Action Language Verifier

Tevfik Bultan   Tuba Yavuz-Kahveci
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
{bultan,tuba}@cs.ucsb.edu

Abstract

Action Language is a specification language for reactive software systems. In this paper we present the Action Language Verifier which consists of 1) a compiler that converts Action Language specifications to composite symbolic representations, and 2) an infinite-state symbolic model checker which verifies (or falsifies) CTL properties of Action Language specifications. Our symbolic manipulator (Composite Symbolic Library) combines a BDD manipulator (for boolean and enumerated types) and a Presburger arithmetic manipulator (for integers) to handle multiple variable types. Since we allow unbounded integer variables, model checking queries become undecidable. We present several heuristics used by the Action Language Verifier to achieve convergence.

1. Introduction

Action Language Verifier is an infinite-state CTL model checker based on Action Language, a formal specification language for reactive systems. Action Language supports both synchronous and asynchronous compositions as basic operations [2]. Translations of Statecharts [9] and SCR [10] specifications to Action Language are compact and Action Language translations preserve the original structure of the specifications.

Action Language Verifier translates an Action Language specification to a composite symbolic representation provided by our Composite Symbolic Library [12]. Composite Symbolic Library combines different symbolic representations, such as BDDs for representing boolean logic formulas and polyhedral representations for Presburger arithmetic formulas (formulas of integer arithmetic where multiplication among variables is not allowed), using composite symbolic representation. Since Composite Symbolic Library uses an object-oriented design, Action Language Verifier is polymorphic. It can dynamically select symbolic representations provided by the Composite Symbolic Library based on the variable types in the input specification.

In general, model checking queries for the infinite-state systems are undecidable. In this paper, we present several heuristics used by the Action Language Verifier to achieve convergence such as approximate fixpoint computations, loop-closures and approximate reachability analysis. Approximate fixpoint computations based on truncated fixpoints and widening operator have been used in the abstract interpretation context before [6, 7, 8, 11]. Our use of loop-closures is similar to the meta-transitions used in [1] for reachability analysis. The idea of computing an approximation to the set of reachable states by a forward fixpoint computation, and then using this result to prune the iterates of the backward fixpoint computations has been used in [11]. However, our use of these techniques in the context of composite symbolic representation is unique and extends our previous work on composite symbolic representation [3].

The rest of the paper is organized as follows. In Section 2 we give an overview of the Action Language using an example specification. We discuss the Composite Symbolic Library in Section 3. In Sections 4 and 5 we present the fixpoint computations and the heuristics used in the Action Language Verifier, respectively. Finally, in Section 6 we give directions for future work.

2. Action Language

Statecharts specification of a light-control-system for an office is given in Figure 1. Figure 2 shows its translation to Action Language. The variable `c` and the state `Occupants` keep track of the number of people in the office. Events `enter` and `exit` signal people entering and exiting the room and events `s_on` and `s_off` signal light switch being turned on and off, respectively. Light cannot be turned off if there are more than one people in the office.

This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

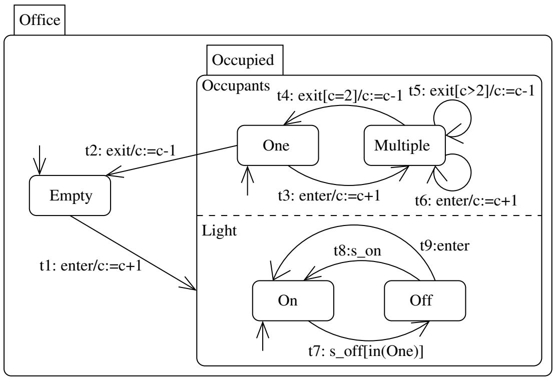

Figure 1. Statecharts specification

```
module main()
  enumerated Office {Empty, Occupied};
  enumerated Occupants {One, Multiple};
  enumerated Light {On, Off};
  boolean enter, exit, s_on, s_off;
  integer c;
  initial : c=0 and Office=Empty;
  restrict: (enter => not(exit or s_on or s_off))
    and (exit => not(enter or s_on or s_off))
    and (s_on => not(enter or exit or s_off))
    and (s_off => not(enter or exit or s_on));
  t1: Office=Empty and enter and Office'=Occupied
    and Occupants'=One and Light'=On and c'=c+1;
  t2: Office=Occupied and Occupants=One and exit
    and Office'=Empty and c'=c-1;
  t3: Office=Occupied and Occupants=One and enter
    and c'=c+1 and Occupants'=Multiple;
  t4: Office=Occupied and Occupants=Multiple and
    exit and c=2 and Occupants'=One and c'=c-1;
  t5: Office=Occupied and Occupants=Multiple
    and c>2 and exit and c'=c-1;
  t6: Office=Occupied and Occupants=Multiple
    and enter and c'=c+1;
  t7: Office=Occupied and Light=On and s_off
    and Occupants=One and Light'=Off;
  t8: Office=Occupied and Light=Off and s_on
    and Light'=On;
  t9: Office=Occupied and Light=Off and enter
    and Light'=On;
  environment: (enter => not(enter')) and
    (exit => not(exit')) and (s_on => not(s_on'))
    and (s_off => not(s_off'));
  main: (t1 | t2  | ((t3 | t4 | t5 | t6)
              & (t7 | t8 | t9))) & environment;
  spec: invariant([(Office=Occupied
          and Occupants=Multiple) => Light=On])
  spec: invariant([c>1 <=>
          Office=Occupied and Occupants=Multiple])
  spec: invariant([c=1 <=>
          Office=Occupied and Occupants=One])
endmodule
```

Figure 2. Action Language specification

An action language specification consists of a set of module definitions. Semantically, each module corresponds to a transition system with a set of states, a set of initial states and a transition relation. Variable declarations define the set of states of the module. Set of states can be restricted using a restrict expression. Initial expression defines the set of initial states of the module. A module expression (which starts with the name of the module) defines the transition relation of the module in terms of its actions and submodules using asynchronous and synchronous composition operators. Each action in an Action Language specification defines a single execution step. In an action expression, primed variables denote the next-state values for the variables and unprimed variables denote the current-state values.

Asynchronous composition of two actions a_1 and a_2, denoted $a_1 \mid a_2$, is defined as the disjunction of their transition relations. However, we also assume that an action preserves the values of the variables which are not modified by itself [2]. Two actions a_1 and a_2 can also be combined with synchronous composition a_1 `&` a_2. Semantics of synchronous composition corresponds to conjunction if two actions are always enabled. However, if one component of synchronous composition is not enabled in a state, this would deadlock the composed system if we use conjunction as its semantics. To prevent this, we assume that, in such a state, the disabled component makes a synchronous idle transition and stays in the same state which allows other components to progress [2].

The Action Language translation (Figure 2) uses one enumerated variable to encode each OR state in the Statecharts specification (Figure 1). Each transition in the Statecharts specification is represented by one action in the Action Language translation. Then, the overall transition relation is exactly the expression which corresponds to combining the transitions of OR states with asynchronous composition |, and the transitions of AND states with synchronous composition `&`. Hence, Action Language translations preserve the structure of Statecharts specifications. Same principle also holds if we use submodules in the translation. We can get an equivalent translation of the above Statecharts example to Action Language by creating two submodules `Occupants` and `Light` defined as: `Occupants:t3 | t4 | t5 | t6` and `Light:t7 | t8 | t9`. Then, the specification of the main module would be `main:(t1 | t2 | (Occupants() & Light())) & environment` again preserving the structure of the Statecharts specification.

Restrict condition in Figure 2 imposes the one-input-assumption (i.e., only one external event occurs at a time). The synchronous composition of the `environment` action with the rest of the system makes sure that generated events are reset immediately. Other semantic interpretations of Statecharts specifications can also be implemented using `restrict` expression to restrict the state space and using synchronous composition to enforce the restrictions on the transition relation.

Although the specification given in Figure 2 is an infinite-state system (the variable `c` is unbounded), when we used Action Language Verifier to verify the first invariant, the exact fixpoint computation (discussed in Section 4) converged in one iteration and we were able to verify the property in 0.04 seconds on a SUN ULTRA 10 workstation with 768 MBytes of main memory running Solaris. Similarly, the exact fixpoint computation for the second invariant converged in 5 iterations and the property was verified in 0.20 seconds. For the third invariant, however, the exact fixpoint computation did not converge. When we used approximations (discussed in Section 5) the fixpoint computation converged in 5 iterations and we were able to prove the property in 0.37 seconds.

3. Composite Symbolic Representation

Action Language parser translates an action language specification to a transition system $T = (S, I, R)$ that consists of a *state space* S, a *set of initial states* $I \subseteq S$, and a *transition relation* $R \subseteq S \times S$. Generally, in model checking transition systems are restricted to be finite (i.e., S is finite). and the transition relation R is assumed to be total (i.e., for each state $s \in S$ there exits a next state s' such that $(s, s') \in R$). We, let go of both of these assumptions. For the infinite-state systems that can be specified in Action Language, CTL model checking is undecidable. In this paper, we present several heuristics used by the Action Language Verifier to achieve convergence. Since we allow non-total transition systems also some fixpoint computations have to be modified.

Composite Symbolic Library is the symbolic manipulator used by the Action Language Verifier. It combines different symbolic representations using the *composite model checking* approach [3]. Our current implementation of the Composite Symbolic Library uses two basic symbolic representations: BDDs for boolean logic formulas and polyhedral representation for Presburger arithmetic formulas. Boolean and enumerated variables in the Action Language specifications are mapped to BDD representation, and integers are mapped to arithmetic constraint representation.

To analyze a system using Composite Symbolic Library, one has to specify its initial condition, transition relation, and state space using a set of *composite formulas*. A composite formula is obtained by combining integer arithmetic formulas on integer variables with boolean variables using logical connectives. Enumerated variables are mapped to boolean variables by the Action Language parser. Since integer representation in the Composite Symbolic Library currently supports only Presburger arithmetic, we restrict arithmetic operators to $+$ and $-$. However, we allow multiplication with a constant and quantification.

A composite formula, p, is represented in disjunctive normal form as

$$p = \bigvee_{i=1}^{n} \bigwedge_{t=1}^{T} p_{it}$$

where p_{it} denotes the the formula of basic symbolic representation type t in the ith disjunct, and n and T denote the number of disjuncts and the number of basic symbolic representation types, respectively. Our Composite Symbolic Library implements methods such as intersection, union, complement, satisfiability check, subset test, which manipulate composite representations in the above form.

Given a set of states p and a transition relation R, precondition $\text{PRE}(p, R)$ are all the states that can reach a state in p with a single transition in R (i.e., the set of predecessors of all the states in p). $\text{POST}(p, R)$ is defined similarly. Given a set p and a transition relation R both represented using composite symbolic representation as $p = \bigvee_{i=1}^{n_p} \bigwedge_{t=1}^{T} p_{it}$ and $R = \bigvee_{i=1}^{n_R} \bigwedge_{t=1}^{T} r_{it}$ the pre-condition can be computed as

$$\text{PRE}(p, R) = \bigvee_{i=1}^{n_R} \bigvee_{j=1}^{n_p} \bigwedge_{t=1}^{T} \text{PRE}(p_{jt}, r_{it})$$

The above property holds because the existential variable elimination in the $\text{PRE}(p, R)$ computation distributes over the disjunctions, and due to the partitioning of the variables based on the basic symbolic types, the existential variable elimination also distributes over the conjunction above.

4. Fixpoint Computations

The CTL temporal operator EX corresponds to precondition computation, i.e., EX $p \equiv \text{PRE}(p, R)$. AX can also be computed as AX $p \equiv \neg\text{PRE}(\neg p, R)$. Rest of the CTL operators can be computed as least and greatest fixpoints using EX and AX [5]

$$p \text{ EU } q \equiv \mu x \,.\, q \ \vee \ (p \ \wedge \ \text{EX}\, x) \qquad \text{EG}\, p \equiv \nu x \,.\, p \ \wedge \ \text{EX}\, x$$
$$p \text{ AU } q \equiv \mu x \,.\, q \ \vee \ (p \ \wedge \ \text{AX}\, x) \qquad \text{AG}\, p \equiv \nu x \,.\, p \ \wedge \ \text{AX}\, x$$

However, above characterizations of AU and EG are not complete if we do not restrict the transition relation to be total. Since a non-total transition system can have states which do not have any next states, AX **false** will be satisfied in such states vacuously. Hence, those states will satisfy AF **false** too. This creates a problem, since we will have states which satisfy AF p without p being satisfied in any future state. To prevent this we alter the fixpoint computation for AU (and similarly for AF) as follows

$$p \text{ AU } q \equiv \mu x \,.\, q \ \vee \ (p \ \wedge \ \text{AX}\, x \ \wedge \ AtLeastOne)$$

where $AtLeastOne$ is the set of states which have at least one successor.

Dual of this problem appears in the EG fixpoint. If all the states in a finite path that ends at a state which does not have any successors satisfies p, then the states on that path should satisfy EGp. Then, we need to change the EG fixpoint as:

$$\text{EG}\, p \equiv \nu x \,.\, p \ \wedge \ (\text{EX}\, x \ \ \vee \ \ None)$$

where $None$ are the set of states which have no successors (i.e., $None \equiv \neg AtLeastOne$).

Action Language Verifier iteratively computes the fixpoints for the temporal operators. In an infinite-state model checker convergence is not guaranteed. Although each iteration takes us closer to the fixpoint we are not guaranteed to reach it. However, if a fixpoint is reached we are sure that it is the least or the greatest fixpoint based on the type of the iteration.

5. Heuristics for Infinite-State Verification

If we cannot directly compute the truth set of a temporal property p for a transition system $T = (S, I, R)$, we can try to generate a *lower bound* for p, denoted p^-, such that $p^- \subseteq p$. Then, if we determine that the set of initial states are included in this lower bound (i.e., $I \subseteq p^-$), we have also showed that $I \subseteq p$, i.e., we proved that transition system T satisfies the property p. However, if $I \not\subseteq p^-$, we cannot conclude anything because it can be a *false negative*. In that case we can compute a lower bound for the negated property: $(\neg p)^-$. If we can find a state s such that $s \in I \cap (\neg p)^-$, then we can generate a counter example which would be a *true negative*. If both cases fail, i.e., both $I \not\subseteq p^-$ and $I \cap (\neg p)^- \equiv \emptyset$, then the verifier cannot report a definite answer.

Since Action Language Verifier computes the temporal formulas recursively starting from the innermost temporal operators, we have to compute an approximation to a formula by first computing approximations for its subformulas. All temporal and logical operators other than "$\neg$" are monotonic. This means that any lower/upper approximation for a negation free formula can be computed using the corresponding lower/upper approximation for its subformulas. To compute a lower bound for a negated property like $p = \neg q$, we can compute an upper bound q^+ for the subformula q where $q^+ \supseteq q$, and then let $p^- \equiv S - q^+$. Similarly we can compute an upper bound for p using a lower bound for q. Thus, we need algorithms to compute both lower and upper bounds of temporal formulas.

Truncated Fixpoints Computations Each iteration of a least fixpoint computation gives a lower bound for the least fixpoint. Hence, if we truncate the fixpoint computation after a finite number of iterations we will have a lower bound for the least-fixpoint. Similarly, each iterate of a greatest fixpoint computation gives an upper bound for the greatest fixpoint. Action Language Verifier has a flag which can be set to determine the bound on number of fixpoint iterations. If the obtained result is not precise enough to prove the property of interest, it can be improved by running more fixpoint iterations.

Widening and Collapsing Operators For computing upper bounds for least-fixpoints we use the *widening* technique [6] generalized to the composite symbolic representation [3]. Assume that p and q are two sets of states, then the widening operator $\bigtriangledown$ satisfies the following constraint: $p \cup q \subseteq p \bigtriangledown q$. Intuitively, $\bigtriangledown$ operator guesses the direction of growth in the fixpoint iterates, and extends the successive iterates in that direction. The least fixpoint computations are modified so that at each iteration the current iterate p_i is set to $p_{i-1} \bigtriangledown p_i$. For the polyhedral representation we use the widening operator defined in [4] for Presburger arithmetic constraints by generalizing the convex widening operator in [7]. The basic idea is to find pairs of polyhedra p and q such that $p \subseteq q$ and set $p \bigtriangledown q$ to conjunction of constraints in p which are also satisfied by q. Intuitively, if a constraint of p is not satisfied by q this means that the iterates are increasing in that direction. By removing that constraint we extend the iterates in the direction of growth as much as possible without violating other constraints.

To compute lower-bounds for greatest fixpoint computations we define the dual of the widening operator and call it the *collapsing* operator (and denote it with $\bigtriangledown^{-1}$). Given two set of states p and q the collapsing operator $\bigtriangledown^{-1}$ satisfies the following: $p \cap q \supseteq p \bigtriangledown^{-1} q$. Intuitively, $\bigtriangledown^{-1}$ operator finds which parts of the fixpoint iterates are decreasing and removes them to accelerate the fixpoint computation. The greatest fixpoint computations are modified so that at each iteration the current iterate p_i is set to $p_{i-1} \bigtriangledown^{-1} p_i$. In our symbolic representation for integers each Presburger arithmetic formula is represented as a disjunction of polyhedra. Given two such representations p and q, our collapsing operator looks for a polyhedron in p which subsumes a polyhedron in q. When a pair is found the subsumed polyhedron is removed from q. The result of the collapsing operation is the union of the polyhedra remaining in q.

Loop-Closures Another heuristic we use to accelerate convergence is to compute closures of self-loops in the specifications. Given a transition relation R we can use any relation R' which satisfies the constraint $R \subseteq R' \subseteq R^*$ (where R^* denotes the reflexive-transitive closure of R) to accelerate the fixpoint computations for temporal operators EF and EU [4].

To exploit this idea, given a transition relation R in composite symbolic representation $R \equiv \bigvee_{i=1}^{n_R} \bigwedge_{t=1}^{T} r_{it}$ Action Language Verifier transforms it to

$$R \equiv R \ \vee \ \bigvee_i (r_{it_I} \ \wedge \bigwedge_{t=1, t \neq t_I}^{T} IR_t)$$

where IR_t is the identity relation for the variables represented with the basic symbolic representation type t, and t_I is the symbolic representation type for integers. Note that, $\bigwedge_{t=1,t\neq t_I}^{T} IR_t$ corresponds to identity relation for all the variables other than integers. Hence, $\bigvee_i (r_{it_I} \wedge \bigwedge_{t=1,t\neq t_I}^{T} IR_t)$ denotes the part of the transition relation where all the variables other than the integer variables stay the same. To compute r_{it_I}'s we intersect the transition relation R with $\bigwedge_{t=1,t\neq t_I}^{T} IR_t$ and collect the resulting disjuncts which are satisfiable. Then, for each r_{it_I} we compute r'_{it_I}, where $r_{it_I} \subseteq r'_{it_I} \subseteq r^*_{it_I}$. We take the union of the result with the original transition relation R to compute

$$R' = R \vee \bigvee_i (r'_{it_I} \wedge \bigwedge_{t=1,t\neq t_I}^{T} IR_t)$$

Then, we use R' in the fixpoint computations for EF and EU instead of R to accelerate the fixpoint computations. Note that we can not use closure computations for EG or AU fixpoints since they may introduce cycles which do not exist in the original transition system.

Reachable States The fixpoint algorithms described thus far are *backward* techniques. They start with a property p, and then use PRE to determine which states can reach p. The last step is to determine whether the set of initial states I is included in the derived set. Alternatively, it may be useful to start with the initial states I, compute an upper approximation RS^+ to the reachable state-space RS, and then use RS^+ to help in the model-checking process. We can accomplish this by altering the symbolic model checker to restrict its computations to states in RS^+. To generate the upper bound RS^+, we used the POST function. The (exact) reachable state-space of a transition system is the least fixpoint $RS \equiv \mu x \,.\, I \vee \text{POST}(x, R)$, and it can be computed using the techniques we previously developed for EU. Moreover, we can use the widening method to compute an upper bound for RS as well. After computing RS^+, we restrict the result of every operation in the model checker to RS^+.

6. Future Work

We plan to extend the Action Language Verifier with new variable types such as reals and new symbolic representations such as automata for arithmetic constraints. The modular structure of our Composite Symbolic Library should make such extensions relatively easy. The verification procedures do not need to be changed. We plan to investigate using hierarchical and compositional verification strategies in Action Language Verifier. We would also like to develop visual (e.g., Statecharts) and tabular (e.g., SCR) specification front ends for Action Language Verifier. Another direction we are considering is generating concurrent Java programs from Action Language specifications.

Composite Symbolic Library and Action Language Verifier are available at:
http://www.cs.ucsb.edu/˜bultan/composite/

References

[1] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. of the 6th International Conference on Computer Aided Verification*, volume 818 of LNCS, pages 55–67. Springer-Verlag, June 1994.

[2] T. Bultan. Action language: A specification language for model checking reactive systems. In *Proc. of the 22nd International Conference on Software Engineering*, pages 335–344, June 2000.

[3] T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, January 2000.

[4] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[5] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th Annual ACM Symposium on Principles of Programming*, pages 84–97, 1978.

[8] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proc. of International Symposium on Static Analysis*, volume 864 of LNCS. Springer-Verlag, September 1994.

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

[10] 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.

[11] T. A. Henzinger and P. Ho. A note on abstract-interpretation strategies for hybrid automata. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of LNCS, pages 252–264. Springer-Verlag, 1995.

[12] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of LNCS, pages 335–344. Springer-Verlag, April 2001.