# Action Language Verifier, Extended*

Tuba Yavuz-Kahveci[1], Constantinos Bartzis[2], and Tevfik Bultan[3]

[1] University of Florida
[2] Carnegie Mellon University
[3] UC, Santa Barbara

## 1 Introduction

Action Language Verifier (ALV) is an infinite state model checker which specializes on systems specified with linear arithmetic constraints on integer variables. An Action Language specification consists of integer, boolean and enumerated variables, parameterized integer constants and a set of modules and actions which are composed using synchronous and asynchronous composition operators [3, 7]. ALV uses symbolic model checking techniques to verify or falsify CTL properties of the input specifications. Since Action Language allows specifications with unbounded integer variables, fixpoint computations are not guaranteed to converge. ALV uses conservative approximation techniques, reachability and acceleration heuristics to achieve convergence.

Originally, ALV was developed using a Polyhedral representation for linear arithmetic constraints [4]. In the last couple of years we extended ALV by adding an automata representation for linear arithmetic constraints [2]. ALV also uses BDDs to encode boolean and enumerated variables. These symbolic representations can be used in different combinations. For example, polyhedral and automata representations can be combined with BDDs using a disjunctive representation. ALV also supports efficient representation of bounded arithmetic constraints using BDDs [2]. Other extensions to ALV include several techniques to improve the efficiency of fixpoint computations such as marking heuristic and dependency analysis, and automated counting abstraction for verification of arbitrary number of finite state processes [7].

## 2 Symbolic Representations

ALV uses the Composite Symbolic Library [8] as its symbolic manipulation engine. Composite Symbolic Library provides an abstract interface which is inherited by every symbolic representation that is integrated to the library. ALV encodes the transition relation and sets of states using a disjunctive, composite representation, which uses the same interface and handles operations on multiple symbolic representations.

**Polyhedral vs. Automata Representation:** Current version of the Composite Symbolic Library uses two different symbolic representations for integer variables: 1) *Polyhedral representation*: In this approach the valuations of integer variables are represented in a disjunctive form where each disjunct corresponds to a convex polyhedron

* This work is supported by NSF grant CCR-0341365.

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 413–417, 2005.

and each polyhedron corresponds to a conjunction of linear arithmetic constraints. This approach is extended to full Presburger arithmetic by including divisibility constraints (which is represented as an equality constraint with an existentially quantified variable). 2) *Automata representation*: In this approach a Presburger arithmetic formula on $v$ integer variables is represented by a $v$-track automaton that accepts a string if it corresponds to a $v$-dimensional integer vector (in binary representation) that satisfies the formula. Both of these symbolic representations are integrated to the Composite Symbolic Library by implementing the intersection, union, complement, backward and forward image operations, and subsumption, emptiness and equivalence tests, which are required by the abstract symbolic interface. We implemented the polyhedral representation by writing a wrapper around the Omega Library [1]. We implemented the automata representation using the automata package of the MONA tool [6] and based on the algorithms discussed in [2].

**BDD Representation for Bounded Integers:** We also integrated algorithms for constructing efficient BDDs for linear arithmetic formulas to ALV [2]. The size of the BDD for a linear arithmetic formula is linear in the number of variables and the number of bits used to encode each variable, but can be exponential in the number of *and* and *or* operators [2]. This bounded representation can be used in three scenarios: 1) all the integer variables in a specification can be bounded, 2) infinite state representations discussed above may exhaust the available resources during verification, or 3) infinite state fixpoint computations may not converge. Note that, for cases 2 and 3, verification using the bounded representation does not guarantee that the property holds for the unbounded case, i.e., the bounded representation is used for finding counter-examples.

**Polymorphic Verification:** Due to the object oriented design of the ALV, implementation of the model checking algorithms are polymorphic. This enables the users to choose different encodings without recompiling the tool. For example, one can first try the polyhedral encoding and if the verification takes too long or the memory consumption is too much the same specification can be checked using the automata encoding. The user specifies the encoding to be used as a command line argument to the ALV. When there are no integer variables in the specification or if the bounded BDD representation for integers is used, ALV automatically runs as a BDD based model checker.

## 3 Fixpoint Computations

ALV is a symbolic model checker for CTL. It uses the least and greatest fixpoint characterizations of CTL operators to compute the truth set of a given temporal property. It uses iterative fixpoint computations starting from the fixpoint for the innermost temporal operator in the formula. At the end, it checks if all the initial states are included in the truth set. ALV supports both the $\{\mathrm{EX}, \mathrm{EG}, \mathrm{EU}\}$ basis and the $\{\mathrm{EX}, \mathrm{EU}, \mathrm{AU}\}$ basis for CTL. ALV uses various heuristics to improve the performance of the fixpoint computations. We discuss some of them below. The reader is referred to [7] for the experimental analysis of these heuristics.

**Marking Heuristic:** Since composite representation is disjunctive, during the least fixpoint computations the result of the $k$th iteration includes the disjuncts from the $k-1$st iteration. A naive approach that applies the image computation to the result of the $k$th iteration to obtain the result of the $k + 1$st iteration performs redundant computations, i.e., it recomputes the image for the disjuncts coming from the result of the $k - 1$th iteration. We alleviate this problem by marking the disjuncts coming from the $k - 1$st iteration when we compute the result of the $k$th iteration [7]. Hence, at the $k + 1$st iteration, we only compute the images of the disjuncts that are not marked, i.e., disjuncts that were added in the $k$th iteration. Markings are preserved during all the operations that manipulate the disjuncts and they are also useful during subsumption check and simplification. When we compare the result of the current iteration to the previous one, we only check if the unmarked disjuncts are subsumed by the previous iteration. During the simplification of the composite representation (which reduces the number of disjuncts) we try to merge two disjuncts only if one of them is unmarked.

**Dependency Analysis:** Typically, in software specifications, the transition relation corresponds to a disjunction of a set of atomic actions. Since the image computation distributes over disjunctions, during fixpoint computation one can compute the image of each action separately. It is common to have pairs of actions $a_1$ and $a_2$ such that, when we take the backward-image of $a_2$ with respect to $\mathtt{true}$ and then take the backward-image of $a_1$ with respect to the result, we get $\mathbf{false}$. I.e., there are no states $s$ and $s'$ such that $s'$ is reachable from $s$ by execution of $a_1$ followed by execution of $a_2$. This implies that, during the $k$th iteration of a backward (forward) fixpoint computation, when we take the backward-image (forward-image) of $a_1$ ($a_2$) with respect to the result of the backward-image (forward-image) of $a_2$ ($a_1$) from the $k - 1$st iteration, the result will be $\mathbf{false}$. We use a dependency analysis to avoid such redundant image computations [7]. First, before we start the fixpoint computation, we identify the dependencies among the actions using the transition relation. Then, during the fixpoint computation, we tag the results of the image computations with the labels of the actions that produce them, and avoid the redundant image computations using the dependency information.

**Approximations, Reachability, and Accelerations:** For the infinite state systems that can be specified in Action Language, model checking is undecidable. Action Language Verifier uses several heuristics to achieve convergence: 1) Truncated fixpoint computations to compute lower bounds for least fixpoints and upper bounds for greatest fixpoints, 2) Widening heuristics (both for polyhedra and automata representations) to compute upper bounds for least fixpoints (and their duals to compute lower bounds for greatest fixpoints), 3) Approximate reachability analysis using a forward fixpoint computation and widening heuristics, 4) Accelerations based on loop-closures which extract disjuncts from the transition relation that preserve the boolean and enumerated variables but modify the integer variables, and then compute approximations of the transitive closures of the integer part.

## 4 Counting Abstraction

We integrated the counting abstraction technique [5] to ALV in order to verify properties of parameterized systems with arbitrary number of finite state modules. When a module is marked to be parameterized, ALV generates an abstract transition system in which the local states of the parameterized module are abstracted away (by removing all the local variables) but the number of instances in each local state is counted by introducing an auxiliary integer variable for each local state. An additional parameterized constant is introduced to denote the number of instances of the module. Counting abstraction preserves the CTL properties that do not involve the local states of the abstracted processes. When we verify properties of a system using counting abstraction we know that the result will hold for any number of instances of the parameterized module and if we generate a counter-example it corresponds to a concrete counter-example. Note that counting abstraction technique works only for modules with finite number of local states.

## 5 An Example

Here, we will briefly describe the verification of the concurrency control component of an airport ground traffic control simulation program (this and other examples and the ALV tool are available at `http://www.cs.ucsb.edu/~bultan/composite/`). The simulation program uses an airport ground network model which consists of two runways (16R, 16L) and 11 taxiways. The Action Language specification has one main module and two submodules representing departing and arriving airplanes. We use integer variables to denote the number of airplanes in each runway and taxiway. A local enumerated variable for each submodule denotes the locations of the airplanes. A set of actions for each submodule specifies how the airplanes move between the runways and taxiways based on the airport topology. The specification has 13 integer variables and 2 and 4 boolean variables for each instantiation of the departing and arriving airplane modules, respectively (these boolean variables are generated by the ALV compiler to encode the enumerated variables).

The property “At any time there is at most one airplane in either runway,” is expressed as `AG(num16R<=1 and num16L<=1)`. ALV verified this property on an instance with 8 departing and 8 arriving airplanes (13 integer variables, 48 boolean variables) in 1.20 seconds using 46.5 MBytes of memory (on a 2.8 GHertz Pentium 4 processor with 2 GBytes of main memory). We also verified this property on the parameterized specification for arbitrary number of arriving and departing airplanes using automated counting abstraction (which generates 20 additional integer variables and 2 parameterized integer constants). ALV verified the property above on the parameterized instance in 9.38 seconds using 6.7 MBytes of memory using the option to compute an approximation of the reachable states (using widening) and the marking heuristic.

## References

1. The Omega project. Available at `http://www.cs.umd.edu/projects/omega/`
2. C. Bartzis. *Symbolic Representations for Integer Sets in Automated Verification*. PhD thesis, University of California, Santa Barbara, 2004.
3. T. Bultan. Action language: A specification language for model checking reactive systems. In *Proc. ICSE 2000*, pages 335–344, June 2000.
4. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proc. of ASE 2001*, pages 382–386, November 2001.

5. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. CAV 2000*, pages 53–68, 2000.
6. J. G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS 1995*, 1995.
7. T. Yavuz-Kahveci. *Specification and Automated Verification of Concurrent Software Systems*. PhD thesis, University of California, Santa Barbara, 2004.
8. T. Yavuz-Kahveci and T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *STTT*, 5(1):15–33, November 2003.