© ACM 2016. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in MiSE '16: Proceedings of the 8th International Workshop on Modeling in Software Engineering https://doi.org/10.1145/2896982.2896990.

Representing Hierarchical State Machine Models in SMT-LIB

Nancy A. Day
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
nday@uwaterloo.ca

Amirhossein Vakili
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
avakili@uwaterloo.ca

ABSTRACT

We motivate and present a proposal for how to represent the syntax of behavioural models written in extended finite-state machine languages with hierarchical states (*e.g.,* the Statecharts family) in SMT-LIB. By including the state structure explicitly in the SMT-LIB model, our goal is to facilitate effective automated deductive reasoning, which can exploit the structure found in the state hierarchy. We present a novel method that combines deep and shallow encoding techniques to describe models that have both state hierarchy and use the rich datatypes found in SMT-LIB. Our representation permits varying semantics to be chosen for the syntax recognizing the rich variety of semantics that exist for this family of modelling 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 hierarchical extended finite-state machine based languages (HSMs) (*e.g.,* the Statecharts family [24, 14]) within SMT-LIB [3][1]. HSMs have state hierarchy, parallel states, and transitions with labels that manipulate data. SMT-LIB is an existing, well-used, and well-accepted standard for representing satisfiability and validity problems in first-order logic (FOL). Many existing verification tools accept SMT-LIB as input, including solvers (*e.g.,* [9, 2]) and APIs (*e.g.,* [17, 16, 10]). We propose a representation of the syntax of HSMs in SMT-LIB that explicitly represents the state structure, motivated by the goal of allowing SMT solvers to use this structure to improve their deductive analysis performance for tasks such as model checking. Translators can be written from representations of models in sophisticated modelling IDEs to our proposed representation in SMT-LIB.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

What kinds of HSMs? We are interested in supporting state machine based modelling formalisms that allow the user to describe the system using hierarchical and parallel states. Transitions between states are labelled with guards, events, and actions. There are many modelling languages in this family including Statecharts [14] and UML StateMachines [21], and there are a variety of semantics proposed for this kind of syntactic representation [24, 12]. Much of the complexity and variety in the semantics comes in the meaning of the big-steps (macro and micro steps in Statecharts terminology).

Why represent HSMs in SMT-LIB? There has been recent progress by us [22, 23] and others [6, 7] on using SMT solvers for model checking finite and infinite state systems. During these investigations, we painfully wrote primitive Kripke structures models in SMT-LIB for input to solvers and we would like to analyze models written in more user-friendly modelling languages. Usually, a translation process from HSMs to analysis tools incorporates the semantics of the language into the representation of the model in the destination language, and the hierarchical state structure is lost: it is either flattened and lost, or represented in primitive variables and ignored in analysis. Automated first-order provers (*e.g.,* SMT solvers [4]) are deductive reasoners. We believe that the deductive-based reasoning of SMT-solvers can exploit the hierarchical state structure to perform reasoning in a more efficient and modular manner. In our proposed approach, we create an explicit representation of the syntax of the state hierarchy in SMT-LIB and plan to use separate FOL axioms to describe the semantics of the states in the HSMs. We hope that discussion of the syntactic representation issues will facilitate model sharing for investigation of analysis techniques.

What are the modelling issues? This work encountered three main challenges:

Staying within a decidable fragment of FOL. A challenge in formalizing the state hierarchy of these models in FOL is the lack of recursive datatypes. We chose a representation of the syntax of the state hierarchy that stays within the logic of uninterpreted functions [1], which is a decidable fragment of FOL. Thus, if the data and operations of the transitions of the model are within a decidable fragment of FOL, our representation of the hierarchy of states and transitions keeps the entire representation of the model within a decidable fragment of logic.

Supporting rich datatypes native in SMT-LIB. One of the advantages of using SMT-LIB as a base language is that our models with hierarchical states can include operations on rich 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 state hierarchy with 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.

Supporting variable semantics. We are pursuing the goal of applying deductive reasoning to these models while accomodating the variety of semantics that modellers employ for very similar syntax. We have created a 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.

Our contribution in this paper is a format in SMT-LIB for representing hierarchical state machines that 1) explicitly represents the syntax of the hierarchical state structure; 2) allows the use of general FOL on transition guards and actions; and 3) represents the state structure within a decidable fragment of FOL, and therefore does not add complexity to the analysis problem. Because we have formulated the syntax of the state hierarchy only, the semantics of the language can be described separately in FOL allowing users to choose their semantics for the model, rather than having the semantics integrated into the model.

Related Work. There are many existing standards for representing the syntax of models of hierarchical state machines. We created Composed Hierarchical Transition Systems (CHTSs) [18] to represent the syntax of such models 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 work differs from these approaches in that it is focused on representing HSMs within an existing logic for analysis, but without choosing a fixed semantics (as is the case in fUML), and also allowing general FOL for specification of transition guards and actions.

2. BACKGROUND

SMT-LIB. A satisfiability modulo theories (SMT) solver, for short SMT solver, is an automatic tool to check the satisfiability of a set of formulae in many-sorted FOL [4]. An SMT solver differs from a general-purpose FOL satisfiability checker in one major way: if a built-in type such as integers is used in a formula, the SMT solver considers only the standard interpretation for that type and the defined operations over it. SMT-LIB is a standard notation that state-of-the-art SMT solvers accept as input [3]. A specification of a problem in SMT-LIB consists of four parts: 1) declarations of user-defined types using the keyword **declare-sort**, 2) declarations of functional symbols used in the model[2] using the keyword **declare-fun**, 3) definitions that are used to simplify the model using the keyword **define-fun**, and 4) a set of assertions, where each assertion is a formula. All functions in SMT-LIB are total. SMT-LIB does not distinguish between terms and formulae. A formula is a term of type `Bool`. To ease the parsing of SMT-LIB specifications by SMT solvers, each SMT-LIB specification is a sequence of S-expressions.

[2]A relational symbol is a functional symbol of type `Bool`.

HSMs. At its most basic level, a behavioural model describes a set of traces of configurations. A configuration is a mapping from variables to values. 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 named states (sometimes called modes). Transitions originate and end at states and have labels. We consider models that are hierarchical state machines with And, Or, and Basic states[3]. Each transition has a source state, a destination state, a name, and optionally an event, a guard condition, an action, and a generated event. Transitions are labelled in the form:
transname: event [guard] / action ^generated_event
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, the event occurs, and the guard on that transition's label is true of the current configuration, then in the next configuration the source state of the transition is replaced by its destination state and the action labelling the transition has taken effect. The complications in the semantics for these languages (which give rise to the variations in this family of languages) come in how the states in the current configuration effect the set of transitions chosen to be executed in a step.

3. SIMPLE EXAMPLE

In this section, we give an example of our proposed format for representing HSMs in SMT-LIB. We use a deep embedding of the state hierarchy, but a shallow embedding of the guards and actions of the transitions. The consequence of using a shallow embedding is that supporting the creation of separate semantic functions requires some auxiliary functions, which we illustrate in this example and describe further in the next sections.

Figure 1 is an simple example of a heating system model with hierarchical states. Figure 2 shows how we represent the state hierarchy through function definitions in SMT-LIB. Each state name is declared as a constant of sort **_State**. The state name **_root** exists for all models. We also require an assertion that all state constants represent distinct state names. The state hierarchy is introduced through a set of definitions stating the kind of each state (Basic, And, or Or), the parent of each state, and the default state of each state. Because all functions in SMT-LIB are total, to represent partial functions, we introduce constants that represent no value. Thus, the parent of the root state is defined to be **_no_state**. **ite** is a keyword in SMT-LIB for the if-then-else function.

[3]Other kinds of states could be added.

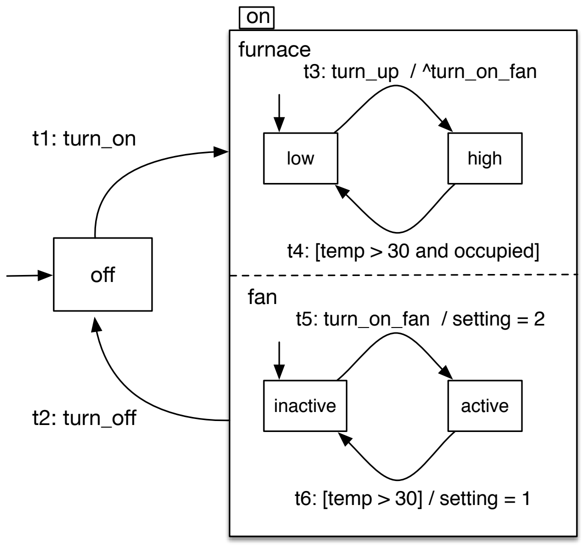
on
furnace
t3: turn_up / ^turn_on_fan
low
high
t4: [temp > 30 and occupied]
t1: turn_on
off
fan
t5: turn_on_fan / setting = 2
inactive
active
t6: [temp > 30] / setting = 1
t2: turn_off

Figure 1: Simple Heating System Model

```
; declare every state name
(declare-fun off () _State)
(declare-fun on () _State)
(declare-fun furnace () _State)
(declare-fun high () _State)
(declare-fun low () _State)
(declare-fun fan () _State)
(declare-fun active () _State)
(declare-fun inactive () _State)
(assert (distinct _root off on furnace high
    low fan active inactive _no_state))

; represent the state hierarchy
(define-fun _kind ((s _State)) _Kind
  (ite (or (= s _root) (= s furnace) (= s fan)) _or
  (ite (= s on) _and
  (ite (or (= s off) (= s high) (= s low)
          (= s active) (= s inactive)) _basic
      _no_kind))))

(define-fun _parent ((s _State)) _State
  (ite (= s _root) _no_state
  (ite (or (= s off) (= s on)) _root
  (ite (or (= s furnace) (= s fan)) on
  (ite (or (= s high) (= s low)) furnace
  (ite (or (= s active) (= s inactive)) fan
      _no_state))))))

(define-fun _default ((s _State)) _State
  (ite (= s _root) off
  (ite (= s furnace) low
  (ite (= s fan) inactive
      _no_state))))
```

Figure 2: Representing State Hierarchy in SMT-LIB

The statements representing the transitions of our simple model in SMT-LIB are shown in Fig 3. The names of the transitions are declared as distinct constants of sort **_Tran**. Each part of the transition is stated in a separate definition. The source and destination states of each transition are defined in the functions **_src** and **_dest** using conditionals on the transition names. The source and destination states can be anywhere in the state hierarchy.

```
; declare every transition name
(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))

; declare basic events
(declare-fun turn_on () _Event)
(declare-fun turn_off () _Event)
(declare-fun turn_up () _Event)
(declare-fun turn_on_fan () _Event)
(assert (distinct turn_on turn_off turn_up
            turn_on_fan _no_event))

(define-fun _src ((t _Tran)) _State
  (ite (= t t1) off
  (ite (= t t2) on
  (ite (= t t3) low
  (ite (= t t4) high
  (ite (= t t5) inactive
  (ite (= t t6) active _no_state)))))))

(define-fun _dest ((t _Tran)) _State
  (ite (= t t1) on
  (ite (= t t2) off
  (ite (= t t3) high
  (ite (= t t4) low
  (ite (= t t5) active
  (ite (= t t6) inactive _no_state)))))))

(define-fun _event ((t _Tran)) _Event
  (ite (= t t1) turn_on
  (ite (= t t2) turn_off
  (ite (= t t3) turn_up
  (ite (= t t5) turn_on_fan _no_event)))))

(define-fun _guard ((t _Tran)
                    (temp Int) (occupied Bool)
                               (setting Int) ) Bool
  (or (and (= t t4) (and (> temp 30) occupied))
      (and (= t t6) (> temp 30))
      (not (or (= t t4) (= t t6)))))

(define-fun _action ((t _Tran)
                     (temp Int) (occupied Bool)
                     (setting Int)
                     (temp_n Int) (occupied_n Bool)
                     (setting_n Int))
                    Bool
  (or (and (= t t5) (= setting_n 2))
      (and (= t t6) (= setting_n 1))
      (not (or (= t t4) (= t t6)))))

; per configuration element
; does a transition constrain it?
(define-fun _change_temp ((t _Tran)) Bool
    false)
(define-fun _change_occupied ((t _Tran)) Bool
    false)
(define-fun _change_setting ((t _Tran)) Bool
    (or (= t t5) (= t t6)))

(define-fun _gen_event ((t _Tran)) _Event
    (ite (= t t3) turn_on_fan _no_event))
```

Figure 3: Representing Transitions in SMT-LIB

We have deeply embedded the state hierarchy in SMT-LIB, meaning that the state names and transition names are new sorts and semantic functions can be written to compare elements of these sorts. However, we want the guards and actions of a transition to be shallowly embedded in SMT-LIB. The advantage of using a shallow embedding is that the expressions can use sorts native to SMT-LIB (*e.g.,* integers, arrays) without any additional infrastructure (such as declaring the sort and creating constructors and operators of the sort). The definition of the function **_guard** takes a transition name as an argument and a vector of the variables manipulated by the model (a configuration). This function returns a Boolean result. For example, the term

_guard (t1 temp occupied setting)

is true when the guard of transition **t1** is true in the configuration represented by the vector **temp occupied setting**.

The function **_action** takes a transition name, the current configuration (a vector of variables), and the next configuration (variables with **_n** appended) as arguments. It returns true when the action of the transition has happened, *i.e.,* the relationship between the current and next configuration variables used in the action holds. Each transition action changes only some of the variables of the model. The semantic functions will need to ensure that all configuration variables are constrained according to the model's semantics, which likely include the rule that a unchanged variable retains its previous value. Thus, we need to provide a function that says which transitions change the values of each variable in the configuration. This information is described in a **_change_** function for each variable.

We use a deep embedding for events. Basic events are declared as constants of sort **_Event** and required to be distinct from each other. The function **_event** takes a transition and returns the event guarding that transition (possibly none). The function **_gen_event** takes a transition and returns the event generated by the transition (possibly none).

4. DEFINITION OF REPRESENTATION

For our representation format, there is a generic part and a part specific to each model. Figure 4 show the sorts and functions that are the same for every model in our format. Figure 5 is a template for the functions that must be declared/defined for an individual model.

The syntax of the representation must conform to SMT-LIB. The order of these elements is not fixed except that definitions/declarations must occur before their use as in SMT-LIB. 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.

Our format requires the following:

- Declarations of constants for state names, transition names, and event names.
- Assertions that all state names are distinct, all transition names are distinct, and all event names are distinct.
- 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.

A modeller can use more definitions than those found in Figures 4 and 5 to create the model. The guards and actions of transitions can be any terms in SMT-LIB that match the sorts required by our template.

We expect that it would be easy to translate from another representation of the syntax to this format and that the user would do any editing of the model in its original format. Transition names are the only elements that might not be present in the original form of the model. These can be added in the translation process to give every transition a unique name.

The independent semantic functions will be written to use the sorts and functions in our format as arguments.

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

Figure 4: Generic Part of Representation Format

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

; represent the state hierarchy
(define-fun _kind ((s _State)) _Kind ...)
(define-fun _parent ((s _State)) _State ...)
(define-fun _default ((s _State)) _State ...)

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

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

(define-fun _src ((t _Tran)) _State ...)

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

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

(define-fun _guard ((t _Tran)
                    <variables of configuration> )
                    Bool
                ...)

(define-fun _action ((t _Tran)
                     <variables of configuration>
                     <copy of variables of configuration>)
                     Bool
                ...)

(define-fun _gen_event ((t _Tran)) _Event ...)

; per configuration element
; does a transition constrain it?
(define-fun _change_var ((t _Tran)) Bool ...)
...
```

Figure 5: Template of Representation Format

5. DESIGN DECISIONS

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

Deep vs Shallow Embeddings: We use a representation format where the state hierarchy and transition names are deeply embedded in SMT-LIB using datatypes that we create, and the guards and actions of transitions are shallowly embedded.

By using a deep embedding of the state hierarchy and transition names, we allow the semantic functions to describe separately the language's rules for which set of transitions should be taken in a step. These rules are where the variation points exist in the semantics of this family of languages and these rules can be used in deductive reasoning.

A deep embedding of the guards and actions of transitions would have required us to define a syntax for the language of these labels of the transitions, whereas a shallow embedding allows the model to take full advantage of the expressiveness of the underlying logic. However, using a shallow embedding means that semantic functions cannot access the contents of the guard or action, only its effect. Thus, we need to include in our representation the right set of information so the independent semantic functions can still define the overall meaning of the model.

Drawing on our past experience with embedding languages in logics [8], we recognize that the semantic functions will need to create a next configuration relation between a current and a next configuration. This configuration is a vector of variables. The semantic functions will take two configurations and use the following interface to the guards and actions of the transitions to create the next configuration relation:

- Test if a guard is true/false in a configuration of the model. Thus, we require the **_guard** function in our representation to take a transition name and a configuration and return a Boolean.
- Assert that the action of a transition occurs in a step. Thus, we require the **_action** function to take a transition name and two configurations and return a Boolean.
- Assert constraints on the entire set of variables in a configuration. Usually, in this family of languages, a variable that is not changed in a step must retain its previous value. Thus, we require the **_change** function for each variable to determine which transitions have actions that change that variable. Since the set of transitions is finite, from this information (and the set of configuration variables), the semantic functions can deduce when a configuration variable is not changed in a step.

In our current representation, the vectors of configuration elements are passed as arguments to the functions. Records would make writing these function definitions more convenient, but SMT-LIB does not yet support records[4].

The independent semantic functions will take two configurations as arguments and use the semantics of the language to determine which set of transitions are taken and enforce their constraints using the above interface to the guards and actions of the transitions. A limit of our chosen form of representation is that the semantic functions 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.

[4]Support for records is proposed for SMT-LIB 2.6.

Definitions vs Axioms: The usual way to define a state hierarchy in a deep embedding of a language is to use a recursive datatype. Since we cannot use a recursive datatype definitions in SMT-LIB[5], we formalize accessor functions to describe all the information that would be found in a recursive datatype for the state hierarchy. The meaning of these accessor functions could have been described by definitions (as we chose), or assertions such as, `(assert (= (_kind (_root)) _or))`. The axiomatization approach would require the use of a quantifier in the axiom to express the default case of the definition, *e.g.,* that any other states have the **_kind** of **_no_kind**. These quantified asssertions would move our representation of the state hierarchy outside of a decidable fragment of FOL. The definitional approach that we chose allows the representation of the state hierarchy to stay within a decidable fragment of FOL, namely the logic of uninterpreted functions [1]. As long as the guards and actions of the transitions are decidable then the whole model stays within a decidable fragment, and representing the state hierarchy has not increased the complexity of the verification problem.

Functions in SMT-LIB are total, thus we faced the classic issue of how to handle describing partial functions, such as the **_default** function when every state does not have a default state. SMT-LIB does not have an Option type (often available in functional programming languages), so we 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 (*e.g.,* there are no states besides those declared)[5]. Such a universally quantified axiom may be required for some kinds of analysis.

Events: In our study of language variants [12], we found that this family of languages has a range of ways of describing events, which are instantaneous occurrences to which the system reacts. We deeply embed events in the SMT-LIB representation, but the modeller can add functions that create events (*e.g.,* entered(state)) and event expressions (*e.g.,* a disjunction of events). 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]The SMT-LIB community is discussing standardizing a format for algebraic datatypes (enumerative and recursive) and decision procedure support for them, which may remove the need for some of these definitions and axioms.

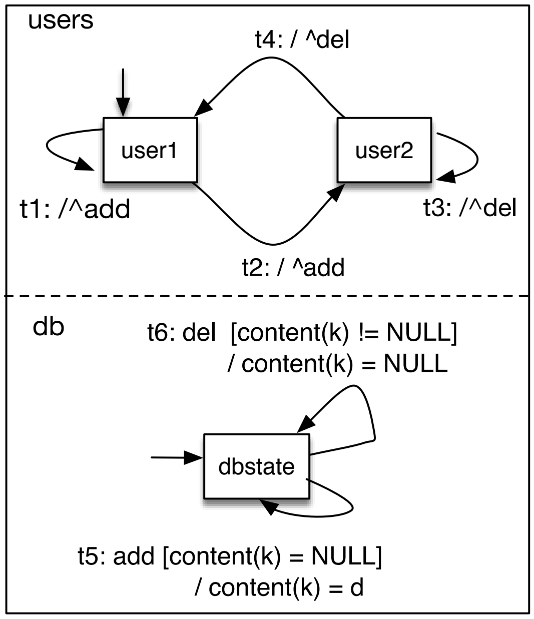
users
t4: / ^del
user1
user2
t1: /^add
t3: /^del
t2: / ^add
db
t6: del [content(k) != NULL] / content(k) = NULL
dbstate
t5: add [content(k) = NULL] / content(k) = d

Figure 6: Database Model

6. EXAMPLE: INTEGRATING CONTROL WITH RICH DATATYPES IN MODELS

By using a shallow embedding in SMT-LIB of the guards and actions of transitions, we have the ability to write abstract models (similiar to those written in Alloy [15]) that also have state hierarchy. We can therefore create models that are rich in both control and data structures.

Figure 6 is a simple model of a database that has two users and has both interesting datatypes and state hierarchy. One user can add items to the database and the other delete items, both by issuing events to which the database reacts. Figure 7 shows an interesting fragment of the SMT-LIB representation of this model, which uses uninterpreted sorts and functions. Because a configuration cannot include a function type[6] (*i.e.,* the function representing a database mapping keys to values), we represent this function using three uninterpreted types: **DB**, **Data**, and **Key**. A function **content** maps a database and a key to data.

7. FUTURE WORK

There are two major directions for future work before the usefulness of our proposed representation format can be established.

Analysis. We motivate the need for a representation of the state hierarchy in SMT-LIB with the idea that the deductive-based reasoning of SMT-solvers can use the state structure in analysis. In search-based reasoning for model-checking, each configuration must be examined until a counterexample is reached or all configurations have been explored. In symbolic-based reasoning for model checking (*e.g.,* binary decision diagrams or bounded model checking), one propositional logic formula can represent many configurations, reducing both the memory required and the time taken to examine all configurations. In our recent work, we showed that for a subset of CTL, called CTL-Live (which includes liveness properties but not safety), the entire model checking problem can be encoded in first-order logic and therefore given to an SMT-solver in one call [22, 23]. SMT-solvers include decision procedures that exploit knowledge regarding particular data structures prior to reducing the problem to a SAT problem. In the context of model checking of modelling languages based on hierarchical state machines, the state hierarchy is a type of data ready to be used to improve the analysis performance similar to other datatypes. We chose our representation of the syntax to be compatible with a set of first-order semantic functions very similar to those described in Esmaeilsabzali and Day [11]. These semantic functions declaratively describe the meaning of the syntax and can be used by a deductive reasoning tool in analysis. For example, a typical semantic rule is that configurations that include an And state must include a child state from all of the components of the And state. Usually, in translations to analysis tools 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. The goal of directly representing the state structure

[6]This is because quantification over an element of type function would be higher-order quantification. Quantification over the elements of the configuration is required for model checking analysis.

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

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

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

; uninterpreted function represents
; the contents of the database
; content: DB x Key -> Data
(declare-fun content (DB Key) Data)

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

(define-fun _gen_event ((t _Tran)) _Event
    (ite (or (= t t1) (= t t2)) add
    (ite (or (= t t3) (= t t4)) del
         _no_event)))

(define-fun _guard ((t _Tran)
                    (k Key) (d Data) (c DB)) 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)))))

(define-fun _action ((t _Tran)
                     (k Key) (d Data) (c DB)
                     (k_n Key) (d_n Data) (c_n DB))
                     Bool
    (or (and (= t t5) (= (content c_n k) d))
        (and (= t t6) (= (content c_n k) NULL)
        (not (or (= t t5) (= t t6))))))
```

Figure 7: Fragment of SMT-LIB of Database Model

in these models is to allow deductive reasoning to exploit this structure.

Translators. SMT-LIB is verbose and the standard format that we are proposing would not be written by hand. Rather we need translators from current representations (textual or graphical within an IDE) to produce a model in the SMT-LIB format. Because the structure of the state hierarchy is captured in the SMT-LIB, the translation should be reversible, *i.e.,* there is a 1-1 relationship between the two representations. 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 the meaning of these elements will either have to be incorporated into the translation process or our format would have to be extended to accommodate them.

8. CONCLUSION

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 hierarchical states in SMT-LIB. The value in our contribution is that:

1. It represents the state hierarchy explicitly in SMT-LIB so that it can be exploited in deductive reasoning without increasing the complexity of the problem (*i.e.,* our format stays within a decidable fragment of FOL).
2. It allows for the creation of abstract models rich in both control and data structures.
3. It permits the representation of models in a family of languages because the semantic functions can be expressed separately.

We accomplished the above through a combination of deep and shallow embeddings. We motivated our decisions for the sorts and functions used in the representation based on wanting the flexibility to include interesting datatypes and allowing semantic functions for the language to be described independently of the model. 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.

9. ACKNOWLEDGMENTS

We thank the reviewers for their insightful comments. Our work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

10. REFERENCES

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

[2] C. Barrett, C. L. Conway, M. Deters, et al. CVC4. In *CAV*, LNCS, pages 171–177. Springer, 2011.

[3] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

[4] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

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

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

[7] R. Cavada, A. Cimatti, et al. The nuXmv symbolic model checker. In *CAV*, pages 334–342, 2014.

[8] N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, University of British Columbia, 1998.

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

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

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

[12] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu. Deconstructing the semantics of big-step modelling languages. *REJ*, 15(2):235–265, 2010.

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

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[15] D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

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

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

[18] J. 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] Object Management Group (OMG). OMG unified modeling language (OMG UML), v2.5, 2015.

[22] A. Vakili and N. A. Day. Reducing CTL-Live model checking to first-order logic validity checking. In *FMCAD*, pages 215–218. IEEE, 2014.

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

[24] M. 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.