and **_change** (per configuration variable). The function **_guard** must take a vector of configuration elements. The function **_action** must take two vectors of configuration elements.
- Assertions that all state names are distinct, all transition names are distinct, and all introduced events are distinct

A modeller can use extra definitions to create the definitions required in our format. We expect that it would be easy to translate from another language to this format. The independent semantic functions will be written using the sorts and functions in our format.

```
; declare every transition name
(declare-fun t1 () _Trans)
(declare-fun t2 () _Trans)
(assert (distinct t1 t2))

; declare basic events
(declare-fun turn_on_ac () _Event)
(declare-fun ac_turned_off () _Event)
(assert (distinct turn_on_ac ac_turned_off _no_event))

; transitions
(define-fun _src (t _Trans) _State
  (ite (= t t1) cold
  (ite (= t t2) hot
       _no_state )))

(define-fun _dest (t _Trans) _State
  (ite (= t t1) hot
  (ite (= t t2) cold
       _no_state )))

(define-fun _event (t _Trans) _Event
  (ite (= t t1) turn_on_ac
  (ite (= t t2) _no_event
       _no_event)))

(define-fun _guard ((t _Trans) (temp Int)) Bool
  (or (and (= t t1) (> temp 23))
      (and (= t t2) (<= temp 23))))

; "fan" is the only configuration variable in this example
(define-fun _action ( (t _Trans) (fan Bool) (fan' Bool) ) Bool
  (or (and (= t t1) (= fan' true))
      (= t t2)))

(define-fun _change_fan (t _Trans) Bool =
  (= t t1))

(define-fun _gen_event (t _Trans) _Event
  (ite (= t t1) _no_event
  (ite (= t t2) ac_turned_off
       _no_event)))
```

Figure 5: Representing Transitions

4 Design Decisions

In this section, we discuss the most significant design decisions we made in creating our representation.

Definitions vs Axioms: Since we cannot use recursive datatype definitions in SMT-LIB [2], we chose to use accessor functions to represent all the information that would be found in a recursive datatype for the state hierarchy. Functions in SMT-LIB are total, thus we faced the classic issue of how to handle describing partial functions, such as the **_default** function where every state does not have a default state. If we chose to axiomatize the meaning of the function, then we would be required to introduce quantifiers in the axioms, which would move our representation out of a decidable fragment of logic. Also, SMT-LIB does not have an Option type (often available in functional programming languages), so we chose to define these functions and declare constants, such as **_no_state**, to represent cases where the function is not defined. We use the SMT-LIB short-hand of the **distinct** keyword to assert that declared constants are distinct. We have chosen not to include an axiom that the constants declared are the complete set of values for the sorts. Such a universally quantified axiom may be required for some kinds of analysis.

Integration of Data and Control: We used a representation format where the state hierarchy is deeply embedded in SMT-LIB using datatypes that we create. A shallow embedding where functions in the destination language/logic are created that "look like" the syntax of the source language was not possible

[2]The SMT-LIB community is discussing standardizing a format for such definitions, but decision procedure support would be challenging.

because the meaning of the composition structures in this family of languages does not match function composition. However, we did not want to describe an embedded syntax for the language of the guards and actions of the transitions. Instead, we wanted to take advantage of the SMT-LIB language to describe these operations and allow rich datatypes and operations, thus we decided to use a shallow embedding of the guards and actions in SMT-LIB. However, in a configuration relation, these guards and actions must be evaluated relative to a configuration. The guards must be defined relative to a vector containing the configuration elements of the model, and the actions must be a relation between vectors of the current configuration and of the next configuration. Records would make writing these function definitions more convenient, but SMT-LIB does not yet support records. We found that additionally we needed to supply the semantic functions with information on what configuration variables are not changed by a transition in order to specify correctly the semantics that an unchanged variable retains its value. A limit of our chosen form of representation is that the semantic functions using actions defined this way will have to enforce either all or none of the action of a transition, potentially causing inconsistencies in the configuration relation if transitions with race conditions can be taken at the same time. The independent semantic functions will also require a vector of configuration elements of the specific model. Once SMT-LIB supports records, writing functions that take this vector can be done without specific knowledge of the model.

Events: In our studies of language variants [12], we found that these languages have a range of ways of describing events, which are meant to be instantaneous occurrences that the system reacts to. Thus, we chose to leave the description of events open-ended in the model. The modeller can add functions that create events (*e.g.,* entered(state)) and event expressions (*e.g.,* the disjunction of events). However, semantic functions will need to be provided for any event that the user introduces.

Invariants: Some languages include invariants as a way to express parts of a model declaratively rather than operationally. Similarly, many models benefit from using declarative invariants to capture environmental constraints. We chose not to include invariants explicitly in our representation because they can be described as axioms in SMT-LIB directly. Invariants can be conjuncted with the configuration relation to form the meaning of the model. Environmental constraints can be antecedents of an analysis question. Axioms of theories describing the behaviour of operations on SMT-LIB data structures are implicit invariants.

5 Conclusions and Future Work

With the goal of facilitating model sharing, we have proposed a format for representing behavioural models written in extended finite state machine-based languages with complex control hierarchy and composition structures in SMT-LIB. We motivated our decisions for the sorts and functions used in the representation and the shallow embedding of the data operations based on wanting flexibility to include interesting datatypes and allowing semantic functions for the language to be described independently of the model. Our representation does not include some of the syntactic sugar used in this family of languages (*e.g.,* in_ state predicates, history states, generating multiple events on a transition) so we will need to evaluate whether our format is useful as it is or if it needs to be extended. Our format allows the specification of abstract behavioural models with rich control and rich datatypes. Our next step is to define a set of semantic functions in SMT-LIB and investigate their use in model checking. A further avenue for exploration is the use of our format to describe parameterized systems.

The appendices contain a template for our format and several examples including the complete AC example.

A Template

```
(declare-sort _State 0)
(declare-fun _no_state () _State)

(declare-sort _Kind 0)
(declare-fun _basic () kind)
(declare-fun _and () kind)
(declare-fun _or () kind)
(declare-fun _no_kind () kind)
(assert (distinct _basic _and _or _no_kind))

(declare-sort _Trans 0)

(declare-sort _Event 0)
(declare-fun _no_event () _Event)

; declare every state name
(declare-fun _root () _State)
(declare-fun statename1 () _State)
(declare-fun statename2 () _State)
...
(assert (distinct statename1 statename2 ... _no_state))

; represent the state hierarchy
(define-fun _kind (s _State) _Kind
  (ite (= s statename1) (...)
  (ite (= s statename2) (...)
  ...
       (_no_kind))))
(define-fun _parent (s _State) _State ...)
(define-fun _default (s _State) _State ...)

; declare every transition name
(declare-fun t1 () _Trans)
(declare-fun t2 () _Trans)
...
(assert (distinct t1 t2 ... _no_trans))

; declare basic events
(declare-fun ev1 () _Event)
(declare-fun ev2 () _Event)
...
(assert (distinct ev1 ev2 ... _no_event))

; declare functions that create other kinds of event

; transitions
(define-fun _src (t _Trans) _State ...)
  (ite (= t t1) (...)
  (ite (= t t2) (...)
  ...
       (_no_state))))

(define-fun _dest (t _Trans) _State ...)

(define-fun _event (t _Trans) _Event ...)

(define-fun _guard ((t _Trans) [elements of configuration] ) Bool ...)

(define-fun _action ((t _Trans) [elements of configuration]
                                [copy of elements of configuration]) Bool ...)

; per configuration element, transitions that constrain/change it
(define-fun _change_v1 (t _Trans) Bool ...)
(define-fun _change_v2 (t _Trans) Bool ...)

(define-fun _gen_event (t _Trans) _Event ...)
```

B Air Conditioning Example

```
(declare-sort _State 0)
(declare-fun _root () _State)
(declare-fun _no_state () _State)
```

```
(declare-sort _Kind 0)
(declare-fun _basic () _Kind)
(declare-fun _and () _Kind)
(declare-fun _or () _Kind)
(declare-fun _no_kind () _Kind)
(assert (distinct _basic _and _or _no_kind))

(declare-sort _Trans 0)

(declare-sort _Event 0)
(declare-fun _no_event () _Event)

; declare every state name
(declare-fun cold () _State)
(declare-fun hot () _State)
(assert (distinct _root cold hot _no_state))

; represent the state hierarchy
(define-fun _kind (s _State) _Kind
  (ite (= s _root) _or
  (ite (= s cold) _basic
  (ite (= s hot) _basic
       (_no_kind)))))
(define-fun _parent (s _State) _State
  (ite (= s _root) _no_state
  (ite (= s cold) _root
  (ite (= s hot) _root
       (_no_state)))))
(define-fun _default (s _State) _State
  (ite (= s _root) cold
  (ite (= s cold) _no_state
  (ite (= s hot) _no_state
       (_no_state)))))

; declare every transition name
(declare-fun t1 () _Trans)
(declare-fun t2 () _Trans)
(assert (distinct t1 t2))

; declare basic events
(declare-fun turn_on_ac () _Event)
(declare-fun ac_turned_off () _Event)
(assert (distinct turn_on_ac ac_turned_off _no_event))

; transitions
(define-fun _src (t _Trans) _State
  (ite (= t t1) cold
  (ite (= t t2) hot
       _no_state )))

(define-fun _dest (t _Trans) _State
  (ite (= t t1) hot
  (ite (= t t2) cold
       _no_state )))

(define-fun _event (t _Trans) _Event
  (ite (= t t1) turn_on_ac
  (ite (= t t2) _no_event
       _no_event)))

(define-fun _guard ((t _Trans) (temp Int)) Bool
  (or (and (= t t1) (> temp 23))
      (and (= t t2) (<= temp 23))))

; "fan" is the only configuration variable in this example
(define-fun _action ( (t _Trans) (fan Bool) (fan' Bool) ) Bool
  (or (and (= t t1) (= fan' true))
      (= t t2)))

(define-fun _change_fan (t _Trans) Bool =
  (= t t1))

(define-fun _gen_event (t _Trans) _Event
  (ite (= t t1) _no_event
  (ite (= t t2) ac_turned_off
       _no_event)))
```

C Traffic Light Example (model from [15])

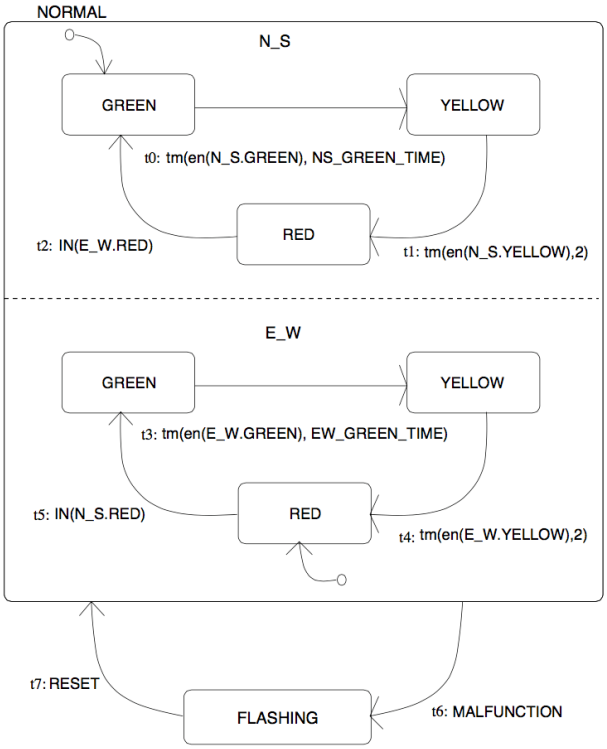


Figure 2.1: Traffic light statechart

From Figure 6-22 in [15]

```
; generic declarations for all models
(declare-sort _State 0)
(declare-fun _root () _State)
(declare-fun _no_state () _State)

(declare-sort _Kind 0)
(declare-fun _basic () _Kind)
(declare-fun _and () _Kind)
(declare-fun _or () _Kind)
(declare-fun _no_kind () _Kind)
(distinct _basic _and _or _no_kind)

(declare-sort _Trans 0)

(declare-sort _Event 0)
(declare-fun _no_event () _Event)

; model-specific information

; declarations of states
(declare-fun normal () _State)
(declare-fun flashing () _State)
(declare-fun n_s () _State)
(declare-fun e_w () _State)
(declare-fun e_w_yellow () _State)
```

```
(declare-fun e_w_red () _State)
(declare-fun e_w_green () _State)
(declare-fun n_s_green () _State)
(declare-fun n_s_red () _State)
(declare-fun n_s_yellow () _State)
(assert (distinct _root normal flashing n_s e_w e_w_yellow e_w_red e_w_green n_s_green n_s_red n_s_yellow _no_state))

; kind of each state
(define-fun _kind ((s _State)) _Kind
  (ite (= s normal) _and
  (ite (= s flashing) _basic
  (ite (= s n_s) _or
  (ite (= s e_w) _and
  (ite (= s e_w_yellow) _basic
  (ite (= s e_w_red) _basic
  (ite (= s e_w_green) _basic
  (ite (= s n_s_green) _basic
  (ite (= s n_s_red) _basic
  (ite (= s n_s_yellow) _basic
       _no_kind)))))))))))

; default states
(define-fun _default ((s _State)) _State
  (ite (= s n_s) n_s_green
  (ite (= s e_w) e_w_red
  _no_state)))

; parent of a state
(define-fun _parent ((s _State)) _State
  (ite (= s normal) _root
  (ite (= s flashing) _root
  (ite (= s n_s) noraml
  (ite (= s e_w) noraml
  (ite (= s n_s_green) n_s
  (ite (= s n_s_yellow) n_s
  (ite (= s n_s_red) n_s
  (ite (= s e_w_green) e_w
  (ite (= s e_w_yellow) e_w
  (ite (= s e_w_red) e_w
       _no_state)))))))))))

; declaration of transitions
(declare-fun t0 () _Tran)
(declare-fun t1 () _Tran)
(declare-fun t2 () _Tran)
(declare-fun t3 () _Tran)
(declare-fun t4 () _Tran)
(declare-fun t5 () _Tran)
(declare-fun t6 () _Tran)
(declare-fun t7 () _Tran)
(assert (distinct t0 t1 t2 t3 t4 t5 t6 t7 ))

; declaration of events
(declare-fun en (_State) _Event)
(declare-fun tm (_Event Int) _Event)
(declare-fun malfunction () _Event)
(declare-fun reset () _Event)

; some constants (not configuration elements)
(declare-fun E_W_GREEN_TIME () Int)
(declare-fun N_S_GREEN_TIME () Int)

; source of a transition
(define-fun _src ((t _Tran)) _State
  (ite (= t t0) n_s_green
  (ite (= t t1) n_s_yellow
  (ite (= t t2) n_s_red
  (ite (= t t3) e_w_green
  (ite (= t t4) e_w_yellow
  (ite (= t t5) e_w_red
  (ite (= t t6) normal
  (ite (= t t7) flashing
       _no_state)))))))))

; destination of a transition
(define-fun _dest ((t _Tran)) _State
  (ite (= t t0) n_s_yellow
```

```
  (ite (= t t1) n_s_red
  (ite (= t t2) n_s_green
  (ite (= t t3) e_w_yellow
  (ite (= t t4) e_w_red
  (ite (= t t5) e_w_green
  (ite (= t t6) flashing
  (ite (= t t7) normal
       _no_state)))))))))

; event corresponding to a transition
(define-fun _event ((t _Tran)) _Event
  (ite (= t t0) (tm (en n_s_green) N_S_GREEN_TIME)
  (ite (= t t1) (tm (en n_s_yellow) 2)
  (ite (= t t2) _no_event
  (ite (= t t3) (tm (en e_w_green) E_W_GREEN_TIME)
  (ite (= t t4) (tm (en e_w_yellow) 2)
  (ite (= t t5) _no_event
  (ite (= t t6) malfunction
  (ite (= t t7) reset
       _no_event)))))))))

; here, we see the need for records in SMT-LIB for the configuration vector
(define-fun _guard ((t _Tran)
                    (in_normal Bool) (in_flashing Bool) (in_n_s Bool) (in_e_w Bool)
                    (in_n_s_red Bool) (in_n_s_yellow Bool) (in_n_s_green Bool)
                    (in_e_w_red Bool) (in_e_w_yellow Bool) (in_e_w_green Bool))
                    Bool
  (or (= t t0)
  (or (= t t1)
  (or (and (= t t2) in_red_e)
  (or (= t t3)
  (or (= t t4)
  (or (and (= t t5) in_red_n)
  (or (= t t6)
      (= t t7)))))))))

; there are no actions in this model
(define-fun _action ((t _Tran)) Bool true)
```

D Cruise Control Example (model and figure from [9])

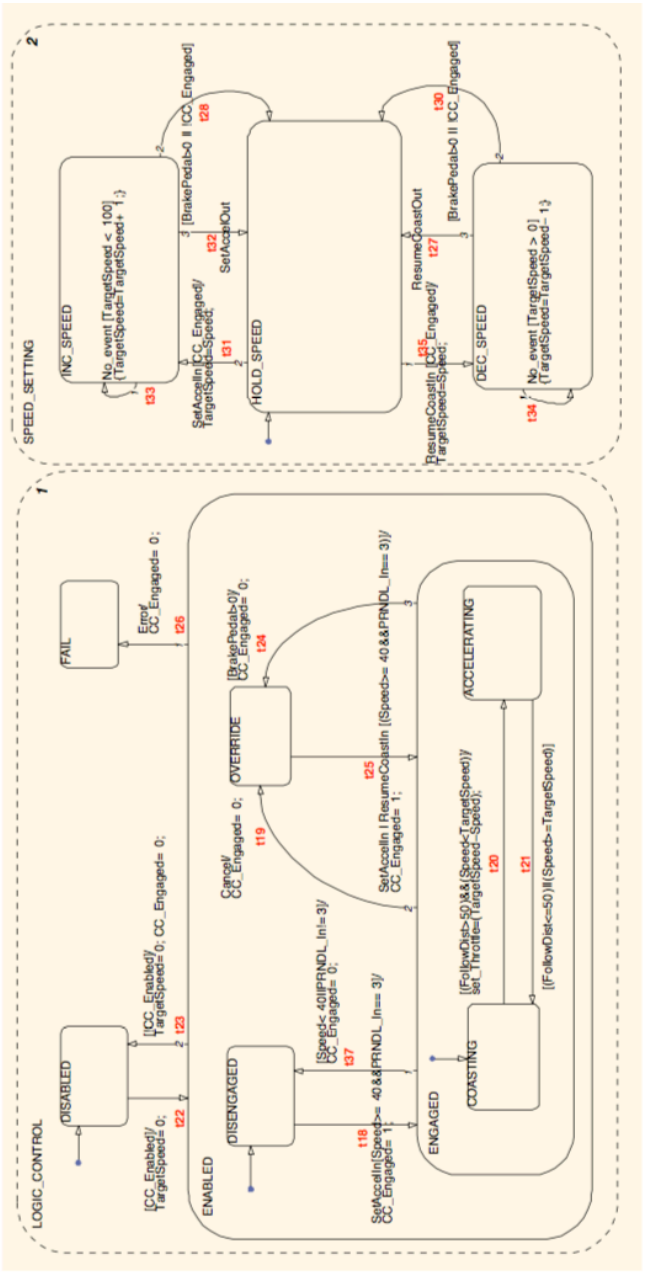


Figure A.2: Cruise Control (CC) STATEFLOW design model

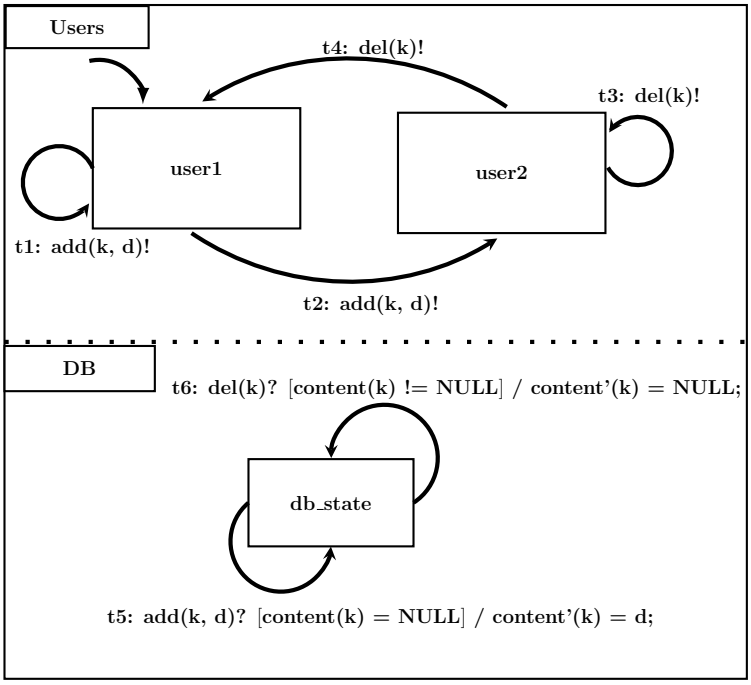
```
; generic declarations for all models
(declare-sort _State 0)
(declare-fun _root () _State)
(declare-fun _no_state () _State)

(declare-sort _Kind 0)
(declare-fun _basic () _Kind)
(declare-fun _and () _Kind)
```

```
(declare-fun _or () _Kind)
(declare-fun _no_kind () _Kind)
(distinct _basic _and _or _no_kind)

(declare-sort _Trans 0)

(declare-sort _Event 0)
(declare-fun _no_event () _Event)

; model-specific information

; declaration of states
(declare-fun FAIL () _State)
(declare-fun HOLD_SPEED () _State)
(declare-fun DISENGAGED () _State)
(declare-fun DISABLED () _State)
(declare-fun SPEED_SETTING () _State)
(declare-fun LOGIC_CONTROL () _State)
(declare-fun OVERRIDE () _State)
(declare-fun ACCELERATING () _State)
(declare-fun DEC_SPEED () _State)
(declare-fun ENGAGED () _State)
(declare-fun COASTING () _State)
(declare-fun INC_SPEED () _State)
(declare-fun ENABLED () _State)
(assert (distinct _root
                  _no_state
                  FAIL
                  HOLD_SPEED
                  DISENGAGED
                  DISABLED
                  SPEED_SETTING
                  LOGIC_CONTROL
                  OVERRIDE
                  ACCELERATING
                  DEC_SPEED
                  ENGAGED
                  COASTING
                  INC_SPEED
                  ENABLED))


(define-fun _kind ((s _State)) _Kind
  (ite (= s _root) _and
  (ite (= s LOGIC_CONTROL) _or
  (ite (= s SPEED_SETTING) _or
  (ite (= s FAIL) _basic
  (ite (= s DISABLED) _basic
  (ite (= s ENABLED) _or
  (ite (= s INC_SPEED) _basic
  (ite (= s HOLD_SPEED) _basic
  (ite (= s DEC_SPEED) _basic
  (ite (= s DISENGAGED) _basic
  (ite (= s ENGAGED) _or
  (ite (= s OVERRIDE) _basic
  (ite (= s COASTING) _basic
  (ite (= s ACCELERATING) _basic
  _no_kind))))))))))))))

(define-fun _default ((s _State)) _State
  (ite (= s LOGIC_CONTROL) DISABLED
  (ite (= s SPEED_SETTING) HOLD_SPEED
  (ite (= s ENABLED) DISENGAGED
  (ite (= s ENGAGED) COASTING
  _no_state)))))

(define-fun _parent ((s _State)) _State
  (ite (= s LOGIC_CONTROL) _root
  (ite (= s SPEED_SETTING) _root
  (ite (= s FAIL) LOGIC_CONTROL
  (ite (= s DISABLED) LOGIC_CONTROL
  (ite (= s ENABLED) LOGIC_CONTROL
  (ite (= s INC_SPEED) SPEED_SETTING
  (ite (= s HOLD_SPEED) SPEED_SETTING
  (ite (= s DEC_SPEED) SPEED_SETTING
  (ite (= s DISENGAGED) ENABLED
  (ite (= s ENGAGED) ENABLED
```

```
  (ite (= s OVERRIDE) ENABLED
  (ite (= s COASTING) ENGAGED
  (ite (= s ACCELERATING) ENGAGED
  _no_state)))))))))))))))

; declaration of transitions
(declare-fun t31 () _Tran)
(declare-fun t34 () _Tran)
(declare-fun t18 () _Tran)
(declare-fun t25 () _Tran)
(declare-fun t24 () _Tran)
(declare-fun t35 () _Tran)
(declare-fun t26 () _Tran)
(declare-fun t23 () _Tran)
(declare-fun t33 () _Tran)
(declare-fun t37 () _Tran)
(declare-fun t21 () _Tran)
(declare-fun t27 () _Tran)
(declare-fun t22 () _Tran)
(declare-fun t19 () _Tran)
(declare-fun t32 () _Tran)
(declare-fun t28 () _Tran)
(declare-fun t30 () _Tran)
(declare-fun t20 () _Tran)
(assert (distinct t31 t24 t18 t2 t24 t35 t26 t23 t33 t37 t21 t27 t22 t19 t32 t28 t30 t20))

; declaration of events
(declare-fun No_event () _Event)
(declare-fun Cancel () _Event)
(declare-fun ResumeCoastOut () _Event)
(declare-fun SetAccelOut () _Event)
(declare-fun SetAccelIn () _Event)
(declare-fun ResumeCoastIn () _Event)
(declare-fun Error () _Event)
(declare-fun event_or (_Event _Event) _Event)
(assert (distinct _no_event No_event Cancel ResumeCoastIn ResumeCoastOut SetAccelIn SetAccelOut Error))


(define-fun _src ((t _Tran)) _State
  (ite (= t t22) DISABLED
  (ite (= t t23) ENABLED
  (ite (= t t26) ENABLED
  (ite (= t t18) DISABLED
  (ite (= t t37) ENABLED
  (ite (= t t19) ENGAGED
  (ite (= t t24) ENGAGED
  (ite (= t t25) OVERRIDE
  (ite (= t t20) COASTING
  (ite (= t t21) ACCELERATING
  (ite (= t t33) INC_SPEED
  (ite (= t t31) HOLD_SPEED
  (ite (= t t32) INC_SPEED
  (ite (= t t28) INC_SPEED
  (ite (= t t35) HOLD_SPEED
  (ite (= t t27) DEC_SPEED
  (ite (= t t30) DEC_SPEED
  (ite (= t t34) DEC_SPEED
  _no_state))))))))))))))))))))

(define-fun _dest ((t _Tran)) _State
  (ite (= t t22) ENABLED
  (ite (= t t23) DISABLED
  (ite (= t t26) FAIL
  (ite (= t t18) ENGAGED
  (ite (= t t37) DISENGAGED
  (ite (= t t19) OVERRIDE
  (ite (= t t24) OVERRIDE
  (ite (= t t25) ENGAGED
  (ite (= t t20) ACCELERATING
  (ite (= t t21) COASTING
  (ite (= t t33) INC_SPEED
  (ite (= t t31) INC_SPEED
  (ite (= t t32) HOLD_SPEED
  (ite (= t t28) HOLD_SPEED
  (ite (= t t35) DEC_SPEED
  (ite (= t t27) HOLD_SPEED
  (ite (= t t30) HOLD_SPEED
```

```
  (ite (= t t34) DEC_SPEED
  _no_state)))))))))))))))))))

(define-fun _event ((t _Tran)) _Event
  (ite (= t t26) Error
  (ite (= t t18) SetAccelIn
  (ite (= t t19) Cancel
  (ite (= t t25) (event_or SetAccelIn ResumeCoastIn)
  (ite (= t t33) No_event
  (ite (= t t31) SetAccelIn
  (ite (= t t32) SetAccelOut
  (ite (= t t35) ResumeCoastIn
  (ite (= t t27) ResumeCoastOut
  (ite (= t t34) No_event
  _no_event)))))))))))

(define-fun _guard ((t _Tran) (CC_Enabled Bool) (FollowDist Int) (BrakePedal Int) (AccelPedal Int) (Speed Int) (PRNDL_in I
  (or
    (and (= t t22) CC_Enabled)
    (and (= t t23) (not CC_Enabled))
    (= t t26)
    (and (= t t18) (>= Speed 40) (= PRNDL_in 3))
    (and (= t t37) (or (< Speed 40) (not (= PRNDL_in 3))))
    (= t t19)
    (and (= t t24) (> BrakePedal 0))
    (and (= t t25) (>= Speed 40) (= PRNDL_in 3))
    (and (= t t20) (> FollowDist 50) (< Speed TargetSpeed))
    (and (= t t33) (< TargetSpeed 100))
    (and (= t t31) CC_Engaged)
    (= t t32)
    (and (= t t28) (or (> BrakePedal 0) (not CC_Engaged)))
    (and (= t t35) CC_Engaged)
    (= t t27)
    (and (= t t30) (or (> BrakePedal 0) (not CC_Engaged)))
    (and (= t t34) (> TargetSpeed 0))))

; one _action an _change per each variable

(define-fun _action_CC_Engaged ((t _Tran) (Speed Int) (TargetSpeed Int) (set_Throttle Int) (CC_Engaged' Bool)) Bool
  (or
    (and (= t t23) (= CC_Engaged' false))
    (and (= t t26) (= CC_Engaged' false))
    (and (= t t18) (= CC_Engaged' true))
    (and (= t t37) (= CC_Engaged' false))
    (and (= t t19) (= CC_Engaged' false))
    (and (= t t24) (= CC_Engaged' false))
    (and (= t t25) (= CC_Engaged' true))))
(define-fun _change_CC_Engaged ((t _Tran)) Bool
  (or (= t t23)
      (= t t26)
      (= t t18)
      (= t t37)
      (= t t19)
      (= t t24)
      (= t t25)))


(define-fun _action_TargetSpeed ((t _Tran) (Speed Int) (TargetSpeed Int) (set_Throttle Int) (TargetSpeed' Int)) Bool
  (or
    (and (= t t22) (= TargetSpeed' 0))
    (and (= t t23) (= TargetSpeed' 0))
    (and (= t t33) (= TargetSpeed' (+ TargetSpeed 1)))
    (and (= t t34) (= TargetSpeed' (- TargetSpeed 1)))))
(defin-fun _change_TargetSpeed ((t _Tran)) Bool
  (or (= t t22)
      (= t t23)
      (= t t33)
      (= t t34)))

(define-fun _action_set_Throttle ((t _Tran) (Speed Int) (TargetSpeed Int) (set_Throttle Int) (set_Throttle' Int)) Bool
  (or
    (and (= t t20) (= set_Throttle' (- TargetSpeed Speed)))))
(define-fun _change_set_Throttle ((t _Tran)) Bool
  (= t t20))
```

E Database Example




```
; generic declarations for all models
(declare-sort _State 0)
(declare-fun _root () _State)
(declare-fun _no_state () _State)

(declare-sort _Kind 0)
(declare-fun _basic () _Kind)
(declare-fun _and () _Kind)
(declare-fun _or () _Kind)
(declare-fun _no_kind () _Kind)
(distinct _basic _and _or _no_kind)

(declare-sort _Trans 0)

(declare-sort _Event 0)
(declare-fun _no_event () _Event)

; model-specific information

; declarations for states
(declare-fun users () _State)
(declare-fun DB () _State)
(declare-fun db_State () _State)
(declare-fun user1 () _State)
(declare-fun user2 () _State)
(assert (distinct _root users user1 user2 db db_state _no_state))

; declarations for transitions
(declare-fun t1 () _Tran)
(declare-fun t2 () _Tran)
(declare-fun t3 () _Tran)
(declare-fun t4 () _Tran)
(declare-fun t5 () _Tran)
(declare-fun t6 () _Tran)
(assert (distinct t1 t2 t3 t4 t5 t6))

; declaration of events
(declare-fun del_r () _Event)
(declare-fun add_s () _Event)
(declare-fun del_s () _Event)
(declare-fun add_r () _Event)
```

```
(assert (distinct add_r add_s del_r del_s _no_event))

; user defined sorts for this specific model

; DBState represents the state of the database.
(declare-sort DBState 0)

; Data represents all the possible data that can be
; stored in the database
(declare-sort Data 0)

; Key represents the available keys
(declare-sort Key 0)

; content represent the content of the database
(declare-fun content (DBState Key) Data)

; if there is no data associated with a key,
; the value of that key is NULL
(declare-fun NULL () Data)

; macros that must be defined for every statechart.

(define-fun _kind ((s _State)) _Kind
    (ite (or (= s user1) (= s user2) (= s db_State)) _basic
    (ite (or (= s users) (= s DB)) _or
    (ite (= s _root) _and
    _no_kind))))

(define-fun _parent ((s _State)) _State
    (ite (or (= s user1) (= s user2)) users
    (ite (= s db_State) DB
    (ite (or (= s users) (= s DB)) _root
    _no_state))))

(define-fun _default ((s _State)) _State
    (ite (= s users) user1
    (ite (= s DB) db_State
    _no_state)))

(define-fun _src ((t _Tran)) _State
    (ite (= t t1) user1
    (ite (= t t2) user1
    (ite (= t t3) user2
    (ite (= t t3) user2
    (ite (= t t5) db_State
    (ite (= t t6) db_State
    _no_state)))))))

(define-fun _dest ((t _Tran)) _State
    (ite (= t t1) user1
    (ite (= t t2) user2
    (ite (= t t3) user2
    (ite (= t t3) user1
    (ite (= t t5) db_State
    (ite (= t t6) db_State
    _no_state)))))))

(define-fun _event ((t _Tran) (k Key) (d Data)) _Event
    (ite (or (= t t1) (= t t2)) add_s
    (ite (or (= t t3) (= t t4)) del_s
    (ite (= t t5) add_r
    (ite (= t t6) del_r
    _no_event)))))

(define-fun _guard ((t _Tran) (k Key) (d Data) (c DBState)) Bool
    (or (= t t1)
        (= t t2)
        (= t t3)
        (= t t4)
        (and (= t t5) (= (content c k) NULL))
        (and (= t t6) (not (= (content c k) NULL)))))

; one _action and _change relation per each configuration variable:
;
(define-fun _action_c ((t _Tran) (k Key) (d Data) (c DBState)
                               (k' Key) (d' Data) (c' DBState) ) Bool
```

```
    (or (and (= t t1) (= (content c' k) (content c k)))
        (and (= t t2) (= (content c' k) (content c k)))
        (and (= t t3) (= (content c' k) (content c k)))
        (and (= t t4) (= (content c' k) (content c k)))
        (and (= t t5) (= (content c' k) d))
        (and (= t t6) (= (content c' k) NULL))))
(define-fun _change_c ((t _Tran)) Bool
  (or (= t t1)
      (= t t2)
      (= t t3)
      (= t t4)
      (= t t5)
      (= t t6)))
```

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem. Studies in Logic and the Foundations of Mathematics*. North-Holland, 1954.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, LNCS, pages 171–177. Springer Berlin Heidelberg, 2011.

[3] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard Version 2.0 Reference Manual*, January 2010.

[5] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and Van Tassel John. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[6] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *CAV*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.

[7] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *CAV*, pages 334–342, 2014.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, LNCS, pages 337–340. Springer, 2008.

[9] Alma L. Juarez Dominguez. *Detection of Feature Interactions in Automotive Active Safety Feature*. PhD thesis, University of Waterloo, 2012.

[10] Levent Erkok. SBV: SMT based verification in Haskell. http://leventerkok.github.io/sbv/.

[11] Shahram Esmaeilsabzali and Nancy Day. Prescriptive semantics for big-step modelling languages. In *FASE*, volume 6013 of *LNCS*, pages 158–172, 2010.

[12] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering*, 15(2):235—265, 2010.

[13] Robert J. Hall and Andrea Zisman. OMML: A behavioural model interchange format. In *RE*, pages 272–282. IEEE Computer Society, 2004.

[14] David Harel. Statecharts: A visual formalism for complex systems. *Sci. of Comp. Prog.*, 8(3):231—274, 1987.

[15] MA i Logix Inc., Burlington. Statemate 4.0 analyzer user and reference manual, April 1991.

[16] Kyle Krchak and Aaron Stump. ocaml-smt2. http://www.cs.uiowa.edu/~astump/software/ocaml-smt2.zip.

[17] Andrea Micheli and Marco Gario. pysmt 0.4.1. https://pypi.python.org/pypi/pySMT.

[18] Jianwei Niu. *Template Semantics: A Parameterized Approach to Semantics-Based Model Compilation.* PhD thesis, University of Waterloo, 2005.

[19] Object Management Group (OMG). Action language for foundational UML (Alf), version 1.0.1, 2013.

[20] Object Management Group (OMG). Semantics of a foundational subset for executable UML models (fUML), version 1.1, 2013.

[21] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[22] Amirhossein Vakili and Nancy A. Day. Reducing CTL-live model checking to first-order logic validity checking. In *FMCAD*, pages 215–218. IEEE; New York, NY, 2014.

[23] Amirhossein Vakili and Nancy A. Day. Verifying CTL-live properties of infinite state models using an SMT solver. In *FSE*, pages 213–223. ACM, 2014.

[24] Michael von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128– 148. Springer, 1994.