Action Language Verifier: An Infinite-State Model Checker for Reactive Software Specifications

Tuba Yavuz-Kahveci* and Tevfik Bultan**

*Computer and Information Sciences and Engineering

University of Florida

Gainesville, FL 32611, USA

tyavuz@cise.ufl.edu

**Department of Computer Science, University of California

Santa Barbara, CA 93106, USA

bultan@cs.ucsb.edu

This work is supported in part by NSF grants CCF-0614002 and CCF-0716095.

Abstract

Action Language is a specification language for reactive software systems. In this paper, we present the syntax and the semantics of the Action Language and we also present an infinite-state symbolic model checker called Action Language Verifier (ALV) that verifies (or falsifies) CTL properties of Action Language specifications. ALV is built on top of the Composite Symbolic Library, which is a symbolic manipulator that combines multiple symbolic representations. ALV is a polymorphic model checker that can use different combinations of the symbolic representations implemented in the Composite Symbolic Library. We describe the heuristics implemented in ALV for computing fixpoints using the composite symbolic representation. Since Action Language specifications allow declaration of unbounded integer variables and parameterized integer constants, verification of Action Language specifications is undecidable. ALV uses several heuristics to conservatively approximate the fixpoint computations. ALV also implements an automated abstraction technique that enables parameterized verification of a concurrent system with an arbitrary number of identical processes.

Keywords: Symbolic model checking, infinite state model checking, automated abstraction

I. Introduction

Computer systems are becoming increasingly pervasive. They are employed in a wide range of industries such as avionics, energy, and medicine to implement safety critical applications. Correct operation of these systems depends on the correctness of the underlying software. 21st century has already witnessed failure of several space exploration missions that were caused by software errors [Low]. These incidents show the importance of software verification, which can help in finding the errors in software systems before they are deployed.

In this paper, we present the syntax and the semantics of the *Action Language* [Bul00], a specification language for reactive software systems. We also present an automated verification tool called the *Action Language Verifier* (ALV) [BYK01], [YKBB05], which can be used to check the correctness properties of infinite-state reactive software specifications written in Action Language [Bul00].

An Action Language specification includes both the behavior specification and the correctness properties of a system. The correctness properties are described using the Computation Tree Logic (CTL) [CGP99]. The Action Language can be used to specify *data parameterized* and/or *control parameterized systems*. Data parameterized system specifications contain explicitly declared parameterized integer constants. Such a specification must be verified for all possible valuations of the parameterized constant, which could be an infinite set. Control parameterized system specifications contain asynchronous composition of an arbitrary number of identical processes. Such specifications must be verified for an infinite number of instances, each representing a specific number of processes.

ALV uses a symbolic manipulator, called the Composite Symbolic Library [YKB03], to encode the state space and the transition system of the input Action Language specification. Using infinite-state symbolic model checking techniques, ALV checks the symbolic transition system for the specified CTL properties. If ALV is unable to prove a property, it can search and report counter-example behaviors for ACTL properties.

The contributions of this paper can be summarized as follows:

- *Formal syntax and semantics of Action Language:* The basic operations in Action Language was discussed in [Bul00]. In this paper we formalize both the syntax and the semantics of the Action Language. We present the formal semantics of the Action Language by mapping the syntactic elements of the Action Language to the three important components of a transition system: the state space, the initial states, and the transition relation.
- *Verification heuristics:* ALV uses two heuristics, the marking and the dependence heuristics, to improve the verification performance. The marking heuristic avoids redundant computations in least fixed point computations by marking a constraint when the pre-image computation is performed on it the first time. This way, in the upcoming iterations, such constraints can be skipped during image computation. The dependence heuristic, on the other hand, analyzes the control-flow of the transition system to avoid the redundant pre- and post-image computations.
- *Automatic generation of parameterized transition systems:* A reactive software specification typically involves concurrent processes. Verifying such a specification with a certain number of concurrent processes yields a validation for that number of concurrent processes only. Using an abstraction technique called *counting abstraction* [Del03], ALV automatically generates an abstract transition system that is parameterized in the number of identical concurrent processes and checks the generated system using infinite-state model checking techniques. CTL properties that are verified on the abstract parameterized system are preserved for the transition system for any number of concurrent processes.
- *Counter-example generation for infinite-state verification* We present a witness generation algorithm that can be used to generate a counter-example for properties that are not satisfied by the given transition system. Our witness generation algorithm provides witnesses for each subformula in the negated property and targets infinite state systems.

The rest of the paper is organized as follows. Section II briefly explains the airport ground traffic control simulation software specification that we use as a case study. Section III presents the Action Language and explains its syntax and semantics. Section IV presents ALV, explains the symbolic encoding and manipulation techniques, discusses the heuristics for infinite-state verification and symbolic verification, and introduces the counter-example generation algorithm for infinite domains. Section V introduces the counting abstraction algorithm for translating a control parameterized Action Language specification to a data parameterized one. Section VI reports the experimental results for our case study on the airport ground traffic control simulation software specification. Section VII compares our work with the related work. Finally, Section VIII concludes the paper.

II. An Example Specification: Airport Ground Traffic Control

We use an infinite state reactive system specification for an airport ground traffic control system as a case study. Airport ground traffic control handles allocation of the airport ground network resources such as runways, taxiways, and gates for the arriving and the departing airplanes. Simulations play an important role for the safety of the airport ground traffic control. Simulations enable early prediction of possible runway incursions, which is a growing problem at the busy airports throughout the world. [Zho97] presents a concurrent simulation program

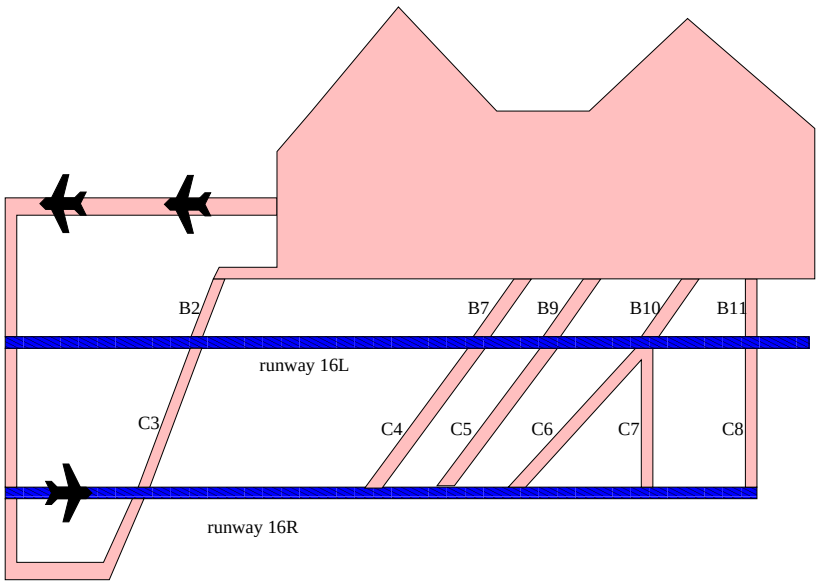


Fig. 1. An airport ground network similar to that of the Seattle Tacoma International Airport

for modeling airport ground traffic control using Java threads. As a case study, we model the concurrency control component of this simulation program in the Action Language [YKB02]. We use the same airport ground network model used in [Zho97] (shown in Figure 1) similar to that of the Seattle/Tacoma International Airport. There are two runways: 16R and 16L. The Runway 16R is used by the arriving airplanes during landing. After landing, an arriving airplane takes one of the exits C3-C8. After taxiing on C3-C8, the arriving airplanes need to cross the runway 16L. After crossing the runway 16L, they continue on to one of the taxiways B2, B7, B9-B11 and reach the gates in which they park. The departing airplanes use the runway 16L for takeoff. The control logic for the ground traffic of this airport must implement the following rules:

1) An airplane can land (takeoff) using the runway 16R (16L) only if no airplane is using the runway 16R (16L) at the moment.
2) An airplane taxiing on one of the exits C3-C8 can cross the runway 16L only if no airplane is taking off at the moment.
3) An airplane can start using the runway 16L for taking off only if none of the crossing exits C3-C8 are occupied at the moment. (The arriving airplanes have priority over the departing airplanes.)
4) Only one airplane can use a taxiway at a time.

In the following sections we show that the above control logic can be specified in the Action Language, and its properties can be automatically verified using the Action Language Verifier.

III. The Action Language

Formal specification is the first step of the formal reasoning process. The goal of formal specification languages is to ease the task of precise formal modeling of both the system behavior and the correctness properties. The Action

Language has been designed to support specification of infinite state reactive systems in a compact and simple manner. The Action Language supports both synchronous and asynchronous compositions as basic operations [Bul00]. In this paper, we extend the Action Language by introducing a module hierarchy and the associated scoping rules. We have also added parameters to modules, which enables renaming of the variables for each instantiation of a module. We formalize the semantics of the Action Language using denotational semantics.

Currently, the Action Language supports variables with boolean, enumerated, and (unbounded) integer types. Additionally, one can declare parameterized integer constants, which enables specification of both data parameterized and control parameterized systems. Parameterized constants can be thought of variables that have unknown initial values and that do not change their values.

In Section III-A we present the syntax of the Action Language and in Section III-B we formally define the semantics of the Action Language.

A. Syntax

An Action Language specification consists of a set of module definitions. Figure 2 shows the Action Language specification of the control logic for the airport ground traffic control simulation software discussed in Section II. The specification consists of three modules: `main`, `Departing`, and `Arriving`. The module `main` (lines 1-36) models the airport ground traffic control system, the module `Departing` (lines 5-13) models a departing airplane, and the module `Arriving` (partially shown in lines 14-30) models an arriving airplane.

The abstract syntax for the Action Language is given in Figure 3. An Action Language module consists of the formal parameter declarations, the local variable declarations, the initial state and the state space specifications, the submodule definitions, the action definitions, the transition relation definition, and the temporal property specifications. In every Action Language specification the top level module is named `main` and `main` module does not have any formal parameters.

A module definition starts with the variable declarations. The variable declarations consist of type definitions of the formal parameters, the local variable declarations, and the parameterized constants. The local variables, formal parameters and parameterized constants of a module are only visible to that module and the submodules of that module, i.e., lexical scoping is used. When an identifier for a variable (which is either a local variable or a formal parameter) is used in a formula, it denotes the value of that variable in the current state. One can refer to the value of a variable in the next state using a primed identifier, i.e., by appending a "`'`" character at the end of the identifier. We distinguish the two by calling the former *current state variable* and the latter *next state variable*.

In Action Language, a parameterized integer constant can take any integer value (however the value it takes does not change during the execution). When Action Language Verifier successfully verifies a specification with a parameterized constant, it means that the verified property holds for any possible valuation of that parameterized constant.

In the specification shown in Figure 2, integer variables model the shared resources of the airport ground traffic control, which are runways and taxiways. For example, variables `numRW16R` and `numC3` (line 2) denote the

```
1  module main()
2   integer numRW16R, numRW16L, numC3 ...;
3   initial: numRW16R=0 and numRW16L=0 numC3=0 ...;
4   restrict: numRW16R>=0 and numRW16L>=0 and numC3>=0...;
5   module Departing()
6     enumerated pc {parked, depFlow,takeOff};
7     initial: pc=parked;
8     reqTakeOff: pc=parked and numRW16L=0 and numC3+numC4+numC5+
9                 numC6+numC7+numC8=0 and pc'=takeOff and
10                numRW16L'=numRW16L+1;
11    leave: pc=takeOff and pc'=depFlow and numRW16L'=numRW16L-1;
12    Departing: reqTakeOff | leave;
13  endmodule
14  module Arriving()
15    enumerated pc {arFlow, touchDown, taxiTo16LC3, taxiTo16LC4,
16    taxiTo16LC5, taxiTo16LC6, taxiTo16LC7, taxiTo16LC8,
17    taxiFr16LB2, taxiFr16LB7, taxiFr16LB9, taxiFr16LB10,
18    taxiFr16LB11};
19    initial: pc=arFlow;
20    reqLand: pc=arFlow and numRW16R=0 and pc'=touchDown and
21              numRW16R'=numRW16R+1;
22    exitRW3: pc=touchDown and numC3=0 and numC3'=numC3+1 and
23              numRW16R'=numRW16R-1 and pc'=taxiTo16LC3;
24    crossRW3: pc=taxiTo16LC3 and numRW16L=0 and numB2A'=numB2A+1
25             and pc'=taxiFr16LB2 and numC3'=numC3-1 and numB2A=0;
26    park2: pc=taxiFr16LB2 and pc'=parked and numB2A'=numB2A-1;
27     . . .
28    Arriving: reqLand | exitRW3 | crossRW3 | park2 | ... ;
29    spec: invariant(pc=arFlow => eventually(pc=parked))   // P4
30  endmodule
31  main: Arriving() | Departing() ;
32  spec: invariant(numRW16R<=1 and numRW16L<=1)   // P1
33  spec: invariant(numC3<=1)   // P2
34  spec: invariant((numRW16L=0 and numC3+numC4+numC5+numC6+numC7+numC8>0)
35        => next(numRW16L=0))    // P3
36 endmodule
```

Fig. 2. An airport ground traffic control specification in the Action Language

number of airplanes on the runway 16R and on the taxiway C3, respectively. The enumerated variables (the `pc` of the module `Departing` (line 6) and the `pc` of the module `Arriving`, (lines 15-18) are used to encode the states of the arriving and the departing airplanes. A departing airplane can be in one of the following states: `parked`, `takeOff`, and `depFlow`, where the state `parked` denotes that the airplane is parked at the gate, the state `takeOff` denotes that the airplane is taking off from the runway 16L, and the state `depFlow` denotes that

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \texttt{module}\ \mathit{Id}\ \texttt{(}\ \mathit{IdL}\ \texttt{)}\ \mathit{VarDecl}\ \mathit{Sys}\ \mathit{ModuleL}\ \mathit{Action}\ \mathit{ModTrans}\ \mathit{Prop}\ \texttt{endmodule} \\
\mathit{ModuleL} & ::= & \mathit{Module}\ \mathit{ModuleL} \mid \epsilon \\
\mathit{VarDecl} & ::= & \texttt{boolean}\ \mathit{Id}\ \texttt{;} \mid \texttt{integer}\ \mathit{Id}\ \texttt{;} \mid \texttt{parameterized integer}\ \mathit{Id}\ \texttt{;} \\
 & & \mid \texttt{enumerated}\ \mathit{Id}\ \{\ \mathit{IdL}\ \}\ \texttt{;} \mid \mathit{VarDecl}\ \mathit{VarDecl} \mid \epsilon \\
\mathit{Sys} & ::= & \texttt{initial :}\ \mathit{Form}\ \texttt{;} \mid \texttt{restrict :}\ \mathit{Form}\ \texttt{;} \mid \mathit{Sys}\ \mathit{Sys} \mid \epsilon \\
\mathit{ModInst} & ::= & \mathit{Id}\ \texttt{(}\ \mathit{IdL}\ \texttt{)} \\
\mathit{Comp} & ::= & \mathit{Id} \mid \mathit{ModInst} \mid \mathit{Comp} \mid \mathit{Comp} \mid \mathit{Comp}\ \texttt{\&}\ \mathit{Comp} \\
\mathit{Action} & ::= & \mathit{Id}\ \texttt{:}\ \mathit{Form}\ \texttt{;} \mid \mathit{Action}\ \mathit{Action} \mid \epsilon \\
\mathit{ModTrans} & ::= & \mathit{Id}\ \texttt{:}\ \mathit{Form}\ \texttt{;} \mid \mathit{Id}\ \texttt{:}\ \mathit{Comp}\ \texttt{;} \\
\mathit{Prop} & ::= & \texttt{spec :}\ \mathit{CtlForm}\ \texttt{;} \mid \mathit{Prop}\ \mathit{Prop} \mid \epsilon
\end{array}
$$

Fig. 3. Syntax of the Action Language

the airplane is in the air departing from the airport. Similarly, an arriving airplane can be in one of the following states: `arFlow`, `touchDown`, `taxiToXY`, `taxiFrXY` and `parked`, where the state `arFlow` denotes that the airplane is in the air approaching to the airport, the state `touchDown` denotes that the airplane has just landed and is on the runway 16R, the state `taxiToXY` denotes that the airplane is currently in the taxiway Y and is going to cross the runway X, the state `taxiFrXY` denotes that the airplane is currently in the taxiway Y and has just crossed the runway X, and finally, the state `parked` denotes that the airplane is parked at the gate.

In an Action Language specification, the initial states and the state space of a system are specified in terms of *composite formulas* (Form) based on the syntax given in Figure 4. A composite formula is obtained by combining boolean and integer formulas with logical connectives. A boolean formula ($\mathit{BoolForm}$) consists of boolean variables or constants combined with logical connectives. An integer formula ($\mathit{IntForm}$) consists of integer variables or constants combined with arithmetic operators, arithmetic predicates, logical connectives, and existential or universal quantifiers. Note that, only multiplication with an integer constant is allowed ($\mathit{Integer}$ denotes an integer constant). For the formulas defining the initial states and the state space, we additionally impose the restriction that only the current state variables appear in the formula. For example, in the specification of Figure 2, a departing airplane is initially in `parked` mode (line 7), whereas an arriving airplane is initially in `arFlow` mode (line 19). Additionally, the state space of the system is restricted to nonnegative values of the integer variables modeling the runways and the taxiways (line 4).

In the Action Language, actions model the atomic transitions of a system. A module can have multiple actions. An action is defined as a composite formula on the current and the next state variables. For example, in the specification of Figure 2, the action `reqTakeOff` (lines 8-10) models the request of a departing airplane for takeoff: when the airplane is in `parked` mode it checks whether all of the exits C3-C8 are empty. If so, it transitions to `takeOff` mode and occupies the runway 16L.

Actions and module instantiations can be composed (denoted by Comp in Figure 3) synchronously (&) or

$$
\begin{array}{rcl}
Form & ::= & Form \texttt{ and } Form \mid Form \texttt{ or } Form \mid \texttt{not } Form \\
 & & \mid (\texttt{exists } IdL \texttt{ : } Form) \mid (\texttt{forall } IdL \texttt{ : } Form) \\
 & & \mid BoolForm \mid EnumForm \mid IntForm \\
IntForm & ::= & IntTerm \texttt{ > } IntTerm \mid IntTerm \texttt{ < } IntTerm \mid IntTerm \texttt{ >= } IntTerm \\
 & & \mid IntTerm \texttt{ <= } IntTerm \mid IntTerm \texttt{ = } IntTerm \mid IntTerm \texttt{ != } IntTerm \\
IntTerm & ::= & IntTerm \texttt{ + } IntTerm \mid IntTerm \texttt{ - } IntTerm \mid \texttt{ - } IntTerm \mid \\
 & & IntTerm \texttt{ * } Integer \mid Id \mid Id\texttt{'} \mid Integer \\
BoolForm & ::= & BoolTerm \mid BoolTerm \texttt{ = } BoolTerm \mid BoolTerm \texttt{ != } BoolTerm \\
BoolTerm & ::= & Id \mid Id\texttt{'} \mid \texttt{true} \mid \texttt{false} \\
EnumForm & ::= & EnumTerm \texttt{ = } EnumTerm \mid EnumTerm \texttt{ != } EnumTerm \\
EnumTerm & ::= & Id \mid Id\texttt{'} \\
CtlForm & ::= & Form \mid \texttt{EX (} CtlForm \texttt{)} \mid \texttt{AX (} CtlForm \texttt{)} \mid \texttt{EF (} CtlForm \texttt{)} \\
 & & \mid \texttt{AF (} CtlForm \texttt{)} \mid \texttt{EG (} CtlForm \texttt{)} \mid \texttt{AG (} CtlForm \texttt{)} \\
 & & \mid \texttt{EU (} CtlForm\texttt{, } CtlForm \texttt{)} \mid \texttt{AU (} CtlForm\texttt{, } CtlForm \texttt{)} \\
 & & \mid CtlForm \texttt{ and } CtlForm \mid CtlForm \texttt{ or } CtlForm \mid \texttt{not } CtlForm
\end{array}
$$

Fig. 4. Syntax of a composite formula

asynchronously (`|`). Transition relation of a module is defined in terms of either a composite formula or a composition of its actions and submodule instantiations ($ModTrans$). Submodules can be instantiated with different actual parameters. In each instantiation of a module the formal parameters are replaced with the corresponding actual parameters and all the local variables are uniquely renamed. For example, in the specification of Figure 2, the behavior of a departing airplane (module `Departing`) is specified in terms asynchronous composition of the actions `reqTakeOff` and `leave` (line 12) and the behavior of the whole system is specified as asynchronous composition of instantiations of the module `Departing` and the module `Arriving` (line 28).

The temporal properties of a module are defined in CTL. A CTL formula (*CtlForm* in Figure 4) consists of composite formulas combined with the temporal operators (EX, AX, EF, AF, EG, AG, EU, AU) and logical connectives. For example, in the specification of Figure 2, four temporal properties are specified (lines 29, 32, 33, and 34). The property at line 29 states that it is always the case that if an arriving airplane is in the flow mode then it will eventually be in the parked mode. The properties given in lines 22-34 correspond to the rules listed in Section II.

B. Semantics

The formal semantics of an Action Language specification is defined by a tuple (M, ϕ), where M is a transition system and ϕ is a CTL property. Transition system M is a tuple (I, S, R), where I, S, and R denote the initial states, the state space, and the transition relation, respectively. An Action Language specification is called a correct

specification iff $M \models \phi$, i.e., M satisfies the property ϕ. Correctness of an Action Language specification can be checked using the Action Language Verifier as we will discuss in the following sections. Below, we describe the semantics of the Action Language using denotational semantics.

a) Notation: In explaining the Action Language semantics we use the following notational conventions: Given a function $f : X \rightarrow Y$ and $x_1, x_2 \in X$ and $y \in Y$, then the function $f[y/x_1] : X \rightarrow Y$ is defined as follows

$$f[y/x_1](x_2) = \begin{cases} y & \text{if } \ x_2 = x_1 \\ f(x_2) & \text{otherwise} \end{cases}$$

We use $[\![]\!]$-type brackets to denote the semantic domains. Depending on the context, $[\![Id]\!]$ denotes an identifier which can be one of the following: an action name, a module name, a variable name, or a parameterized constant; $[\![IdL]\!]$ denotes a list of identifiers; $[\![Form]\!]$ and $[\![CtlForm]\!]$ denote a composite formula and a CTL formula, respectively.

We define several functions that map actions or modules to their attributes.

- An action function $\alpha \in Act = [\![Id]\!] \rightarrow [\![Form]\!]$ maps each action to the composite formula that corresponds to that action.
- The initial states, the state space, and the transition relation functions $Init, State, Trans = [\![Id]\!] \rightarrow [\![Form]\!]$ map each module to the composite formulas that characterize its initial states, state space, and transition relation, respectively. Note that, the formulas for the initial states and the states space use only current state variables, whereas the formula for the transition relation can use both the current state and the next state variables.
- Functions for the formal parameters, the local variables and the parameterized constants $Formals$, $Locals$, $Params = [\![Id]\!] \rightarrow [\![IdL]\!]$ map each module to the set of identifiers that correspond to its formal parameters, local variables and parameterized constants, respectively.
- The instantiation counter function $InstCount = [\![Id]\!] \rightarrow N$ maps each module to its instantiation counter, which keeps track of the number of instantiations.
- The CTL property function $\tau \in CtlProp = [\![Id]\!] \rightarrow [\![CtlForm]\!]$ maps each module to the CTL property associated with that module.

We define the following tuples based on the functions defined above:

- The variable environment tuple is defined as $\vartheta \in VarEnv = Locals \times Formals \times Params$.
- The system environment tuple is defined as $\epsilon \in SysEnv = Init \times State$.
- The current environment tuple is defined as: $\rho \in CurEnv = Init \times State \times [\![Form]\!] \times CtlProp \times Locals \times InstCount$.
- The environment tuple is defined as: $\beta \in Env = Init \times State \times CtlProp \times InstCount \times Locals \times Formals \times Params \times Act \times Trans$.

We make use of the following functions:

- $Rename_F : ([\![Form]\!] \cup [\![CtlForm]\!]) \times [\![IdL]\!] \times [\![IdL]\!] \times N \times [\![IdL]\!] \rightarrow ([\![Form]\!] \cup [\![CtlForm]\!])$ function takes a formula, the set of actual parameters, the set of formal parameters, the current value of the instantiation

counter, and the set of local variables as input and returns the formula in which the formal parameters are replaced with the corresponding actual parameters and the local variables are renamed uniquely using the value of the instantiation counter.

- $Rename_V$: $[\![IdL]\!] \times N \rightarrow [\![IdL]\!]$ function takes a set of variables and an instantiation counter as input and renames the set of input variables uniquely using the value of the instantiation counter.
- $NextStateVar$: $[\![Form]\!] \rightarrow [\![IdL]\!]$ function takes a composite formula as input and returns the set of variables whose next state versions appear in the composite formula.
- $Identity$: $[\![IdL]\!] \rightarrow [\![Form]\!]$ function takes a set of variables as input and returns a composite formula that preserves the current value of every current state variable in the input in the next state. If the input is an empty set then it returns $\mathbf{true}$.
- $Guard$: $[\![Form]\!] \rightarrow [\![Form]\!]$ function takes a composite formula that denotes a transition relation as input and returns a composite formula that corresponds to the domain of the transition relation. This can be computed by existentially quantifying out all the next state variables in the input formula.

Finally, a tuple is denoted by enclosing the list of its components with $\langle\rangle$. Projections of a tuple are shown using the subscripts consisting of the first character or the first two characters of the component domains, e.g., given a tuple $\epsilon \in SysEnv = Init \times State$, $\epsilon_I \in Init$ and $\epsilon_S \in State$.

b) Semantic Functions: We define the semantics of the Action Language by mapping an Action Language specification to a tuple (M, ϕ) using denotational semantics, where M is a transition system and ϕ is the CTL property of the system. The transition system M is a tuple (I, S, R) where I, S, and R denote the initial states, the state space, and the transition relation, respectively. Each module in an Action Language specification is associated with a tuple that consists of: a composite formula describing the initial states ($\mathbf{true}$ by default), a composite formula describing the state space ($\mathbf{true}$ by default), a CTL formula describing the property, a counter keeping the number of instantiations, a set of local variables, a set of formal parameters, a set of parameterized integer constants, a set of action definitions, and a composite formula describing its transition relation. Note that the set Env defined above is the set of such tuples.

Below we present the definitions of the semantics functions and their explanations.

1) Ξ : $VarDecl \rightarrow [\![Id]\!] \rightarrow VarEnv \rightarrow VarEnv$

a) $\Xi[\![\texttt{boolean}\ Id]\!]m\vartheta =$
$$\begin{cases} \langle \vartheta_L[[\![Id]\!] \cup \vartheta_L(m)/m], \vartheta_F, \vartheta_P \rangle & \text{if } [\![Id]\!] \notin \vartheta_F(m) \\ \langle \vartheta_L, \vartheta_F, \vartheta_P \rangle & \text{otherwise} \end{cases}$$

b) $\Xi[\![\texttt{integer}\ Id]\!]m\vartheta =$
$$\begin{cases} \langle \vartheta_L[[\![Id]\!] \cup \vartheta_L(m)/m], \vartheta_F, \vartheta_P \rangle & \text{if } [\![Id]\!] \notin \vartheta_F(m) \\ \langle \vartheta_L, \vartheta_F, \vartheta_P \rangle & \text{otherwise} \end{cases}$$

c) $\Xi[\![\texttt{enumerated}\ Id\ \{IdL\}]\!]m\vartheta =$
$$\begin{cases} \langle \vartheta_L[[\![Id]\!] \cup \vartheta_L(m)/m], \vartheta_F, \vartheta_P \rangle & \text{if } [\![Id]\!] \notin \vartheta_F(m) \\ \langle \vartheta_L, \vartheta_F, \vartheta_P \rangle & \text{otherwise} \end{cases}$$

d) $\Xi[\![\texttt{parameterized}\ Id]\!]m\vartheta = \langle \vartheta_L, \vartheta_F, \vartheta_P[[\![Id]\!] \cup \vartheta_P(m)/m] \rangle$.

e) $\Xi[\![\ VarDecl_1\ \ VarDecl_2\ ]\!]m\vartheta = \Xi[\![\ VarDecl_2\ ]\!]m\Xi[\![\ VarDecl_1\ ]\!]m\vartheta$.

A variable becomes the local variable of the module in which it is defined provided that it is not used as a formal parameter in that module (cases 1.a, 1.b, and 1.c). Since the values of the parameterized variables do not change, they need to be kept separately and treated in a different way (see 6.a).

2) $\Gamma : Sys \rightarrow [\![Id]\!] \rightarrow SysEnv \rightarrow SysEnv$

a) $\Gamma[\![\texttt{initial}\ :\ Form]\!]m\epsilon = \langle \epsilon_I[\epsilon_I(m)\ \wedge\ [\![Form]\!]/m], \epsilon_S \rangle$.

b) $\Gamma[\![\texttt{restrict}\ :\ Form]\!]m\epsilon = \langle \epsilon_I, \epsilon_S[\epsilon_S(m)\ \wedge\ [\![Form]\!]/m] \rangle$.

c) $\Gamma[\![\ Sys_1\ \ Sys_2\ ]\!]m\epsilon = \Gamma[\![\ Sys_2\ ]\!]m\Gamma[\![\ Sys_1\ ]\!]m\epsilon$.

The initial states of a module are described by the conjunction of the composite formulas from the system definitions inside that module with the `initial` keyword (cases 2.a and 2.c) and the composite formulas describing the initial states of its submodules that are instantiated in the transition relation definition (case 3, equation 2 below). The state space of a module is described by the conjunction of the composite formulas in the system definitions using the `restrict` keyword inside that module (cases 2.b and 2.c) and the composite formulas describing the state space of its submodules that are instantiated in the transition relation definition (case 3, equation 3 below). When the initial states or the state space is not specified (i.e., there are no system definitions) the default value, **true**, is used.

3) $\Theta : ModInst \rightarrow [\![Id]\!] \rightarrow Env \rightarrow CurEnv$

$\Theta[\![Id\ \texttt{(}\ IdL\ \texttt{)}]\!]m_1\beta = \rho$ where $m = [\![Id]\!]$, $l = [\![IdL]\!]$ and

$$\rho_{IC} = \beta_{IC}[\beta_{IC}(m) + 1/m] \quad (1)$$

$$\rho_I = \beta_I[Rename_F(\beta_I(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m)) \wedge \beta_I(m_1)/m_1] \quad (2)$$

$$\rho_S = \beta_S[Rename_F(\beta_S(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m) \wedge \beta_S(m_1)/m_1] \quad (3)$$

$$\rho_T = Rename_F(\beta_T(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m)) \quad (4)$$

$$\rho_C = \beta_C[Rename_F(\beta_C(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m)) \wedge \beta_C(m_1)/m_1] \quad (5)$$

$$\rho_L = \beta_L[Rename_V(\beta_L(m), \rho_{IC}(m)) \cup \beta_L(m_1)/m_1] \quad (6)$$

The local variables of a module are uniquely renamed for each instantiation using the instantiation counter for that module. Each instantiation causes the instantiation counter to be incremented by one (equation 1). The environment for a new instantiation of a module is defined by renaming the local variables and by replacing the formal parameters with the corresponding actual parameters in the composite formulas describing the initial states and the state space of the module (equations 2 and 3), the composite formula describing the transition relation of the module (equation 4), and the CTL formula describing the property of the module (equation 5). All these transformations are achieved using the $Rename_F$ function. Finally, after renaming (using the $Rename_V$ function), the local variables of the instantiated module are added to that of the parent module in which it is instantiated (equation 6).

4) $\Lambda : Comp \rightarrow [\![Id]\!] \rightarrow Env \rightarrow CurEnv$

a) $\Lambda[\![Id]\!]m\beta = \rho$ where $\rho_T = \beta_A([\![Id]\!])$ and $\forall X \in \{I, S, C, L, IC\}, \rho_X = \beta_X$.

b) $\Lambda[\![ModInst]\!]m\beta = \Theta[\![ModInst]\!]m\beta$.

c) Let $\rho' = \Lambda[\![Comp_1]\!]m\beta$ and $\rho'' = \Lambda[\![Comp_2]\!]m\langle\rho'_I, \rho'_S, \rho'_C, \rho'_{IC}, \rho'_L, \beta_F, \beta_P, \beta_A, \beta_T\rangle$.
$\Lambda[\![Comp_1 \mid Comp_2]\!]m\beta = \rho'''$ where

$$\rho'''_T = (\rho'_T \wedge Identity(NextStateVar(\rho''_T) \setminus NextStateVar(\rho'_T))) \vee (\rho''_T \wedge Identity(NextStateVar(\rho'_T) \setminus NextStateVar(\rho''_T)))$$

and $\forall X \in \{I, S, C, L, IC\},\ \rho'''_X = \rho''_X$.

d) Let $\rho' = \Lambda[\![Comp_1]\!]m\beta$ and $\rho'' = \Lambda[\![Comp_2]\!]m\langle\rho'_I, \rho'_S, \rho'_C, \rho'_{IC}, \rho'_L, \beta_F, \beta_P, \beta_A, \beta_T\rangle$.
$\Lambda[\![Comp_1 \,\&\, Comp_2]\!]\beta = \rho'''$ where

$$\rho'''_T = (\rho'_T \vee \neg Guard(\rho'_T) \wedge Identity(NextStateVar(\rho'_T))) \wedge (\rho''_T \vee \neg Guard(\rho''_T) \wedge Identity(NextStateVar(\rho''_T)))$$

and $\forall X \in \{I, S, C, L, IC\},\ \rho'''_X = \rho''_X$.

The Action Language supports both asynchronous and synchronous composition of actions and module instantiations. Asynchronous composition (denoted by $\mid$) models interleaving semantics of concurrency, which corresponds to combining the transition relations of individual components using disjunction (case 4.c). In asynchronous composition, when a transition is executed, the values of the variables that are modified solely by the other transition are preserved. In synchronous composition (denoted by &) two transitions are executed in parallel, which corresponds to combining the transition relations of individual components using conjunction (case 4.d). However, if one of the transitions is disabled then it does not block the other transition.

5) $\Omega : Action \rightarrow Act \rightarrow Act$

a) $\Omega[\![Id : Form]\!]\alpha = \alpha[[\![Form]\!]/[\![Id]\!]]$.

b) $\Omega[\![Action_1\ Action_2]\!]\alpha = \Omega[\![Action_2]\!]\Omega[\![Action_1]\!]\alpha$.

Actions model atomic transitions of the system. Actions are specified as composite formulas on current and next state variables.

6) $\Psi : ModTrans \rightarrow Env \rightarrow Env$

a) $\Psi[\![Id : Form]\!]\beta = \beta'$ where
$\beta'_T = \beta_T[[\![Form]\!] \wedge Identity(\beta_P([\![Id]\!]))/[\![Id]\!]]$
and $\forall X \in \{I, S, C, IC, L, F, A, T\},\ \beta'_X = \beta_X$.

b) $\Psi[\![Id : Comp]\!]\beta = \beta'$ where

$$\begin{aligned} \rho &= \Lambda[\![Comp]\!][\![Id]\!]\beta \\ \beta'_T &= \beta_T[\rho_T \wedge Identity(\beta_P([\![Id]\!]))/[\![Id]\!]] \\ \beta'_X &= \rho_X,\ \forall X \in \{I, S, C, IC, L\} \\ \beta'_F &= \beta_F,\ \beta'_P = \beta_P,\ and\ \beta'_A = \beta_A \end{aligned}$$

Behavior of a module is defined either as a composite formula on current and next state variables or as a composition of instantiations of its submodules and its actions.

7) $\Phi : Prop \rightarrow [\![Id]\!] \rightarrow CtlProp \rightarrow CtlProp$

   a) $\Phi[\![\texttt{spec :}\ CtlForm]\!]m\tau = \tau[\tau(m) \wedge [\![CtlForm]\!]/m]$.

   b) $\Phi[\![Prop_1\ Prop_2]\!]m\tau = \Phi[\![Prop_2]\!]m\Phi[\![Prop_1]\!]m\tau$.

   The CTL property of a module is described by conjunction of the CTL formulas given in all property specifications of that module (cases 7.a and 7.b) and the CTL formulas describing the CTL properties of its submodules that are instantiated in that module's transition relation definition (case 3, equation 5).

8) $\Upsilon : Module \rightarrow Env \rightarrow Env$

   a) $\Upsilon[\![\texttt{module}\ Id\ \texttt{(}\ IdL\ \texttt{)}\ VarDecl\ Sys\ Module\ Action\ ModTrans\ Prop\ \texttt{endmodule}]\!]\beta = \beta''$ where

$$\begin{aligned}
m &= [\![Id]\!] \\
l &= [\![IdL]\!] \\
\epsilon &= \Gamma[\![Sys]\!]m\langle\beta_I, \beta_S\rangle \\
\beta' &= \Upsilon[\![Module]\!]\langle\epsilon_I, \epsilon_S, \beta_C, \beta_{IC}, \beta_L, \beta_F[l/m], \beta_P, \beta_A, \beta_T\rangle \\
\alpha &= \Omega[\![Action]\!]\beta'_A \\
\tau &= \Phi[\![Prop]\!]m\beta'_C \\
\beta'' &= \Psi[\![ModTrans]\!]m\langle\beta'_I, \beta'_S, \tau, \beta'_{IC}, \beta'_L, \beta'_F, \beta'_P, \alpha, \beta'_T\rangle
\end{aligned}$$

   Semantics of a module m is defined by $((I, S, R), \phi)$ where

$$I = \beta''_I(m),\ S = \beta''_S(m),\ R = \beta''_T(m),\ and\ \phi = \beta''_C(m).$$

   Therefore, the transition system $((I, S, R), \phi)$ that is defined by an Action Language specification is defined as

$$I = \beta''_I(\texttt{main}),\ S = \beta''_S(\texttt{main}),\ R = \beta''_T(\texttt{main}),\ and\ \phi = \beta''_C(\texttt{main}).$$

   where the initial environment β is defined as

$$\beta_I = \lambda x.\mathbf{true},\ \beta_S = \lambda x.\mathbf{true}, \beta_C = \lambda x.undefined,\ \beta_{IC} = \lambda x.0,$$

$$\beta_L = \lambda x.\emptyset,\ \beta_F = \lambda x.\emptyset,\ \beta_P = \lambda x.\emptyset,\ \beta_A = \lambda x.\emptyset,\ \beta_T = \lambda x.\emptyset.$$

In Figure 5 we show a small Action Language specification inspired by the case study discussed in Figure 1. The goal of this example is to demonstrate the semantics of the Action Language composition operators. In this example specification, there are two submodules of module `main`: `runway` and `environment`. Variable `rw16L` models availability status of the runway 16L and `ev16L` models events that denote either an enter request or an exit request. Module `runway` models status change of a runway and its behavior is modeled by asynchronous

```
module main()
   enumerated ev16L {enter, exit};
   boolean rw16L;

   module runway(rw, ev)
      boolean rw;
      enumerated ev {enter, exit};
      initial: rw;
      r1: rw and ev=enter and !rw';
      r2: !rw and ev=exit and rw';
      runway: r1 | r2;
   endmodule

   module environment(ev)
      enumerated ev {enter,exit};
      initial: ev=enter;
      e1: ev=enter and ev'=exit;
      e2: ev=exit and ev'=enter;
      e3: ev=exit and ev'=exit;
      environment: e1 | e2 | e3;
   endmodule

   main: runway(rw16L,ev16L) & environment(ev16L);

   spec: AG(!rw16L => AX(rw16L))
endmodule
```

Fig. 5. A sample Action Language specification modeling status change of runway 16L as the relevant event occurs.

composition of its actions `r1` and `r2`. Module `environment` models the occurrences of the enter and exit events for a particular runway. Whenever an enter event occurs, in the next state an exit event occurs (action `e1`). After the exit event occurs, either an enter event (`action e2`) or an exit event (`action e3`) occurs nondeterministically. The behavior of the `environment` module is modeled by the asynchronous composition of its actions `e1`, `e2`, and `e3`. The behavior of the whole system is defined by synchronous composition of instantiations of the modules `runway` and `environment` using `rw16L` and `ev16L`. The correctness property states that whenever the runway 16L is occupied it is emptied in the next state. Table I shows the transition systems that correspond to the modules `runway`, `environment`, and `main`.

IV. The Action Language Verifier

The Action Language Verifier (ALV) is an automated verification tool for analyzing Action Language specifications using infinite state model checking techniques. The main challenge in model checking [CGP99] is to alleviate the *state explosion* problem caused by the exponential growth of the state space with the increasing number of concurrent components and variables. Symbolic model checking [McM93] provides a way to address the state

Module	I	S	R
`runway`	rw	**true**	$(rw \wedge ev = enter \wedge \neg rw') \vee$ $(\neg rw \wedge ev = exit \wedge rw')$
`environment`	$ev = enter$	**true**	$(ev = enter \wedge ev' = exit) \vee$ $(ev = exit \wedge ev' = enter) \vee$ $(ev = exit \wedge ev' = exit)$
`main`	$rw16L \wedge ev16L = enter$	**true**	$((rw16L \wedge ev16L = enter \wedge \neg rw16L')$ $\vee (\neg rw16L \wedge ev16L = exit \wedge rw16L')$ $\vee \neg(rw16L \wedge ev16L = enter \vee$ $\neg rw16L \wedge ev16L = exit) \wedge$ $rw16L' = rw16L) \wedge ((ev16L = enter \wedge$ $ev16L' = exit) \vee (ev16L = exit \wedge$ $ev16L' = enter) \vee (ev16L = exit \wedge$ $ev16L' = exit) \vee \neg(ev16L = enter \vee$ $ev16L = exit) \wedge ev16L' = ev16L)$

TABLE I

THE TRANSITION SYSTEMS THAT CORRESPOND TO THE MODULES `runway`, `environment`, AND `main` OF THE ACTION LANGUAGE SPECIFICATION IN FIGURE 5. I, S, AND R DENOTE THE INITIAL STATES, THE STATE SPACE, AND THE TRANSITION RELATION, RESPECTIVELY.

explosion problem by encoding the state space symbolically (as, for example, Boolean logic formulas) instead of explicitly enumerating the states. Since the size of a symbolic representation can be much smaller than the set of states it represents, symbolic representations enable verification of very large (and even infinite) state systems.

Given a transition system $T = (I, S, R)$, where I, S, and R denote the *initial states*, the *state space*, and the *transition relation*, respectively, and a CTL property ϕ, model checking problem is to decide whether $M \models \phi$. Let $[\![\phi]\!]$ denotes the states that satisfy ϕ, then $M \models \phi$ if and only if $I \Rightarrow [\![\phi]\!]$. Throughout this paper we assume that the set of initial states, the state space and the transition relation are symbolically represented as formulas. A symbolic model checker first computes a formula that characterizes $[\![\phi]\!]$ and then checks if $I \Rightarrow [\![\phi]\!]$ holds.

ALV is a symbolic model checker. Its distinguishing feature is to encode the transition system using the composite symbolic representation to provide the flexibility and the extensibility required for analyzing software specifications. Moreover, ALV can be used to analyze infinite-state systems. In general, CTL model checking for infinite-state systems is undecidable. However, ALV employs several heuristics for speeding up or guaranteeing the termination of the fixpoint computations required in symbolic model checking. These heuristics are conservative and generate approximations of the least or greatest fixpoints, i.e., ALV does not generate any false positives or false negatives, however, its analysis may be inconclusive.

ALV uses the composite symbolic representation, as implemented by the Composite Symbolic Library [YKB03], to encode the sets of states and the transition relation. In the Composite Symbolic Library, different symbolic representations are combined using the *composite model checking* approach [BGL00]. Our current implementation

of the Composite Symbolic Library uses two symbolic representations: BDDs for boolean logic formulas and polyhedral [BGP99] or automata-based [BB03] representations for Presburger arithmetic formulas. We call these representations *basic symbolic representations*.

The atomic properties (AP) in ALV are not restricted to propositional properties as in finite state model checkers [CGP99]. Any property that can be expressed using a combination of Presburger arithmetic and Boolean logic can be used as an atomic property. These atomic properties combined with the CTL temporal operators and the Boolean connectives form the property specification language of ALV.

Each variable type in the input Action Language specification is assigned to the corresponding basic symbolic representation for that variable type. Boolean and enumerated variables in the Action Language specifications are mapped to BDD representation, and integers are mapped to an arithmetic constraint representation. Currently, the Composite Symbolic Library uses CUDD package [CUD] for the BDD representation, the Omega Library [Ome] for the polyhedral representation and an automata encoding of the Presburger arithmetic built on top of the MONA [HJJ$^+$95] package for the automata-based representation [BB03]. We encode the set of states and the transition relation in Disjunctive Normal Form (DNF), as a disjunction of conjunctions of basic symbolic representations (e.g., a disjunct consists of conjunction of a boolean formula stored as a BDD representing the states of boolean and enumerated variables, and a Presburger arithmetic constraint representing the states of integer variables). We call this DNF representation a *composite symbolic representation* since it combines different basic symbolic representations. A composite formula, p, is represented in DNF as

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

where p_{it} denotes 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.

Heuristics for efficient manipulation of composite symbolic representation, including implementations of the basic operations such as negation, conjunction, and disjunction has been implemented in the Composite Symbolic Library and discussed in [YKB03]. So, we do not discuss implementations of these basic operations in this paper, rather, we focus on the verification heuristics implemented in ALV on top of the functionality provided by the Composite Symbolic Library.

The rest of this section is organized as follows. Section IV-A discusses the fixpoint computations for symbolic CTL model checking. Section IV-B presents the heuristics for accelerating or guaranteeing the convergence of the fixpoint computations and the heuristics for efficient fixpoint computations. Section IV-C explains the counter-example generation algorithms.

A. Fixpoint Computations

The pre- and post-condition computations are among the basic operations in a symbolic model checker. Given a set of states p and a transition relation R, the pre-condition, $\text{PRE}(p, R)$, is the set of all states that can reach a state

in p with a single transition in R (i.e., the predecessors of all the states in p). The post-condition, $\text{POST}(p, R)$, is defined similarly.

Given states p and a transition relation R, both represented using the composite symbolic representation as

$$p = \bigvee_{i=1}^{n_p} \bigwedge_{t=1}^{T} p_{it} \quad R = \bigvee_{i=1}^{n_R} \bigwedge_{t=1}^{T} r_{it},$$

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 existential variable elimination in $\text{PRE}(p, R)$ computation distributes over the disjunctions, and due to the partitioning of the variables based on the basic symbolic types, existential variable elimination also distributes over the conjunction above for the composite symbolic representation [BGL00]. Since the pre-condition computation distributes over both the disjunction and the conjunction for the composite symbolic representation, we are able to compute the pre-condition of a composite representation using pre-condition computations of the basic symbolic representations.

The temporal operator EX corresponds to the pre-condition computation, i.e., $[\![\text{EX } p]\!] \equiv \text{PRE}(p, R)$. AX can also be computed as $[\![\text{AX } p]\!] \equiv \neg\text{PRE}(\neg p, R)$. The rest of the CTL operators can be computed as least and greatest fixpoints using EX and AX [CGP99]

$$\begin{aligned}
[\![p \text{ EU } q]\!] &\equiv \mu x \,.\, q \vee (p \wedge [\![\text{EX } x]\!]) \\
[\![p \text{ AU } q]\!] &\equiv \mu x \,.\, q \vee (p \wedge [\![\text{AX } x]\!]) \\
[\![\text{EG } p]\!] &\equiv \nu x \,.\, p \wedge [\![\text{EX } x]\!] \\
[\![\text{AG } p]\!] &\equiv \nu x \,.\, p \wedge [\![\text{AX } x]\!]
\end{aligned}$$

However, the above characterizations of AU and EG are not complete if we do not restrict the transition relation to be total. A transition relation is total if every state has a next state, which is a common assumption in model checking literature [CGP99]. Since a non-total transition system can have states that do not have any next states, AX $\mathbf{false}$ will be satisfied in such states vacuously. Hence, those states will satisfy AF $\mathbf{false}$ too. This creates a problem, since we will have states that 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

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

where $\mathit{AtLeastOne}$ denotes the states that have at least one successor, and can be computed as:

$$\mathit{AtLeastOne} \equiv \neg[\![\text{AX } \mathbf{false}]\!] \equiv [\![\text{EX } \mathbf{true}]\!] \equiv \text{PRE}(\mathbf{true}, R)$$

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

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

```
1  EG(p): composite formula
2  p, s, s_old, None: composite formula
3  let R denote the transition relation
4  let None denote the states with no successors
5  s ← p
6  s_old ← false
7  while ¬isEquivalent(s, s_old) do
8        s_old ← s
9        s ← (PRE(s, R) ∨ None) ∧ s_old
10 return s
```

Fig. 6. The algorithm for computing the states that satisfy EGp

```
1  EU(p, q): composite formula
2  p, q, s, s_old: composite formula
3  s ← q
4  s_old ← false
5  let R denote the transition relation
6  while ¬isEquivalent(s, s_old) do
7        s_old ← s
8        s ← PRE(s, R) ∧ p ∨ s_old
9  return s
```

Fig. 7. The algorithm for computing the states that satisfy p EU q

where $None$ denotes the states that have no successors (i.e., $None \equiv \neg AtLeastOne$). Note that, this fixpoint always considers all the paths that end in a state with no successors. In the Action Language Verifier $AtLeastOne$ and $None$ are pre-computed and stored with the transition system, so that they are not recomputed in each fixpoint iteration.

Based on the equivalences among the CTL operators [CGP99], one can show that $\{\text{EX}, \text{EG}, \text{EU}\}$ forms a basis for CTL, i.e., all CTL formulas can be expressed using only these temporal operators. Similarly, another basis for CTL is $\{\text{EX}, \text{EU}, \text{AU}\}$. In the Action Language Verifier both basis are implemented and can be chosen by the user. Another option is to leave the temporal operators as they are. In that case the Action Language Verifier computes each temporal operator directly using the corresponding fixpoint.

Figures 6, 7 and 8 show the algorithms for computing the states that satisfy the CTL formulas EGp, p EU q, and p AU q based on the fixpoint characterizations of these temporal operators. Note that, in an infinite state model checker like ALV termination of these fixpoint computations 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

1 AU(p, q): composite formula
2 p, q, s, s_{old}, $AtLeastOne$: composite formula
3 $s \leftarrow q$
4 $s_{old} \leftarrow \mathbf{false}$
5 **let** R denote the transition relation
6 **let** $AtLeastOne$ denote the states with at least one successor
7 **while** $\neg isEquivalent(s, s_{old})$ **do**
8 $\quad s_{old} \leftarrow s$
9 $\quad s \leftarrow \neg \text{PRE}(\neg s, R) \wedge p \wedge AtLeastOne \vee s_{old}$
10 **return** s

Fig. 8. The algorithm for computing the states that satisfy p AU q

least or the greatest fixpoint based on the type of the iteration. In the next section we will discuss heuristics for computing approximations of least and greatest fixpoints.

B. Heuristics

We use the approximate fixpoint computation approach from [BGP99] to compute approximations of least and greatest fixpoints. Assume that we wish to compute the states that satisfy a temporal property ϕ in a transition system $M = (S, I, R)$. If M is an infinite state system the fixpoint computations described above may not converge. Instead of computing $[\![\phi]\!]$, i.e., the set of states that satisfy the temporal property ϕ, assume that we compute a *lower bound* for $[\![\phi]\!]$, denoted $[\![\phi]\!]^-$, such that $[\![\phi]\!]^- \Rightarrow [\![\phi]\!]$. Note that, $I \Rightarrow [\![\phi]\!]^-$ implies that $I \Rightarrow [\![\phi]\!]$, i.e., showing $I \Rightarrow [\![\phi]\!]^-$ means that the transition system M satisfies the property ϕ. However, if $I \not\Rightarrow [\![\phi]\!]^-$, we cannot conclude that the transition system does not satisfy the property since it is possible that $I \not\Rightarrow [\![\phi]\!]^-$ but $I \Rightarrow [\![\phi]\!]$. In that case, we can compute a lower bound for the negated property: $[\![\neg\phi]\!]^-$. If $I \wedge [\![\neg\phi]\!]^-$ is satisfiable then we can conclude that the transition system M does not satisfy the property ϕ. If both cases fail, i.e., both $I \not\Rightarrow [\![\phi]\!]^-$ and $I \wedge [\![\neg\phi]\!]^- \equiv \mathbf{false}$, our verification effort will be inconclusive.

Since ALV computes the temporal formulas recursively starting from the innermost temporal operators, in order to implement this approach in ALV, we have to compute an approximation to a formula by first computing the 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 such as $\neg\phi$, we can compute an upper bound $[\![\phi]\!]^+$ for the subformula ϕ where $[\![\phi]\!] \Rightarrow [\![\phi]\!]^+$, and then let $[\![\neg\phi]\!]^- \equiv \neg([\![\phi]\!]^+)$. Similarly, we can compute an upper bound for $\neg\phi$ using a lower bound for ϕ. Thus, we need to implement algorithms to compute both lower and upper bounds of temporal formulas.

In this section we explain heuristics for computing lower and upper approximations of least and greatest fixpoint computations using truncated fixpoint calculations and the widening and the collapsing operators. We also discuss

```
1   module main()
2    parameterized integer size;
3    integer x,y;
4    enumerated pc {a,b,c};
5    restrict: size>0;
6    initial: x=-size and y=size;
7    a1: pc=a and x<0 and x'=x+1 and pc'=a;
8    a2: pc=a and x=0 and pc'=b;
9    a3: pc=b and y>0 and y'=y-1 and pc'=b;
10   a4: pc=b and y=0 and pc'=c;
11   a5: pc=c and x'=-size and y'=size and pc'=a;
12   main: a1 | a2 | a3 | a4 | a5;
13   spec: AG(x<=y)
14  endmodule
```

Fig. 9. A sample Action Language specification.

heuristics for accelerating fixpoint computations based on self-loop-closures and restricting the state space to an over-approximation of the reachable states. We also propose two new heuristics, which are marking and dependency heuristics, for avoiding the redundant computations during fixpoint computations. We have implemented all these heuristics in ALV.

We will explain these heuristics using the sample specification given in Figure 9, in which two integer variables (`x` and `y`) are periodically assigned values that have the same absolute value and are of different signs (`x=-size` and `y=size`, by action `a5`). Between two such consecutive assignments the negative value is incremented until it becomes zero (actions `a1` and `a2`) and then the positive value is decremented until it becomes zero (actions `a3` and `a4`). The correctness property is specified as `x` is always smaller than or equal to `y`.

Truncated Fixpoint 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, the result of each iteration of a greatest fixpoint computation gives an upper bound for the greatest fixpoint. For instance, for the specification in Figure 9, truncating the fixpoint computation of $EF(x > y)$, given in Figure 10, after two iterations yields I_2, which is a lower approximation for the least fixpoint. ALV has a flag that 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 [CC77] generalized to the composite symbolic representation [BGL00]. Let p and q denote two composite formulas such that $p \Rightarrow q$ where $p = \bigvee_{i=1}^{n} \bigwedge_{t=1}^{T} p_{it}$ and $q = \bigvee_{i=1}^{m} \bigwedge_{t=1}^{T} q_{it}$. Then the widening operator for the composite symbolic representation is defined as:

$$p \bigtriangledown q \equiv \bigvee_{1\le i\le n,\ 1\le j\le m,\ p_i\Rightarrow q_j} \bigwedge_{t\in T} p_{it} \bigtriangledown_t q_{jt} \ \vee \bigvee_{1\le j\le m,\forall i,1\le i\le n,\ p_i\not\Rightarrow q_j} q_j \ \vee \bigvee_{1\le i\le n,\forall j,1\le j\le m,\ p_i\not\Rightarrow q_j} p_i$$

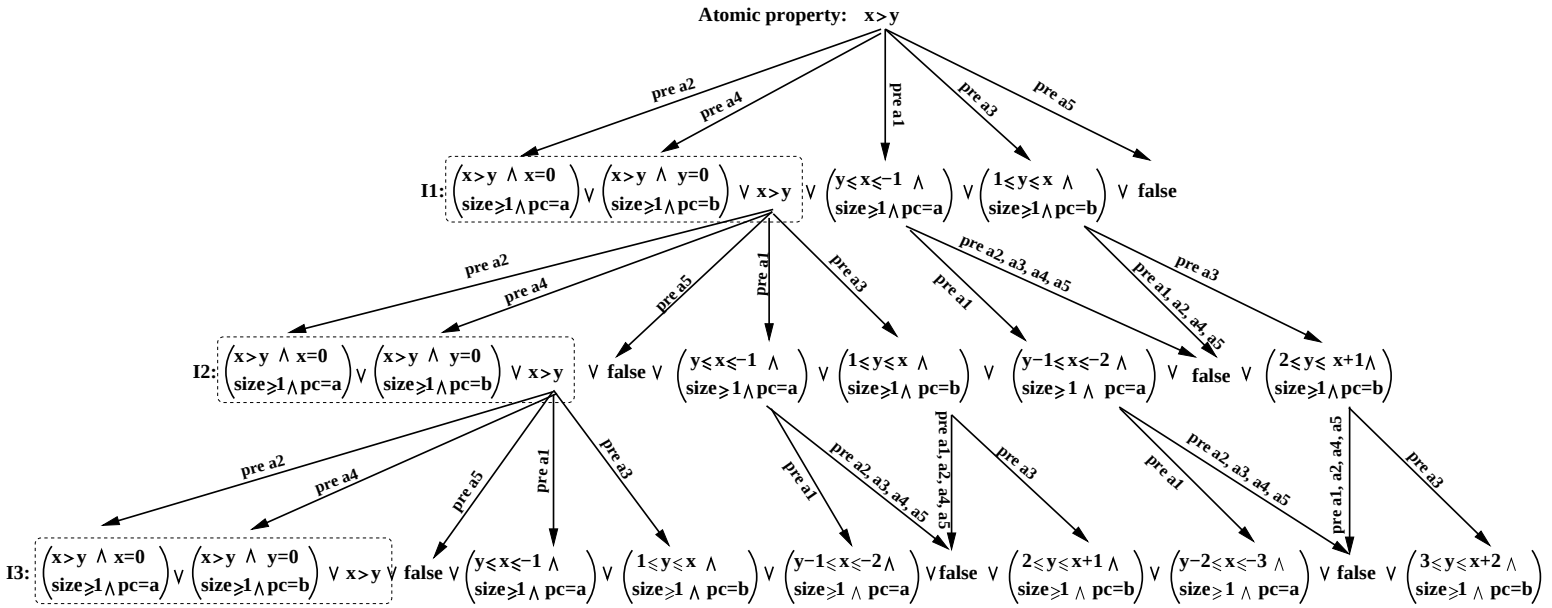


Fig. 10. The result of the first three iterations of computing $EF(x > y)$ naively for the Action Language specification given in Figure 9. I_1, I_2, and I_3 denote the result of the first, the second, and the third iteration, respectively. $pre\ a_i$ denotes that the constraint pointed by the arrow is obtained by performing the pre-condition computation on the source constraint using action a_i, where $1 \leq i \leq 5$. The constraints in each of the dashed rounded boxes are simplified into constraint, $x > y$, by the Composite Symbolic Library.

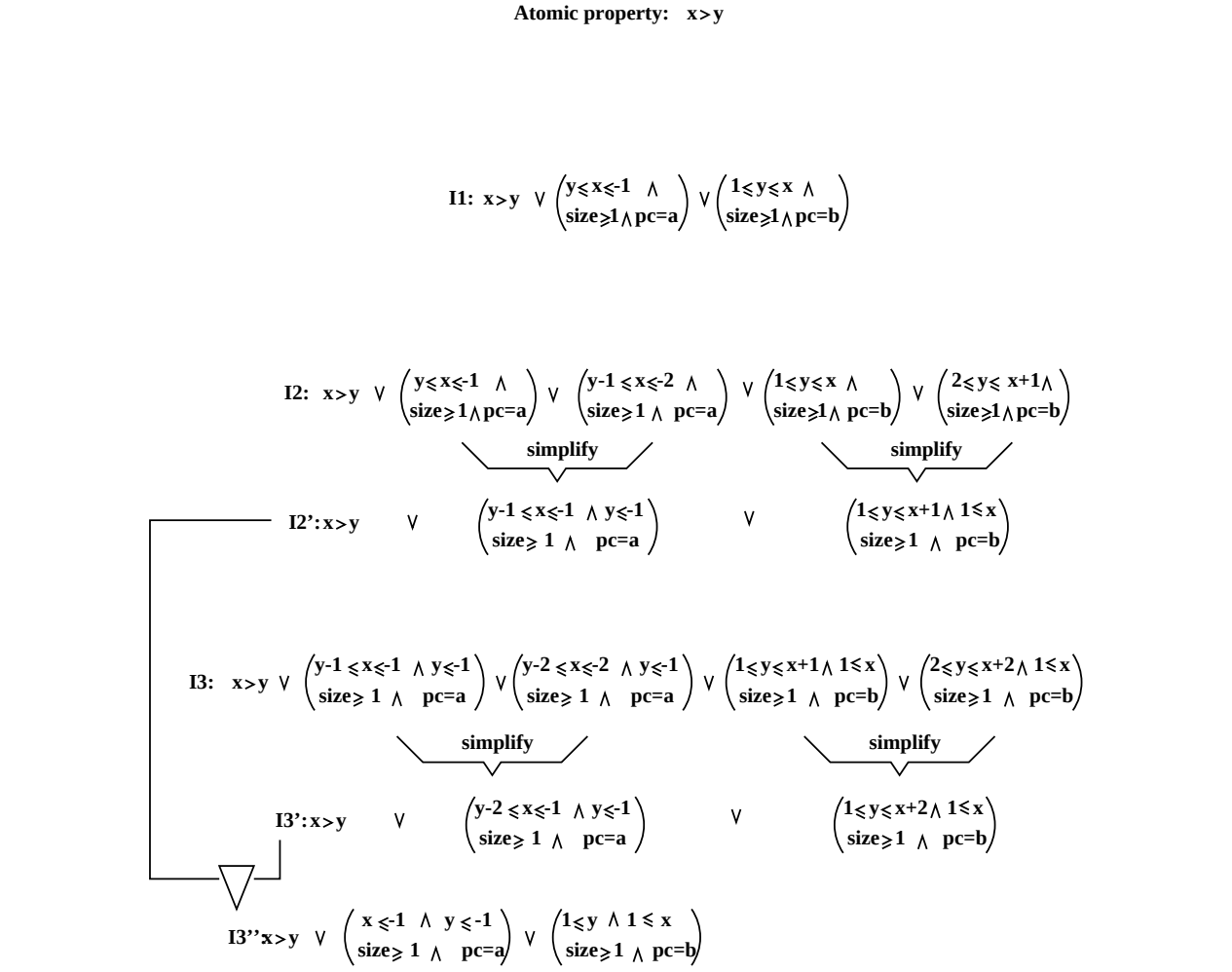


Fig. 11. The results of the first three iterations of computing $EF(x > y)$ for the Action Language specification given in Figure 9. I_1, I_2, and I_3 denote the results of the first, the second, and the third iteration, respectively. I_2' and I_3' denote the results of the iterations after the simplification operation and I_3'' denotes the result of the iteration after the widening operation ($\triangledown$).

where $p_i = \bigwedge_{t=1}^{T} p_{it}$, $q_j = \bigwedge_{t=1}^{T} q_{jt}$, and n, T, p_{it}, and $\triangledown_t$ denote the number of basic symbolic representations, the set of basic symbolic representations, a symbolic representation of type t, and the type-specific widening operator for type t, respectively. Note that the widening operator for the composite representation simply uses the widening operators for the basic symbolic representations on pairs of disjuncts from p and q that satisfy the constraint $p_i \Rightarrow q_j$.

As the widening operator for the boolean representation ($\triangledown_{bool}$), we simply use the disjunction operation. There are two widening operators ($\triangledown_{int}$) for the integer domain based on the type of symbolic representation used for the integer domain. For the automata representation we use the widening operator defined in [BB04]. For the polyhedral representation we use the widening operator defined in [BGP99] that generalizes the convex widening operator in

[CH78] to Presburger arithmetic formulas.

A widening operator has to satisfy the following two constraints: 1) For any p and q, $p \vee q \Rightarrow p \bigtriangledown q$, i.e., widening operator should provide an upper bound for the disjunction operation. 2) The approximate fixpoint sequence computed using the widening operator should eventually converge. The widening operator described above satisfies the first constraint. In order to satisfy the second constraint, we have to bound the number of disjuncts in the composite symbolic representation. This means that after we reach the bound, we need to merge the disjuncts in the composite representation. We can merge two disjuncts $p_1 = \bigwedge_{t=1}^{T} p_{1t}$ and $p_2 \bigwedge_{t=1}^{T} p_{2t}$ as

$$p_{1,2} = \bigwedge_{t=1}^{T} (p_{1t} \vee p_{2t})$$

where $p_{1,2}$ provides an upper bound for $p_1 \vee p_2$, i.e., $p_1 \vee p_2 \Rightarrow p_{1,2}$. Using this merging operation we can limit the number of disjuncts in the composite symbolic representation and guarantee convergence of the fixpoint iterations to an upper approximation of the least fixpoint.

Figure 11 shows an example application of the widening operator, which is used for the computation of $EF(x > y)$ fixpoint for the specification in Figure 9. In this example ALV was directed to start applying the widening operation after the second iteration. For computing $I'_2 \bigtriangledown I'_3$, ALV compares each pair of disjuncts where one of them comes from I'_2 and the other comes from I'_3. For pairs that satisfy the subsumption relation it applies the widening operation. For instance, the disjunct $y - 1 \leq x \leq -1 \wedge y \leq -1 \wedge size \geq 1 \wedge pc = a$ that comes from I'_2 is subsumed by the disjunct $y - 2 \leq x \leq -1 \wedge y \leq -1 \wedge size \geq 1 \wedge pc = a$ that comes from I'_3. Applying integer based widening operation on the integer parts of these two disjuncts, i.e., $(y - 1 \leq x \leq -1 \wedge y \leq -1 \wedge size \geq 1) \bigtriangledown_{int} (y - 2 \leq x \leq -1 \wedge y \leq -1 \wedge size \geq 1)$, yields $x \leq -1 \wedge y \leq -1 \wedge size \geq 1$.

To compute lower-bounds for the greatest fixpoint computations we define the dual of the widening operator and call it the *collapsing* operator (and denote it with $\bar{\bigtriangledown}$). Let p and q denote two composite formulas such that $q \Rightarrow p$. Then $p \bar{\bigtriangledown} q$ is defined as

$$p \bar{\bigtriangledown} q \equiv \bigvee_{1 \leq i \leq n,\ 1 \leq j \leq m,\ p_i \Rightarrow q_j} \bigwedge_{t \in T} p_{it} \bar{\bigtriangledown}_t q_{jt} \ \vee \bigvee_{1 \leq i \leq n,\ \neg \exists 1 \leq j \leq m,\ p_i \Rightarrow q_j} p_i \ \vee \bigvee_{1 \leq j \leq m,\ \neg \exists 1 \leq i \leq n,\ p_i \Rightarrow q_j} q_j$$

where $p_i = \bigwedge_{t=1}^{T} p_{it}$, $q_j = \bigwedge_{t=1}^{T} q_{jt}$, and n, T, p_{it}, and $\bar{\bigtriangledown}_t$ denote the number of basic symbolic representations, the set of basic symbolic representations, a symbolic representation of type t, and the type-specific collapsing operator for type t, respectively.

The collapsing operators satisfy the following constraint: $p \bar{\bigtriangledown} q \Rightarrow p \wedge q$. Intuitively, $\bar{\bigtriangledown}$ operator finds the decreasing parts of the fixpoint iterations and removes them to accelerate the fixpoint computation. The greatest fixpoint computations are modified so that at each iteration the result p_i is set to $p_{i-1} \bar{\bigtriangledown} p_i$. For the boolean representation the collapsing operator ($\bar{\bigtriangledown}_{bool}$) is simply the conjunction operator.

Note that the collapsing operator is different than the narrowing operator [CC77]. The narrowing operator is used to improve an over approximation of a least fixpoint in finite number of steps. The collapsing operator, on the other hand, is used to compute a lower approximation of a greatest fixpoint.

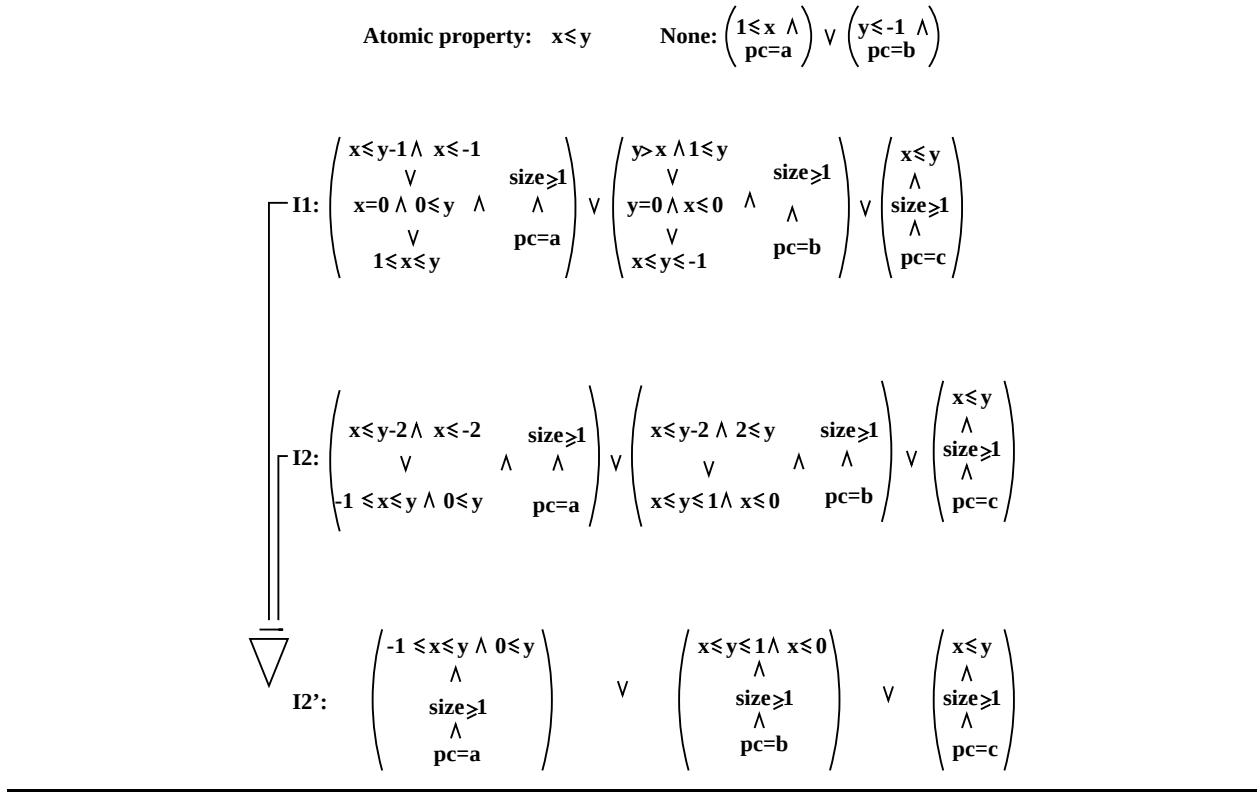


Fig. 12. The results of the first three iterations of computing $EG(x \leq y)$ for the Action Language specification given in Figure 9. $None$ denotes the states with no successors. I_1 and I_2 denote the results of the first and the second iterations, respectively. I_2' denotes the result of the iteration after the collapsing operation ($\bar{\bigtriangledown}$).

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 for linear arithmetic constraints ($\bar{\bigtriangledown}_{int}$) looks for a polyhedron in p that subsumes and is not equal to 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.

For the automata representation we use the widening operator in computing the collapsing operator as follows: $p\bar{\bigtriangledown}_{int}q = \neg(\neg p \bigtriangledown_{int} \neg q)$. Note that this approach is inefficient for polyhedra representation since computing the negation is inefficient in polyhedra encoding.

Figure 12 shows an example of the collapsing operator, which is used for the computation of $EG(x \leq y)$ fixpoint for the specification in Figure 9. In this example ALV was directed to start applying the collapsing operation after the first iteration. For computing $I_1\bar{\bigtriangledown}I_2$, ALV compares each pair of disjuncts where one of them comes from I_1 and the other comes from I_2. For pairs that satisfy the subsumption relation it applies the collapsing operation. For instance, the disjunct $(x \leq y-2 \wedge x \leq -2 \ \vee \ -1 \leq x \leq y \wedge 0 \leq y) \wedge size \geq 1 \wedge pc = a$ that comes from I_2 is subsumed by the disjunct $(x \leq y-1 \wedge x \leq -1 \ \vee \ x = 0 \wedge y \geq 0 \ \vee \ 1 \leq x \leq y) \wedge size \geq 1 \wedge pc = a$

that comes from I_1. Applying integer based collapsing operation on the integer parts of these two disjuncts, i.e., $(x \leq y-1 \wedge x \leq -1 \ \vee \ x = 0 \wedge y \geq 0 \ \vee \ 1 \leq x \leq y \ \vee \ 1 \leq x \leq y) \wedge \mathit{size} \geq 1 \bar{\triangledown}_{int} (x \leq y-2 \wedge x \leq -2 \ \vee \ -1 \leq x \leq y \wedge 0 \leq y) \wedge \mathit{size} \geq 1$, yields $-1 \leq x \leq y \wedge y \geq 0 \wedge \mathit{size} \geq 1$. Note that the disjunct $x \leq y-2 \wedge x \leq -2$ is subsumed by and is not equal to $x \leq y-1 \wedge x \leq -1$, so it does not appear in the result.

Self-Loop-Closures: Another heuristic we use to accelerate convergence is to compute the closures of self-loops in the specifications. Given a transition system (I, S, R) we can use any relation R' that satisfies the constraint

$$\forall s \Rightarrow S, \textsc{post}(s, R) \Rightarrow \textsc{post}(s, R') \Rightarrow \textsc{post}(s, R^*)$$

(where R^* denotes the reflexive-transitive closure of R) to accelerate the fixpoint computations for temporal operators EF and EU [BGP99].

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

$$R \equiv R \ \vee \ \bigvee_{i=1}^{n} (r_{i,int} \ \wedge \bigwedge_{t \in T, t \neq int} IR_t)$$

where IR_t is the identity relation for the variables represented with the basic symbolic representation type t, and the subscript int denotes the symbolic representation for integers. Note that, $\bigwedge_{t \in T, t \neq int}^{T} IR_t$ corresponds to identity relation for all the variables other than integers. Hence, $\bigvee_{i=1}^{n} (r_{i,int} \ \wedge \ \bigwedge_{t \in T, t \neq int} IR_t)$ denotes the part of the transition relation where all the variables other than the integer variables stay the same. To compute $r_{i,int}$'s we conjunct the transition relation R with $\bigwedge_{t \in T, t \neq int} IR_t$ and collect the resulting disjuncts that are satisfiable. Then, for each $r_{i,int}$ we compute an $r'_{i,int}$, where $\forall s \Rightarrow S, \textsc{post}(s, r_{i,int}) \Rightarrow \textsc{post}(s, r'_{i,int}) \Rightarrow \textsc{post}(s, r^*_{i,int})$i [BGP99]. We take the disjunction of the result with the original transition relation R to compute

$$R' = R \ \vee \ \bigvee_{i=1}^{n} (r'_{i,int} \ \wedge \bigwedge_{t \in T, t \neq int}^{T} IR_t)$$

Then, we use R' in the fixpoint computations for EF and EU instead of R to accelerate the fixpoint computations. Note that we cannot use closure computations for EG or AU fixpoints since they may introduce cycles that do not exist in the original transition system. Table II shows the transitions that correspond to the actions in Figure 9 for both cases with and without using the self-loop-closures heuristic. Note that when the self-loop-closures heuristic is used only the transitions that correspond to the actions `a1` and `a3` change, since they are the ones that correspond to self-loops, they model iterative increment and decrement operations, respectively.

Reachable States: The fixpoint algorithms described thus far are *backward* reachability techniques. They start with a property ϕ, and then use PRE to determine which states can reach ϕ. The last step is to determine whether there are initial states I that are included in the resulting set of states. 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. For example we can alter the symbolic model checker to restrict its computations to the states in RS^+. To generate the upper bound RS^+, we can use the POST function. The (exact) reachable state-space of a transition system is the least fixpoint $RS \equiv \mu x \ . \ I \ \vee \ \textsc{post}(x, R)$, and it can be computed using

Action	Transition Relation	Transition Relation with Loop-Closures
a1	$pc = a \wedge x < 0 \wedge$ $x' = x + 1 \wedge pc' = a$	$(pc = a \wedge x < 0 \wedge x' = x + 1 \wedge pc' = a) \vee$ $(pc = a \wedge x + 1 \leq x' \leq 0 \wedge pc' = a)$
a2	$pc = a \wedge x = 0 \wedge pc' = b$	$pc = a \wedge x = 0 \wedge pc' = b$
a3	$pc = b \wedge y > 0 \wedge$ $y' = y - 1 \wedge pc' = b$	$(pc = b \wedge y > 0 \wedge y' = y - 1 \wedge pc' = b) \vee$ $(pc = b \wedge 0 \leq y' < y \wedge pc' = b)$
a4	$pc = b \wedge y = 0 \wedge pc' = c$	$pc = b \wedge y = 0 \wedge pc' = c$
a5	$pc = c \wedge x' = -size \wedge$ $y' = -size \wedge pc' = a$	$pc = c \wedge x' = -size \wedge y' = -size \wedge pc' = a$

TABLE II

THE TRANSITIONS THAT CORRESPOND TO THE ACTIONS OF THE ACTION LANGUAGE SPECIFICATION IN FIGURE 9. $-$ AND $+$ DENOTE EXCLUSION AND INCLUSION, RESPECTIVELY.

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^+.

Marking Heuristic: The states that satisfy EFϕ are characterized by the least fixpoint $\mu Z.\phi \vee \text{PRE}(Z)$. The states that satisfy EFϕ can be computed iteratively such that the result of the kth iteration denotes the states that can reach a state that satisfies ϕ in at most k transitions. Since composite symbolic representation is a disjunctive representation, and, since in the least fixpoint computations the result of the kth iteration includes the disjuncts from the previous iteration ($k-1$st iteration), a naive approach that computes the pre-condition on the result of the $k-1$st iteration to obtain the result of the kth iteration would perform redundant computations by recomputing the pre-condition for the disjuncts coming from the result of the $k-2$nd iteration. We can alleviate this problem by marking the disjuncts from the result of the $k-1$st iteration after computing the result of the kth iteration. Hence, at the kth iteration the pre-condition is computed only on the disjuncts that are not marked, i.e., the disjuncts that were computed at the $k-1$st iteration. Table III shows the results of the first 4 iterations for computing EFϕ for both with and without the marking heuristic. At the kth iteration the fixpoint algorithm without the marking heuristic computes $k-1$ more pre-condition computation than that computed by the the fixpoint algorithm with the marking heuristic. Another benefit of marking heuristic is to reduce the number of the widening operations performed when ALV runs in the approximate fixpoint computation mode. The Marking heuristic can also be used during the least fixpoint computations for the properties of type pEUq and the reachable states RS, since these fixpoints are also characterized as: $\mu Z.\phi \vee F(Z)$.

Figure 13 shows the results of the first three iterations for computing $EF(x > y)$ using the Marking heuristic for the specification given in Figure 9. The disjuncts that are enclosed by rounded-corner boxes denote the marked ones. The Marking heuristic makes sure that the pre-condition computations on the disjuncts $x > y$, $y \leq x \leq -1 \wedge size \geq 1 \wedge pc = a$, and $y - 1 \leq x \leq 2 \wedge size \geq 1 \wedge pc = a$ are performed only in the first, the second, and

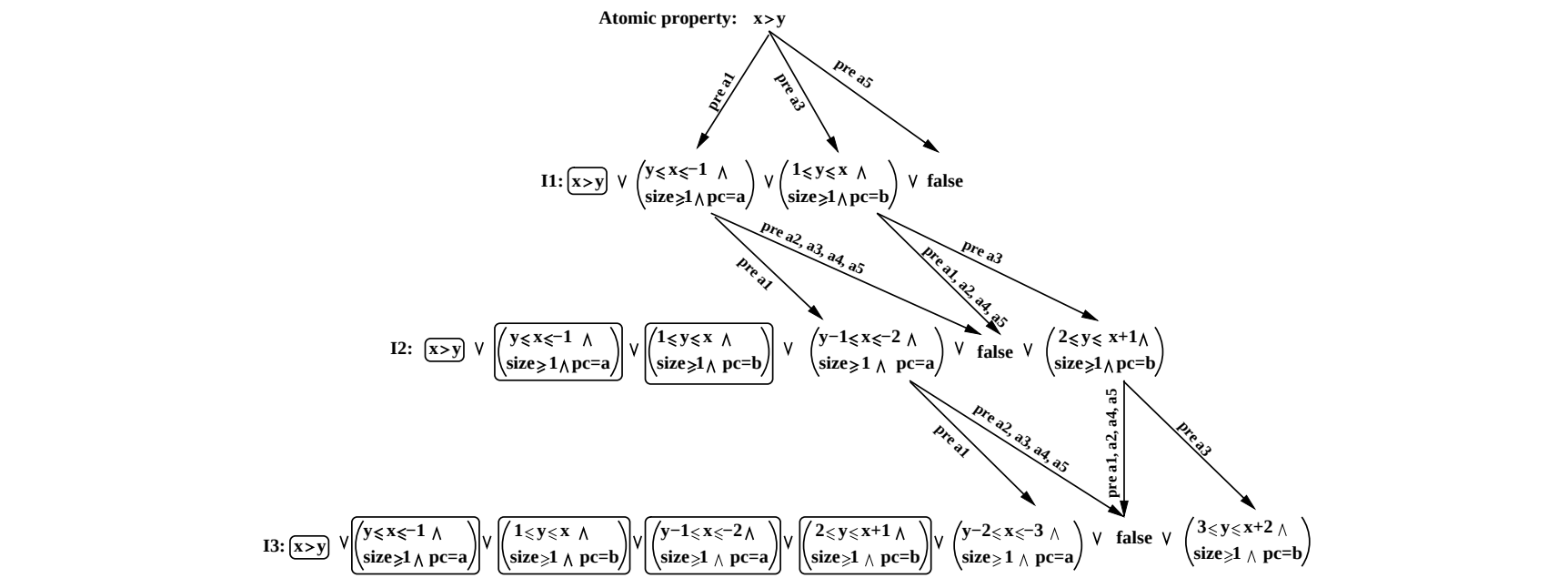


Fig. 13. The result of the first three iterations of computing $EF(x > y)$ using the Marking heuristic for the Action Language specification given in Figure 9. I_1, I_2, and I_3 denote the results of the first, the second, and the third iterations, respectively. $pre\ a_i$ denotes that the constraint pointed by the arrow is obtained by performing the pre-condition computation on the source constraint using action a_i, where $1 \leq i \leq 5$. The constraints that are marked by the Marking heuristic are enclosed in rounded-corner boxes.

Iter.	Fixpoint Iterations	New computed	Fixpoint Iterations with Marking computed	marked
0	ϕ	none	none	none
1	$\phi \vee \text{PRE}(\phi)$	$\text{PRE}(\phi)$	$\text{PRE}(\phi)$	ϕ
2	$\phi \vee \text{PRE}(\phi)$ $\vee\text{PRE}(\text{PRE}(\phi))$	$\text{PRE}(\phi)$, $\text{PRE}(\text{PRE}(\phi))$	$\text{PRE}(\text{PRE}(\phi))$	ϕ, $\text{PRE}(\phi)$
3	$\phi \vee \text{PRE}(\phi)$ $\vee\text{PRE}(\text{PRE}(\phi))\vee$ $\text{PRE}(\text{PRE}(\text{PRE}(\phi)))$	$\text{PRE}(\phi)$, $\text{PRE}(\text{PRE}(\phi))$, $\text{PRE}(\text{PRE}(\text{PRE}(\phi)))$	$\text{PRE}(\text{PRE}(\text{PRE}(\phi)))$	ϕ, $\text{PRE}(\phi)$, $\text{PRE}(\text{PRE}(\phi))$

TABLE III

THE RESULTS OF THE FIRST 4 ITERATIONS OF COMPUTING $\text{EF} = \mu Z.\phi \vee \text{PRE}(Z)$ WITH AND WITHOUT THE MARKING HEURISTIC. $-$ AND $+$ DENOTE EXCLUSION AND INCLUSION, RESPECTIVELY.

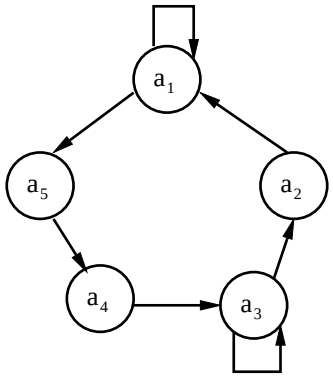


Fig. 14. The dependency graph for the Action Language specification in Figure 9. A directed edge between the nodes a_i and a_j means that the constraints generated as a result of the pre-condition computation using the action a_i may enable the action a_j for computing the pre-condition computation.

the third iterations, respectively.

In addition to eliminating the redundant pre and post-condition computations, the Marking heuristic eliminates the redundant simplification operations among the sets that have been computed in the previous iterations. However, we allow merging an unmarked disjunct with a marked disjunct (resulting in an unmarked disjunct) in order to reduce the number of disjuncts in the composite representation. Although this reduces the effectiveness of the marking heuristic, it improves the overall performance by reducing the composite representation size.

Dependency Heuristic: Given a state s and the transition relation $R = \bigvee_{i=1}^{n} r_i$ where each r_i is an atomic transition, the pre-condition (post-condition) is computed by distributing the pre-condition (post-condition) operator over the disjuncts of R. However, for the case of the pre-condition computation there may be an atomic transition r_k such that there are no states from which s can be reached by executing the transition r_k and for the case of the post-condition computation there may be an atomic transition r_j that is not enabled at state s. Computing the pre-

condition or the post-condition for such cases is redundant. Below we show how this kind of redundancies can be eliminated for the pre-condition computations. This approach can also be used for the post-condition computations.

We first compute a directed graph, which we call the *dependency graph*, (N, E) where $N = \{r_1, r_2, ..., r_n\}$ and E denotes the set of edges. $(r_i, r_j) \in E$ if and only if the following holds:

$$\textsc{Pre}(\textsc{Pre}(true, r_i), r_j) \neq \mathbf{false}$$

The dependency graph, in a way, describes all the feasible interleavings of the atomic transitions. Figure 14 shows the dependency graph for the Action Language specification given in Figure 9. During the fixpoint computations that use the pre-condition computation, we associate every state with the *enable backward set*, which denotes the set of transitions that it can enable via the pre-condition computation. For instance, let $s_2 = \textsc{Pre}(s_1, a_1)$ where s_1 and s_2 represent states, a_1 represents the atomic transition that corresponds to the action `a1` given in Figure 9. The *enable backward set* for s_2 is $\{a_1, a_5\}$, which consists of the neighbors of the transition a_1 according to the dependency graph. Before performing the pre-condition computation on s_2 using a transition a_i, one can first check whether a_i is in the backward enable set of s_2. If it is the case, then we perform the computation, otherwise we skip the computation. For instance, the pre-condition computation on s_2 using the transition a_3 can be skipped since a_3 is not an element of the set $\{a_1, a_5\}$.

Figure 15 shows the results of the first three iterations for computing $EF(x > y)$ for the specification given in Figure 9. The disjunct $y \leq x \leq -1 \wedge size \geq 1 \wedge pc = a$ is generated in the first iteration as a result of the pre-condition computation using the a_1. Since the neighbors of node a_1 are a_1 and a_5, the enable backward set for this constraint is set to $\{a_1, a_5\}$. As a result, on $y \leq x \leq -1 \wedge size \geq 1 \wedge pc = a$ the pre-condition is not computed using a_2, a_3, and a_4, by which we avoid computing some of the pre-condition computations that would yield unsatisfiable constraints.

In addition to using the dependency information for avoiding the redundant pre-condition computations, ALV uses this information during the simplification of the results of the fixpoint iterations. Two disjuncts are compared for equivalence during the simplification phase only if their enable backward sets are the same.

If we compare Figures 13 and 15 with Figure 10, we can see that the Marking heuristic prunes the computation tree such that the pre-condition computations that are performed on the constraints already generated are eliminated, whereas the Dependency heuristic prunes the computation tree such that some of the pre-condition computations that would yield unsatisfiable constraints are eliminated. The effectiveness of the Dependency heuristic highly depends on the dependency graph, i.e., the number of edges between the nodes. However, since the Dependency heuristic is sound for both the least fixpoint computations and the greatest fixpoint computations, it can be used for any property verification, whereas the Marking heuristic can only be used for the least fixpoint computations. The Marking and the Dependency heuristics can be combined to achieve a greater degree of reduction as long as ALV is computing a least fixpoint.

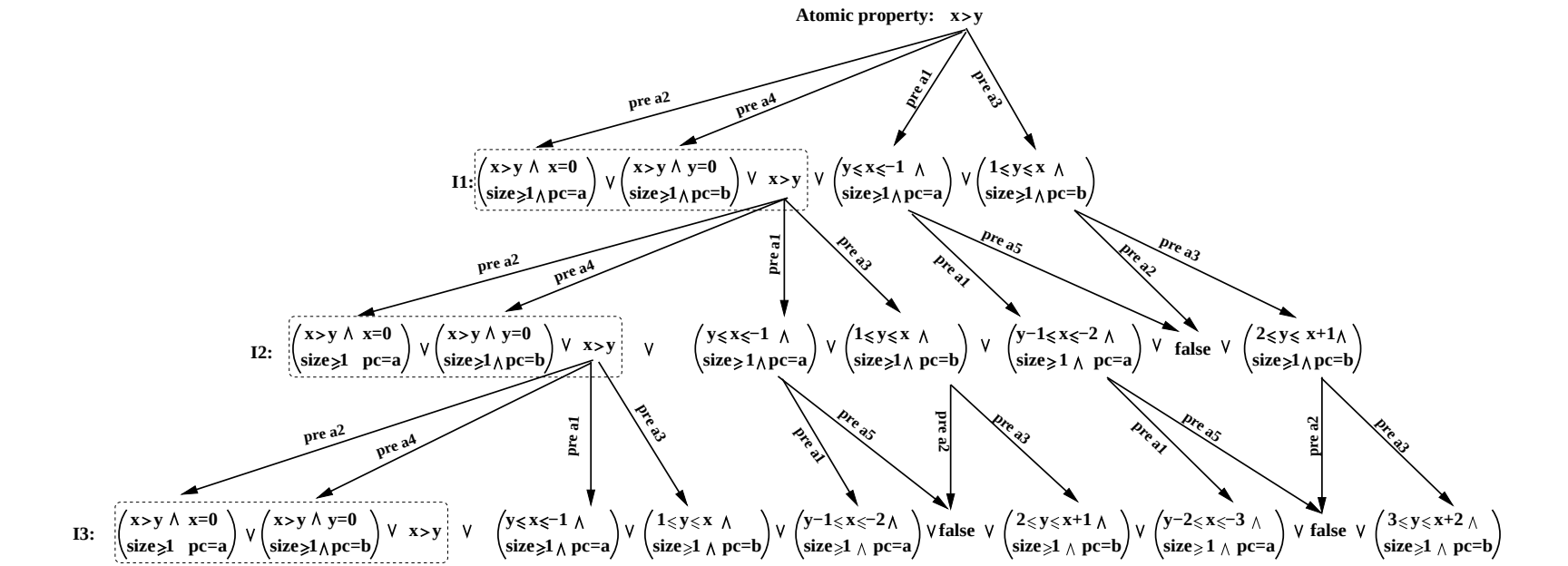


Fig. 15. The results of the first three iterations of computing $EF(x > y)$ using the Dependency heuristic for the Action Language specification given in Figure 9. I_1, I_2, and I_3 denote the results of the first, the second, and the third iterations, respectively. $pre\ a_i$ denotes that the constraint pointed by the arrow is obtained by performing the pre-condition computation on the source constraint using action a_i, where $1 \leq i \leq 5$. The constraints in each of the dashed rounded boxes are simplified into constraint, $x > y$, by the Composite Symbolic Library.

C. Counter-Example Generation

An important feature of model checkers is their ability to generate counter-example behaviors. ALV is able to generate counter-examples for the properties that it falsifies. Generating a counter-example for a property ϕ corresponds to generating a witness for its negation $\neg\phi$. We cannot generate witnesses for universal properties, since we need to list all the paths in the system to demonstrate that the property holds. This is equivalent to saying that we cannot generate a counter-example for an existential property. Hence, ALV generates counter-examples only for ACTL properties. A counter-example for an ACTL property can be represented as a tree-like structure that starts from the initial states and demonstrates that the property is violated [CJLV02].

When asked to generate a counter-example, ALV negates the input property ϕ, converts it to $\{\text{EX}, \text{EG}, \text{EU}\}$ basis, and pushes all the negations inside, i.e., there exists no negation in front of a temporal formula. Then it computes the states that satisfy the sub formulas bottom-up, starting from the atomic properties. However, it also stores the intermediate fixpoint computations for EG and EU when it is looking for a counter-example. After the computation ends it looks for an initial state $s \Rightarrow I \wedge [\![\neg\phi]\!]$. If there is no such state, it reports that no counter-example has been found. (If the computed fixpoints are exact this means that the property is proved.) Otherwise, it starts constructing a witness for $\neg\phi$ (i.e., a counter-example for ϕ) starting from s in a top-down manner, i.e., first it generates the witness that corresponds to the top-most temporal operator and then it continues to generate the witnesses for the sub formulas.

Figure 16 shows a recursive algorithm for computing a witness for a given CTL formula in {EX,EU,EG} basis starting from a given state. The algorithm accumulates the results in a global list called *witness*. Each call to the GENERATE_WITNESS function adds a new entry to this witness list. Each entry in the witness list is a tuple that consists of a state, a formula, and a path, where the path forms the basis of a witness for that formula starting from that state. Note that the path itself may not be enough as a witness, and the later entries in the witness list may contain paths starting from the states in that path. As discussed in [CJLV02], it is possible to construct a tree-like structure by combining the paths reported in the witness list.

The GENERATE_WITNESS algorithm first reverses the results of fixpoint iterations for the input CTL formula. The reason is that fixpoint computations propagate backwards starting from the inner CTL formula, whereas witness computation propagates forwards starting from an initial state.

To generate a witness for the property EXϕ starting from a state s that satisfies EXϕ, the algorithm saves s as the initial state of the witness path. Then it computes $\text{POST}(s, R)$ and conjoins the result with the states that satisfy ϕ. One state chosen from the conjunction is saved as the next state in the witness path. The resulting path is recorded as the witness for the property EXϕ. Then the witness generation algorithm is invoked recursively to generate a witness for ϕ starting from the last state of the witness path for EXϕ.

To generate a witness for the property $\text{EU}(\phi_1, \phi_2)$ starting from a state s that satisfies $\text{EU}(\phi_1, \phi_2)$, the algorithm starts with the result of the last iteration of the fixpoint for $\text{EU}(\phi_1, \phi_2)$. Since that corresponds to the states that satisfy $\text{EU}(\phi_1, \phi_2)$, it is guaranteed that s is in it. If s satisfies ϕ_2 then the algorithm stops. Otherwise, $\text{POST}(s, R)$

```
witness: List of ⟨state, CTL formula, List of states⟩
GENERATE_WITNESS(s, φ)
  s, s1: Symbolic, φ: CtlFormula, path: List of states, iterReverse: List of set of states
  if φ.isAtomic() then return
  iterReverse ← φ.iterates.reverse()
  path.add(s)
  case φ = EXφ1:
    path.add(choose(POST(s, R) ∧ iterReverse.get(1)))
    witness.add(⟨s,φ,path⟩)
    GENERATE_WITNESS(path.getLast(),φ1)
  case φ = φ1EUφ2:
    for 0 < i < iterReverse.size() and not path.getLast().isSubsumed(iterReverse.getLast()) do
      path.add(choose(POST(path.getLast(), R) ∧ iterReverse.get(i)))
    witness.add(⟨s,φ,path⟩)
    for 0 ≤ i < path.size() do
      if i = path.size() − 1 then
        GENERATE_WITNESS(path.get(i),φ2)
      else
        GENERATE_WITNESS(path.get(i),φ1)
  case φ = EGφ1:
    cycleNotReached ← true
    for 1 ≤ i ≤ MAX_ITER and cycleNotReached do
      s1 ← POST(path.getLast(), R)
      if not s1.isSatisfiable() then break
      path.add(choose(s1 ∧ iterReverse.get(i)))
      for 0 ≤ j < i do
        if path.getLast().isSubsumed(path.get(j)) then
          cycleNotReached ← false break
    witness.add(⟨s,φ,path⟩)
    for 0 ≤ i < path.size() do
      GENERATE_WITNESS(path.get(i),φ1)
```

Fig. 16. The algorithm for generating witness for the CTL formula ϕ in {EX,EU,EG} basis starting from state s.

is conjoined with the next one of the reversed results of the fixpoint iterations. Note that, this conjunction cannot be false, since, based on the fixpoint computation for EU, s must have a next state in the result of the next iteration. The algorithm chooses one of the states that satisfy the conjunction as the next state in the witness path. It continues until a state that satisfies ϕ_2 is reached. This state is used to generate a witness for ϕ_2. Note that, the algorithm also generates witnesses for property ϕ_1 starting from all the states that were visited before the state satisfying ϕ_2 was reached. All these witnesses are added to the witness list.

To generate a witness for the property EGϕ, the algorithm only needs the last of the reversed results of the fixpoint iterations, which corresponds to the states that satisfy EGϕ. It starts from a state s that satisfies EGϕ. Then,

POST(s, R) is computed and conjoined with the states that satisfy EGϕ. The algorithm chooses a state that satisfies the conjunction and continues this iteration until a cycle or a state that does not have any next states is found. In either of these cases the algorithm returns the generated path as the witness path. However, since the Action Language specifications can be infinite, it is not guaranteed to find a witness that contains a cycle or a finite path. It is possible that all the witnesses are infinite paths that do not have any repeating states. Hence, the algorithm puts a bound (MAX_ITER) on witness search. When it reaches that bound it adds the path computed so far as the prefix of a witness path to the output witness list. Then, the algorithm generates witnesses for property ϕ starting from all the states in the witness path for EGϕ and all these witnesses are added to the witness list.

The algorithm in Figure 16 does not show how the logical operators *not*, *and*, and *or* are handled. As we have stated before, before generating a witness for a CTL formula, ALV pushes all the negations inside the atomic property. Therefore, the witness generation algorithm does not need to handle the *not* operator. It handles the *and* operator by generating a witness for each subformula, whereas it handles the *or* operator by choosing one of the subformulas that yields a witness.

We have to be careful with counter-example generation when we are using the approximate fixpoint computations. Assume that we are using the $\{\text{EX}, \text{EG}, \text{EU}\}$ basis for the CTL and we try to verify the property AGp. Then we would compute $\neg(EF\neg p)$. If we are computing the approximate fixpoint computations, then this will require us to compute an upper bound for $EF\neg p$ to get a lower bound for AGp. If we can show that $I \Rightarrow [\![\text{AG}p]\!]^-$ then we are done. However, if $I \not\Rightarrow [\![\text{AG}]\!]^-$ we cannot use our computations for $[\![EF\neg p]\!]^+$ to generate a counter-example. Since $[\![\text{EF}\neg p]\!]^+$ is an upper bound it can include spurious counter-examples. If we want to generate a counter-example, then we need to compute a lower bound for EF$\neg p$ (negation of the original property). If we can generate a counter-example using $[\![\text{EF}\neg p]\!]^-$ we are sure that it is a valid counter-example. Because of this issue ALV works in two phases; 1) the verification phase and 2) the falsification phase. During the verification phase ALV does not try to generate a counter-example. If ALV is unable to prove the property during the verification phase, it recomputes the fixpoints and then tries to generate a counter-example. As explained above, these phases will use different types of approximations if approximate fixpoints are being used. Either of these cases can be skipped by the user using the input flags of ALV.

V. Parameterized Verification

In this section we present the adaptation of an automated abstraction technique called *counting abstraction* [Del00] to the parameterized verification of specifications in the Action Language. Using counting abstraction we can automatically verify the properties of a system with arbitrary number of finite-state processes. The basic idea is to define an abstract transition system in which the local states of the processes are abstracted away but the number of processes in each local state is counted by introducing an auxiliary integer variable for each local state. As we will show below, counting abstraction preserves the CTL properties that do not involve the local states of the processes. For this abstraction technique to work we need the local states of the submodules to be finite. Each local state corresponds to a valuation of all the local variables of a submodule, i.e., the set of local states of a submodule

	`Departing`	`Departing*`
S	pc = parked $\vee$ pc = depFlow $\vee$ pc = takeOff	parkedC ≥ 0 $\wedge$ depFlowC ≥ 0 $\wedge$ takeOffC ≥ 0 $\wedge$ parkedC+ depFlowC + takeOffC $= C$
I	pc = parked	parkedC = C $\wedge$ depFlowC $= 0$ $\wedge$ takeOffC $= 0$
R		
r_1	pc = parked $\wedge$ numRW16L $= 0$ $\wedge$ numC3 + numC4 + numC5+ numC6 + numC7 + numC8 $= 0$ $\wedge$ pc$'$ = takeOff $\wedge$ numRW16L$'$ = numRW16L + 1	parkedC > 0 $\wedge$ numRW16L $= 0$ $\wedge$ numC3 + numC4 + numC5+ numC6 + numC7 + numC8 $= 0$ $\wedge$ takeOffC$'$ = takeOffC + 1 $\wedge$ numRW16L$'$ = numRW16L + 1 $\wedge$ parkedC$'$ = parkedC $-$ 1 $\wedge$ depFlowC$'$ = depFlowC
r_2	pc = takeOff $\wedge$ pc$'$ = depFlow $\wedge$ numRW16L$'$ = numRW16L $-$ 1	takeOffC > 0 $\wedge$ depFlowC$'$ = depFlowC + 1 $\wedge$ numRW16L$'$ = numRW16L $-$ 1 $\wedge$ takeOffC$'$ = takeOffC $-$ 1 $\wedge$ parkedC$'$ = parkedC

TABLE IV

TRANSITION SYSTEM INFORMATION FOR AN INSTANTIATION OF `Departing` MODULE IN FIGURE 2 AND ARBITRARY NUMBER OF INSTANTIATIONS OF `Departing` MODULE USING COUNTING ABSTRACTION. S DENOTES THE STATE SPACE, I DENOTES THE INITIAL STATES, AND R DENOTES THE TRANSITION RELATION. r_1 AND r_2 DENOTE THE ATOMIC TRANSITIONS THAT CORRESPOND TO REQTAKEOFF AND LEAVE, RESPECTIVELY. PARKEDC, DEPFLOWC, AND TAKEOFFC DENOTE THE NUMBER OF AIRPLANES IN PARKED, DEPFLOW, AND TAKEOFF MODES, RESPECTIVELY. C IS A PARAMETERIZED CONSTANT THAT DENOTES THE NUMBER OF DEPARTING AIRPLANES.

is the Cartesian product of the domains of the local variables of that submodule. For example, if a submodule has a local variable that is an unbounded integer, we cannot directly use the counting abstraction.

In the Action Language a module instantiation can be parameterized by appending the '*' character to the module instantiation, e.g., `main: Arriving() | Departing()*` indicates that the transition system is an asynchronous composition of an instantiation of the module `Arriving` and an arbitrary number of instantiations of the module `Departing`. Table IV shows the components of the transition system with one departing airplane (`Departing`) versus the transition system with arbitrary number of departing airplanes (`Departing*`) using counting abstraction. The only local variable of the module `Departing` is `pc`, which is an enumerated variable and can take one of the values `parked`, `depFlow`, and `takeOff`. Therefore, the local state space of the module `Departing` consists of `pc` taking one of these values. For the parameterized system, we need to introduce

three counters, `parkedC`, `depFlowC`, and `takeOffC`, which denote the number of the departing airplanes in `parked` mode, the number of the departing airplanes in `depFlow` mode, and the number of the departing airplanes in `takeOff` mode, respectively. We introduce an additional parameterized integer constant, `C`, which denotes the number of the departing airplanes. The state space for the parameterized system consists of non-negative values for `parkedC`, `depFlowC`, and `takeOffC` where their sum is restricted to be equal to `C`. In the initial state of the transition system for a single departing airplane the airplane is in `parked` mode. For the parameterized system, in the initial state `parkedC` is equal to `C` and `depFlowC` and `takeOffC` are equal to zero to denote the fact that all the departing airplanes are initially in the `parked` mode. A departing airplane can perform any of the two atomic actions: `reqTakeOff` or `leave`. For the transition system for a single departing airplane, `reqTakeOff` represents a departing airplane's transition from `parked` mode to `takeOff` mode provided that the runway 16L (`numRWL=0`) is not occupied and there are no airplanes on the taxiways C3-C8 (`numC3+numC4+...+numC8=0`). For the transition system for arbitrary number of departing airplanes, `reqTakeOff` represents transition of *any departing airplane that is in parked mode* (`parkedC>0`) to `takeOff` mode (`parkedC'=parkedC-1`, `takeOffC'=takeOffC+1`). Note that the execution of `reqTakeOff` does not change the status of the departing airplanes in `depFlow` mode, which is taken care of by keeping the value of `depFlow` same in the next state (`depFlowC'=depFlowC`).

Section V-A presents the formal definition of counting abstraction and Section V-B explains application of counting abstraction to a module in an Action Language specification.

A. Formal Definition

Let T be a transition system $T = (I, S, R)$ that is defined over a set of variables V. Let $D(v_i)$ denote the domain of variable v_i in V. Both the initial states I and the state space S are subsets of the Cartesian product of the domains of all variables in V, i.e., $I, S \subseteq \prod_{v_i \in V} D(v_i)$. We partition V into two nonintersecting sets: the set of local variables (V_L) and the set of global variables (V_G). We define the set of local states as $L = \prod_{v_i \in V_L} D(v_i)$. Similarly, the set of global states is defined as $G = \prod_{v_i \in V_G} D(v_i)$. We can specify T by distinguishing the local and global states as follows $T = T_{L,G} = (I_{L,G}, S_{L,G}, R_{L,G})$, where

$$I_{L,G} = \{s | s = \langle s_L, s_G \rangle \ \wedge \ s_L \in L \ \wedge \ s_G \in G \ \wedge \ s \in I\}, \tag{7}$$

$$S_{L,G} = \{s | s = \langle s_L, s_G \rangle \ \wedge \ s_L \in L \ \wedge \ s_G \in G \ \wedge \ s \in S\}, \tag{8}$$

$$R_{L,G} = \{r | r = (\langle s_L, s_G \rangle, \langle s'_L, s'_G \rangle) \ \wedge \ s_L, s'_L \in L \ \wedge \ s_G, s'_G \in G \ \wedge \ r \in R\}. \tag{9}$$

Let T^N denote the asynchronous composition of N *identical* transition systems, $T_i = (I_i, S_i, R_i)$, $1 \leq i \leq N$, where each T_i is defined over a separate set of local variables, V_{L_i}, with the same cardinality, and the same set of global variables. The set of local states, L_i, of each transition system, T_i, will be of the same cardinality, which we denote by M. The transition systems T_i are identical in terms of their initial states, state spaces and transition relations, i.e., $\forall 1 \leq i, j \leq N.\ S_i = S_j \ \wedge \ I_i = I_j \ \wedge \ R_i = R_j$. The initial states, the state space, and the transition

relation of the composed transition system $T^N = (I^N_{L,G}, S^N_{L,G}, R^N_{L,G})$, where $I^N_{L,G} \subseteq S^N_{L,G}$ and $R^N_{L,G} \subseteq S^N_{L,G} \times S^N_{L,G}$ are defined as:

$$I^N_{L,G} = \{s | s = \langle s_1, s_2, \ldots, s_N, s_G\rangle \ \wedge \ \forall 1 \leq i \leq N.\ s_i \in L_i \ \wedge \ \langle s_i, s_G\rangle \in I_i\}, \tag{10}$$

$$S^N_{L,G} = \{s | s = \langle s_1, s_2, \ldots, s_N, s_G\rangle \ \wedge \ \forall 1 \leq i \leq N.\ s_i \in L_i \ \wedge \ \langle s_i, s_G\rangle \in S_i\}, \tag{11}$$

$$\begin{aligned} R^N_{L,G} = \{r | r = (\langle s_1, s_2, \ldots, s_N, s_G\rangle, \langle s'_1, s'_2, \ldots, s'_N, s'_G\rangle) \ \wedge \ \exists 1 \leq i \leq N.\ (\langle s_i, s_G\rangle, \langle s'_i, s'_G\rangle) \in R_i \\ \wedge \forall 1 \leq j \leq N.\ j \neq i \ \Rightarrow \ s_j = s'_j\}. \end{aligned} \tag{12}$$

Equation 10 (11) states that the initial states (state space) of the composed system is the composition of the local initial states (local state spaces) of the identical transition systems and the global initial states (global state space) that are common to all of the individual transition systems. Equation 12 states that when the composed transition system transitions from one state to another state only one of the individual transition systems changes state based on its transition system, which follows from the definition of asynchronous composition.

Let us label the local states $s_L \in L = L_1 = L_2 = \ldots = L_N$ with numbers between 1 and M, where each state has a unique label. We denote this labeling with $l(s_L)$. We introduce one integer variable $v_{l(s_L)}$ for each local state s_L to denote the number of transition systems, T_i, that are currently in the local state s_L. Since the cardinality of L is M, M integer variables are introduced. Given a counter v_i the corresponding local state is s_L where $i = l(s_L)$. We define an abstraction function $\alpha_{CA} : \prod_{1 \leq i \leq N} L_i \rightarrow \prod_{1 \leq i \leq M} Z$ that maps a state, $\langle s_1, s_2, \ldots, s_N\rangle$, that is defined over the local variables of T^N to the valuation of counters $\langle v_1, v_2, \ldots, v_M\rangle$ such that valuation of each v_i represents the number of transition systems that are currently in a state s_L where $i = l(s_L)$:

$$\alpha_{CA}(\langle s_1, s_2, \ldots, s_N\rangle) = \langle v_1 = k_1, v_2 = k_2, \ldots, v_M = k_M\rangle$$

where $k_i = |\{s_j | 1 \leq j \leq N \ \wedge \ i = l(s_j)\}|$.

We transform the composed system T^N into an abstract transition system $T^N_{CA} = (I^N_{CA}, S^N_{CA}, R^N_{CA})$, where

$$\langle s_1, s_2, \ldots, s_N, s_G\rangle \in I^N \ \Leftrightarrow \ \langle \alpha_{CA}(\langle s_1, s_2, \ldots, s_N\rangle), s_G\rangle \in I^N_{CA}, \tag{13}$$

$$\langle s_1, s_2, \ldots, s_N, s_G\rangle \in S^N \ \Leftrightarrow \ \langle \alpha_{CA}(\langle s_1, s_2, \ldots, s_N\rangle), s_G\rangle \in S^N_{CA}, \tag{14}$$

$$\begin{aligned} (\langle s_1, s_2, \ldots, s_N, s_G\rangle, \langle s'_1, s'_2, \ldots, s'_N, s'_G\rangle) \in R^N \ \Rightarrow \ (\langle \alpha_{CA}(\langle s_1, s_2, \ldots, s_N\rangle), s_G\rangle, \\ \langle \alpha_{CA}(\langle s'_1, s'_2, \ldots, s'_N\rangle), s'_G\rangle) \in R^N_{CA}. \end{aligned} \tag{15}$$

Lemma 5.1: T^N and T^N_{CA} are bisimulation equivalent with respect to the set of atomic properties AP_G that are defined over the global variables V_G.

Proof: We define the following bisimulation relation $B \subseteq S^N \times S^N_{CA}$ where, for all $\langle s_1, s_2, \ldots, s_N, s_G\rangle \in S^N$ and $\langle s_{CA}, s_G\rangle \in S^N_{CA}$

$$(\langle s_1, s_2, \ldots, s_N, s_G\rangle, \langle s_{CA}, s_G\rangle) \in B \ \Leftrightarrow \ s_{CA} = \alpha_{CA}(\langle s_1, s_2, \ldots, s_N\rangle) \tag{16}$$

Given any $s = \langle s_1, s_2, \ldots, s_N, s_G \rangle \in S^N$ and $s_A = \langle s_{CA}, s_G \rangle \in S^N_{CA}$, where $(s, s_A) \in B$, following three conditions hold:

1) For any property $\phi \in AP_G$, $s \models \phi \Leftrightarrow s_A \models \phi$, since the global state is the same for s and s_A according to the definition of B in 16.
2) For every state $s' = \langle s'_1, s'_2, \ldots, s'_N, s'_G \rangle \in S^N$ such that $(s, s') \in R^N$, there exists a state $s'_A = \langle s'_{CA}, s'_G \rangle \in S^N_{CA}$, such that $(s_A, s'_A) \in R^N_{CA}$ and $(s', s'_A) \in B$ where $s'_{CA} = \alpha_{CA}(\langle s'_1, s'_2, \ldots, s'_N \rangle)$. This follows directly from the definition 15.
3) For every state $s'_A = \langle s'_{CA}, s'_G \rangle \in S^N_{CA}$ such that $(s_A, s'_A) \in R^N_{CA}$, we want to show that there exists a state $s' = \langle s'_1, s'_2, \ldots, s'_N, s'_G \rangle \in S^N$ such that $(s, s') \in R^N$, and $(s', s'_A) \in B$ where $s'_{CA} = \alpha_{CA}(\langle s'_1, s'_2, \ldots, s'_N \rangle)$. If there exists a transition $(s_A, s'_A) \in R^N_{CA}$ then there must exist two states $s'' = \langle s''_1, s''_2, \ldots, s''_N, s''_G \rangle \in S^N$ and $s''' = \langle s'''_1, s'''_2, \ldots, s'''_N, s'''_G \rangle \in S^N$ where $(s'', s_A) \in B$, $(s''', s'_A) \in B$, $(s'', s''') \in R^N$ and $\alpha_{CA}(\langle s''_1, s''_2, \ldots, s''_N \rangle) = \alpha_{CA}(\langle s_1, s_2, \ldots, s_N \rangle)$ according to 16 and 15. If $s'' = s$, then we can choose $s' = s'''$. If $s'' \neq s$, consider the transition system T_i that executes the transition from s'' to s''' (as defined in 12). Since $\alpha_{CA}(\langle s''_1, s''_2, \ldots, s''_N \rangle) = \alpha_{CA}(\langle s_1, s_2, \ldots, s_N \rangle)$ there must exist a transition system T_j in s that is in the same state that T_i is in in s''. If T_j executes the same transition executed by T_i in s'' in s, then the next state will be a state s' such that $(s', s'_A) \in B$.

Finally, note that, for any initial state $s = \langle s_1, s_2, \ldots, s_N, s_G \rangle \in I^N$, there exists a state $s_A = \langle s_{CA}, s_G \rangle \in S^N_{CA}$ such that $(s, s_A) \in B$ where $s_{CA} = \alpha_{CA}(\langle s_1, s_2, \ldots, s_N \rangle)$. And, for any abstract initial state $s_A = \langle s_{CA}, s_G \rangle \in S^N_{CA}$ there exists a state $s = \langle s_1, s_2, \ldots, s_N, s_G \rangle \in I^N$ such that $(s, s_A) \in B$ where $s_{CA} = \alpha_{CA}(\langle s_1, s_2, \ldots, s_N \rangle)$. This follows directly from the definition 13. Hence, we conclude that T^N and T^N_{CA} are bisimulation equivalent with respect to the set of atomic properties AP_G.

■

Theorem 5.1: For any CTL formula ϕ that is defined over the global atomic properties AP_G, $T^N_{CA} \models \phi \Leftrightarrow T^N \models \phi$.

Proof: The proof follows from the fact that bisimulation relation preserves CTL properties [CGP99] and Lemma 5.1. ■

Finally, we consider the *parameterized verification problem*: Given a CTL formula ϕ over the global atomic properties AP_G, check if $\forall N, N \geq 1, T^N \models \phi$. In order to achieve this, we generate a parameterized-abstract transition system T^P_{CA} that contains a parameterized integer constant, N_P, where $N_P \geq 1$. Given a transition system $T^N = (I^N_{L,G}, S^N_{L,G}, R^N_{L,G})$ and its abstraction $T^N_{CA} = (I^N_{CA}, S^N_{CA}, R^N_{CA})$ as defined above, the parameterized-abstract transition system $T^P_{CA} = (I^P_{CA}, S^P_{CA}, R^P_{CA})$ has the following property:

$$I^P_{CA} \wedge (N_P = N) \equiv I^N_{CA} \qquad S^P_{CA} \wedge (N_P = N) \equiv S^N_{CA} \qquad R^P_{CA} \wedge (N_P = N) \equiv R^N_{CA}$$

Then, we have the following result:

Theorem 5.2: Given the transition system T^P_{CA}, and a CTL formula ϕ over the global atomic properties AP_G, $T^P_{CA} \models \phi \Leftrightarrow \forall N, N \geq 1, T^N \models \phi$.

1 COUNTABS(m)
2 m: module name
3 s: composite formula
4 $s \leftarrow \sum_{i=0}^{M} counter_i = paramCons \ \wedge \ \bigwedge_{i=0}^{M} counter_i \geq 0 \ \wedge \ paramCons > 0$
5 $State(m) \leftarrow \exists \ Locals(m), State(m) \ \wedge \ s$
6 $Init(m) \leftarrow$ COUNTABS_STATES($Init(m)$) $\wedge \ s$
7 **for each** action a of module m **do**
8 $\quad Act(a) \leftarrow$ COUNTABS_TRANSITIONS($Act(a)$) $\wedge \ s \ \wedge \ s'$
9 $Locals(m) \leftarrow \emptyset$

Fig. 17. The algorithm for applying counting abstraction to a module m of an Action Language specification. s' represents the constraint that s represents using the next state version of the variables, e.g., $counter'_i$ instead of $counter_i$.

Note that, if using ALV we can show that T^P_{CA} satisfies the property ϕ then this means that the property holds *for any* T^N, i.e., $T^P_{CA} \models \phi \ \Rightarrow \ \forall N, N \geq 1, T^N \models \phi$. However, if ALV finds a counter-example for the property ϕ for the transition system T^P_{CA} demonstrating that the property does not hold, then this means that the property does not hold *for some* T^N, i.e., $T^P_{CA} \not\models \phi \ \Rightarrow \ \exists N, N \geq 1, T^N \not\models \phi$.

B. Implementation

ALV automatically generates the parameterized-abstract transition system T^P_{CA} whenever the input Action Language specification contains a transition formula of the form `m()*`, where m is a module name. The algorithm for parameterization of a module of an Action Language specification using counting abstraction is given in Figure 17. The algorithm accepts a module name, m, as the input and generates M number of auxiliary integer variables, where M denotes the size of the local state space of m and one parameterized constant that denotes the number of processes. Each of the M auxiliary variables denotes the number of processes in a particular local state. We call these auxiliary integer variables the *counters*. We distinguish two counters that correspond to two different local states by assigning an integer value (i.e., an index) to each local state and using this value as the subscript. For instance, let local state s be represented by the integer value i, then $counter_i$ is the counter that corresponds to s. The algorithm in Figure 17 changes the state formula, initial formula, and the actions of the input module m by replacing the constraints on the local variables with constraints that use the counters and the parameterized constant. It changes the state formula so that the sum of the counters is equal to the parameterized constant, each counter is a nonnegative value, and the parameterized constant is a positive value (line 4). It changes the initial formula by calling the algorithm COUNTABS_STATES and conjoining it with the constraint on the counters (represented by s) (line 6) given in Figure 18, and it changes the transition formula of each action by calling the algorithm COUNTABS_TRANSITIONS (lines 7-8) given in Figure 19 and conjoining it with the constraints on the current (s) and next state counter variables (s'). Conjoining s (lines 5, 6, and 8) and s' (line 8) with the partial constraints generated for counting abstraction makes sure that the second part of the implication in 14, 13, and 15 are realized

1 COUNTABS_STATES(m,f) : composite formula
2 m: module name
3 f, l, s, $resultDis$, $result$: composite formula
4 $index$: integer
5 **let** V_s be the list of all the boolean variables other than m's local boolean variables
6 **let** V_l be the list of m's local boolean variables
7 $result \leftarrow \mathbf{false}$
8 **for each** composite atom $d = \bigwedge_{t \in T} d_t$ of f **do**
9 $resultTerm \leftarrow 0$
10 $indexSet \leftarrow \emptyset$
11 **for each** minterm e of d_{bool} **do**
12 $l \leftarrow \exists\ V_s, e$
13 $indexSet \leftarrow indexSet\ \cup\ l$
14 **let** $index$ denote an integer value that uniquely represents l
15 $resultTerm \leftarrow resultTerm\ +\ counter_{index}$
16 $result \leftarrow result\ \vee\ \bigwedge_{t \in T, t \neq bool} d_t \wedge\ resultTerm = paramCons\ \wedge\ \bigwedge_{i \notin indexSet} counter_i = 0\ \wedge\ \exists V_l, e$
17 **return** $result$

Fig. 18. The algorithm for applying counting abstraction to a formula f.

in the implementation.

The algorithm COUNTABS_STATES accepts a module name m and a composite formula f as the input. It enumerates all the minterms e of the boolean part of each composite atom[1]. By existentially quantifying out the non-local variables, the algorithm extracts a local state l and makes sure that the counter ($counter_l$) that corresponds to l is included in the summation term ($resultTerm$) (line 15), which sums up all the counters that correspond to local states in which a transition system can be. Later (line 16) $resultTerm$ is equated to the parameterized constant $paramConst$ to make sure that all the transition systems are in one of the local states that are extracted from the minterms. Also, the remaining counters are set to zero indicating that none of the transition systems can be in those states (line 16). It conjoins this constraint with the non-local part of the minterm (line 16) and the non-boolean parts of the composite atom. It performs this for all composite atoms (lines 8-16) and computes the disjunction of the resulting constraints (line 16).

The algorithm COUNTABS_TRANSITIONS accepts a module name m and an action name a as the input. It enumerates the minterms of the boolean part of $Act(a)$, which denotes the transition formula that corresponds to the action a. Similar to the algorithm COUNTABS_STATES it existentially quantifies out the non-local variables from the boolean part to obtain an atomic local transition formula on the boolean variables. It obtains the local state l_d

[1]Note that counting abstraction can only be applied on finite local state spaces and in the Action Language finite local state spaces can be defined by boolean and enumerated variables. Since enumerated variables are converted to boolean variables, in a composite atom, boolean part is the one that encodes the finite state space.

1 COUNTABS_TRANSITIONS(m,a): composite formula
2 m: module name a: action name $index$:integer
3 l, s, $result$: composite formula
4 $result \leftarrow \mathbf{false}$
5 **let** V_s denote the list of boolean variables other than m's local boolean variables
6 **let** V_{snext} denote the list of next state boolean variables other than m's local next state
7 boolean variables
8 **let** V_l denote the list of m's local boolean variables
9 **let** V_{lnext} denote the list of m's local next state boolean variables
10 **let** $Act(a) = \bigwedge_{t \in T} r_t$
11 **for each** minterm e of r_{bool} **do**
12 $l_d \leftarrow \exists\, V_{lnext}, \exists\, V_{snext}, \exists\, V_s, e$
13 $l_r \leftarrow \exists\, V_l, \exists\, V_{snext}, \exists\, V_s, e$
14 $s \leftarrow \exists\, V_{lnext}, \exists\, V_l, e$
15 **let** $index_d$ denote an integer value that uniquely represents l_d
16 **let** $index_r$ denote an integer value that uniquely represents l_r
17 **if** $index_d = index_r$ **then**
18 $resultAbs \leftarrow counter_{index_d} > 0 \ \wedge \ counter'_{index_d} = counter_{index_d}$
19 **else**
20 $resultAbs \leftarrow counter_{index_d} > 0 \ \wedge \ counter'_{index_d} = counter_{index_d} - 1 \ \wedge$
21 $counter'_{index_r} = counter_{index_r} + 1$
22 **for each** $0 \leq i < L \wedge i \neq index_d \wedge i \neq index_r$ **do**
23 $resultAbs \leftarrow resultAbs \wedge counter'_i = counter_i$
24 $result \leftarrow result \ \vee \ resultAbs \ \wedge \ s$
25 **return** $\bigwedge_{t \in T, t \neq bool} r_t \ \wedge \ result$

Fig. 19. The algorithm for applying counting abstraction to an action a of module m.

at which the local transition is enabled by existentially quantifying the next state variables (line 12) and obtains the local state l_r at which one can reach by executing the local transition by existentially quantifying out the current state variables (line 13). Then, depending on whether l_d and l_r denote the same state, it generates a constraint. If l_d and l_r denote the same state then the constraint states that the counter that corresponds to l_d is greater than zero and in the next state the counter that corresponds to l_d keeps its value (line 18). Otherwise, the constraint states that the counter that corresponds to l_d is greater than zero and in the next state the counter that corresponds to l_d is decremented by one and the counter that corresponds to l_r is incremented by one (lines 20-21). In both cases the constraint states that the other counters keep their value in the next state (lines 22-23). It conjoins the constraint with the non-local part of the minterm (line 24). It performs this for all minterms and gets the disjunction of the resulting constraints (lines 11-24). Finally, it conjoins the generated constraint with the non-boolean parts of the action a (line 25).

Problem Instance	Transition Relation Size					
	Composite	Polyhedra	EQ, GEQ	BDD	# int.	# bool.
PA2D	22	22	1388	518	29	4
PA4D	26	26	1642	986	29	8
PA8D	34	34	2150	2258	29	16
PA16D	50	50	3166	6146	29	32
PAPD	20	20	1481	326	33	6

TABLE V

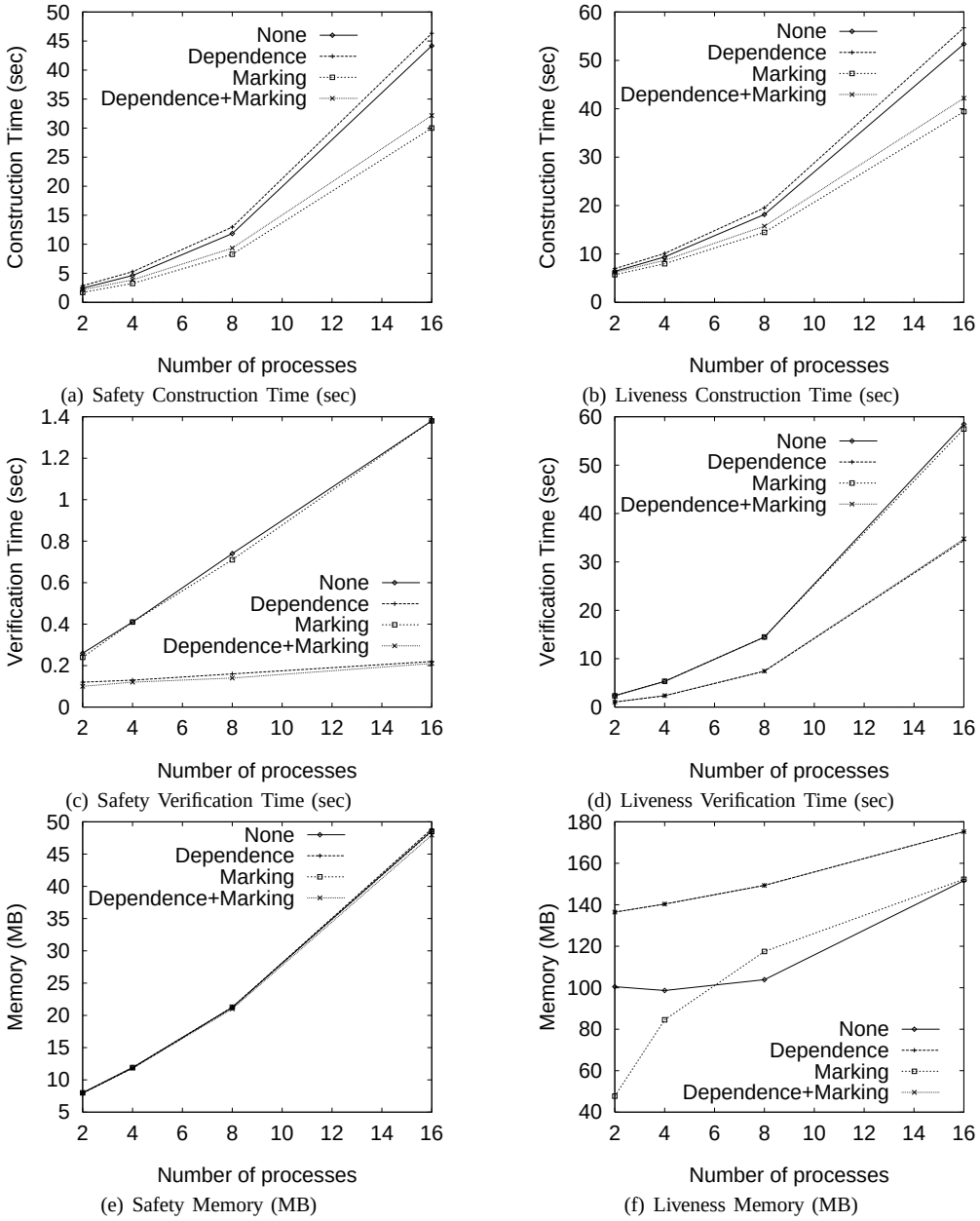
SIZES OF THE TRANSITION RELATIONS FOR THE PROBLEM INSTANCES USED IN THE EXPERIMENTS. $PAXD$ DENOTES ARBITRARY NUMBER (P) OF ARRIVING (A) AIRPLANES AND X NUMBER OF DEPARTING (D) AIRPLANES. $PAPD$ DENOTES ARBITRARY NUMBER OF DEPARTING AND ARRIVING AIRPLANES.

VI. EXPERIMENTS WITH THE AIRPORT GROUND TRAFFIC CONTROL SPECIFICATION

This section presents experimental results that are obtained using the airport ground traffic control specification from Figure 2 discussed earlier. Table V shows the size of the transition systems used in the experiments. We used the polyhedra encoding for the integer variables. For each transition system and the corresponding composite symbolic representation the table shows the number of disjuncts in the composite representation (Composite), the number of polyhedra (Polyhedra), and the number of equality and inequality constraints (EQ, GEQ), and the number of BDD nodes (BDD), the number of integer variables (# int) and the number of Boolean variables (# bool). We have varied the number of the departing airplanes and kept the number of the arriving airplanes arbitrary (parameterized). There is also an instance where both the number of the arriving airplanes and the number of the departing airplanes are both arbitrary (parameterized).

Figure 20 shows performance of the Dependence and the Marking heuristics for the verification of the safety and liveness properties in terms of the construction time, verification time, and memory usage. We used ALV with the widening and the approximate reachable states heuristics for this experiment. Results show that the Marking heuristic performs better than the Dependence heuristic in terms of the construction time, which includes the time spent for the approximate reachable state computation. Note that the approximate reachable state computation involves least fixpoint computation where Marking heuristic is effective. Moreover, the Dependence heuristic incurs a startup cost due to construction of the dependence graphs. On the other hand, the Dependence heuristic performs better than the Marking heuristic in terms of the verification time since it performs significant savings during the simplification phase by avoiding redundant equality checks (see Section IV-B) using the dependence information. The Dependence heuristic uses more memory than that is used by the Marking heuristics since it stores the dependence graph, which is of size n^2 where n is the number of atomic transitions, and the dependence information for each composite atom during the analysis.

The performance of ALV for the verification of the fully parameterized case, where both the number of the



(a) Safety Construction Time (sec)
(b) Liveness Construction Time (sec)
(c) Safety Verification Time (sec)
(d) Liveness Verification Time (sec)
(e) Safety Memory (MB)
(f) Liveness Memory (MB)

Fig. 20. Comparison of the Dependence and the Marking Heuristics

arriving and the number of the departing airplanes are parameterized, is as follows: For the safety property, the transition system construction time is 6.59 seconds, the verification time is 0.02 seconds and the total memory usage is 6.8 MBytes. For the liveness property, the transition system construction time is 28.15 seconds, the verification time is 5.98 seconds and the total memory usage is 103.05 MBytes. Our experimental results indicate that the verification performance for the fully parameterized case is smaller than the partially parameterized case, where the number of the arriving airplanes is arbitrary and the number of the departing airplanes is a constant, when the number of the departing airplanes is greater than or equal to 16. I.e., in this case, the use of the counting abstraction

improves the verification performance.

We have changed the specification in Figure 2 by redefining action `reqTakeOff` as

```
reqTakeOff: pc=parked and numRW16L=0 and pc'=takeOff and numRW16L'=numRW16L+1;
```

By doing so we have introduced an error, since a departing airplane can start taking off even though there may be some airplanes at the exits C3-C8. This violates rule 3 given in Section II. ALV checked this erroneous specification for the CTL property that corresponds to rule 3, and falsified the property by providing a counter-example path. The counter-example path is a witness path for the negated property

```
EF(numRW16L=0 and numC3+numC4+numC5+numC6+numC7+numC8>0 and EX(!numRW16L=0))
```

and consists of two sub-witness paths:

1) The witness path for the property

   ```
   EF(numRW16L=0 and numC3+numC4+numC5+numC6+numC7+numC8>0 and EX(!numRW16L=0))
   ```

   consists of three states. According to the counter-example path, initially, the departing airplane, denoted by `Departing.pc`, is in the `parked` state and the arriving airplane, denoted by `Arriving.pc`, is in the `arFlow` state. Then the arriving airplane lands and transitions to the `touchDown` state. Having landed, the arriving airplane selects exit C3 and starts taxiing on it by transitioning into `taxiTo16LC3` state. During this transition the departing airplane is still in `parked` state.
2) The witness path for property `EX(!(numRW16L=0))` consists of two states. It starts from the state where the departing airplane is still in the `parked` state and the arriving airplane is in the `taxiTo16LC3` state of the sub-witness path. Then the departing airplane starts the takeoff and transitions into the `takeOff` state. This violates the property since exit C3 is occupied while the departing airplane is taking off.

VII. Related Work

The initial structure of the Action Language was presented in [Bul00]. An overview of ALV was presented in [BYK01] and [YKBB05]. The composite symbolic representation used by ALV was discussed in [YKB03]. ALV has been used in verification of various types of specifications including parameterized cache coherence protocols [DB01], parameterized hierarchical state machines [YKB05], workflow specifications [FBHS01], concurrent programs [YKB02], [BCB04], requirements specifications [BH08] and implementation of safety critical software components [BCBL$^+$07].

The main difference between ALV and the well known symbolic model checkers SMV [McM93] and NuSMV [CCG$^+$02] is the fact that ALV targets infinite state specifications. Another distinguishing feature of ALV is the use of the composite symbolic representation. To analyze infinite-state spaces with finite state model checkers such as SMV, NuSMV or SPIN [Hol97]. one needs to first generate an abstraction of the original specification. ALV, on the other hand, uses various automated heuristics for infinite-state verification. Additionally, the counter-example

paths that are generated by ALV is on the concrete system, whereas with the model checkers mentioned above the generated counter-example paths for the abstraction of an infinite-state system would be on the abstract system and needs to be mapped backed to the original infinite-state system in order to understand the source of the error.

There have been earlier work on model checking real-time systems (for example, UPPAAL [BBD+02]) or hybrid systems (for example, Hytech [HHWT97]). UPPAAL focuses on real-time systems and uses the timed automata model which is too restricted as a computation model for the types of specifications that ALV targets. Hytech uses a polyhedra based representation for arithmetic constraints, such as the one used in ALV, for analyzing hybrid systems. However, ALV uses the polyhedra representation on integer domains. Although the Action Language Verifier currently focuses on integers, by integrating the linear arithmetic representation for reals to the Composite Symbolic Library it can be extended to verification of specifications with integer and real variables.

FAST [BFL04] is a verification tool that uses an automata-based representation for unbounded integers and uses fixpoint acceleration techniques for the automata-based representation that are similar to the loop closure heuristic of ALV for the composite representation.

LEVER [VV06] is a verification tool that uses learning algorithms to compute the fixpoints required for verification. LEVER can be used to verify infinite state specifications and has been used on verification of systems with integer variables [VV07]. ALV uses a more traditional iterative approach for computing fixpoints, as pioneered by symbolic model checkers such as SMV. ALV also enables integration of multiple symbolic representations based on the composite symbolic representation.

[SS07] applies bounded model checking techniques to infinite state systems with unbounded integer variables. ALV uses truncated fixpoints and widening operations to under and over approximate least fixpoint computations, respectively. Using these approaches together allows ALV to do sound analysis during verification and look for concrete counter-examples during a separate falsification phase.

[YWGI06] uses the frontier concept to avoid redundant computations during reachable state computation. This technique is similar to the marking heuristic presented in this paper. However, the frontier approach requires computation of the back edges which incurs a startup cost to the fixpoint computation whereas the marking heuristic does not incur such an overhead.

[Del03] uses counting abstraction to verify safety properties of parameterized cache coherence protocols. Our work differs from [Del03] in the following ways: 1) The Action Language Verifier can *automatically* translate an Action Language specification to its parameterized version, whereas [Del03] uses hand translation, 2) The translation algorithm that the Action Language Verifier employs is more general since it can handle any type of predicate that can be specified in the Action Language, whereas in [Del03] it is required that the predicates defining the enabling condition of a transition involve only the number of caches in a local state, 3) The Action Language Verifier can verify CTL properties of parameterized systems using counting abstraction, whereas the technique that is presented in [Del03] is specialized for the verification of safety properties of parameterized cache coherence protocols.

VIII. Conclusions

We presented the formal syntax and the semantics of the Action Language, and we presented ALV, which is an infinite-state symbolic model checking tool for the Action Language. ALV is built on top of the Composite Symbolic Library, which is a symbolic manipulator for systems with heterogeneous data domains. ALV is a polymorphic verification tool. Depending on the type of variables declared in the Action Language specification, ALV may become a BDD-based model checker, a polyhedra-based model checker or a composite model checker where both BDDS and polyhedra-based representation are used for symbolic encoding. Since Composite Symbolic Library can be extended with new symbolic representations, the range of applications that can be analyzed with ALV can also increase in the future.

We discussed the infinite-state verification heuristics implemented in ALV that conservatively approximate the fixpoints and can be used to both verify or falsify Action Language specifications. We also showed that ALV can automatically verify parameterized Action Language specifications.

We have performed a case study on Airport Ground Traffic Control to evaluate the effectiveness of ALV for specifying and verifying a reactive software system. ALV is able to verify many important properties of this case study including the ones specified in [Zho97] for arbitrary number of arriving and departing airplane processes.

References

[BB03] Constantinos Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

[BB04] C. Bartzis and T. Bultan. Widening arithmetic automata. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–333. Springer-Verlag, July 2004.

[BBD+02] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal implementation secrets. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002)*, pages 3–22, 2002.

[BCB04] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 248–257, September 2004.

[BCBL+07] Aysu Betin-Can, Tevfik Bultan, Mikael Lindvall, Benjamin Lux, and Stefan Topp. Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. *Autom. Softw. Eng.*, 14(2):129–178, 2007.

[BFL04] Sébastien Bardin, Alain Finkel, and Jérôme Leroux. Faster acceleration of counter automata in practice. In *TACAS*, pages 576–590, 2004.

[BGL00] T. Bultan, R. Gerber., and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions of Software Engineering and Methodology*, 9(1):3–50, January 2000.

[BGP99] 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.

[BH08] Tevfik Bultan and Connie Heitmeyer. Applying infinite state model checking and other analysis techniques to tabular requirements specifications of safety-critical systems. *Design Automation for Embedded Systems*, 12(1-2):97–137, 2008.

[Bul00] T. Bultan. Action Language: A specification language for model checking reactive systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 335–344, June 2000.

[BYK01] T. Bultan and T. Yavuz-Kahveci. Action Language Verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CGP99] E. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT PRess, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming*, pages 84–97, 1978.

[CJLV02] Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 19–29, 2002.

[CUD] CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`

[DB01] G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, December 2001.

[Del00] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68, 2000.

[Del03] G. Delzanno. Constraint-based verification of parameterized cache-coherence protocols. *Formal Methods in System Design*, 23:257–301, 2003.

[FBHS01] X. Fu, T. Bultan, R. Hull, and J. Su. Verification of Vortex workflows. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, April 2001.

[HHWT97] T. A. Henzinger, P. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[HJJ+95] 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.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Low] Michael Lowry. Software construction and software analysis tools for future space missions. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*.

[McM93] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.

[Ome] The Omega project. `http://www.cs.umd.edu/projects/omega/`

[SS07] Tobias Schüle and Klaus Schneider. Bounded model checking of infinite state systems. *Formal Methods in System Design*, 30(1):51–81, 2007.

[VV06] Abhay Vardhan and Mahesh Viswanathan. Lever: A tool for learning based verification. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, pages 471–474, 2006.

[VV07] Abhay Vardhan and Mahesh Viswanathan. Learning to verify branching time properties. *Formal Methods in System Design*, 31(1):35–61, 2007.

[YKB02] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proc. of International Symposium on Software Testing And Analysis*, 2002.

[YKB03] T. Yavuz-Kahveci and T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):15–33, November 2003.

[YKB05] T. Yavuz-Kahveci and T. Bultan. Verification of parameterized hierarchical state machines using action language verifier. In *Proceedings of the 3rd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2005)*, July 2005.

[YKBB05] Tuba Yavuz-Kahveci, Constantinos Bartzis, and Tevfik Bultan. Action language verifier, extended. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 413–417, 2005.

[YWGI06] Zijiang Yang, Chao Wang, Aarti Gupta, and Franjo Ivancic. Mixed symbolic representations for model checking software programs. In *MEMOCODE*, pages 17–26, 2006.

[Zho97] C. Zhong. *Modeling of Airport Operations Using An Object-Oriented Approach*. PhD thesis, Virginia Polytechnic Institute and State University, 1997.