Action Language: A Specification Language for Model Checking Reactive Systems

Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
(805) 893 3735
bultan@cs.ucsb.edu

ABSTRACT
We present a specification language called Action Language for model checking software specifications. Action Language forms an interface between transition system models that a model checker generates and high level specification languages such as Statecharts, RSML and SCR—similar to an assembly language between a microprocessor and a programming language. We show that Action Language translations of Statecharts and SCR specifications are compact and they preserve the structure of the original specification. Action Language allows specification of both synchronous and asynchronous systems. It also supports modular specifications to enable compositional model checking.

Keywords
Reactive systems, specification languages, model checking.

1 INTRODUCTION

Developing reliable software for reactive systems is one of the most challenging goals in information technology. A reactive system is one which interacts with its environment continuously without terminating [22]. Typical examples are concurrent programs, embedded systems, protocols, and hardware systems. Classical notions of program verification such as partial correctness and termination are not suitable abstractions for characterizing properties of such systems. Properties of reactive systems involve notions such as invariance (a property holds all the time) and eventuality (some property eventually becomes true, for example, a request is eventually served). Most safety critical systems, such as control software for embedded systems, are reactive. Also increasing use of technologies such as Internet and mobile computing which involve mostly reactive software components implies that developing software for reactive systems will be a key part of software engineering practice in the future. With the integration of such systems to everyday life, their reliability will become increasingly important.

This work was supported in part by NSF grant CCR-9970976.

Recently, research in two areas addressed the problem of developing reliable software for reactive systems from two different directions:

- Software specification languages for reactive systems: Specification languages such as Statecharts [15], RSML (Requirements State Machine Language) [20], and SCR (Software Cost Reduction method) [16] target reliable software development for safety critical systems such as airline traffic control and nuclear power plants. The main motivation is to provide languages for formal and unambiguous statement of software specifications so that the bugs will be exposed before they creep into the implementation.
- Automated verification methods for reactive systems: In recent years, there has been a surge of progress in automated verification methods for reactive systems. In areas like hardware design, these technologies are rapidly augmenting key phases of testing and validation [12]. To date, one of the most successful of these methods has been model checking [10]. Given a transition system and a temporal property, model checking procedures exhaustively search the state space of the input transition system to find out if it satisfies the given temporal property.

Bugs in software specifications are very costly to fix if they are found late in the design cycle. It is critical to provide tools for analyzing specifications so that the errors can be fixed as early as possible. Specification languages such as Statecharts, RSML and SCR help designers in stating the specifications of software systems precisely and unambiguously. Formal semantics of these languages make it possible to use automated verification techniques such as model checking in analyzing properties of specifications. The state-space of a software spec-

ification (defined by the semantics of the specification language used) can be explored using model checking to verify or falsify its properties [2].

Model Checking Software Specifications
There are two main approaches to model checking, 1) symbolic model checking based on CTL (Computation Tree Logic, a branching time temporal logic) [8], and 2) explicit state model checking based on LTL (Linear time Temporal Logic) [24]. The important difference between these two approaches is the verification technique used. Although expressive powers of LTL and CTL are not comparable, basic safety and liveness properties can be expressed in both of them. Two popular model checking tools which implement these techniques are SMV (a symbolic model checker) [21] and SPIN (an explicit state model checker) [18].

SMV is based on a data structure called Binary Decision Diagrams (BDDs), which can encode boolean functions in a highly compact format [5]. The main idea in BDD based model checking is to represent sets of system states and transitions as boolean formulas, and manipulate them efficiently using the BDD data structure [8, 21]. This type of model checking is called *symbolic* since the system states are represented implicitly by the BDD data structure during the state space search.

SPIN on the other hand is an explicit state model checker based on an automata theoretic approach. SPIN converts both the negation of the input temporal property and the input transition system to finite automata. (In order to model the nonterminating behavior of reactive systems, both the property and the system are represented as finite automata that accept infinite words.) Then, SPIN generates the product automaton which represents all the behaviors of the input system which can violate the input LTL property. Using an efficient depth first search algorithm it searches for accepting runs of the product automaton which correspond to violating behaviors.

Recently, both SPIN and SMV have been used for analyzing software specifications with encouraging results [9, 17, 23]. Researchers who used SPIN and SMV to verify software specifications followed the steps shown in Figure 1. To check specifications written in languages such as Statecharts, RSML, or SCR, researchers have translated such specifications to Promela (the input language of SPIN) or to the input language of SMV. Most of the time these translations are not automatic but require simplifications and abstractions. Once the specifications are translated to its input language, a model checker automatically generates a transition system model for the system to be verified.

Specification languages such as Statecharts, RSML, and SCR were not designed to be analyzed by model checkers. Rather, their mission is to provide tools for precisely stating software specifications. On the other hand, input language of SMV was designed for model checking synchronous hardware systems [21]. Hence, for example, the translation between Statecharts and SMV could change the structure of the original specification and make it difficult for a user to see the correspondence. Since scalability of model checking is limited because of the state space explosion problem, we think that it will be crucial to have some user input (for example in forming abstractions [11] or deciding the order of compositions in compositional analysis [6]) during the model checking of large systems. Hence the user should be able to understand the language that the model checker uses and see how it relates to the original specification.

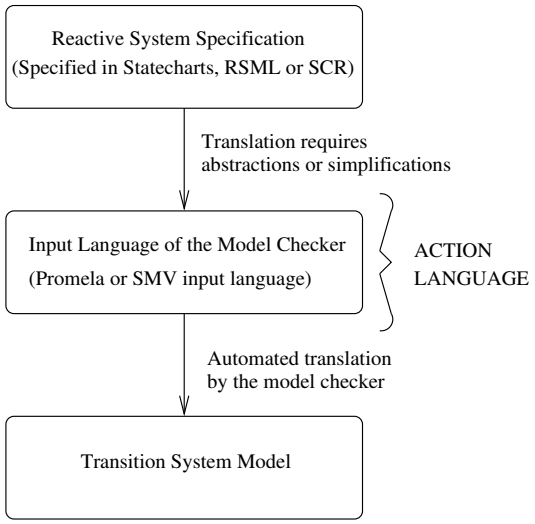

Figure 1: Translation of reactive system specifications for model checking

We are proposing a language, called Action Language, which will bridge the gap between transition system models and high level specification languages (Figure 1). It can be viewed as an assembly language which will form an abstraction layer between software specification languages and the transition system models required for model checking. Action Language specifications are readable—a person who understands a high level specification can easily understand its Action Language translation. Also someone who understands the transition system models used for model checking can easily understand the transition system that an Action Language specification would generate.

Another important property of the Action Language is that it supports both synchronous and asynchronous composition as basic operations. In Action Language synchronous systems can be expressed as easily as asynchronous systems and visa versa.

We show how Statecharts and SCR specifications can be translated to Action Language. Both of these languages have been used for model checking software specifications. We show that their translation to Action Language is compact and the Action Language translations preserve the original form of the specifications.

We will use Action Language as the input language of our composite model checker [7]. In the composite model checking approach each variable in the input system is mapped to a symbolic representation type. Our composite model checker maps boolean and enumerated variables to BDD representation, and integers to arithmetic constraint representation. Each atomic action in the input system is conjunctively partitioned where each conjunct specifies the effect of the action on the variables mapped to a single symbolic representation. Conjunctive partitioning of the atomic actions allows pre- and post-condition computations to distribute over different symbolic representations. A composite model checker is limited by the variable types that can be represented by its symbolic representations. To make further extensions possible, we define the Action Language as a family of languages, the variable types can be extended based on the types the model checker supports.

Action Language is similar to Temporal Logic of Actions (TLA) [19]. As in TLA, in Action Language a system is specified using logical connectives. However, we deviate from a pure logical semantics to make asynchronous and synchronous compositions more readable. Also Action Language does not use temporal operators to specify the behaviors of systems as in TLA. Instead we use a dual language approach and use temporal logic to specify the properties of Action Language specifications.

Reactive Module Language is a specification language for hardware-software codesign and verification [1]. It is designed to support model checking. Although the target applications for Reactive Module Language and Action Language are different, the goal of bridging the gap between specifications and model checkers is the same for both of them.

We start with an overview of the Action Language in section 2. We discuss the variable types, the actions and the modules in the action language on a simple producer-consumer system specification. We present the asynchronous and the synchronous composition constructs in Action Language and discuss their semantics. In Section 3 we show translation of a Statecharts specification and a SCR specification to Action Language. Finally, in Section 4 we state our conclusions.

2 ACTION LANGUAGE

Action Language is a specification language for reactive systems. Semantics of an Action Language specification is given as a transition system $TS = (S, I, R)$ where S is the set of states, $I \subseteq S$ is the set of initial states and $R \subseteq S \times S$ is the transition relation. The set of states S of an Action Language specification is defined by the Cartesian product of the domains of its variables.

```
module producer_consumer
  integer produced, consumed, count;
  parameterized integer size;
  initial :  produced = consumed = count = 0;
  restrict : size >= 1;
  producer : count < size & count' = count + 1
    & produced' = produced + 1;
  consumer : count > 0 & count' = count - 1
    & consumed' = consumed + 1;
  producer_consumer : producer | consumer;
  spec : invariant(produced - consumed = count
                   & count <= size);
endmodule
```

Figure 2: A simple producer-consumer system specification in Action Language.

An Action Language specification consists of a set of modules. Semantics of each module is defined as a transition system. In Figure 2 we show a simple Action Language specification of a producer-consumer system which consists of a single module `producer_consumer`.

Each module starts with a set of variable and module declarations. Module declarations are used to declare submodules. (Recursive module definitions are not allowed.) For example in Figure 3 we show a modular specification of the same producer-consumer system. Modules `producer` and `consumer` are submodules of module `producer_consumer` in Figure 3.

Action Language allows declaration of parameterized variables such as `size` in Figure 2. We assume that the value of a parameterized variable is unspecified and stays constant throughout the execution of the system.

Declarations are followed by an initial condition statement which specifies the set of initial states of the system. A restrict statement can be used to restrict the state space. For example, in module `producer_consumer` in Figure 2 the value of `size` is restricted to be greater than or equal to 1 using the restrict statement.

The transition relation of a module is defined using a set of actions. For example in Figure 2, module `producer_consumer` has two action specifications `producer` and `consumer`. The last action is a module specification and it must have the same name as the module. This action defines the overall transition relation of the module.

In addition to its actions, a module specification can also refer to its submodules. For example in Figure 3,

```
module producer
  integer produced;
  shared integer count;
  shared parameterized integer size;
  initial : produced = 0;
  producer : count < size & count' = count + 1
    & produced' = produced + 1;
  spec : invariant(0 <= produced & produced <= size);
endmodule

module consumer
  integer consumed;
  shared integer count;
  shared parameterized integer size;
  initial : consumer = 0;
  consumer : count > 0 & count' = count - 1
    & consumed' = consumed + 1;
endmodule

module producer_consumer
  module producer, consumer;
  shared integer count;
  initial : count = 0;
  producer_consumer : producer | consumer;
  spec : invariant(produced - consumed = count
                       & count <= size);
endmodule
```

Figure 3: A modular Action Language specification of the same producer-consumer system.

the module definition for `producer_consumer` uses its two submodules `producer` and `consumer`.

After the module specification, temporal properties of the system are specified. The `invariant` operator used in Figures 2 and 3 is equivalent to CTL operator AG. We provide both CTL operators (so that complex temporal properties can be expressed) and also more user friendly operators such as `invariant` and `eventually` which can easily be translated to CTL.

Let VAR(m) denote the variables declared in module m and its submodules. A state of module m corresponds to a valuation of all the variables in VAR(m). The state space S of the module m is defined by the Cartesian product of the domains of the variables in VAR(m) intersected with the states which satisfy the restrict clause. The initial states (I) are the states in S which satisfy the initial clause of the module and initial clauses of its submodules.

A variable in Action Language could be local, shared (other modules which declare it can both read and write access it), exported (giving read access to other modules), or imported (it can be read accessed). (Similar constructs have been used for real-time systems in ASTRAL [13].) A module can access variables of its submodules in its property specifications. For example, in Figure 3, module `producer_consumer` can use all the variables of its submodules `producer` and `consumer` in its property specifications.

Variable Types

We are proposing Action Language as a tool for model checking. Hence our goal is to design the Action Language so that the systems specified in it are amenable to automated verification. Complexity of model checking procedures increase with the increasing size of the state space of the input system. It is well-known that the main obstacle in scalability of model checking is the state space explosion problem. State space explosion has two causes 1) the exponential increase in the state space by the increasing number of variables, 2) the exponential increase in the state space by the increasing number of concurrent components. Since we define the state space of an Action Language specification as the Cartesian product of the domains of its variables, it is clear that both the number of variables and the type of variables will effect the size of the state space. Hence, if we want Action Language specifications to be verifiable we have to be careful in defining the variable types.

We define Action Language as a family of languages which is parameterized by the variable types in it. This is motivated by our composite model checking approach [7]. Recently, various symbolic representations have been proposed for model checking which provide efficient encodings for various variable types. Variable types also effect the efficiency of the search techniques used in explicit state model checking. We define Action Language as a family of languages so that it can be customized based on the capabilities of a given model checker.

The simplest Action Language is the one with only boolean variables. Boolean variables can have two values `true` and `false`. Boolean expressions in Action Language are constructed using boolean variables, boolean constants `true` and `false` and boolean operators `!` (negation), `&` (conjunction), `|` (disjunction), `->` (implication), and `<->` (equivalence). We assume that every variable type in Action Language supports the equality predicate `=`.

Any language with only boolean variables is a good candidate for BDD based verification. However, such a restriction on variable types would mean that we have to represent each variable using a binary encoding. This would make Action Language specifications unreadable in most cases. Enumerated variables can easily be handled with both symbolic (using a binary encoding) and explicit state model checkers. Expressions on enumerated variables are constructed using the equality predicate `=`, the values of the enumerated type, and the enumerated variables.

Next we consider the integer type. Integers cause two

complications in model checking 1) if we allow unbounded integer variables model checking becomes undecidable, 2) there are no efficient representations for symbolic model checking of systems with nonlinear constraints. The first problem can be resolved using either conservative approximation techniques or bounding the domains of integer variables. For the second problem we currently do not know any good solution. Since we plan to use Action Language as the input language for our composite model checker, we do not allow nonlinear constraints in Action Language. (This restriction can be lifted if one uses an explicit state model checker such as SPIN.) Hence, an integer expression in Action Language is composed of logical operators `!`, `&`, `|`, `->`, `<->`, arithmetic operators `+`, `-`, multiplication by a constant; arithmetic predicates `<=`, `<`, `=` `>=`, and `>`; and integer variables and integer constants.

In Action Language expressions on different variable types can be combined with logical operators `!`, `&`, `|`, `->` and `<->` to form mixed-type expressions. A *simple expression* in an Action Language module m consists of (possible mixed-type) expressions on its variables ($\text{VAR}(m)$). Simple expressions are used for specifying the initial clause, the restrict clause and the atomic properties of the temporal logic used for specifying temporal properties of the system.

Actions

In addition to everything that can appear in a simple expression, an *action expression* also includes primed variables. Primed variables correspond to *next state variables* and unprimed variables correspond to *current state variables*. Given a module m the variables that can appear in its action expressions can be partitioned into five sets: $\text{LOCAL}(m)$, $\text{SHARED}(m)$, $\text{PARAM}(m)$, $\text{IMPORTED}(m)$, and $\text{EXPORTED}(m)$, where

$$\begin{aligned}\text{VAR}(m) \;=\; & \text{LOCAL}(m) \cup \text{SHARED}(m) \cup \text{PARAM}(m) \\ & \cup \text{IMPORTED}(m) \cup \text{EXPORTED}(m)\end{aligned}$$

In an action expression for module m only variables in $\text{LOCAL}(m) \cup \text{SHARED}(m) \cup \text{EXPORTED}(m)$ can be primed. The reason is that the variables in $\text{PARAM}(m)$ stay constant by definition (i.e., if $v \in \text{PARAM}(m)$ we assume by default that $v' = v$) and values of imported variables can be only read accessed (i.e., if $v \in \text{IMPORTED}(m)$, v can appear in an action expression in m but v' cannot).

Actions can be combined using operators `&` and `|` to specify other actions. Operators `&` and `|` correspond to synchronous and asynchronous composition, respectively, as described below. We do not allow recursive action definitions, each action can only refer to the actions defined before it.

An action expression defines a transition relation on its state space. State space of an action may be different than the state space of the module that includes it.

Let $\text{VAR}(a)$ denote the set of variables of an action. These are current or next variables which either appear in the corresponding action expression, or are in $\text{VAR}(a_i)$ for some action a_i which appears in the corresponding action expression. $\text{DVAR}(a)$ denotes the current state variables in action a, and $\text{RVAR}(a)$ denotes the next state variables where $\text{VAR}(a) = \text{DVAR}(a) \cup \text{RVAR}(a)$. For example, for the action `producer` in Figure 2

$$\text{VAR}(\texttt{producer}) = \{\texttt{count}, \texttt{size}, \texttt{produced}\}$$

and $\text{DVAR}(\texttt{producer})$ and $\text{RVAR}(\texttt{producer})$ are equal to $\text{VAR}(\texttt{producer})$. Since parameterized variables have implicit constraints in the form $v' = v$ which preserve their values, they are also included in the range variables.

The state space S_a of action a is the Cartesian product of the domains of the variables in $\text{VAR}(a)$ intersected with the states which satisfy the restrict clause of the module. The transition relation $R_a \subseteq S_a \times S_a$ of a is defined by its action expression.

For example, the transition relation of the action `producer` in Figure 2 is any tuple ((`count`, `size`, `produced`), (`count'`, `size'`, `produced'`)) of integers which satisfies the following formula

$$\texttt{size} \geq 1 \;\wedge\; \texttt{count} < \texttt{size} \;\wedge\; \texttt{count'} = \texttt{count} + 1 \;\wedge\; \texttt{produced'} = \texttt{produced} + 1 \;\wedge\; \texttt{size'} = \texttt{size}$$

So, the semantics of the actions are derived from the expressions which define them with the interpretation that primed variables correspond to next state variables and unprimed variables correspond to current state variables.

Given an action a with the transition relation R_a, we denote its domain and range as

$$\begin{aligned}\text{DOMAIN}(a) &= \{\; s \;:\; \exists t \;,\; (s,t) \in R_a \;\} \\ \text{RANGE}(a) &= \{\; s \;:\; \exists t \;,\; (t,s) \in R_a \;\}\end{aligned}$$

Composition

In Action Language both the actions and the modules can be composed. Composition is achieved using logical connectives `&` (for synchronous composition) and `|` (for asynchronous composition). Semantics of composition follows immediately from semantics of logical connectives `&` and `|` if the state spaces of two actions or modules are the same. For example, given two modules m_1 and m_2, where $\text{SHARED}(m_1) = \text{SHARED}(m_2)$, $\text{PARAM}(m_1) = \text{PARAM}(m_2)$, and either module has no local, imported or exported variables, assume that the semantics of module m_1 is given by the transition system $TS_1 = (S, I_1, R_1)$ and the semantics of the module m_2 is given by the transition system $TS_2 = (S, I_2, R_2)$,

then the transition system that corresponds to synchronous composition m_1 & m_2 of these two modules is defined as

$$TS_{m_1 \ \& \ m_2} = (S, I_1 \cap I_2, R_1 \cap R_2).$$

Similarly, the semantics of asynchronous composition $m_1 \mid m_2$ is defined as

$$TS_{m_1 \ \mid \ m_2} = (S, I_1 \cap I_2, R_1 \cup R_2).$$

(Note that we always intersect the initial states of two modules to get the initial states of the composed system.)

Two actions can be composed similarly (they have to be in the same module). Given two actions a_1, a_2 we define the domain variables of their compositions as the union of the domain variables of a_1 and a_2, i.e.,

$$\text{DVAR}(a_1 \ \& \ a_2) = \text{DVAR}(a_1 \mid a_2) = \text{DVAR}(a_1) \cup \text{DVAR}(a_2)$$

We define the range variables of the compositions similarly,

$$\text{RVAR}(a_1 \ \& \ a_2) = \text{RVAR}(a_1 \mid a_2) = \text{RVAR}(a_1) \cup \text{RVAR}(a_2)$$

The state space of the composition is defined based on the union of the domain and the range variables.

Given two actions a_1 and a_2 if we have $\text{DVAR}(a_1) = \text{DVAR}(a_2)$ and $\text{RVAR}(a_1) = \text{RVAR}(a_2)$ (which implies that the compositions of a_1 and a_2 have the same domain and range variables as a_1 and a_2) then we have:

$$R_{a_1 \ \& \ a_2} = R_{a_1} \cap R_{a_2}$$
$$R_{a_1 \ \mid \ a_2} = R_{a_1} \cup R_{a_2}$$

Note that if we define the semantics of action composition based on logical connectives & (and) and | (or) this is exactly what we would get. One problem which prevents us from using a fully logical semantics is the specification of the behaviors of the variables which are not mentioned in the action, i.e., we have to decide what happens when the range variables of two actions are different. For example, action a_1 may restrict the next state of a variable v (i.e., $v \in \text{RVAR}(a_1)$) while the action a_2 may not have any constraints for the next state value of v (i.e., $v \notin \text{RVAR}(a_2)$). If we assume that when action a_2 is executed next state value of v can take any value, then our asynchronous composition rule will fail. What we really want from asynchronous composition is that a_2 should preserve the values of the variables if it does not explicitly change them, i.e., we should assume that there is an implicit constraint $v' = v$ in a_2.

Consider the two actions `producer` and `consumer` in Figure 2. The transition relation of `producer` and `consumer` are defined with the following formulas:

$$\texttt{size} \geq 1 \ \wedge \ \texttt{count} < \texttt{size} \ \wedge \ \texttt{count'} = \texttt{count} + 1$$
$$\wedge \ \texttt{produced'} = \texttt{produced} + 1 \ \wedge \ \texttt{size'} = \texttt{size}$$

and

$$\texttt{size} \geq 1 \ \wedge \ \texttt{count} > 0 \ \wedge \ \texttt{count'} = \texttt{count} - 1$$
$$\wedge \ \texttt{consumed'} = \texttt{consumed} + 1 \ \wedge \ \texttt{size'} = \texttt{size}$$

respectively. Note that the constraint $\texttt{size} \geq 1$ comes from the restrict clause and the constraint $\texttt{size'} = \texttt{size}$ is there since `size` is parameterized. Now consider the asynchronous composition of these two actions (`producer | consumer`) as in Figure 2. If we join the two formulas which define these two actions using a disjunction, we will get

$$(\texttt{size} \geq 1 \ \wedge \ \texttt{count} < \texttt{size} \ \wedge \ \texttt{count'} = \texttt{count} + 1$$
$$\wedge \ \texttt{produced'} = \texttt{produced} + 1 \ \wedge \ \texttt{size'} = \texttt{size})$$
$$\vee \ (\texttt{size} \geq 1 \ \wedge \ \texttt{count} > 0 \ \wedge \ \texttt{count'} = \texttt{count} - 1$$
$$\wedge \ \texttt{consumed'} = \texttt{consumed} + 1 \ \wedge \ \texttt{size'} = \texttt{size})$$

If we assume that this is the semantics of `producer | consumer` then note that when `producer` action is taken the value of `consumed` can take any arbitrary value since there is no constraint on it. This is not what we want asynchronous composition operator to do. What we want is

$$(\texttt{count} < \texttt{size} \ \wedge \ \texttt{count'} = \texttt{count} + 1$$
$$\wedge \ \texttt{produced'} = \texttt{produced} + 1 \ \wedge \ \texttt{size'} = \texttt{size}$$
$$\wedge \ \texttt{size} \geq 1 \ \wedge \ \texttt{consumed'} = \texttt{consumed})$$
$$\vee \ (\texttt{count} > 0 \ \wedge \ \texttt{count'} = \texttt{count} - 1$$
$$\wedge \ \texttt{consumed'} = \texttt{consumed} + 1 \ \wedge \ \texttt{size'} = \texttt{size}$$
$$\wedge \ \texttt{size} \geq 1 \ \wedge \ \texttt{produced'} = \texttt{produced})$$

We define the asynchronous composition operator so that this problem is resolved automatically without asking user to specify the frame constraints such as $\texttt{produced'} = \texttt{produced}$. As defined above, given two actions a_1 and a_2 the state space of $a_1 \mid a_2$ is defined based on $\text{VAR}(a_1 \mid a_2)$ which includes domain and range variables of both a_1 and a_2. We extend the transition relation of a_1 to the state space of $a_1 \mid a_2$ by conjoining it with a frame constraint $v' = v$ for each variable $v \in \text{RVAR}(a_2) - \text{RVAR}(a_1)$ (where $-$ is set difference). Similarly, we extend the transition relation of a_2 by conjoining it with a frame constraint $v' = v$ for each variable $v \in \text{RVAR}(a_1) - \text{RVAR}(a_2)$. Assume that R'_{a_1} and R'_{a_2} denote the transformed transition relations. Then, we define $R_{a_1 \ \mid \ a_2}$ as

$$R_{a_1 \ \mid \ a_2} = R'_{a_1} \cup R'_{a_2}$$

The rule for asynchronous module composition is defined similarly.

Similar to asynchronous composition, there is a caveat in the definition of synchronous composition & which prevents us from interpreting its semantics simply as conjunction. If we treat the synchronous composition as conjunction this will force users to specify a next state

behavior for each state of a synchronous component. Otherwise in each state where one of the synchronous components does not have any next state specified, the composed system would deadlock. To prevent this we assume that if there is a state where one of the components does not have any next state specified, then that component should make a synchronous transition where it stays in the same state (i.e., instead of causing a deadlock, the disabled component would have a self-loop which would allow other components to progress).

Given two actions a_1, a_2 with transition relations R_{a_1} and R_{a_2} we define two transition relations R'_{a_1} and R'_{a_2} as follows. Let I_{a_1} denote the relation over the variables in $\text{VAR}(a_1)$ which has a constraint $v' = v$ for each variable $v \in \text{RVAR}(a_1)$. (Similarly assume I_{a_2} denotes the corresponding relation for action a_2.) Let $\text{RESTRICT}(R, S)$ denote the relation formed by restricting the domain of relation R to set S. Then, we define R'_{a_1} as

$$R'_{a_1} = R_{a_1} \cup \text{RESTRICT}(I_{a_1}, (S_{a_1} - \text{DOMAIN}(a_1)))$$

i.e., R'_{a_1} adds transitions which preserve the values of range variables to states which do not have any next state in the definition of a_1. Similarly

$$R'_{a_2} = R_{a_2} \cup \text{RESTRICT}(I_{a_2}, (S_{a_2} - \text{DOMAIN}(a_2)))$$

Then, we define $R_{a_1 \ \& \ a_2}$ as

$$R_{a_1 \ \& \ a_2} = R'_{a_1} \cap R'_{a_2}$$

Verification

One could generate a transition system model $TS = (I, S, R)$ for an Action Language specification based on the definitions given above. Our composite model checker [7] can translate Action Language specifications to a composite representation which consists of arithmetic constraints and BDDs if the operations on the integer type are restricted as explained above. Our composite representation is able to represent integers with unbounded domains. If the domains are bounded, Action Language specifications can be translated to a pure BDD representation as in SMV or an explicit state representation as in SPIN.

We would like to point out some properties of the Action Language which supports modular verification. The variables of a module m which can be modified by other modules are given as $\text{IMPORTED}(m) \cup \text{SHARED}(m)$. The rest of the variables are either modified exclusively by m (i.e., variables in $\text{LOCAL}(m) \cup \text{EXPORTED}(m)$) or stay constant (i.e., variables in $\text{PARAM}(m)$). We can use this information in compositional model checking techniques for verifying properties of m. We can generate a transition system from module m's specification which will give an upper bound to the actual transition system that will be generated when we compose m with the modules defining the behaviors of variables in $\text{IMPORTED}(m) \cup \text{SHARED}(m)$. Now, assume that we want to verify an invariant of m. Our composite model checker would do this by computing the set of states which can violate the invariant. If during the search for the violating states we use an upper bound for the actual transition system, then we would compute and upper bound for the states which would violate the condition. If the initial states are not in this upper bound then we can conclude that the system satisfies the property.

Consider the property

```
invariant(0 <= produced & produced <= size)
```

for the module `producer` in Figure 3. This property can be verified using our composite model checker using only the specification of module `produced`. And this result will hold for any system which uses `produced` as a submodule. This idea can be generalized to full model checking of CTL if one uses both lower and upper bounds for the transition system of a module [14].

3 TRANSLATION TO ACTION LANGUAGE

In this section we will present two example specifications—one in Statecharts and the other one in SCR—and demonstrate their translations to Action Language.

Statecharts Translation

The Statecharts specification in Figure 4 is from [9] where it is translated to SMV input language. In Figure 5 we show its translation to Action Language. Note that (as in the SMV translation in [9]) we created one enumerated variable for each OR state in the Statecharts specification. The nice property of the Action Language specification is that each transition in the Statecharts representation can be represented by one action. Then, the overall system is exactly the expression which corresponds to combining OR states with asynchronous composition `|`, and AND states with synchronous composition `&`. Hence, the Action Language specification preserves the structure of the Statecharts specifications precisely. A user familiar with the Statecharts specification should be able to understand the translation easily. We believe this is not the case for the SMV translation given in [9]. SMV has a synchronous semantics whereas Statecharts specifications can specify synchronous and asynchronous transitions at any level. Action Language has the same property, it does not favor asynchronous or synchronous compositions, they can both be expressed easily. Additional constraints such as the synchrony hypothesis [9] can also be modeled in Action Language.

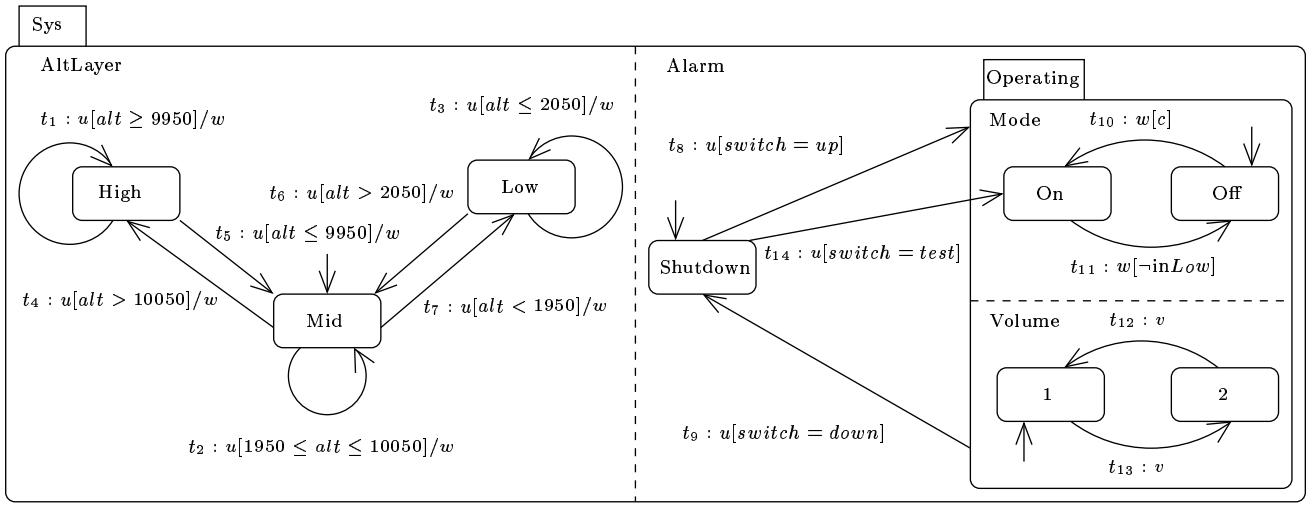

Figure 4: Statecharts example

SCR Translation

In Figure 6 we show an SCR specification given in [3, 4]. In Figure 7 we give its translation to Action Language. Note that each row of the mode transition table for mode class `Pressure` is represented by an action (`p1`, `p2`, `p3`, `p4`) Their asynchronous composition gives the semantics of the whole mode transition table. Each column of the event table for `Overridden` is also represented with an action (`o1`, `o2`), and overall table is represented by composing them. The condition table for `Inject` is translated similarly. Then the overall composition corresponds to the synchronous composition of these tables. In [3] this specification is converted to Promela and verified using SPIN model checker. We think that the Action Language translation of SCR exposes the semantics of the specification better than the Promela specification. In fact Action Language specification is similar to the logical semantic definition used in [16] to explain the semantics of SCR specification.

We can also put the one input assumption [16] (the assumption that only one monitored variable can change at a time) in the Action Language specification by restricting the behavior of variables `Block` and `Reset`.

Both the SCR translation and Statecharts translation can be made modular using submodules. The similarity of action and module composition rules makes it easy to modularize a specification in Action Language. A modular approach is crucial for scaling the model checking technique to larger systems.

4 CONCLUSIONS

We presented a specification language called Action Language for model checking software specifications. Action Language specifications are low level in the sense that one can easily understand the transition system that a model checker would generate for a given Action Language specification. However Action Language translations of higher level languages such as Statecharts and SCR are also readable and compact.

We think that low level specification languages which would bridge the gap between model checkers and high level specification languages could be very helpful in increasing the usability of model checking tools. Fully automated model checking does not scale very well because of the state space explosion problem. Abstraction and compositional techniques can be helpful, but they are more effective when they are guided by the user. Having a low level interface between the model checker and the high level specification languages would make it easier for the users to guide the model checking process. One of the issues that has to be investigated in this context is the presentation of counter-example behaviors by the model checker.

Our goal is to extend the Action Language further. For example, variable hiding [19] could be an important tool in reducing the size of the state space, hence, improving the efficiency of model checking. We plan to investigate integrating such constructs to Action Language to enable model checking of larger systems.

REFERENCES

[1] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, July 1999.

[2] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

```
module Sys
  enumerated AltLayer {High, Mid, Low};
  enumerated Alarm {Shutdown, Operating};
  enumerated Mode {On, Off};
  enumerated Volume {1, 2};
  enumerated switch {up, down, test};
  boolean u, v, w;
  integer alt;
  initial : AltLayer=Mid & Alarm=Shutdown & Mode=Off
    & Volume=1 & !w;
  t1 : AltLayer=High & u & alt>=9950
    & w' & AltLayer'=High;
  t2 : AltLayer=Mid & u & 1950<=alt<=10050
    & w' & AltLayer'=Mid;
  t3 : AltLayer=Low & u & alt<=2050
    & w' & AltLayer'=Low;
  t4 : AltLayer=Mid & u & alt>10050
    & w' & AltLayer'=High;
  t5 : AltLayer=High & u & alt<9950
    & w' & AltLayer'=Mid;
  t6 : AltLayer=Low & u & alt>2050
    & w' & AltLayer'=Mid;
  t7 : AltLayer=Mid & u & alt<1950
    & w' & AltLayer'=Low;
  t8 : Alarm=Shutdown & u & switch=up
    & Alarm'=Operating & Mode'=Off & Volume'=1;
  t9 : Alarm=Operating & u & switch=down
    & Alarm'=Shutdown;
  t10 : Alarm=Operating & Mode=Off & w & c
    & Mode'=On;
  t11 : Alarm=Operating & Mode=On & w & c
    & Mode'=Off;
  t12 : Alarm=Operating & Volume=1 & v
    & Volume'=2;
  t13 : Alarm=Operating & Volume=2 & v
    & Volume'=1;
  t14 : Alarm=Shutdown & u & switch=test &
    Alarm'=Operating & Mode'=On & Volume'=1;
  Sys : (t1 | t2 | t3 | t4 | t5 | t6 | t7)
    & (t8 | t14 | t9 | ((t10 | t11) & (t12 | t13)));
endmodule
```

Figure 5: Action Language translation of the Statecharts example

Constants: Low, Permit : positive integer; Low $<$ Permit;
Monitored Variables: WaterPres: integer;
Block, Reset : { On, Off };
Controlled variables: Inject : { On, Off };
Terms: Overridden : boolean;
Mode Class: Pressure : { TooLow, Permitted, High };
Initial Conditions: Block, Reset, Inject = Off;
Overridden = False; Pressure = Low;
Low $\leq$ WaterPres $<$ Permit;

Old Mode	Event	New Mode
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

Mode	Events	
High	False	@T(InMode)
TooLow, Permitted	@T(Block=On) WHEN Reset=Off	@T(InMode) OR @T(Reset=On)
Overridden	True	False

Mode	Conditions	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
Inject	Off	On

Figure 6: SCR Specification of the Safety Injection System

SOFT International Symposium on Software Testing and Analysis, pages 224–238, January 1996.

[3] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[4] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstractions. *Automated Software Engineering*, 6(1):37–68, January 1999.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model checking for counterexamples. In *Proceedings of the 1996 ACM SIG-*

[7] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, March 1998.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, January 1990.

[9] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

```
module SafetyInjectionSystem
  enumerated Pressure {TooLow, Permitted, High};
  enumerated Inject {On, Off};
  boolean Overridden;
  enumerated Block, Reset {On, Off};
  parameterized integer Low, Permit;
  initial : Block=Off & Reset=Off & Inject=Off
    & !Overridden & Pressure=Low
    & Low<=WaterPres<Permit;
  p1 : Pressure=TooLow & Pressure'=Permitted
    & WaterPres'>=Low & !(WaterPres>=Low);
  p2 : Pressure=Permitted & Pressure'=High
    & WaterPres'>=Permit & !(WaterPres>=Permit);
  p3 : Pressure=Permitted & Pressure'=TooLow
    & WaterPres'<Low & !(WaterPres<Low);
  p4 : Pressure=High & Pressure'=Permitted
    & WaterPres'<Permit & !(WaterPres<Permit);
  p : p1 | p2 | p3 | p4;
  o1 : Overridden' & Block'=On & Block=Off & Reset=Off
    & (Pressure=TooLow | Pressure=Permitted);
  o2 : !Overridden' & (Pressure'=High & !(Pressure=High)
    | Pressure'=TooLow & !(Pressure=TooLow)
    | Pressure'=Permitted & !(Pressure=Permitted)
    | Reset'=On & !(Reset=On) & (Pressure=TooLow
    | Pressure=Permitted));
  o : o1 | o2;
  s : (Inject=On & Pressure=TooLow & !Overridden
    | Inject=Off & (Pressure=High |
    Pressure=Permitted | Pressure=TooLow & !Overridden))
    & (Inject'=On & Pressure'=TooLow & !Overridden'
    | Inject'=Off & (Pressure'=High |
    Pressure'=Permitted |
    Pressure'=TooLow & !Overridden'));
  SafetyInjectionSystem : p & o & s;
  spec : invariant((Reset=On & !(Pressure=High))
                    -> !Overridden);
  spec : invariant((Reset=On & Pressure = TooLow)
                    -> Inject = On);
endmodule
```

Figure 7: Action Language translation of the SCR example

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[12] E. M. Clarke and R. P. Kurshan. Computer aided verification. *IEEE Spectrum*, pages 61–67, June 1996.

[13] A. Coen-Porisini, C. Ghezzi, and R. A. Kemmerer. Specification of real-time systems using ASTRAL. *IEEE Transactions on Software Engineering*, 23(9):572–598, September 1997.

[14] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

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

[16] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[17] C. L. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[18] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[19] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[20] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specifications of process-control systems. *IEEE Transactions on Software Engineering*, 20(9), September 1994.

[21] K. L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Massachusetts, 1993.

[22] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[23] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements: A case study. In *Proceedings of the 11th Annual Conference on Computer Assurance*, pages 77–88, June 1996.

[24] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.