UNIVERSITY of CALIFORNIA
Santa Barbara

Specification and Automated Verification of Concurrent Software Systems

A Dissertation submitted in partial satisfaction of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Tuba Yavuz-Kahveci

Committee in charge:

Professor Tevfik Bultan, Chair
Professor Richard Kemmerer
Professor Ambuj Singh

June 2004

The dissertation of Tuba Yavuz-Kahveci is approved.

Professor Richard Kemmerer

Professor Ambuj Singh

Professor Tevfik Bultan, Committee Chair

June 2004

Specification and Automated Verification of Concurrent Software Systems

To my parents: Cahide and Mustafa Yavuz

Acknowledgements

First and foremost, I would like to thank my advisor Professor Tevfik Bultan. It is his enthusiasm in doing model checking research that attracted me to the field. Throughout the five years in my Ph.D. study, I have always felt his support and encouragement. I am very grateful to him for teaching me how to be a good researcher and, maybe more importantly, how to be a good writer. I would like to thank Professor Ambuj Singh for being a mentor for me from the first day I came to UCSB and being a member of the committee. I would like to thank the committee member Professor Richard Kemmerer for the useful comments and the feedback. I would like to thank Professor Oscar Ibarra for teaching the most exciting course that I have ever taken in my life. I would like to thank my dear husband Tamer Kahveci for his support, understanding, and encouragement, which made the *seamless integration* of the academic life with the family life possible. I would like to thank my parents, Cahide and Mustafa Yavuz, for making the quality of their children's education their number one priority. I would like to thank my brothers Osman Yuksel and Emre Yavuz for their constant love and support. I would like to thank my little son Omer Kahveci for giving me energy through his love. I would like to thank the graduate students Aysu Betin-Can, Constantinos Bartzis, Xiang Fu, Lingli Zhang, Murat Tuncer, and Christian Lang for their friendship. I have always enjoyed discussing research related issues with them. Last, but not least, I would like to thank the computer science department staff for their help in making things work.

Curriculum Vitæ
Tuba Yavuz-Kahveci

June 1997	Bachelor of Science Department of Computer Science Bilkent University, Ankara, Turkey
August 1999	Master of Science Department of Computer Science Bilkent University, Ankara, Turkey
June 2004	Doctor of Philosophy Department of Computer Science University of California, Santa Barbara

Fields of Study

Software engineering, automated verification, model checking, static analysis.

Publications

- Tuba Yavuz-Kahveci and Tevfik Bultan. A Symbolic Manipulator for Automated Verification of Reactive Systems with Heterogeneous Data Types. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, no. 1, pp. 15-33, November 2003.
- Tuba Yavuz-Kahveci and Tevfik Bultan. Automated Verification of Concurrent Linked Lists with Counters. Proceedings of the 9th International Static Analysis Symposium (SAS 2002). M. V. Hermenegildo, G. Pueble eds., LNCS 2477, pp. 69-84, Springer, Madrid, Spain, September 2002.
- Tuba Yavuz-Kahveci and Tevfik Bultan. Specification, Verification, and Synthesis of Concurrency Control Components. Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), pp. 169-179, Via di Ripette, Rome, Italy, July 22-24, 2002.
- Tuba Yavuz-Kahveci and Tevfik Bultan. Heuristics for Efficient Manipulation of Composite Constraints. Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCoS 2002), Alessandro Ar-

mando, ed., LNAI 2309, pp. 57-71, Springer, Santa Margherita Ligure, Italy, April 8-10, 2002.

- Tuba Yavuz-Kahveci, Murat Tuncer, and Tevfik Bultan. A Library for Composite Symbolic Representations. Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), Tiziana Margaria and Wang Yi, eds., LNCS 2031, pp. 52-66, Springer, Genova, Italy, April 2001.
- Tevfik Bultan and Tuba Yavuz-Kahveci. Action Language Verifier. Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001), pp. 382-386, Coronado Island, California, November 2001.
- Tuba Yavuz-Kahveci, Tamer Kahveci, and Ambuj Singh. Buffering of Multimedia Index Structures. The International Society for Optical Engineering (SPIE) - Internet Multimedia Management Systems, Boston, 2000.

Abstract

Specification and Automated Verification of Concurrent Software Systems

by

Tuba Yavuz-Kahveci

Correctness is the most important quality of software systems. This dissertation focuses on automated verification of concurrent software systems using symbolic model checking. Model checking techniques exhaustively explore the state space of a system in order to determine whether it satisfies the given temporal properties such as safety and liveness. Use of symbolic representations (compact data structures for efficiently encoding the state space) in model checking has enabled the verification of systems with large (even infinite) state spaces.

In order to make symbolic model checking a viable technique for software systems, we have designed several tools and techniques. We have built the Composite Symbolic Library to provide a framework for combining different, type-specific symbolic representations. Our infinite-state symbolic model checker, the Action Language Verifier, uses the Composite Symbolic Library to implement the model checking algorithms. We use the Action Language to specify the behavior of concurrent software systems.

Based on these tools, we have proposed an approach for automatic generation of concurrency controller implementations that are correct by construction. We have applied our approach to a case study in airport ground traffic control.

We have extended our verification technique to verify invariant properties of concurrent linked lists. Our composite framework enables the verification of in-

variant properties that relate the shape of the linked list to the integer variables used in the specification.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Correctness is the most important quality of software systems. There are various techniques for assuring the correctness of software systems such as testing and automated verification. Testing involves analysis of a software system for only a subset of all the possible inputs/events. Although testing is widely used in the software industry, it can only show the existence of errors but not their absence [30].

Automated verification techniques require both software systems and their correctness properties to be specified as mathematical objects in order to formally reason about them. Model checking is an automated verification technique, which exhaustively explores the state space of a system in order to verify that it satisfies a given property. Model checking techniques vary in the way they represent the state space. Explicit-state model checking enumerates the set of all possible states; hence, it can only verify properties of finite state systems. Symbolic model checking, on the other hand, uses symbolic representations to encode the state space, which enables the representation of large (even infinite) state spaces in a compact way. Hence, it is possible to verify systems with very large state spaces using symbolic model checking [19]. Although the automated verification of infinite state systems is undecidable, symbolic model checking can be combined with conservative approximation techniques, which can effectively verify or falsify a large set of properties. The disadvantage of approximate analysis is that sometimes the result of the analysis can be inconclusive. In some cases, it is possible to verify (or falsify) a larger set of properties by increasing the precision of the approximate analysis.

In our research, we have focused on symbolic model checking of infinite state concurrent software systems. Our main objective is to make symbolic model checking a viable technique for the verification of concurrent software systems. We think that verifying software systems at the programming language level is not feasible due to the existence of implementation details, which increase the size of the state space. Instead, we verify software systems at a higher level of abstraction using a specification language, which is called the Action Language. Software components that implement these specifications can be automatically synthesized after the specifications are verified for certain correctness properties.

We have designed and built a tool, which is called the Action Language Verifier, for the automated verification of concurrent software specifications. The architecture of our tool is shown in Figure 1.1. Our tool consists of four main parts: 1) The front-end, 2) The symbolic manipulator, 3) The Action Language translator, and 4) The model checker. Our tool accepts a specification in the Action Language as input. The Action Language is a high-level specification language for reactive systems. The front-end of our tool, which consists of a lexer and a parser, generates an abstract syntax tree from the input Action Language specification. The Action Language translator generates a symbolic transition system from the abstract syntax tree using our symbolic manipulator, which is called the

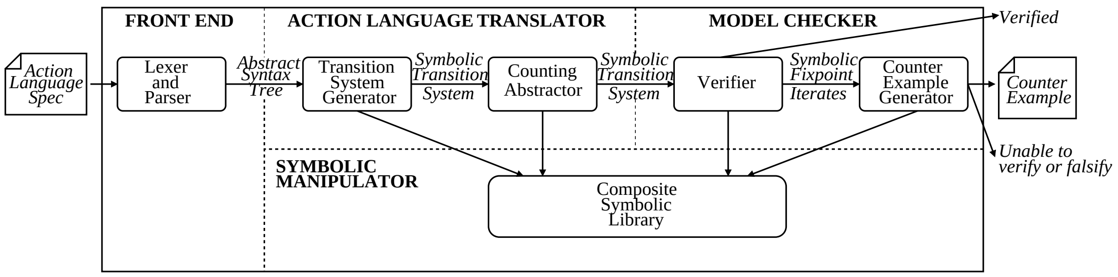

FRONT END
ACTION LANGUAGE TRANSLATOR
MODEL CHECKER
Verified
Action Language Spec
Lexer and Parser
Abstract Syntax Tree
Transition System Generator
Symbolic Transition System
Counting Abstractor
Symbolic Transition System
Verifier
Symbolic Fixpoint Iterates
Counter Example Generator
Counter Example
Unable to verify or falsify
SYMBOLIC MANIPULATOR
Composite Symbolic Library

Figure 1.1: Architecture of the Action Language Verifier Tool

Composite Symbolic Library. The Counting Abstractor, which is a part of the Action Language translator, can automatically generate transition systems that are parameterized in the number of concurrent processes using an abstraction technique called *counting abstraction*. The model checker uses symbolic model checking along with infinite-state verification heuristics to analyze correctness of the symbolic transition system for Computation Tree Logic (CTL) properties. CTL is a formalism that can be used to express safety and liveness properties. For the verification of infinite-state transition systems, the Action Language Verifier may not be able to give a definite answer. When the correctness property is falsified, it can generate a counter-example path, which can be used to find the source of the failure.

Section 1.1 summarizes our contributions along with presentation of the underlying motivation for our work. Section 1.2 explains an outline of the dissertation.

1.1 Summary of Contributions

Specification of a concurrent software system may include multiple data types. Effective symbolic analysis of such a specification would require the ability to manipulate the symbolic representations for several data types compositionally. To serve this purpose we have designed and implemented the Composite Symbolic Library [69, 66, 68], which is a symbolic manipulator for systems with heterogeneous data types. It combines different, type-specific symbolic representations using a disjunctive representation. The implementation of each symbolic representation provides the basic set manipulation and the pre-condition and the post-condition operations, which are sufficient to implement symbolic model checking algorithms. We have designed and implemented several heuristics for efficient manipulation of the composite symbolic representation and shown their effectiveness on a large set of examples. The earlier versions of the Composite Symbolic Library provided the symbolic representations for boolean and integer types. Recently, we have extended the Composite Symbolic Library with a symbolic representation for heap type for the automated verification of concurrent linked lists [65]. The extensible design of the Composite Symbolic Library enabled other researchers to extend it with other symbolic representations and to use it in their research [42, 6].

We have designed and implemented the Action Language Verifier [17], which is an infinite state symbolic model checker. It verifies CTL properties of concurrent systems specified in the Action Language [13]. The Action Language Verifier uses the Composite Symbolic Library to implement the verification algorithms for symbolic model checking. It employs several heuristics to accelerate or to ensure the termination of the analysis. We have also designed and implemented the heuristics for improving the performance of the Action Language Verifier.

The Action Language is the specification language of the Action Language Verifier. The Action Language is a modular language for specifying concurrent software systems. It supports both asynchronous and synchronous semantics of concurrency where the communication is achieved via shared variables. Currently, the Action Language provides boolean, enumerated, and integer data types. We formalized the semantics of the Action Language using denotational semantics approach. We have used the Action Language to specify several concurrent systems including the concurrency controller of an airport ground traffic control simulation software [67].

In [67] we have proposed an approach for developing reliable concurrency controllers. Our approach involves i) the specification of a concurrency controller using the Action Language ii) the verification of the Action Language specification of the concurrency controller using the Action Language Verifier, and iii) the automated generation of an efficient Java implementation for the verified concurrency controller specification using the Composite Symbolic Library.

We have integrated shape analysis to our automated verification framework. Shape analysis involves the exploration of the shape related properties of data structures that reside on the heap, such as linked lists. Since such data structures can grow arbitrarily large due to dynamic memory allocation, compact and abstract representations are needed for encoding their unbounded state spaces. We have proposed an approach for automatically verifying invariant properties of concurrent singly linked lists in [65]. The novelty of our approach lies in its ability to verify properties that relate the integer variables to the heap variables, e.g., we can verify properties such as $head = null \Rightarrow numItems = 0$. Recently, we have extend the proposed approach to linked lists with multiple fields.

1.2 Outline of the Dissertation

Chapter 2 presents the Action Language in terms of its formal syntax and semantics. It also provides a case study on airport ground traffic control. Chapter

3 explains the Composite Symbolic Library in detail. It includes the symbolic manipulation algorithms along with the time complexities, several heuristics for efficient manipulation of the symbolic representations, and experimental evaluation of the composite approach and the heuristics. Chapter 4 presents the Action Language Verifier. It explains the fixpoint computations for non-total transition systems, the heuristics for efficient image computations, the verification of parameterized systems, the counter-example generation, and the experimental evaluation of the heuristics. Chapter 5 explains our approach for automated synthesis of concurrency controllers. It focuses on monitors as a concurrency control mechanism and proposes an approach for the specification and automated synthesis of efficient monitor implementations in Java. Chapter 6 presents our pattern-based approach for the automated verification of concurrent linked list specifications. It explains the pattern-based symbolic representation of heap configurations, presents the manipulation algorithms and the integration of the shape analysis technique to the composite framework, and reports some experimental results.

Chapter 2

The Action Language

Formal specification is the first step of the formal reasoning process. Formal specification languages have been developed in order to ease the task of the users in their mathematical formulation of both the system behavior and the correctness properties. Most specification languages for reactive software systems have been proposed without considering the automated verification. On the other hand, most model checkers have come up with their own input languages for specifying the transition system models instead of adopting the existing specification languages [58, 51]. Hence, to use a model checker, one has to translate the input specification to the input language of the model checker. Most of the time such translations are based on ad hoc techniques and result with unreadable translations. The Action Language [13] has been designed to come up with a specification language which can compactly represent reactive software specifications. The Action Language supports both synchronous and asynchronous compositions as the basic operations. Asynchronous and synchronous composition operators of the Action Language were initially defined in [13]. In this chapter we extend the language by introducing a module hierarchy and the associated scoping rules. We also added parameters to modules, which enables renaming of the variables for each instantiation of a module.

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 are treated as variables that have unknown initial values and do not change value.

This chapter is organized as follows. Section 2.1 presents our case study on airport ground traffic control. Section 2.2 presents the syntax of the Action Language. Section 2.3 presents the semantics of the Action Language. Finally, Section 2.4 discusses the related work.

2.1 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. [70] presents a concurrent simulation program 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. We use the same airport ground network model used in [70] (shown in Figure 2.1) 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 and chapters we will 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.

2.2 Syntax of the Action Language

An Action Language specification consists of a set of module definitions. Figure 2.2 shows the Action Language specification of the control logic for the airport ground traffic control simulation software discussed in Section 2.1. The specification consists of three module definitions: `main`, `Departing`, and `Arriving`. The module `main` (lines 1-36) models the airport ground traffic control system,

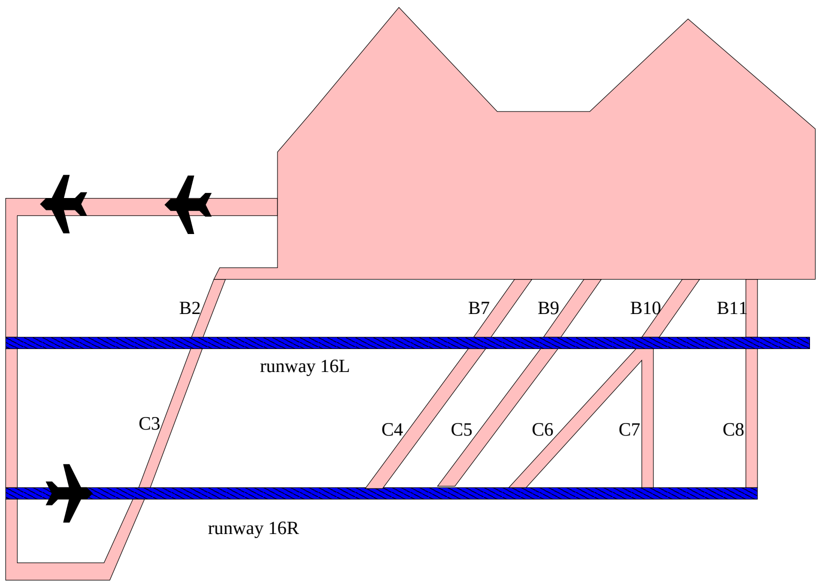


Figure 2.1: An airport ground network similar to that of the Seattle Tacoma International Airport

the module `Departing` (lines 5-13) models a departing airplane, and the module `Arriving` (lines 14-30) models an arriving airplane. The Action Language syntax is given in Figure 2.3. 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 and the local variable declarations. The local variables and the formal parameters of a module are only visible to 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

```
module main()
 integer numRW16R, numRW16L, numC3 ...;
 initial: numRW16R=0 and numRW16L=0 numC3=0 ...;
 restrict: numRW16R>=0 and numRW16L>=0 and numC3>=0...;
 module Departing()
   enumerated pc {parked, depFlow,takeOff};
   initial: pc=parked;
   reqTakeOff: pc=parked and numRW16L=0 and numC3+numC4+numC5+
               numC6+numC7+numC8=0 and pc'=takeOff and
               numRW16L'=numRW16L+1;
   leave: pc=takeOff and pc'=depFlow and numRW16L'=numRW16L-1;
   Departing: reqTakeOff | leave;
 endmodule
 module Arriving()
   enumerated pc {arFlow, touchDown, taxiTo16LC3, taxiTo16LC4,
   taxiTo16LC5, taxiTo16LC6, taxiTo16LC7, taxiTo16LC8,
   taxiFr16LB2, taxiFr16LB7, taxiFr16LB9, taxiFr16LB10,
   taxiFr16LB11};
   initial: pc=arFlow;
   reqLand: pc=arFlow and numRW16R=0 and pc'=touchDown and
            numRW16R'=numRW16R+1;
   exitRW3: pc=touchDown and numC3=0 and numC3'=numC3+1 and
            numRW16R'=numRW16R-1 and pc'=taxiTo16LC3;
   crossRW3: pc=taxiTo16LC3 and numRW16L=0 and numB2A'=numB2A+1
           and pc'=taxiFr16LB2 and numC3'=numC3-1 and numB2A=0;
   park2: pc=taxiFr16LB2 and pc'=parked and numB2A'=numB2A-1;
    . . .
   Arriving: reqLand | exitRW3 | crossRW3 | park2 | ... ;
   spec: invariant(pc=arFlow => eventually(pc=parked))   // P4
 endmodule
 main: Arriving() | Departing() ;
 spec: invariant(numRW16R<=1 and numRW16L<=1)   // P1
 spec: invariant(numC3<=1)   // P2
 spec: invariant((numRW16L=0 and numC3+numC4+numC5+numC6+numC7+
       numC8>0) => next(numRW16L=0))    // P3
endmodule
```

Figure 2.2: The Action Language Specification of the Airport Ground Traffic Control Case Study

Module	::=	`module` *Id* (*IdL*)
		VarDecl Sys ModuleL Action ModTrans Prop
		`endmodule`
ModuleL	::=	*Module ModuleL* | ϵ
VarDecl	::=	`boolean` *IdL* ; | `integer` *IdL* ; | `parameterized integer` *IdL* ;
		| `enumerated` *IdL* { *IdL* } ; | *VarDecl VarDecl* | ϵ
Sys	::=	`initial` : *Form* ; | `restrict` : *Form* ; | *Sys Sys* | ϵ
ModInst	::=	*Id* (*IdL*)
Comp	::=	*Id* | *ModInst* | *Comp* `|` *Comp* | *Comp* `&` *Comp*
Action	::=	*Id* : *Form* ; | *Action Action* | ϵ
ModTrans	::=	*Id* : *Form* ; | *Id* : *Comp* ;
Prop	::=	`spec` : *CtlForm* ; | *Prop Prop* | ϵ

Figure 2.3: Syntax of the Action Language

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 variables* and the latter *next state variables.* In the specification of Figure 2.2, 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 number of airplanes on the runway 16R and on the taxiway C3, respectively. The enumerated variables (`pc` of module `Departing` (line 6) and `pc` of 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 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.

The initial states and the state space of a system are specified in terms of *composite formulas* ($Form$) with the grammar given in Figure 2.4. A composite formula is obtained by combining boolean and integer formulas with logical connectives. A boolean formula ($BoolForm$ in Figure 2.4) consists of boolean variables or constants combined with logical connectives. An integer formula ($IntForm$ in Figure 2.4) 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 ($Integer$ 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. In the specification of Figure 2.2, 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).

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

Figure 2.4: Syntax of a composite formula

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. In the specification of Figure 2.2, the action `reqTakeOff` (lines 8 and 9) 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 (*Comp*) synchronously (`&`) or 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. In the specification of Figure 2.2, 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 consists of composite formulas combined with the temporal operators (EX, AX, EF, AF, EG, AG, EU, AU) and logical connectives. Syntax of a CTL formula (*CtlForm*) is given in Figure 2.4. In the specification of Figure 2.2, 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 specify the rules given in Section 2.1.

2.3 Semantics of the Action Language

The formal semantics of an Action Language specification is defined by a tuple (M, ϕ), where M is a transition system and ϕ denotes the 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. This can be checked using the Action Language Verifier (see Section 4).

Notation We make use of the following notational conventions: Given a function $f : X \to Y$ and $x_1, x_2 \in X$ and $y \in Y$, then the function $f[y/x_1] : X \to Y$ 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 one of the following: an action name, a module name, or a variable name. $[\![IdL]\!]$ denotes a set of strings that correspond to a list of variable names. $[\![Form]\!]$ 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 $\alpha \in Act = [\![Id]\!] \to [\![Form]\!]$ maps each action to the composite formula that corresponds to that action.

- The initial states, the state space, and the transition relation functions $Init, State,$
  $Trans = [\![Id]\!] \to [\![Form]\!]$ 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 $Formals$, $Locals$, $Params = [\![Id]\!] \to [\![IdL]\!]$ 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.

- $\mathit{Rename}_V : [\![\mathit{IdL}]\!] \times N \rightarrow [\![\mathit{IdL}]\!]$ 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.

- $\mathit{NextStateVar} : [\![\mathit{Form}]\!] \rightarrow [\![\mathit{IdL}]\!]$ function takes a composite formula as input and returns the set of variables whose next state versions appear in the composite formula.

- $\mathit{Identity} : [\![\mathit{IdL}]\!] \rightarrow [\![\mathit{Form}]\!]$ 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. Note that if the input is an empty set then it returns true.

- $\mathit{Guard} : [\![\mathit{Form}]\!] \rightarrow [\![\mathit{Form}]\!]$ 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 $\epsilon \in \mathit{SysEnv} = \mathit{Init} \times \mathit{State}$, $\epsilon_I \in \mathit{Init}$ and $\epsilon_S \in \mathit{State}$.

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 (*true* by default), a composite formula describing the state space (*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. $\Xi : VarDecl \rightarrow [\![Id]\!] \rightarrow VarEnv \rightarrow VarEnv$

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

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

   (c) $\Xi[\![\texttt{enumerated}\ IdL_1\ \{IdL_2\}]\!]m\vartheta =$
   $$\begin{cases} \langle \vartheta_L[[\![IdL_1]\!] \cup \vartheta_L(m)/m], \vartheta_F, \vartheta_P \rangle & \text{if } [\![IdL_1]\!] \notin \vartheta_F(m) \\ \langle \vartheta_L, \vartheta_F, \vartheta_P \rangle & \text{otherwise} \end{cases}$$

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

   (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 2.2 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 2.3 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

$$\rho_{IC} = \beta_{IC}[\beta_{IC}(m) + 1/m] \tag{2.1}$$

$$\rho_I = \beta_I[Rename_F(\beta_I(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m))\ \wedge \beta_I(m_1)/m_1] \tag{2.2}$$

$$\rho_S = \beta_S[Rename_F(\beta_S(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m))\ \wedge \beta_S(m_1)/m_1] \tag{2.3}$$

$$\rho_T = Rename_F(\beta_T(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m)) \tag{2.4}$$

$$\rho_C = \beta_C[Rename_F(\beta_C(m), l, \beta_F(m), \rho_{IC}(m), \beta_L(m))\ \wedge \beta_C(m_1)/m_1] \tag{2.5}$$

$$\rho_L = \beta_L[Rename_V(\beta_L(m), \rho_{IC}(m)) \cup \beta_L(m_1)/m_1] \tag{2.6}$$

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 2.1). 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 2.2 and 2.3), the composite formula describing the transition relation of the module (equation 2.4), and the CTL formula describing the property of the module (equation 2.5). 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 2.6).

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

   $$\begin{aligned}\rho'''_T = {} & (\rho'_T \wedge Identity(NextStateVar(\rho''_T) \setminus NextStateVar(\rho'_T))) \;\vee \\ & (\rho''_T \wedge Identity(NextStateVar(\rho'_T) \setminus NextStateVar(\rho''_T)))\end{aligned}$$

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

(d) Let $\rho' = \Lambda[\![Comp_1]\!]m\beta$ and

$\rho'' = \Lambda[\![Comp_2]\!]m\langle \rho'_I, \rho'_S, \rho'_C, \rho'_{IC}, \rho'_L, \beta_F, \beta_P, \beta_A, \beta_T\rangle$.

$\Lambda[\![Comp_1 \texttt{ \& } Comp_2]\!]\beta = \rho'''$ 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 `|`) models interleaving semantics of concurrency (case 4.c). When a transition is executed the values of the variables that are modified only by the other transition are preserved. In asynchronous composition all possible interleavings of the composed transitions are taken into consideration. In synchronous composition (denoted by `&`) two transitions are executed in parallel. However, if one of the transitions is disabled then it does not block the other transition (case 4.d).

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. $\Phi : Prop \to [\![Id]\!] \to CtlProp \to CtlProp$

(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 2.5).

8. $\Upsilon : Module \to Env \to Env$

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

$$\begin{aligned}
m &= [\![Id]\!] \\
l &= [\![IdL]\!]
\end{aligned}$$

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

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

$$\beta_I = \lambda x.true,\ \beta_S = \lambda x.true, \beta_C = \lambda x.undefined,\ \beta_{IC} = \lambda x.0,$$
$$\beta_L = \lambda x.\emptyset,\ \beta_F = \lambda x.\emptyset,\ \beta_P = \lambda x.\emptyset,\ \beta_A = \lambda x.\emptyset,\ \beta_T = \lambda x.\emptyset.$$

Figure 2.5 shows a sample Action Language specification in the context of the case study given in Figure 2.1. There are two submodules of module `main`: `runway` and `environment`. Variable `rw16L` models availability status of the runway 16L and ev16L models events that denotes either an enter request or an exit request. Module `runway` models status change of a runway and its behavior is modeled by asynchronous composition of its actions `r1` and `r2`. Module `environment` models the creation of enter and exit events for a particular runway. Whenever an enter event is created, in the next state an exit event is created (action `e1`). After the

```
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
```

Figure 2.5: A sample Action Language specification modeling status change of runway 16L as the relevant event occurs.

exit event is created, either an enter event (`action e2`) or an exit event (`action e3`) is created nondeterministically. Its behavior 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 instantiation 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 2.1 shows the transition systems that correspond to the modules `runway`, `environment`, and `main`.

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

Table 2.1: The transition systems that correspond to the modules **`runway`**, **`environment`**, and **`main`** of the Action Language specification in Figure 2.5. I, S, and R denote the initial states, the state space, and the transition relation, respectively.

2.4 Related Work

The Action Language has been designed as an input language of an infinite-state symbolic model checker. This design decision has been very influential on the features that it possesses. Unlike the input languages of explicit-state model checkers [51, 38] or the input languages of symbolic model checkers that is based on finite-state symbolic representations [58], it allows unbounded integer variables and parameterized constants. This makes it possible to specify behavior of parameterized systems (see Section 4.3) as well as systems that manipulate unbounded integer domains. Explicit-state model checkers either provide input languages with high-level constructs such as bounded arrays [38] and channels [51] or use an existing high-level programming language and explore the unbounded state space for a certain depth [10]. On the other hand, the underlying model checker of the Action Language explores unbounded state spaces to provide a conservative result, i.e., any property that is verified is actually satisfied by the specified system. This is achieved by employing various abstract interpretation techniques, which requires the existence of precise abstractions for unbounded data domains. Therefore, the Action Language can support a data type provided that the underlying model checker can effectively represent the corresponding data domain. As the Action Language has been designed to specify behavior of concurrent software systems, we envision that it will be extended with new data types and Chapter 6, indeed, serves as an example. The Statechart [47] is a visual formalism for specifying behavior of hierarchical reactive systems and it is included in UML [9], which is a popular modeling language for software systems. The modular nature of the Action Language and the semantics of the asynchronous and synchronous composition operators of the Action Language, enables translating the Statechart specifications to the Action Language such that the hierarchical structure is preserved. The Action Language is similar to Temporal Logic of Actions (TLA) [55]. As in TLA, in the Action Language a system is specified using logical connectives. However, in the Action Language the semantics of the asynchronous and the synchronous composition operations deviates from pure logic to make these operations more readable. Also the Action Language does not use temporal operators to specify the behaviors of systems as in TLA.

Chapter 3

The Composite Symbolic Library

Compact and efficient symbolic representations have enabled the automated verification of large hardware and software systems by overcoming the state-space explosion problem [18, 58, 22]. Symbolic representations are efficient alternatives to explicit state exploration, since they provide a compact representation of the state space. Properties of a system can be verified by manipulating the symbolic representations that represent its transition relation and states. Binary Decision Diagrams (BDDs) [11] (for representing boolean logic formulas) and polyhedral representation [45] (for representing linear arithmetic formulas) are two examples for such symbolic representations. BDDs have been successfully used in the verification of finite-state systems that could not be verified explicitly due to the size of the state space [18, 58, 22]. Linear arithmetic constraint representations have been used in the verification of real-time systems, and infinite-state systems [3, 16, 34, 46], which cannot be verified using the explicit representations.

One problem with these symbolic representations is that they are specialized for certain domains; i.e., BDDs are specialized for encoding boolean variables and the polyhedral representation is specialized for representing the states of integer and real variables as linear arithmetic constraints. As a result, BDDs are restricted to finite domains and the polyhedral representation becomes inefficient when it is used for a large set of boolean variables.

Generally, model checking tools have been built using a single symbolic representation [58, 3]. As model checkers become more widely used, it is not hard to imagine that a user would like to use a model checker built for real-time systems on a system with lots of boolean variables and only a couple of real variables. Similarly another user may want to use a BDD-based model checker to check a system with few boolean variables but lots of integer variables. Currently, such users may need to obtain a new model-checker for these instances, or use various abstraction techniques to solve a problem that may not be suitable for the symbolic representation their model checker is using. More importantly, as symbolic model-checkers are applied to larger problems, they are bound to encounter specifications with different variable types that may not be efficiently representable using a single symbolic representation.

This chapter summarizes our work on the Composite Symbolic Library, which was presented in [69, 66, 68]. The Composite Symbolic Library is a symbolic manipulator for systems with heterogeneous data types. It combines different symbolic representations using the *composite model checking* approach presented in [14, 15]. Each variable type in the input specification is assigned to the most efficient representation for that variable type. The goal is to have a platform where the strength of each symbolic representation is utilized as much as possible, and the deficiencies of a representation are compensated by the existence of other representations.

The rest of the chapter is organized as follows. In Section 3.1 we define the composite symbolic representation and describe the architecture and the design of the Composite Symbolic Library. We present the algorithms for manipulating the composite symbolic representation in Section 3.2. In Section 3.3 we describe some heuristics for improving the performance of the Composite Symbolic Library. We present the experiments that demonstrate the effectiveness of our heuristics in Section 3.4. In Section 3.5 we discuss the related work.

3.1 The Composite Symbolic Representation

To combine different symbolic representations we use the composite model checking approach presented in [14, 15]. The basic idea in composite model checking is to map each variable in the input specification to a symbolic representation type. For example, boolean and enumerated variables can be mapped to the BDD representation, and integers can be mapped to an arithmetic constraint representation. We encode the sets of system states and transitions as a disjunction of conjunctions of type specific representations. For example, a disjunct may consist of a boolean formula stored as a BDD representing the states of the boolean and the enumerated variables, and a linear arithmetic constraint representation representing the states of the integer variables. We call this disjunctive representation a *composite representation*. Each atomic event in the input specification is conjunctively partitioned where each conjunct specifies the effect of the event on the variables represented by a single symbolic representation. For example, one conjunct specifies the effect of the event on the variables encoded using BDDs, whereas another conjunct specifies the effects of the event on the variables encoded using linear arithmetic constraints. The pre- and post-condition computations are computed independently for each symbolic representation by exploiting the conjunctive partitioning of the atomic events. The key observation here is the fact that conjunctive partitioning of the atomic events allows pre- and post-condition computations to distribute over different symbolic representations. We also implement algorithms for conjunction, disjunction, complement, subsumption, equivalence and satisfiability checking for the disjunctive composite representation, which use the corresponding methods for different symbolic representations.

Our current implementation of the Composite Symbolic Library uses two symbolic representations: BDDs for boolean logic formulas and the polyhedral representation for Presburger arithmetic formulas. We call these the *basic symbolic representations*. For the BDD representations we use the Colorado University Decision Diagram Package (CUDD) [1]. For the Presburger arithmetic formula manipulation we use the Omega Library [54, 2].

We implemented the Composite Symbolic Library in C++ and Fig. 3.1 shows its class hierarchy as a UML class diagram[1]. The abstract class `Symbolic` serves as an interface to all symbolic representations including the composite representation. The classes `BoolSym` and `IntSym` are the symbolic representations for boolean and integer variable types, respectively. The class `BoolSym` serves as a wrapper for the BDD library CUDD [1]. It is derived from the abstract class `Symbolic`. Similarly, the class `IntSym` is also derived from the abstract class `Symbolic` and serves as a wrapper for the Omega Library [2].

The class `CompSym` is the class for the composite representation. It is derived from the class `Symbolic` and uses the classes `IntSym` and `BoolSym` (through the `Symbolic` interface) to manipulate the composite representation. There is no dependency among the classes `CompSym`, `IntSym`, and `BoolSym`. Note that this

[1]In UML class diagrams, triangle arcs denote *generalization*, diamond arcs denote *aggregation*, dashed arcs denote *dependency*, and solid lines denote *association* among classes.

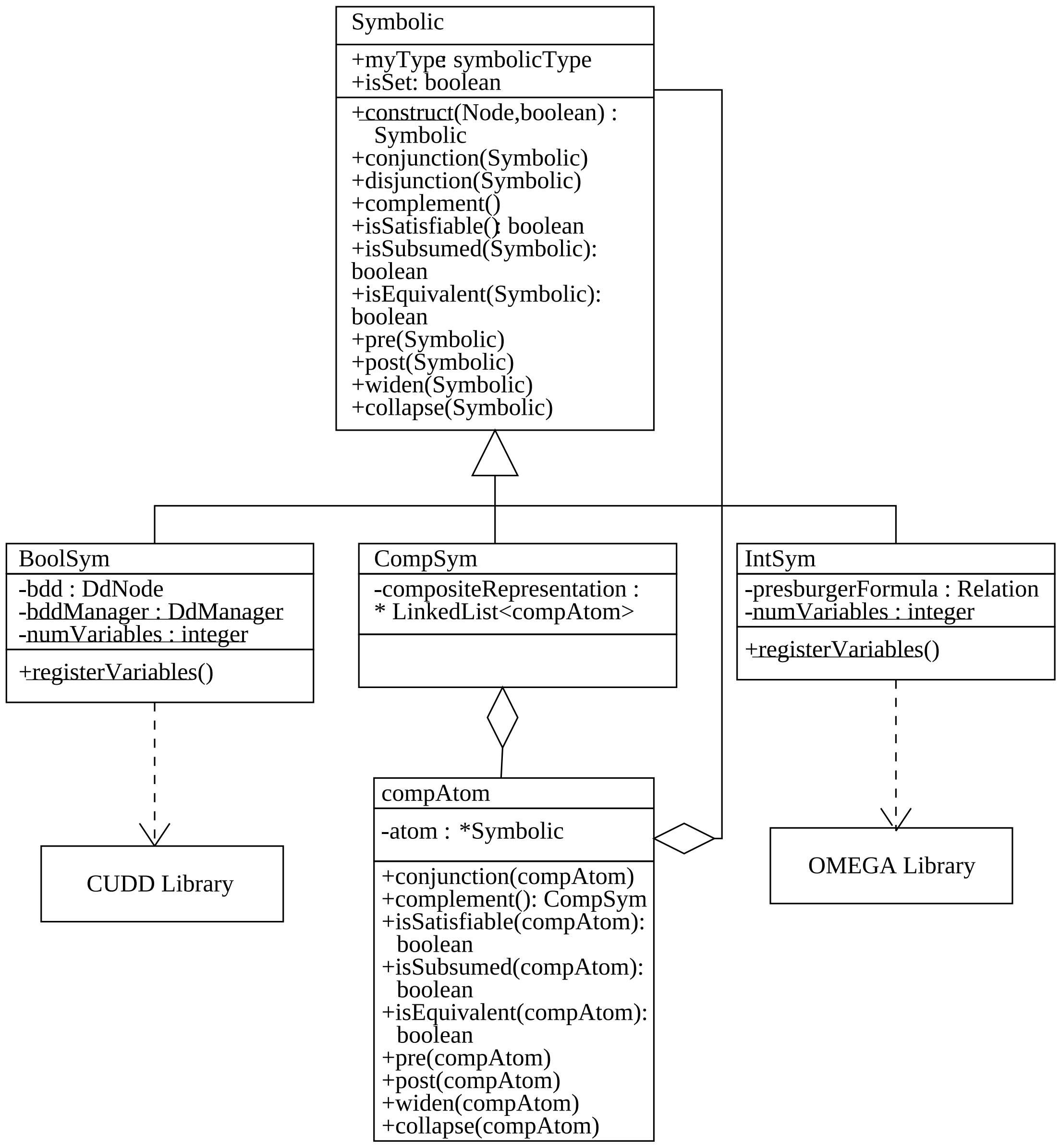


Figure 3.1: Class diagram for the Composite Symbolic Library

design is an instance of the composite design pattern given in [43].

To verify a system with our tool, one has to specify its initial condition, transition relation, and state space using a set of composite formulas, whose syntax is given in Figure 2.4.

A transition relation can be specified using a composite formula by using the unprimed variables to denote the current state variables and the primed variables to denote the next state variables. A method called `registerVariables` in `BoolSym` and `IntSym` is used to register the current and the next state variable names during the initialization of the representation.

Given a composite formula, the method `construct()` in the class `Symbolic` traverses the syntax tree and calls the constructor of the class `BoolSym` when a boolean formula is encountered and calls the constructor of the class `IntSym` when an integer formula is encountered. If the composite formula consists of both integer and boolean formulas then the constructor of the class `CompSym` is called. In the class `CompSym`, a composite formula, A, is represented in our *composite representation* as

$$A \equiv \bigvee_{i=1}^{n} \bigwedge_{t \in T} a_{it}$$

where a_{it} denotes the formula of type t in the ith disjunct, and n and T denote the number of disjuncts and the set of basic symbolic representations, respectively.

We call each disjunct $\wedge_{t \in T} a_{it}$ a *composite atom*. Fig. 3.2 shows the composite atoms in an example composite formula. Each composite atom is implemented as an instance of a class called `compAtom` (see Fig. 3.1). Each `compAtom` object represents a conjunction of formulas each of which is either a boolean or an integer formula.

A composite formula that is stored in a `CompSym` object is implemented as a list of `compAtom` objects, which corresponds to the disjunction in the composite representation. Note that `Symbolic` members of the `compAtom` class cannot be of

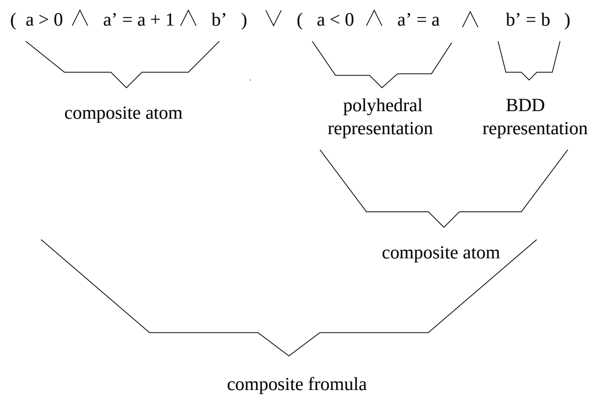

(a > 0 ∧ a' = a + 1 ∧ b') ∨ (a < 0 ∧ a' = a ∧ b' = b)
composite atom
polyhedral representation
BDD representation
composite atom
composite fromula

Figure 3.2: An example composite formula. a is an integer variable and b is a boolean variable.

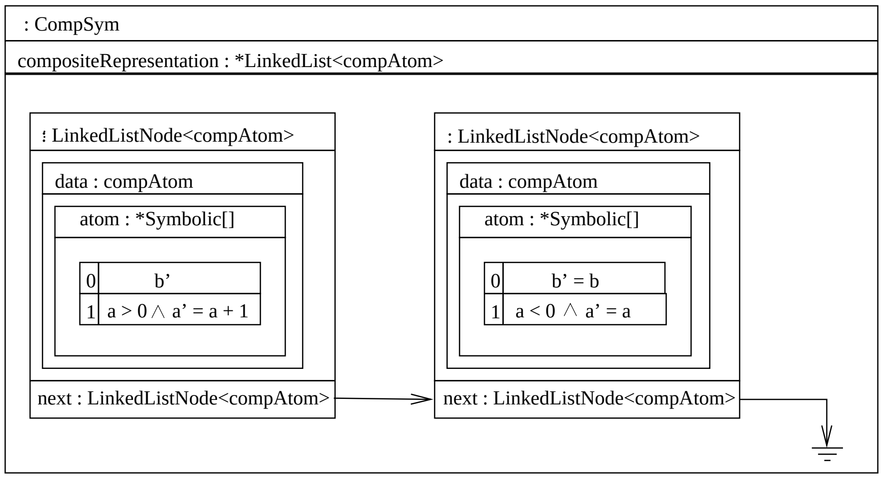

: CompSym
compositeRepresentation : *LinkedList<compAtom>
: LinkedListNode<compAtom>
data : compAtom
atom : *Symbolic[]
0 b'
1 a > 0 ∧ a' = a + 1
next : LinkedListNode<compAtom>
: LinkedListNode<compAtom>
data : compAtom
atom : *Symbolic[]
0 b' = b
1 a < 0 ∧ a' = a
next : LinkedListNode<compAtom>

Figure 3.3: An instance of the **CompSym** class representing the composite formula in Fig. 3.2

type `CompSym`. Fig. 3.3 shows internal representation of the composite formula given in Fig. 3.2 in a `CompSym` object. The field *atom* is an array of pointers to the class `Symbolic` and the size of the array is the number of basic symbolic representations.

The classes `CompSym` and `compAtom` use a `TypeDescriptor` class, which records the variable types used in the input specification. Our library can adapt itself to any subset of the supported variable types, i.e., if a variable type is not present in the input specification, the symbolic library for that type will not be called during the execution. For example, given an input specification with no integer variables our tool will behave as a BDD-based model checker without making any calls to the Omega Library.

3.2 Manipulation of the Composite Representation

In this section we present algorithms for basic set operations and pre- and post-condition computations on our disjunctive composite representation. These algorithms are implemented as methods in the classes `compAtom` and `CompSym` in the Composite Symbolic Library. Note that the algorithms given in this section are independent of the type and the number of the basic symbolic representations used.

Throughout this section, n_A, T, and T^t_{Op} denote the number of composite atoms in composite formula A, the set of basic symbolic representations in the Composite Symbolic Library, and the time complexity of the operation Op for the basic symbolic representation t. The operations are $Op \in \{Conjunction$, $Disjunction$, $Complement$, $IsSubsumed$, $IsEquivalent$, $IsSatisfiable\}$.

Subsumption Check: A composite atom $a \equiv \bigwedge_{t \in T} a_t$ is subsumed by a composite atom $b \equiv \bigwedge_{t \in T} b_t$ iff for each symbolic representation a_t in a, a_t is subsumed by b_t, which is the corresponding symbolic representation in b. For instance let

```
IsSubsumed(a, b): boolean
a, b: composite atom
for each basic symbolic representation $t$ do
    if $a_t \not\Rightarrow b_t$ then
        return $false$
return $true$
```

Figure 3.4: Algorithm for checking the subsumption relation between two composite atoms

composite atoms a and b be

$$a \equiv x \wedge y \wedge z > 0, b \equiv x \wedge z \geq 0$$

where x and y are boolean variables and z is an integer variable. $a \Rightarrow b$ since the valuations of the variables x and y that satisfy the formula $x \wedge y$ is subsumed by the valuations of the variables x and y that satisfy the formula x, and the valuations of the variable z that satisfy the formula $z > 0$ is subsumed by the valuations of the variable z that satisfy the formula $z \geq 0$. Fig. 3.4 shows the *IsSubsumed* algorithm for checking subsumption relation between two composite atoms. The worst case time complexity of the algorithm is $O(\sum_{t \in T} T^t_{IsSubsumed})$.

A composite formula $A \equiv \bigvee_{i=1}^{n_A} a_i$ is subsumed by a composite formula $B \equiv \bigvee_{k=1}^{n_B} b_k$ iff $\forall i$ s.t. $1 \leq i \leq n_A$, $a_i \Rightarrow B$. For instance let A and B be

$$A \equiv (x \wedge z > 0) \vee (x \wedge y \wedge z \leq 0), B \equiv z \geq 0 \vee x$$

where x and y are boolean variables and z is an integer variable. A is subsumed by B since both composite atoms $(x \wedge z > 0)$ and $(x \wedge y \wedge z \leq 0)$ are subsumed by

```
IsSubsumed($A$, $B$): boolean
found: boolean
$A$, $B$, $C$: composite formula
for each composite atom $a$ in $A$ do
    $found \leftarrow$ false
    for each composite atom $b$ in $B$ do
        if $a \Rightarrow b$ then
            $found \leftarrow true$
            break
    if $\neg found$ then
        $C \leftarrow a \wedge \neg B$
        if $isSatisfiable(C)$ then
            return $false$
return $true$
```

Figure 3.5: Algorithm for checking the subsumption relation between two composite formulas

B. Note that $(x \wedge z > 0)$ is subsumed by $z \geq 0$ and $(x \wedge y \wedge z \leq 0)$ is subsumed by x. So the most straightforward way of checking the subsumption relation between two composite formulas A and B is to iterate through the composite atoms in A and check the subsumption relation between each composite atom a_i in A and B. If there exists a composite atom a_i in A such that a_i is not subsumed by B we can conclude that A is not subsumed by B. On the other hand, if there exists no such composite atom in A then we can conclude that A is subsumed by B.

Fig. 3.5 shows the algorithm for checking the subsumption relation between the two composite formulas A and B. For each composite atom a_i in A, first the

```
1 IsSatisfiable(a): boolean
2 a: composite atom
3 for each symbolic representation $t$ do
4     if $\neg isSatisfiable(a_t)$ then
5         return $false$
6 return $true$
```

Figure 3.6: Algorithm for checking emptiness of a composite atom

algorithm checks if a_i is subsumed by any composite atom in B (lines 5-9). If there exists no composite atom b in B such that a is subsumed by b then this does not mean that a is not subsumed by B. Next, the algorithm computes $a \wedge \neg B$ and assigns the result to a composite formula C. If C is satisfiable, then this means that a is not subsumed by B and the algorithm exits by returning false (lines 10-13). Otherwise, the algorithm continues until either it finds out that there exists a composite atom a that is not subsumed by B or it has checked all the composite atoms in A, in which case it returns true (line 14).

The worst case time complexity of the algorithm is

$$O(n_A \times n_B \times \sum_{t \in T} T^t_{IsSubsumed} + n_A \times (n_B \times \sum_{t \in T} T^t_{Complement} + \\ |T|^{n_B} \times \sum_{j=1, t_j \in T}^{n_B} T^{t_j}_{Conjunction} + |T|^{n_B} \times \sum_{t \in T} T^t_{IsSatisfiable})).$$

The exponential component of the formula is due to the complement operation at line 11 in Fig. 3.5, which is of exponential time complexity, as will be explained below.

Satisfiability Check: A composite atom a is not satisfiable iff there exists a symbolic representation a_t in a such that a_t is not satisfiable. Fig. 3.6 shows

```
IsSatisfiable(A): boolean
A: composite formula
for each composite atom $a$ in $A$ do
    if $isSatisfiable(a)$ then
        return $true$
return $false$
```

Figure 3.7: Algorithm for checking satisfiability of a composite formula

IsSatisfiable algorithm for checking satisfiability of a composite atom. The worst case time complexity of the algorithm is $O(\sum_{t \in T} T^t_{IsSatisfiable})$.

A composite formula A is not satisfiable iff for all composite atoms a_i in A, a_i is not satisfiable. Fig. 3.7 shows *IsSatisfiable* algorithm for a composite formula. The worst case time complexity of the algorithm is $O(n_A \times \sum_{t \in T} T^t_{IsSatisfiable})$.

Pre-Condition Computation: Given two composite formulas $A \equiv \bigvee_{i=1}^{n_A} \bigwedge_{t \in T} a_{it}$ and $B \equiv \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} b_{kt}$, where A represents the set of states and B represents the transition relation, the pre-condition of A with respect to B can be computed as

$$Pre(A, B) \equiv \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} Pre(a_i, b_k) \equiv \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} Pre(a_{it}, b_{ky}) \equiv \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} \exists V' a'_{it} \wedge b_{kt}$$

where V' is the set of next state variables and a'_{it} is obtained by replacing every variable in a_{it} with the corresponding next-state variable. Note that the above property holds because the existential variable elimination in the $Pre(A, B)$ computation distributes over the disjunctions, and due to the partitioning of the variables based on the basic symbolic types, the existential variable elimination also distributes over the conjunction above [15].

Computing the pre-condition of A with respect to B is equivalent to computing

```
1 $Pre(a,b)$: composite atom
2 $a,b$: composite atom
3 $c$: composite atom
4 for each symbolic representation $t$ do
5     $c_t \leftarrow \exists V' a'_t \wedge b_t$
6 return $c$
```

Figure 3.8: Algorithm for computing the pre-condition of a composite atom with respect to a composite atom

the set of states that can reach the set of states represented by A by a single transition in B. For instance let A and B be

$$A \equiv y = 1,\ B \equiv (x \wedge y' = 1) \vee (\neg x \wedge y \geq 0 \wedge y' = y + 1)$$

where x is a boolean variable and y is an integer variable. Then $Pre(A, B)$ can be computed as:

$$\begin{aligned} Pre(A, B) &\equiv Pre(y = 1, x \wedge y' = 1) \vee Pre(y = 1, \neg x \wedge y \geq 0 \wedge y' = y + 1) \\ &\equiv x \vee (\neg x \wedge y = 0) \end{aligned}$$

Fig. 3.8 and 3.9 show the pre-condition computation algorithms for composite atoms and composite formulas, respectively. The worst case time complexity of the pre-condition algorithm for composite atoms is $O(\sum_{t \in T} T^t_{\textit{Pre-Condition}})$ and that of the pre-condition algorithm for composite formulas is $O(n_A \times n_B \times \sum_{t \in T} T^t_{\textit{Pre-Condition}})$.

Conjunction: Given two composite formulas, A and B, their conjunction can be computed as:

$$A \wedge B \equiv \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} (a_{it} \wedge b_{kt})$$

```
1 $Pre(A, B)$: composite formula
2 $A$, $B$, $C$: composite formula
3 $C \leftarrow false$
4 for each composite atom $a$ in $A$ do
5     for each composite atom $b$ in $B$ do
6         $C \leftarrow C \vee Pre(a,b)$
7 return $C$
```

Figure 3.9: Algorithm for computing the pre-condition of a composite formula with respect to a composite formula

The worst case time complexity of computing the conjunction of two composite atoms is $O(\sum_{t\in T} T^t_{Conjunction})$ and that of computing the conjunction of two composite formulas is $O(n_A \times n_B \times \sum_{t\in T} T^t_{Conjunction})$.

Complement : The complement of a composite atom $a \equiv \bigwedge_{t\in T} a_t$ is a composite formula $B \equiv \bigvee_{t\in T} \neg a_t$. Given a composite formula A we can compute A's complement as

$$\neg A \equiv \bigvee_{i=1}^{|T|^{n_A}} a_i, \quad a_i \equiv \bigwedge_{j=1,t_j\in T}^{n_A} \neg a_{t_j}.$$

Fig. 3.10 and 3.11 show the complement algorithms for a composite atom and a composite formula, respectively. The worst case time complexity of the complement algorithm for a composite atom is $O(\sum_{t\in T} T^t_{Complement})$ and that of the complement algorithm for a composite formula is

$$O(n_A \times \sum_{t\in T} T^t_{Complement} + |T|^{n_A} \times \sum_{j=1,t_j\in T}^{n_A} T^{t_j}_{Conjunction}).$$

Disjunction: The disjunction of a composite atom a with a composite atom b is a composite formula $A \equiv a \vee b$. Given two composite formulas, A and B, we

```
1 $\mathit{Complement}(a)$: composite formula
2 $a$, $A$: composite formula
3 $A \leftarrow false$
4 for each symbolic representation $a_t$ in $a$ do
5     $A \leftarrow A \vee \neg a_t$
6 return $A$
```

Figure 3.10: Algorithm for computing the complement of a composite atom

```
1 $\mathit{Complement}(A)$: composite formula
2 $A$, $B$: composite formula
3 $B \leftarrow false$
4 let $A \equiv \bigvee_{i=1}^{n_A} \bigwedge_{t \in T} a_{it}$
5 let $n_A$ be the number of composite atoms in $A$
6 let $a_i \equiv \bigwedge_{t \in T}^{T} a_{it}$ be a composite atom in $A$
7 for each combination of $(a_{1_{t_1}}, a_{2_{t_2}}, ..., a_{n_{t_n}})$ s.t. $t_i \in T$ and $1 \leq i \leq n_A$ do
8     $B \leftarrow B \vee \bigwedge_{i=1}^{n_A} \neg a_{i_{t_i}}$
9 return $B$
```

Figure 3.11: Algorithm for computing the complement of a composite formula

define A disjunction B as

$$A \vee B \equiv \bigvee_{i=1}^{n_A+n_B} \bigwedge_{t \in T} c_{it}$$

where for $1 \leq i \leq n_A$ $c_{it} \equiv a_{it}$ and for $n_A + 1 \leq i \leq n_A + n_B$ $c_{it} \equiv b_{it}$.

The worst case time complexity of computing the disjunction of two composite atoms is constant and that of computing the disjunction of two composite formulas is $O(n_A+n_B)$. As the worst case time complexity formulas show, while computing the disjunction of two composite atoms or two composite formulas, no operation is performed on symbolic representation level. Since we use a disjunctive composite representation, the disjunction operation can be performed by a concatenation operation on the linked list structure that is used to implement a `CompSym` object (see Fig. 3.3).

3.3 Heuristics for Efficient Automated Verification with the Composite Representation

In this section we present several heuristics that improve the performance of automated verification using the Composite Symbolic Library [66]. Our heuristics make use of the following observations: 1) the efficient operations on BDDs (e.g., satisfiability checking) can be used to mask expensive operations on polyhedra (e.g., image computations and satisfiability checking), 2) our disjunctive representation can be exploited by interleaving the computation of pre- and post-conditions with subsumption checks, and 3) the size of a composite representation can be minimized by iteratively merging matching constraints and removing redundant ones.

Masking Integer Operations: The Composite Symbolic Library currently supports two basic symbolic representations: BDDs to represent boolean and enumerated variables and polyhedral representation of linear arithmetic formulas to represent integer variables. Existential variable elimination for linear integer arithmetic formulas is NP-hard and it is used in the satisfiability check and the pre- and post-condition computations. However, since the BDD representation is canonical, a satisfiability check for BDDs can be performed in constant time by comparing the root node of a BDD representation to the unique BDD that corresponds to *false*. This discrepancy in the performances of the BDD representation and the polyhedral representation in checking satisfiability can be exploited to speed up the pre- and post-condition computations on the composite symbolic representation.

Given two composite formulas $A \equiv \bigvee_{j=1}^{n_A} a_{jb} \wedge a_{ji}$ and $C \equiv \bigvee_{k=1}^{n_C} c_{kb} \wedge c_{ki}$, where A represents a set and C represents the transition relation, a_{jb} and c_{kb} correspond to boolean formulas, and a_{ji} and c_{ki} correspond to integer formulas, pre-condition of A with respect to C can be written as

$$\mathit{Pre}(A, C) \equiv \bigvee_{j=1}^{n_A} \bigvee_{k=1}^{n_C} \mathit{Pre}(a_{jb}, c_{kb}) \wedge \mathit{Pre}(a_{ji}, c_{ki})$$

Instead of computing $\mathit{Pre}(a_{jb}, c_{kb})$ and $\mathit{Pre}(a_{ji}, c_{ki})$ and then taking the conjunction of the two, we can first compute $\mathit{Pre}(a_{jb}, c_{kb})$ and then check it for satisfiability. Since $\mathit{Pre}(a_{jb}, c_{kb})$ is a boolean formula and represented by BDDs, checking satisfiability of $\mathit{Pre}(a_{jb}, c_{kb})$ is cheaper than checking satisfiability of $\mathit{Pre}(a_{ji}, c_{ki})$, which is represented by polyhedra. We should compute $\mathit{Pre}(a_{ji}, c_{ki})$, which involves the manipulation of the polyhedral representation only if $\mathit{Pre}(a_{jb}, c_{kb})$ is satisfiable. If it is not satisfiable then we will not compute $\mathit{Pre}(a_{ji}, c_{ki})$ since we can deduce that $\mathit{Pre}(a_{jb}, c_{kb}) \wedge \mathit{Pre}(a_{ji}, a_{ki})$ evaluates to *false*. As a result expensive integer manipulation is masked by cheaper boolean manipulation.

Subsumption Check: The subsumption check algorithm given in Fig. 3.5 uses the complement operation at the composite formula level (line 11). The worst case time complexity of the complement operation on a composite formula B is exponential in the number of composite atoms in B. For the subsumption check algorithm, computing the complement of B means that all the composite atoms in B are taken into consideration to decide if a composite atom a is subsumed by B. However, deciding if a composite atom a is subsumed by a composite formula B does not always require one to consider all the composite atoms in B. For instance, let the composite atom a and the composite formula B be,

$$a \equiv (x \wedge y \wedge z \geq 0), \quad B \equiv (x \wedge z = 0) \vee (x \wedge z \geq 1) \vee (\neg x \wedge z < 0)$$

where x and y are boolean variables and z is an integer variable. Since each composite atom in a composite formula corresponds to a disjunct of the composite formula, B has three composite atoms b_1, b_2, and b_3, which correspond to $(x \wedge z = 0)$, $(x \wedge z \geq 1)$, and $(\neg x \wedge z < 0)$, respectively. In order to decide if a is subsumed by B we do not need to consider all the composite atoms in B. For this example, it is sufficient to compare a against b_1 and b_2 (note that a is subsumed by $b_1 \vee b_2$) only to conclude that a is subsumed by B. However, the algorithm given in Fig. 3.5 will process b_1, b_2, and b_3 by computing the complement of B.

In the light of this observation we propose a more efficient solution to the subsumption check problem for composite formulas. Given two composite formulas A and B, for each composite atom a in A, our solution iteratively computes the unsubsumed part of a, U that is not covered by the composite atoms in B that have been examined so far. U is initialized to a and for each k s.t. $1 \leq k \leq n_B$, U is updated as $U \wedge \neg b_k$. After U is updated using b_k it is checked for satisfiability. If it becomes not satisfiable then the algorithm skips checking the remaining composite atoms in B and concludes that a is subsumed by B. Otherwise, it continues with b_{k+1}. After checking all composite atoms in B if U is not satisfiable then the algorithm concludes that a is not subsumed by B.

Simplification Algorithms: The number of composite atoms in a composite formula that results from the conjunction operation is linear in the product of the number of composite atoms of the input composite formulas. The number of composite atoms in a composite formula that results from the complement operation is exponential in the number of composite atoms of the input composite formula. Most of the time these resulting composite formulas are not minimal in terms of the number of composite atoms they have. Since time complexity of manipulating a composite formula is dependent on the number of composite atoms in it, we need to reduce the number of composite atoms in a composite formula as much as possible to make the verification feasible in terms of both time and memory. We describe a simplification method that can be tuned for three different degrees of aggressiveness, which are $S1$, $S2$, and $S3$ in the increasing aggressiveness order.

A composite formula having two composite atoms, a and c, can be simplified and represented by a single composite atom d if one of the following holds:

1. a is subsumed by c. In this case $d \equiv c$.

2. c is subsumed by a. In this case $d \equiv a$.

3. There exists a symbolic representation t_j s.t. for all symbolic representations t_i s.t. $t_i \neq t_j$, $a_{t_i} \equiv c_{t_i}$. In this case for all t_i s.t. $t_i \neq t_j$, $d_{t_i} \equiv a_{t_i}$ and $d_{t_j} \equiv a_{t_j} \vee b_{t_j}$.

The most aggressive version of the simplification method ($S3$) combines pairs of composite atoms in a composite formula until there exists no pair of composite atoms that can be combined according to the above rules. So, for each pair of compatom it performs the above rules in the order 1, 2, and 3. $S3$ simplification

```
ComputeEF(A, C): composite formula
A, C: composite formula
$S_{new} \leftarrow A$
do
    $S_{old} \leftarrow S_{new}$
    $S_{new} \leftarrow Pre(S_{old}, C) \vee S_{old}$
    $S_{new} \leftarrow Simplify(S_{new})$
while ($S_{new} \not\Leftrightarrow S_{old}$)
return $S_{new}$
```

Figure 3.12: Algorithm for computing the least fixpoint for EF

method can be changed into a less aggressive version ($S2$) by checking the equivalence only on the boolean type when applying rule 3. $S1$ simplification method is similar to $S2$ but it eliminates the subsumption check (rules 1 and 2).

Combining Pre-Condition, Subsumption Check, and Disjunction Computations: All CTL operators can be defined in terms of the least and the greatest fixpoints. Temporal operator EF is defined as a least fixpoint as EF $p \equiv \mu x \, . \, p \;\vee\;$ EX x. Fig. 3.12 shows the algorithm for computing the least fixpoint for EF. Given two composite formulas $A \equiv \bigvee_{i=1}^{n_A} a_i$ and $C \equiv \bigvee_{j=1}^{n_C} c_j$, where A represents a set of states and C represents the transition relation, the algorithm computes the set of states that satisfy EF(A) iteratively. S_{new} represents the largest set computed so far. At line 5 of *ComputeEF* algorithm $S_{new} \equiv \bigvee_{k=1}^{n_S} s_k$ and $S_{old} \equiv \bigvee_{k=1}^{n_B} b_k$ where one of the following holds for s_k:

1. $s_k \equiv Pre(b_i, c_j)$, where $1 \leq i \leq n_B$ and $1 \leq j \leq n_C$, and $s_k \not\Rightarrow S_{old}$,

```
1  $PreDisjunction(A, C, isSubsumed)$: composite formula
2  $S_{res}$, $A$, $C$: composite formula
3  $isSubsumed$: boolean
4  $isSubsumed \leftarrow true$
5  $S_{res} \leftarrow A$
6  for each composite atom $a$ in $A$ do
7      for each composite atom $c$ in $C$ do
8          $s_k \equiv Pre(a, c)$
9          if $s_k \not\Rightarrow A$ then
10             $isSubsumed \leftarrow false$
11             $S_{res} \leftarrow S_{res} \vee s_k$
12 return $S_{res}$
```

Figure 3.13: A pre-condition algorithm with subsumption check and disjunction

2. $s_k \equiv Pre(b_i, c_j)$, where $1 \leq i \leq n_B$ and $1 \leq j \leq n_C$, and $s_k \Rightarrow S_{old}$,

3. $s_k \Rightarrow S_{old}$ and there exists no i, j, where $1 \leq i \leq n_B$ and $1 \leq j \leq n_C$, s.t $s_k \equiv Pre(b_i, c_j)$.

Note that composite atoms that satisfy (1) can be used to decide if S_{new} is subsumed by S_{old} earlier during the computation of pre-condition and eliminate subsumption check at line 7 of the algorithm. This may serve as an improvement over the algorithm in Fig. 3.12 since we can eliminate processing composite atoms in S_{new} that satisfy (3) during the subsumption check at line 7 of the algorithm. An additional improvement can be achieved by taking the disjunction of s_k with S_{res} only if s_k is not subsumed by A and prevent the unnecessary increase in

```
1  $\mathit{EfficientEF}(A,\, C)$: composite formula
2  $A,\, C$: composite formula
3  $isSubsumed$: boolean
4  $S_{new} \leftarrow A$
5  do
6      $S_{old} \leftarrow S_{new}$
7      $S_{new} \leftarrow PreDisjunction(S_{old}, C, isSubsumed)$
8      $S_{new} \leftarrow Simplify(S_{new})$
9  while $(\neg isSubsumed)$
10 return $S_{new}$
```

Figure 3.14: A more efficient algorithm for computing the least fixpoint for EF

the number of composite atoms in S_{res}. Fig. 3.13 and 3.14 show the algorithms *PreDisjunction*, which computes the pre-condition along with the subsumption check and the disjunction, and *EfficientEF*, which computes the least fixpoint for EF using the *PreDisjunction* algorithm, respectively.

3.4 Experimental Evaluation of the Heuristics

We have experimented with the heuristics explained in Section 3.3 using a large set of specifications, which we describe below. Table 3.1 shows the different properties verified for each specification. Each instance is labeled using NAME$[n_{pc}]$-$[n_{pr}]$ where n_{pc} and n_{pr} are the number of processes and the property number, respectively. Specifications of all these examples and properties are available at: http://www.cs.ucsb.edu/~bultan/composite/

Problem Instance	Property
BK2-1	$AG(\neg(p1 = cs \wedge p2 = cs))$
BK3-1	$AG(\neg((p1 = cs \wedge p2 = cs) \vee (p1 = cs \wedge p3 = cs) \vee$ $(p2 = cs \wedge p3 = cs)))$
BK[2,3]-2	$AG(\neg(p1 = try) \vee AF(p1 = cs))$
TK2-1	$AG(\neg(p1 = cs \wedge p2 = cs))$
TK3-1	$AG(\neg((p1 = cs \wedge p2 = cs) \vee (p1 = cs \wedge p3 = cs) \vee$ $(p2 = cs \wedge p3 = cs)))$
TK[2,3]-2	$AG(\neg(p1 = try) \vee AF(p1 = cs))$
BR[2,3,4,P]-1	$AG(chair \leq 1)$
BR[2,3,4,P]-2	$AG(open \leq 1)$
BR[2,3,4,P]-3	$AG(barber \leq 1)$
C-1	$AG(\neg((xShared \geq 1 \wedge xExclusive \geq 1) \vee$ $xExclusive \geq 2))$
C-2	$AG(xWaitS \geq 1 \Rightarrow AF(xShared \geq 1))$
C-3	$AG(xWaitS \geq n \Rightarrow AF(xShared \geq n))$
C-4	$AG(xWaitE \geq 1 \Rightarrow AF(xExclusive \geq 1))$
C-REF-1	$AG(\neg((xShared >= 1 \wedge xExclusive >= 1) \vee$ $xExclusive >= 2))$
C-REF-2	$AG(xWaitS \geq 1 \Rightarrow AF(xShared \geq 1))$
C-REF-3	$AG(xWaitS \geq n \Rightarrow AF(xShared \geq n))$
ISORT	$AG(\neg((pc = entryA1 \wedge k \geq n) \vee (pc = entryA1 \wedge$ $k \leq -1) \vee (pc = entryA3 \wedge i \geq n-1) \vee$ $(pc = entryA3 \wedge i \leq -2) \vee (pc = entryA2 \wedge$ $i \leq -1) \vee (pc = entryA2 \wedge i \leq -2) \vee$ $(pc = entryA2 \wedge i \geq n-1)$ $\vee (pc = entryA2 \wedge i \geq n-1)))$
RW[16,32,64,P]	$AG(busy \Rightarrow nr = 0)$
SIS-1	$AG(Inject \Rightarrow Pressure = TooLow)$
SIS-2	$AG((Reset \wedge \neg(Pressure = TooHigh) \Rightarrow \neg Overridden)$ $\wedge (Reset \wedge Pressure = TooLow \Rightarrow Inject))$
LC	$AG(count > 1 \iff Office = Occupied \wedge$ $Occupants = Multiple)$

Table 3.1: List of problem instances used in the experiments

- BK[2,3,4] and TK[2,3,4] are mutual exclusion protocols (for (2,3,4) processes, respectively). We verified both mutual exclusion (BK[2,3,4]-1, TK[2,3,4]-1) and starvation-freedom (BK[2,3,4]-2, TK[2,3,4]-2) properties for these protocols.

- We verified three properties for sleeping barber monitor specifications with 2, 3, and 4 customer processes and one barber process (BR[2,3,4]-[1,2,3]). We also verified the three properties (BRP-[1,2,3]) on the parameterized system.

- We analyzed a parameterized cache coherence protocol specification (C-[1,2,3,4], C-REF-[1,2,3]) given in [33]. We verified all the properties given in [33].

- ISORT is a specification from [34], for array bound checking of an implementation of insertion sort algorithm.

- RW[16,32,64] is a monitor specification for the readers-writers problem for various numbers of processes [4].

- LC and SIS are two reactive software specifications. LC is an office light control system specification written in statecharts [17]. SIS is specification of a safety injection system for a nuclear reactor [27]

We obtained the experimental results on a SUN ULTRA 10 workstation with 768 Mbytes of memory, running SunOs 5.7.

In Table 3.2 we show the sizes of the problem instances we used in our experiments. Each row in Table 3.2 shows the size of the composite symbolic representation for the transition relation used in that instance. The size of a composite representation is shown in terms of the number of composite atoms, the number of polyhedra, the number of equality (EQ) and greater-than-or-equal-to (GEQ) constraints, the number of BDD nodes, the number of integer variables, and the number of boolean variables in it. Table 3.3 shows the size of the maximum

Problem Instance	Transition Relation Size					
	Composite	Polyhedra	EQ, GEQ	BDD	# int.	# bool.
BK2-[1,2]	6	8	32	69	2	4
BK3-[1,2]	9	121	126	165	3	6
TK2-[1,2]	6	6	38	69	4	4
TK3-[1,2]	9	9	66	165	5	6
BR2-[1,2,3]	8	8	48	88	3	4
BR3-[1,2,3]	10	10	60	140	3	5
BR4-[1,2,3]	12	12	72	204	3	6
BRP-[1,2,3]	6	6	62	32	6	2
C-[1,2,3,4]	10	10	120	94	6	4
C-REF-[1,2,3]	10	10	120	88	6	4
ISORT	8	8	27	56	3	3
RW16	32	32	64	1570	1	17
RW32	64	64	128	6210	1	33
RW64	128	128	256	24706	1	65
RWP	4	4	56	11	7	1
SIS-1	8	10	50	117	3	6
SIS-2	8	14	1573	117	6	6
LC	12	12	25	271	1	7

Table 3.2: Sizes of the transition relations for the problem instances used in the experiments

fixpoint iteration result for the optimized version of the Composite Symbolic Library with all the presented heuristics included. For 14 of 35 problem instances the verification procedure runs out of memory if the heuristics are not used.

Experimental results for the verifier with different versions of the simplification algorithm and without simplification are given in Tables 3.4 and 3.5. The label $S1$-$S2$-$S3$ indicates that at each simplification point the simplification algorithm with $S1$, $S2$, and $S3$ are called in this order. So a multi-level simplification is achieved starting with the least aggressive version and continuing by increasing the degree of aggressiveness. Results show that multi-level simplification performs better than single level simplification. It also indicates that the speedup obtained by simplifying the composite representation is significant. Without simplification,

Problem	Maximum Fixpoint Iteration Result Size				Fixpoints	
Instance	Composite	Polyhedra	EQ, GEQ	BDD	EF	EG
BK2-1	6	10	20	28	4	–
BK2-2	4	5	10	21	1	9
BK3-1	22	61	183	171	5	–
BK3-2	16	48	146	141	4	15
TK2-1	6	11	36	31	9	–
TK2-2	8	25	74	41	7	5
TK3-1	19	28	117	168	11	–
TK3-2	27	54	191	195	6	8
BR2-1	14	29	87	60	9	–
BR2-2	4	4	12	13	5	–
BR2-3	9	10	30	42	11	–
BR3-1	16	35	105	90	10	–
BR3-2	4	4	12	15	5	–
BR3-3	12	14	42	75	12	–
BR4-1	18	41	123	128	11	–
BR4-2	4	4	12	17	5	–
BR4-3	15	18	54	122	13	–
BRP-1	4	24	167	10	8	–
BRP-2	4	6	26	10	5	–
BRP-3	5	14	73	13	11	–
C-1	5	7	42	23	4	–
C-2	11	43	258	51	5	3
C-3	15	81	494	72	8	3
C-4	15	20	111	73	13	11
C-REF-1	4	6	36	20	4	–
C-REF-2	7	8	46	33	5	3
C-REF-3	11	186	1116	53	9	3
ISORT	5	8	10	19	5	–
RW16	1	1	1	2	1	–
RW32	1	1	1	2	1	–
RW64	1	1	1	2	1	–
RWP	1	1	7	2	1	–
SIS-1	1	1	1	3	1	–
SIS-2	2	7	49	14	2	–
LC	4	4	5	35	3	–

Table 3.3: Maximum fixpoint iteration result sizes and the number of fixpoint iterations for the optimized version of the Composite Symbolic Library for problem instances used in the experiments

Problem Instance	S1-S2-S3	S1	S2	S3	None
BK2-1	0.21	0.09	0.10	0.2	0.1
(L)BK2-2	0.26	0.53	0.31	0.35	↑
BK3-1	8.26	3.44	3.48	8.85	5.71
(L)BK3-2	51.32	370	109.7	81.23	↑
TK2-1	1.07	0.59	0.60	0.87	1.81
(L)TK2-2	3.13	1.31	1.30	2.19	↑
TK3-1	14.71	15.49	13.56	21.42	↑
(L)TK3-2	29.73	75.48	28.2	119.95	↑
BR2-1	4.62	4.52	3.59	4.52	59.3
BR2-2	0.27	0.25	0.23	0.27	0.28
BR2-3	1.58	1.49	1.51	1.55	2.93
BR3-1	10.93	13.22	9.14	10.59	↑
BR3-2	0.35	0.32	0.33	0.30	0.5
BR3-3	4.12	4.39	4.31	3.93	66.89
BR4-1	21.98	29.92	18.85	21.05	↑
BR4-2	0.43	0.46	0.43	0.39	0.83
BR4-3	8.76	10.09	9.93	8.61	↑
BRP-1	6.76	5.64	6.53	7.8	173
BRP-2	0.22	0.16	0.19	0.20	0.38
BRP-3	1.17	0.78	0.89	1.14	3.77
C-1	0.34	0.25	0.29	0.30	0.23
(L)C-2	2.74	0.74	0.82	4.03	2.1
(L)C-3	11.97	2.11	2.42	18.8	21.97
(L)C-4	13.14	1.93	2.03	27.09	↑
C-REF-1	0.27	0.19	0.22	0.25	0.16
(L)C-REF-2	0.97	>83	>83	>83	↑
(L)C-REF-3	0.48	>139	>139	>139	↑
ISORT	0.2	0.11	0.11	0.23	0.21
RW16	0.02	0.02	0.02	0.02	↑
RW32	0.03	0.03	0.03	0.03	↑
RW64	0.05	0.05	0.05	0.05	↑
RWP	0.01	0.01	0.01	0.01	0.01
SIS-1	0.01	0.01	0.01	0.01	0.01
SIS-2	0.07	0.02	0.03	0.06	0.02
LC	0.12	0.10	0.09	0.09	0.09

Table 3.4: Verification time (in secs) results for different versions of the simplification heuristic vs. no simplification (↑ means that the program ran out of memory, $> x$ means that the execution did not terminate in x seconds, and (L) indicates that the specification has been verified for a liveness property)

for most of the examples, the verifier ran out of memory.

Problem Instance	S1-S2-S3	S1	S2	S3	None
BK2-1	7.8	7.8	7.7	7.7	80
(L)BK2-2	7.9	8.8	8.5	7.8	↑
BK3-1	19.6	21.5	20.3	19.5	30
(L)BK3-2	34.7	323	77.5	34.9	↑
TK2-1	10.2	10.3	10.4	10.2	17.4
(L)TK2-2	13.8	13.8	13.5	13.4	↑
TK3-1	28	58	43.3	35.2	↑
(L)TK3-2	29	173	62	60	↑
BR2-1	17.7	21	17.6	17.6	197
BR2-2	8.8	9	9	8.8	9.3
BR2-3	12.8	13.8	13.8	12.8	20
BR3-1	26.6	34.5	26.4	26.4	↑
BR3-2	9.5	9.7	9.8	9.5	10
BR3-3	18.1	21	21	18	205
BR4-1	38.5	52.7	38.1	38.1	↑
BR4-2	10.1	10.5	10.5	10.1	13.3
BR4-3	25.9	32.2	32	25.8	↑
BRP-1	24	24	24	24	228
BRP-2	9.4	9.3	9.3	9.3	10.3
BRP-3	13.5	13.4	13.4	13.4	19.5
C-1	11.3	11.2	11.2	11.3	11.5
(L)C-2	14.8	13.6	13.1	15.6	24.8
(L)C-3	29.8	22.2	21.1	32.8	161
(L)C-4	24.6	19.3	19.3	36.9	↑
C-REF-1	11	11	11	11	11.1
(L)C-REF-2	11.4	>60	>60	>60	↑
(L)C-REF-3	11.6	>181	>181	>181	↑
ISORT	8.5	8.5	8.4	8.4	8.7
RW16	8.1	8.1	8.1	8.1	↑
RW32	10.8	10.8	10.8	10.8	↑
RW64	20.6	20.6	20.6	20.6	↑
RWP	9	9	9	9	9
SIS-1	7.5	7.5	7.5	7.5	7.7
SIS-2	19.4	19.4	19.4	19.4	23.8
LC	7.9	8	8	7.9	8.4

Table 3.5: Verification memory (in Mbytes) results for different versions of the simplification heuristic vs. no simplification (↑ means that the program ran out of memory, $> x$ means that the execution did not terminate in x seconds, and (L) indicates that the specification has been verified for a liveness property)

The results in Table 3.6 show that combining the subsumption check and the disjunction computations with the pre-condition computation speeds up the verification. There are two reasons for the speedup: 1) Since disjuncts that are computed as the result of the pre-condition computation are not included in the resulting composite formula if they are subsumed by the result from the previous iteration, the resulting composite formula has a smaller size. 2) Only the disjuncts that are the result of the pre-condition computation in the current iteration are checked for subsumption relation against the resulting composite formula from the previous step. Average speedup for this heuristic is 29%.

Verification times with and without masking the integer pre-condition computation by boolean satisfiability check are given in Table 3.6. Results show that masking the integer pre-condition computation speeds up the verification and the speedup becomes higher for the specifications where the temporal property to be checked is a liveness property. The reason may be that the liveness properties involve two fixpoint computations: one for EG and one for EF[2]. Additionally, a fixpoint iteration for EG involves a pre-condition computation followed by conjunction operation whereas a fixpoint iteration for EF involves a pre-condition computation followed by a disjunction operation. Since the conjunction causes a quadratic increase (whereas the disjunction causes a linear increase) in the composite formula size, the results of the EG fixpoint iterations are likely to grow faster. Average speedup for masking heuristic is 12%.

Verification times for the two different versions of the subsumption check algorithm are given in Table 3.6. In most of the experiments the subsumption check

[2]Note that the verifier computes the negation of a liveness property $AGAFp$ and computes the fixpoint for $EFEG(\neg p)$. Then it checks satisfiability of $I \wedge EFEG(\neg p)$. Similarly, for an invariant property AGp it computes the fixpoint for the negation of AGp and checks satisfiability of $I \wedge EF(\neg p)$

Problem	−Subsumption		−PreDisjunction		−Mask		All-Heuristics	
Instance	T	M	T	M	T	M	T	M
BK2-1	0.26	10.9	0.32	7.8	0.24	8.7	0.21	7.8
(L)BK2-2	0.34	9.4	0.3	7.9	0.41	8	0.26	7.9
BK3-1	20.99	268	20.37	16.1	8.34	28.8	8.26	19.6
(L)BK3-2	54	69	50.04	34.7	57.34	36.6	51.32	34.7
TK2-1	1.16	18.3	1.22	9.6	1.14	13.2	1.07	10.2
(L)TK2-2	3.31	23.2	3.26	12.3	3.35	19.7	3.13	13.8
TK3-1	18.87	159	29.14	20.1	14.76	37.4	14.71	28
(L)TK3-2	36.47	204	66.53	34.2	32.67	49.9	29.73	29
BR2-1	8.03	122	6.32	15.7	4.65	21.4	4.62	17.7
BR2-2	0.2	10.1	0.37	8.3	0.24	9	0.27	8.8
BR2-3	1.69	34.4	2	11.1	1.63	14.5	1.58	12.8
BR3-1	21.75	299	11.53	18.8	11.07	31.1	10.93	26.6
BR3-2	0.21	11.2	0.46	8.6	0.36	9.7	0.35	9.5
BR3-3	6.03	105	4.15	13.7	4.12	20.5	4.12	18.1
BR4-1	43.77	554	20.62	22.8	22.21	44.4	21.98	38.5
BR4-2	0.27	12.3	0.59	8.9	0.44	10.4	0.43	10.1
BR4-3	16.94	264	8.38	16.7	8.87	29.1	8.76	25.9
BRP-1	4.64	31.9	6.48	18.8	6.83	25.3	6.76	24
BRP-2	0.25	10.4	0.44	9.4	0.24	9.2	0.22	9.4
BRP-3	1.59	23.5	5.11	21.8	1.27	14	1.17	13.5
C-1	0.37	13.4	0.86	10	0.38	13	0.34	11.3
(L)C-2	4.92	53.9	4.88	15.4	3.43	29.3	2.74	14.8
(L)C-3	21.61	169	26.45	35.4	13.93	91.1	11.97	29.8
(L)C-4	15.65	96.2	13.38	22.1	25.64	49.7	13.14	24.6
C-REF-1	0.28	11.9	0.56	10.1	0.33	12.7	0.27	11
(L)C-REF-2	1.17	21.2	1.93	11.2	1.26	14.3	0.97	11.4
(L)C-REF-3	0.59	18.2	0.50	11.6	1.01	11.8	0.48	11.6
ISORT	0.2	9.1	0.26	8.6	0.3	10.8	0.2	8.5
RW16	0.02	8.1	0.03	8.1	0.02	8.1	0.02	8.1
RW32	0.03	10.8	0.05	10.8	0.04	10.8	0.03	10.8
RW64	0.04	20.6	0.1	20.6	0.08	20.6	0.05	20.6
RWP	0.01	9	0.01	9	0.01	9	0.01	9
SIS-1	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.5
SIS-2	0.07	19.4	0.08	19.4	0.38	27.2	0.07	19.4
LC	0.13	8.8	0.17	7.6	0.12	8.5	0.12	7.9

Table 3.6: Verification time (T, in seconds) and memory (M, in Mbytes) results demonstrating the impact of different heuristics (− denotes exclusion of the specified heuristic and All-Heuristics denotes that all the heuristics are enabled)

algorithm with the heuristic performs better than the subsumption check algorithm shown in Fig. 3.5, which computes the complement operation at the composite formula level. The results demonstrate that processing composite atoms as needed and performing the complement operation at the composite atom level instead of composite formula level is more efficient. Average speedup for the efficient subsumption check heuristic is 10%.

3.5 Related Work

There have been other studies that combine different symbolic representations. In [23], Chan *et al.* present a technique in which (both linear and non-linear) constraints are mapped to BDD variables and a constraint solver is used during model checking computations (in conjunction with SMV) to prune the infeasible combinations of these constraints. Although this technique is capable of handling non-linear constraints, it is restricted to systems where the transitions are either *data-memoryless* (i.e., next state value of a data variable does not depend on its current state value), or *data-invariant* (i.e., data variables remain unchanged). Hence, even a transition that increments a variable (i.e., $x' = x + 1$) is ruled out. It is reported in [23] that this restriction is partly motivated by the semantics of RSML, and it allows modeling of a significant portion of TCAS II system.

In [8], a tool for checking inductive invariants on SCR specifications is described. This tool combines automata based representations for linear arithmetic constraints with BDDs. This approach is similar to our approach but it is specialized for inductive invariant checking. Another difference is, our tool uses polyhedral representations as opposed to automata based representations for linear arithmetic. However, because of the object-oriented design of our tool it should be easy to extend it with the automata-based linear constraint representations.

The Symbolic Analysis Laboratory provides a framework for combining different tools in the verification of concurrent systems [7]. The heart of the Symbolic Analysis Laboratory is a language for specifying concurrent systems in a compositional manner. Our Composite Symbolic Library is a low-level approach compared to the Symbolic Analysis Laboratory. We are combining different libraries at the symbolic representation level as opposed to developing a specification language to integrate different tools.

Techniques that are similar to our heuristics have been used in the literature. In [34], local subsumption test is used during the fixpoint computations to remove the redundant constrained facts. This is similar to our approach for preventing the increase in the size of the disjunctive composite representation during fixpoint computation by removing the redundant disjuncts. However, we use full subsumption test. Local subsumption test can also be used as a heuristic to test the convergence of fixpoint computations [34]. However, there can be cases where a fixpoint computation that uses the local subsumption test does not converge whereas the fixpoint computation that uses the full subsumption test converges.

Hytech, a tool for verification of hybrid systems, simplifies formulas using rewrite rules [3]. The approach used in [3] is for simplification of linear arithmetic formulas on real variables. Our work is different in two respects: 1) We use linear arithmetic formulas on integer variables. 2) Our heuristics are not for simplification of linear arithmetic formulas, this is handled by the constraint manipulator that we use [2]. Rather, our heuristics are for simplification of the composite formulas, which contain a mixture of boolean and integer variables. In Hytech boolean and enumerated variables (for example, control states) are eliminated by partitioning the state space [3].

In [62], a linear partitioning algorithm for convex polyhedra is used to efficiently test if a single convex polyhedron is subsumed by a disjunction of convex polyhedra. This approach is analogous to our subsumption check heuristic where the disjunction of convex polyhedra corresponds to our disjunctive composite representation and the single convex polyhedron corresponds to a single disjunct of a composite representation.

Chapter 4

Action Language Verifier

The Action Language Verifier is an automated verification tool for analyzing the Action Language specifications using model checking technique. Given a transition system $T = (S, I, R)$, 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 $I \Rightarrow [\![\phi]\!]$ holds, where $[\![\phi]\!]$ denotes the states that satisfy ϕ.

The main challenge regarding model checking is to alleviate the so called *state explosion* problem, which indicates the exponential growth of the state space with the increasing number of concurrent components and variables. Symbolic model checking [25] emerged as a partial solution to the state explosion problem by encoding the state space symbolically instead of explicitly. As a result it enabled verification of very large (even infinite) state systems. Symbolic model checking owes its success to compact representations such as Binary Decision Diagrams (BDDs) [12].

The Action Language Verifier 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 efficiency required for analyzing software specifications. The composite representation is implemented as an extensible framework as described in Chapter 3.

The Action Language Verifier can analyze infinite-state systems. CTL model checking for infinite-state systems is undecidable. However, the Action Language Verifier employs several heuristics for speeding or guaranteeing the termination of the analysis. These heuristics make the Action Language Verifier conservative, i.e., it does not generate any false positives, however, the analysis may be inconclusive.

This chapter is organized as follows. Section 4.1 presents the fixpoint characterization of CTL operators for non-total systems. Section 4.2 presents the heuristics for accelerating or guaranteeing the convergence of the fixpoint computations and the heuristics for efficient fixpoint computations. Section 4.3 presents the algorithms for implementing an abstraction technique, called *counting abstraction*, for the composite representation. Section 4.4 explains the counter-example generation algorithms. Section 4.5 presents the experimental results. Finally, Section 4.6 discusses the related work.

4.1 Fixpoint Computations

The Action Language Verifier is built on top of the Composite Symbolic Library. An Action Language specification is translated to a transition system (see Section 2.3) that is encoded symbolically by the Composite Symbolic Library and passed to the verification engine implemented by the class `TransSys` given in Figure 4.1.

The class `TransSys` implements a CTL model checker. The CTL operators have either least fixpoint or greatest fixpoint characterizations [25]. The class

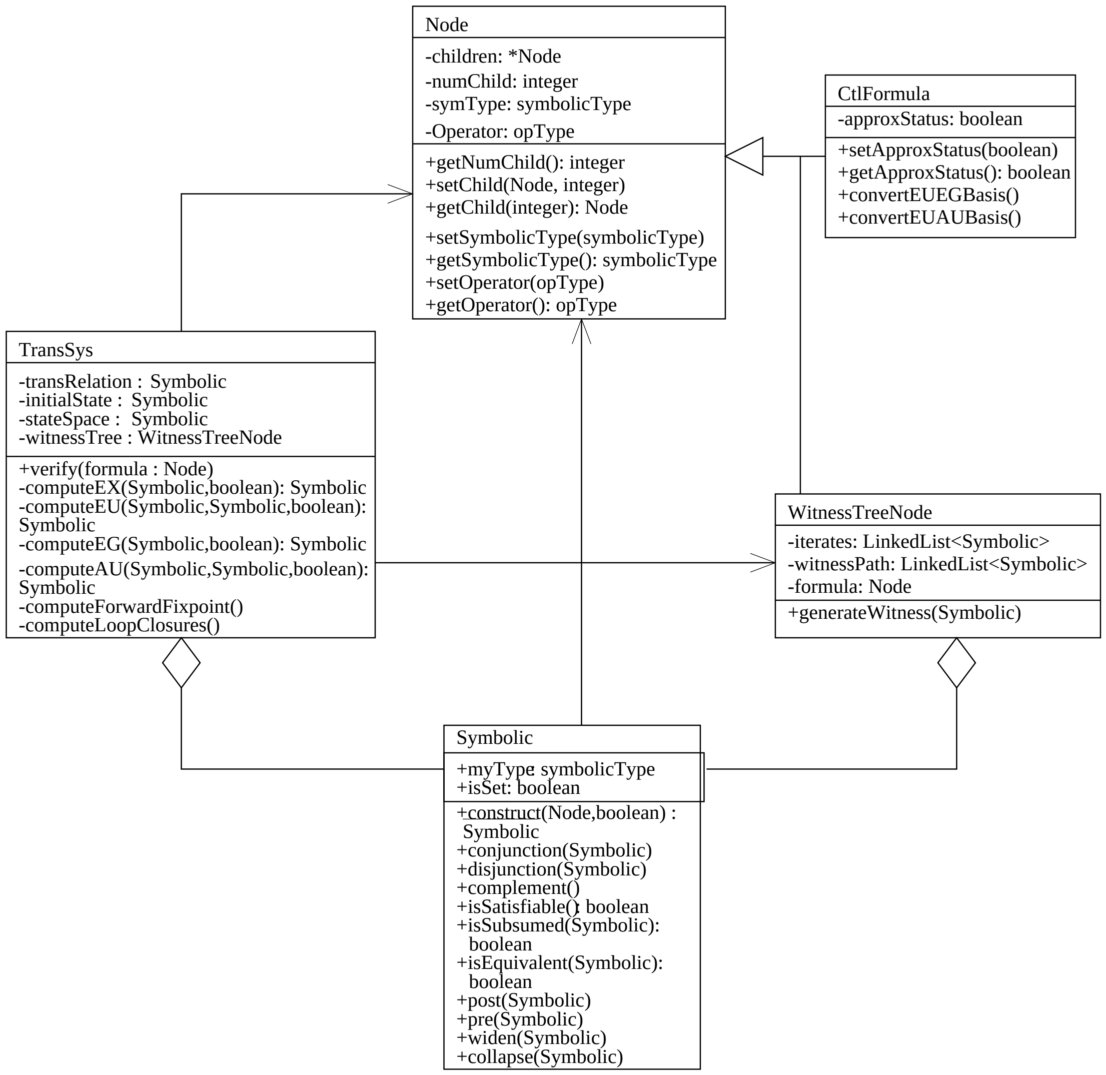

Node
-children: *Node
-numChild: integer
-symType: symbolicType
-Operator: opType
+getNumChild(): integer
+setChild(Node, integer)
+getChild(integer): Node
+setSymbolicType(symbolicType)
+getSymbolicType(): symbolicType
+setOperator(opType)
+getOperator(): opType
CtlFormula
-approxStatus: boolean
+setApproxStatus(boolean)
+getApproxStatus(): boolean
+convertEUEGBasis()
+convertEUAUBasis()
TransSys
-transRelation : Symbolic
-initialState : Symbolic
-stateSpace : Symbolic
-witnessTree : WitnessTreeNode
+verify(formula : Node)
-computeEX(Symbolic,boolean): Symbolic
-computeEU(Symbolic,Symbolic,boolean): Symbolic
-computeEG(Symbolic,boolean): Symbolic
-computeAU(Symbolic,Symbolic,boolean): Symbolic
-computeForwardFixpoint()
-computeLoopClosures()
WitnessTreeNode
-iterates: LinkedList<Symbolic>
-witnessPath: LinkedList<Symbolic>
-formula: Node
+generateWitness(Symbolic)
Symbolic
+myType: symbolicType
+isSet: boolean
+construct(Node,boolean) : Symbolic
+conjunction(Symbolic)
+disjunction(Symbolic)
+complement()
+isSatisfiable(): boolean
+isSubsumed(Symbolic): boolean
+isEquivalent(Symbolic): boolean
+post(Symbolic)
+pre(Symbolic)
+widen(Symbolic)
+collapse(Symbolic)

Figure 4.1: Class diagram of the composite model checker

`TransSys` implements the fixpoint algorithms by calling the method `pre` of the class `Symbolic` in order to compute the states that satisfy a given CTL formula, which is implemented by the class `CtlFormula` given in Figure 4.1.

The pre- and post-condition computations are among the basic operations in a symbolic model checker. Given states p and a transition relation R, the pre-condition $Pre(p, R)$ are all the states that can reach a state in p with a single transition in R (i.e., the predecessors of all the states in p). $Post(p, R)$ 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, as we have already mentioned in Section 3.2, can be computed as

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

Since the pre-condition computation distributes over both the disjunction and the conjunction, we are able to compute the pre-condition of a composite representation using `pre` methods of the basic symbolic representations.

The CTL temporal operators are generated from the basic temporal operators for paths:

- G p (Globally p) p holds in every state
- F p (Future p) p will hold in a future state
- p U q (p Until q) p will hold until q holds
- X p (Next p) p will hold in the next state

by adding one of the path quantifiers A (for all paths) or E (there exists a path) as a prefix. For example, AG p denotes that p is an invariant, AF p denotes that p is always eventually reached, and AG($p \Rightarrow$ AF q) denotes that p leads-to q (i.e., whenever p becomes true, eventually q will be reached).

Based on the equivalences among the CTL operators [25], one can show that $\{\mathrm{EX}, \mathrm{EG}, \mathrm{EU}\}$ forms a basis for CTL, i.e., all CTL formulas can be expressed using only these temporal operators. Similarly, another basis for CTL is $\{\mathrm{EX}, \mathrm{EU}, \mathrm{AU}\}$. 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.

The temporal operator EX corresponds to the pre-condition computation, i.e., EX $p \equiv \mathit{Pre}(p, R)$. AX can also be computed as AX $p \equiv \neg \mathit{Pre}(\neg p, R)$. The rest of the CTL operators can be computed as least and greatest fixpoints using EX and AX [25]

$$\begin{array}{lcl} p \ \mathrm{EU} \ q & \equiv & \mu x \ . \ q \ \vee \ (p \ \wedge \ \mathrm{EX} \ x) \\ p \ \mathrm{AU} \ q & \equiv & \mu x \ . \ q \ \vee \ (p \ \wedge \ \mathrm{AX} \ x) \\ \mathrm{EG} \ p & \equiv & \nu x \ . \ p \ \wedge \ \mathrm{EX} \ x \\ \mathrm{AG} \ p & \equiv & \nu x \ . \ p \ \wedge \ \mathrm{AX} \ x \end{array}$$

However, the above characterizations of AU and EG are not complete if we do not restrict the transition relation to be total. Since a non-total transition system can have states that do not have any next states, AX **false** will be satisfied in such states vacuously. Hence, those states will satisfy AF **false** too. This creates a problem, since we will have states 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 *AtLeastOne* denotes the states that have at least one successor.

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

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

where *None* denotes the states that have no successors (i.e., $\mathit{None} \equiv \neg \mathit{AtLeastOne}$). 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.

The Action Language Verifier iteratively computes the fixpoints for the temporal operators. Given a monotonic functional $\mathcal{F}$ [63]

$$\bigvee_{i=0}^{n} \mathcal{F}^i \ \mathbf{false} \ \ \Rightarrow \ \ \mu x \ . \ \mathcal{F} \ x \tag{4.1}$$

$$\nu x \ . \ \mathcal{F} \ x \ \ \Rightarrow \ \ \bigwedge_{i=0}^{n} \mathcal{F}^i \ \mathbf{true} \tag{4.2}$$

where $\mathcal{F}^i \ p$ denotes the application of $\mathcal{F}$ to p i consecutive times, and $\mathcal{F}^0$ corresponds to the identity relation. Using these properties we can compute the least fixpoints iteratively by starting with **false** and then applying the functional to get the result of the next iteration. Similarly, the greatest fixpoints can be computed starting from **true**. In an infinite state model checker convergence is not guaranteed. Although each iteration takes us closer to the fixpoint, we are not

```
1  EX($p$): composite formula
2  $p$: composite formula
3  **let** $R$ denote the transition relation
4  **return** $Pre(p, R)$
```

Figure 4.2: The algorithm for computing the states that satisfy EXp

guaranteed to reach it. However, if a fixpoint is reached we are sure that it is the least or the greatest fixpoint based on the type of the iteration.

```
1  EG($p$): composite formula
2  $p$, $s$, $sold$, $None$: composite formula
3  **let** $R$ denote the transition relation
4  **let** $None$ denote the states with no successors
5  $s \leftarrow p$
6  $sold \leftarrow false$
7  **while** $\neg isEquivalent(s, sold)$ **do**
8          $sold \leftarrow s$
9          $s \leftarrow (Pre(s, R) \vee None) \wedge sold$
10 **return** $s$
```

Figure 4.3: The algorithm for computing the states that satisfy EGp

Figures 4.2, 4.3, 4.4 and 4.5 show the algorithms for computing the states that satisfy the CTL formulas EXp, EGp, p EU q, and p AU q. The algorithm in Figure 4.2 computes EXp by simply computing the pre-condition computation on p. The algorithm in Figure 4.3 computes the set of states that satisfy EGp. It starts with

```
EU($p$, $q$): composite formula
$p$, $q$, $s$, $sold$: composite formula
$s \leftarrow q$
$sold \leftarrow false$
**let** $R$ denote the transition relation
**while** $\neg isEquivalent(s, sold)$ **do**
        $sold \leftarrow s$
        $s \leftarrow Pre(s, R) \wedge p \vee sold$
**return** $s$
```

Figure 4.4: The algorithm for computing the states that satisfy p EU q

```
AU($p$, $q$): composite formula
$p$, $q$, $s$, $sold$, $AtLeastOne$: composite formula
$s \leftarrow q$
$sold \leftarrow false$
**let** $R$ denote the transition relation
**let** $AtLeastOne$ denote the states with at least one successor
**while** $\neg isEquivalent(s, sold)$ **do**
        $sold \leftarrow s$
        $s \leftarrow \neg Pre(\neg s, R) \wedge p \wedge AtLeastOne \vee sold$
**return** $s$
```

Figure 4.5: The algorithm for computing the states that satisfy p AU q

the states that satisfy p. Then it iteratively restricts these states to the states that can reach the states that satisfy p in one step, in two steps, and so on using the pre-condition computation. The iteration stops when there is no change in the states computed so far and returns the result. The algorithm in Figure 4.4 computes p EU q. It starts with the states that satisfy q. Then it iteratively adds the states that satisfy p and can reach q in one step, in two steps, and so on to these states using the pre-condition computation. The iteration stops when there is no change in the states computed so far and returns the result. The algorithm in Figure 4.5 computes p AU q. It starts with the states that satisfy q. Then it adds states that satisfy p and cannot reach $\neg q$ in one step, in two steps, and so on to these states using the pre-condition computation. The iteration stops when there is no change in the states computed so far and returns the result.

4.2 Heuristics

If we cannot directly compute the states that a temporal property ϕ satisfies for a transition system $M = (S, I, R)$, we can try to generate a *lower bound* for ϕ, denoted ϕ^-, such that $\phi^- \Rightarrow \phi$. Then, if we determine that the initial states satisfy this lower bound (i.e., $I \Rightarrow \phi^-$), we have also shown that $I \Rightarrow \phi$, i.e., we proved that transition system M satisfies the property ϕ. However, if $I \not\Rightarrow \phi^-$, we cannot conclude anything, because it can be a *false negative*. In that case we can compute a lower bound for the negated property: $(\neg\phi)^-$. If $I \wedge (\neg\phi)^-$ is satisfiable, then we can generate a counter example, which would be a *true negative*. If both cases fail, i.e., both $I \not\Rightarrow \phi^-$ and $I \wedge (\neg\phi)^- \equiv false$, then the verifier cannot report a definite answer.

Since the Action Language Verifier computes the temporal formulas recursively starting from the innermost temporal operators, we have to compute an approximation to a formula by first computing the approximations for its sub formulas. 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 sub formulas. To compute a lower bound for a negated property like $p = \neg q$, we can compute an upper bound q^+ for the sub formula q where $q \Rightarrow q^+$, and then let $p^- \equiv \neg q^+$. Similarly we can compute an upper bound for p using a lower bound for q. Thus, we need algorithms to compute both lower and upper bounds of temporal formulas. In this section we explain the heuristics for accelerating the convergence of the fixpoint computations, which are truncated fixpoint calculations, the widening and the collapsing operators, the loop-closures, and the reachable states. We also propose two heuristics, which are marking and dependency heuristics, for avoiding the redundant computations during fixpoint computations. We have implemented these heuristics in the Action Language Verifier. We will explain these heuristics using the sample specification given in Figure 4.6, 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 till it becomes zero (actions `a1` and `a2`) and then the positive value is decremented till 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

```
module main()
 parameterized integer size;
 integer x,y;
 enumerated pc {a,b,c};
 restrict: size>0;
 initial: x=-size and y=size;
 a1: pc=a and x<0 and x'=x+1 and pc'=a;
 a2: pc=a and x=0 and pc'=b;
 a3: pc=b and y>0 and y'=y-1 and pc'=b;
 a4: pc=b and y=0 and pc'=c;
 a5: pc=c and x'=-size and y'=size and pc'=a;
 main: a1 | a2 | a3 | a4 | a5;
 spec: AG(x<=y)
endmodule
```

Figure 4.6: A sample Action Language specification.

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 4.6, truncating the fixpoint computation of $EF(x > y)$, given in Figure 4.7, after two iterations yields I_2, which is a lower approximation for the least fixpoint. The Action Language Verifier 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.

The Widening and Collapsing Operators For computing upper bounds for least-fixpoints we use the *widening* technique [28] generalized to the composite symbolic representation [15]. Let p and q denote two composite formulas such that $p \Rightarrow q$ and $\triangledown$ denote the widening operator. Then $p \triangledown q$ is defined as

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

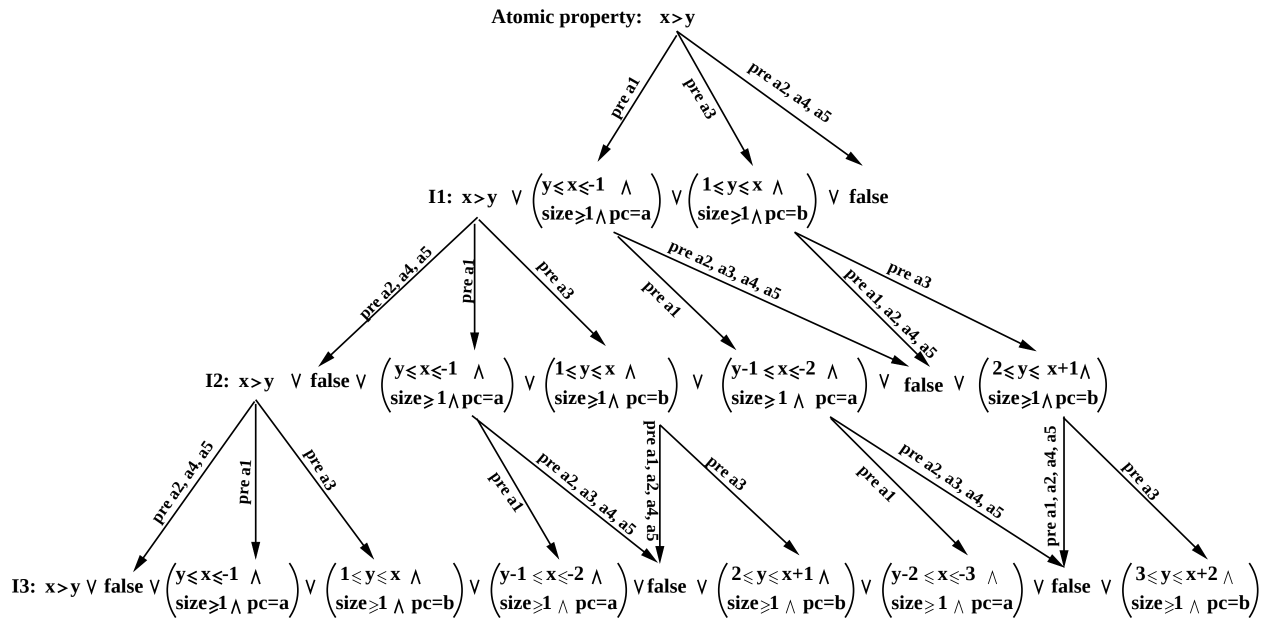

Atomic property: x>y
pre a1
pre a3
pre a2, a4, a5
I1: x>y ∨ (y≤x≤-1 ∧ size≥1∧pc=a) ∨ (1≤y≤x ∧ size≥1∧pc=b) ∨ false
pre a2, a4, a5
pre a1
pre a3
pre a2, a3, a4, a5
pre a1
pre a3
pre a1, a2, a4, a5
I2: x>y ∨ false ∨ (y≤x≤-1 ∧ size≥1∧pc=a) ∨ (1≤y≤x ∧ size≥1∧pc=b) ∨ (y-1 ≤x≤-2 ∧ size≥1 ∧ pc=a) ∨ false ∨ (2≤y≤ x+1∧ size≥1∧pc=b)
pre a2, a4, a5
pre a1
pre a3
pre a1
pre a2, a3, a4, a5
pre a1, a2, a4, a5
pre a3
pre a1
pre a2, a3, a4, a5
pre a1, a2, a4, a5
pre a3
I3: x>y ∨ false ∨ (y≤x≤-1 ∧ size≥1 ∧ pc=a) ∨ (1≤y≤x ∧ size≥1 ∧ pc=b) ∨ (y-1 ≤x≤-2 ∧ size≥1 ∧ pc=a) ∨false ∨ (2≤y≤x+1 ∧ size≥1 ∧ pc=b) ∨ (y-2 ≤x≤-3 ∧ size≥ 1 ∧ pc=a) ∨ false ∨ (3≤y≤x+2 ∧ size≥1 ∧ pc=b)

Figure 4.7: The result of the first three iterations of computing $EF(x > y)$ naively for the Action Language specification given in Figure 4.6. 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 $1 \leq i \leq 5$.

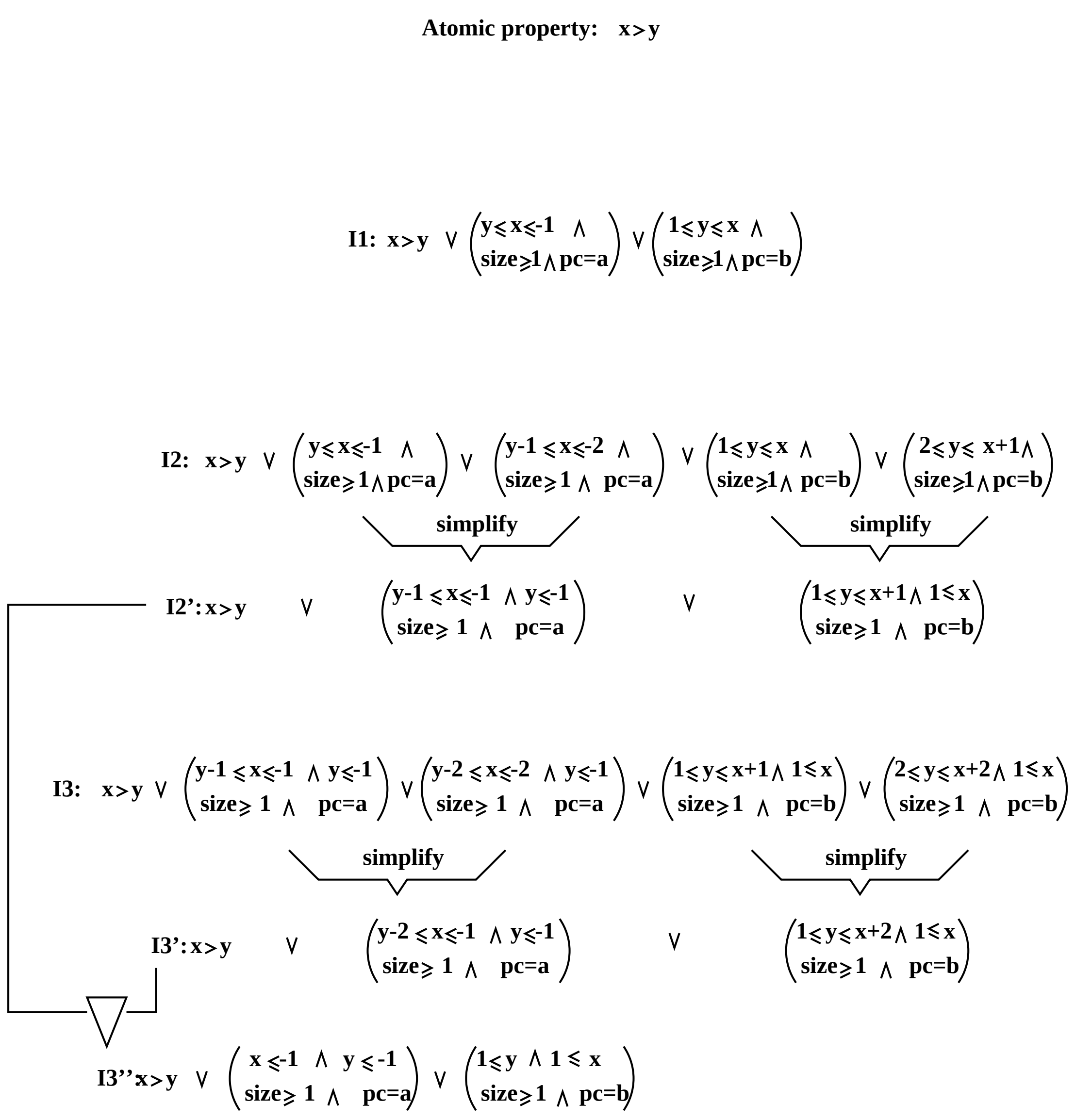

Atomic property: x>y
I1: x>y ∨ (y⩽x⩽-1 ∧ size⩾1∧pc=a) ∨ (1⩽y⩽x ∧ size⩾1∧pc=b)
I2: x>y ∨ (y⩽x⩽-1 ∧ size⩾1∧pc=a) ∨ (y-1 ⩽x⩽-2 ∧ size⩾1 ∧ pc=a) ∨ (1⩽y⩽x ∧ size⩾1∧ pc=b) ∨ (2⩽y⩽ x+1∧ size⩾1∧pc=b)
simplify
simplify
I2':x>y ∨ (y-1 ⩽x⩽-1 ∧ y⩽-1 size⩾ 1 ∧ pc=a) ∨ (1⩽y⩽x+1∧ 1⩽x size⩾1 ∧ pc=b)
I3: x>y ∨ (y-1 ⩽x⩽-1 ∧ y⩽-1 size⩾ 1 ∧ pc=a) ∨ (y-2 ⩽x⩽-2 ∧ y⩽-1 size⩾ 1 ∧ pc=a) ∨ (1⩽y⩽x+1∧ 1⩽x size⩾1 ∧ pc=b) ∨ (2⩽y⩽x+2∧ 1⩽x size⩾1 ∧ pc=b)
simplify
simplify
I3':x>y ∨ (y-2 ⩽x⩽-1 ∧ y⩽-1 size⩾ 1 ∧ pc=a) ∨ (1⩽y⩽x+2∧ 1⩽x size⩾1 ∧ pc=b)
I3''x>y ∨ (x ⩽-1 ∧ y ⩽-1 size⩾ 1 ∧ pc=a) ∨ (1⩽y ∧ 1 ⩽ x size⩾1 ∧ pc=b)

Figure 4.8: The results of the first three iterations of computing $EF(x > y)$ for the Action Language specification given in Figure 4.6. 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.

where n, T, p_{it}, and $\bigtriangledown_t$ denote the number of the set of basic symbolic representations, a symbolic representation of type t, and the type-specific widening operator, respectively. All widening operators satisfy the following constraint: $p \vee q \Rightarrow p \bigtriangledown q$. Intuitively, $\bigtriangledown$ operator guesses the direction of the growth in the results of the fixpoint iterations, and extends the results of the successive iterations in that direction. The least fixpoint computations are modified so that at each iteration the result p_i is set to $p_{i-1} \bigtriangledown p_i$. For the polyhedral representation we use the widening operator defined in [16] for Presburger arithmetic constraints by generalizing the convex widening operator in [29]. The basic idea is to find pairs of polyhedra p and q such that $p \Rightarrow q$ and set $p \bigtriangledown_{int} q$ to conjunction of constraints in p that are also satisfied by q. Intuitively, if a constraint of p is not satisfied by q this means that the results of the iterations are increasing in that direction. By removing that constraint we extend the result of the iterations in the direction of growth as much as possible without violating other constraints. For the boolean representation the widening operator ($\bigtriangledown_{bool}$) is simply the disjunction operation. Figure 4.8 shows an example of the widening operator, which is used for the computation of $EF(x > y)$ fixpoint for the specification in Figure 4.6. In this example the Action Language Verifier was directed to start applying the widening operation after the second iteration. For computing $I'_2 \bigtriangledown I'_3$, the Action Language Verifier 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 $x \leq -1 \wedge y \leq -1 \wedge size \geq 1$. Note that the conjunct $y-2 \leq x$ is not satisfied by $y-1 \leq x \leq -1 \wedge y \leq -1 \wedge size \geq 1$, so it does not appear in the result.

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 the $p \bar{\bigtriangledown} q$ is defined as

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

where n, T, p_{it}, and $\bar{\bigtriangledown}_t$ denote the number of the set of basic symbolic representations, a symbolic representation of type t, and type-specific collapsing operator, respectively. The collapsing operators satisfy the following: $p \bar{\bigtriangledown} q \Rightarrow p \wedge q$. Intuitively, $\bar{\bigtriangledown}$ operator finds which parts of the result of the fixpoint iterations are decreasing 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 $p_{i-1} \bar{\bigtriangledown} p_i$. 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 boolean representation the collapsing operator ($\bar{\bigtriangledown}_{bool}$) is the conjunction operator. Figure 4.9 shows an example of the collapsing operator, which is used for the computation of $EG(x \leq y)$ fixpoint for the specification in Figure 4.6. In this example the Action Language Verifier was directed to start applying the collapsing operation after the first iteration. For computing $I_1 \bar{\bigtriangledown} I_2$, the Action Language

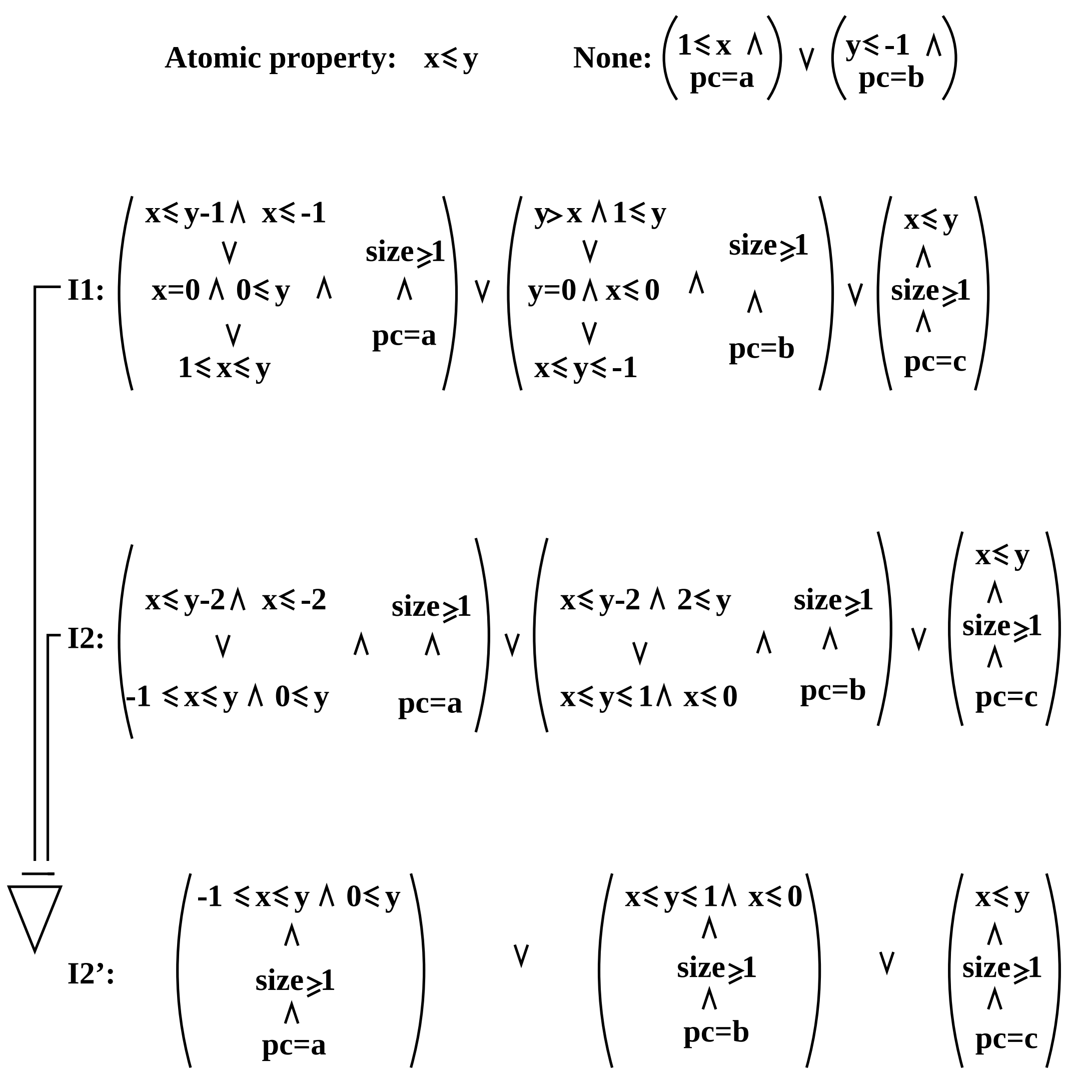

Atomic property: x⩽y
None: (1⩽x ∧ pc=a) ∨ (y⩽-1 ∧ pc=b)
I1: ((x⩽y-1 ∧ x⩽-1 ∨ x=0 ∧ 0⩽y ∨ 1⩽x⩽y) ∧ size⩾1 ∧ pc=a) ∨ ((y>x ∧ 1⩽y ∨ y=0 ∧ x⩽0 ∨ x⩽y⩽-1) ∧ size⩾1 ∧ pc=b) ∨ (x⩽y ∧ size⩾1 ∧ pc=c)
I2: ((x⩽y-2 ∧ x⩽-2 ∨ -1 ⩽x⩽y ∧ 0⩽y) ∧ size⩾1 ∧ pc=a) ∨ ((x⩽y-2 ∧ 2⩽y ∨ x⩽y⩽1 ∧ x⩽0) ∧ size⩾1 ∧ pc=b) ∨ (x⩽y ∧ size⩾1 ∧ pc=c)
I2': (-1 ⩽x⩽y ∧ 0⩽y ∧ size⩾1 ∧ pc=a) ∨ (x⩽y⩽1 ∧ x⩽0 ∧ size⩾1 ∧ pc=b) ∨ (x⩽y ∧ size⩾1 ∧ pc=c)

Figure 4.9: The results of the first three iterations of computing $EG(x \leq y)$ for the Action Language specification given in Figure 4.6. *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.

Verifier 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., $(x \leq y - 1 \wedge x \leq -1 \ \vee \ x = 0 \wedge y \geq 0 \ \vee \ 1 \leq x \leq y \ \vee \ 1 \leq x \leq y) \wedge size \geq 1 \bar{\nabla}_{int} (x \leq y - 2 \wedge x \leq -2 \ \vee \ -1 \leq x \leq y \wedge 0 \leq y) \wedge size \geq 1$, yields $-1 \leq x \leq y \wedge y \geq 0 \wedge size \geq 1$. 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.

The 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

$$\forall s \Rightarrow S, Post(s, R) \Rightarrow Post(s, R') \Rightarrow Post(s, R^*)$$

(where R^* denotes the reflexive-transitive closure of R) to accelerate the fixpoint computations for temporal operators EF and EU [16].

To exploit this idea, given a transition relation R in the composite symbolic representation $R \equiv \bigvee_{i=1}^{n} \bigwedge_{t \in T} r_{it}$, the Action Language Verifier 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* denote 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 $\forall s \Rightarrow S, Post(s, r_{i,int}) \Rightarrow Post(s, r'_{i,int}) \Rightarrow Post(s, r^{*}_{i,int})$. 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 4.1 shows the transitions that correspond to the actions in Figure 4.6 for both cases with and without using the loop-closures heuristic. Note that when the loop-closures heuristic is used only the transitions that correspond to the actions `a1` and `a3` change, since they are the ones that model loops, i.e., they model iterative increment and decrement operations, respectively.

The Reachable States The fixpoint algorithms described thus far are *backward* 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 the initial states I imply the derived 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. We can accomplish this by altering the symbolic model checker to restrict its computations to states in RS^{+}. To

Action	−Loop-Closures	+Loop-Closures
a1	$pc = a \wedge x < 0 \wedge$ $x' = x + 1 \wedge pc' = a$	$(pc = a \wedge x < 0 \wedge x' = x + 1 \wedge pc' = a) \vee$ $(pc = a \wedge x + 1 \geq x' \geq 0 \wedge pc' = a)$
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 4.1: The transitions that correspond to the actions of the Action Language specification in Figure 4.6. − and + denote exclusion and inclusion, respectively.

generate the upper bound RS^+, we used the *Post* function. The (exact) reachable state-space of a transition system is the least fixpoint $RS \equiv \mu x \ . \ I \ \vee \ Post(x, R)$, and it can be computed using the techniques we previously developed for EU. Moreover, we can use the widening method to compute an upper bound for RS as well. After computing RS^+, we restrict the result of every operation in the model checker to RS^+.

The Marking Heuristic The states that satisfy EFϕ are characterized by the least fixpoint $\mu Z.\phi \vee Pre(Z)$. 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. As composite symbolic representation is a disjunctive representation and 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, i.e., recomputes 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

Iter.	Result	−Marking	+Marking	
		computed	computed	marked
0	ϕ	none	none	none
1	$\phi \vee Pre(\phi)$	$Pre(\phi)$	$Pre(\phi)$	ϕ
2	$\phi \vee Pre(\phi)$ $\vee Pre(Pre(\phi))$	$Pre(\phi)$, $Pre(Pre(\phi))$	$Pre(Pre(\phi))$	ϕ, $Pre(\phi)$
3	$\phi \vee Pre(\phi)$ $\vee Pre(Pre(\phi)) \vee$ $Pre(Pre(Pre(\phi)))$	$Pre(\phi)$, $Pre(Pre(\phi))$, $Pre(Pre(Pre(\phi)))$	$Pre(Pre(Pre(\phi)))$	ϕ, $Pre(\phi)$, $Pre(Pre(\phi))$

Table 4.2: The results of the first 4 iterations of computing EF $= \mu Z.\phi \vee Pre(Z)$ with and without the marking heuristic.$-$ and $+$ denote exclusion and inclusion, respectively.

kth iteration. Hence, at the kth iteration the pre-condition is computed only on the disjuncts that are not marked, i.e., disjuncts that were computed at the $k-1$st iteration. Table 4.2 shows the results of the first 4 iterations for computing EFϕ for both with and without the marking heuristic. At the kth iteration the fix-point 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 the Action Language Verifier runs in the approximate fixpoint computation mode. The Marking heuristic can also be used for computing the states satisfied by $p$EU$q$ and for computing the reachable states RS as they also have least fixpoint characterization in the form $\mu Z.\phi \vee F(Z)$.

Figure 4.10 shows the results of the first three iterations for computing $EF(x > y)$ using the Marking heuristic for the specification given in Figure 4.6. The disjuncts that are enclosed by rounded-corner boxes denote the marked ones. The Marking heuristic makes sure that the pre-condition computation on the disjuncts $x > y$, $y \leq x \leq -1 \wedge size \geq 1 \wedge pc = a$, and $y - 1 \leq x \leq 2 \wedge size \geq 1 \wedge pc = a$ is performed only in the first iteration, in the second iteration, and in the third

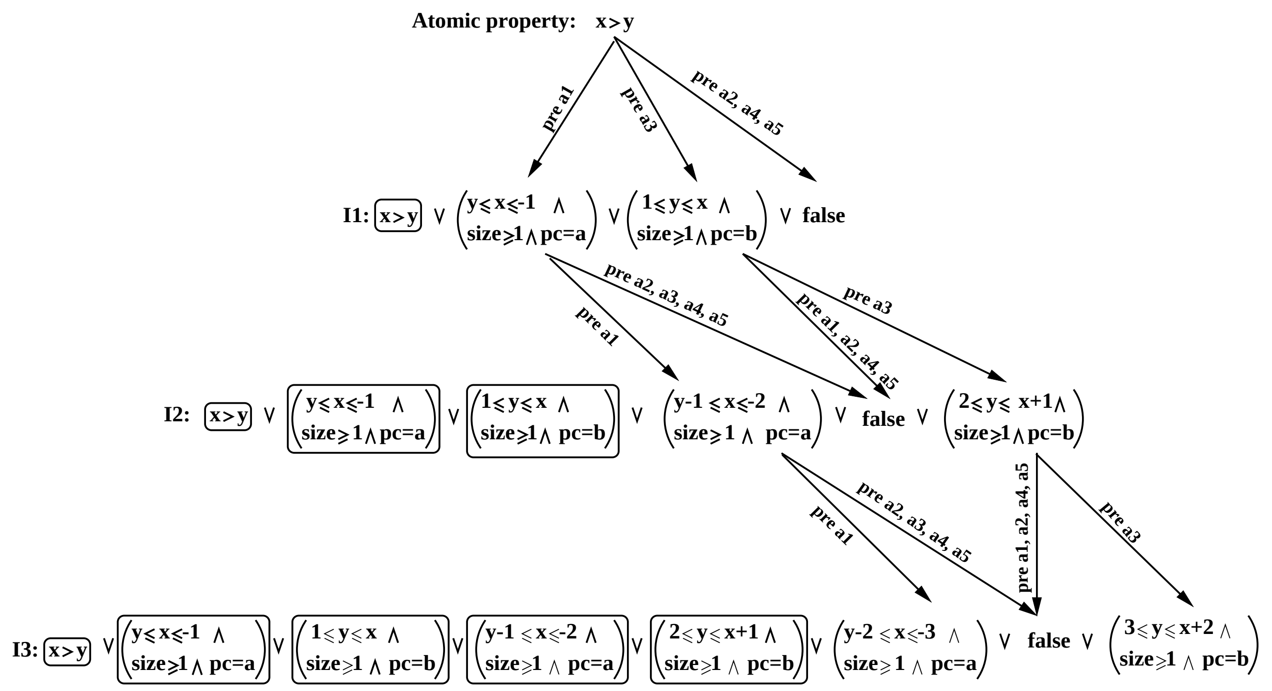

Atomic property: x>y
pre a1
pre a3
pre a2, a4, a5
I1: x>y ∨ (y≤x≤-1 ∧ size≥1 ∧ pc=a) ∨ (1≤y≤x ∧ size≥1 ∧ pc=b) ∨ false
pre a1
pre a2, a3, a4, a5
pre a1, a2, a4, a5
pre a3
I2: x>y ∨ (y≤x≤-1 ∧ size≥1 ∧ pc=a) ∨ (1≤y≤x ∧ size≥1 ∧ pc=b) ∨ (y-1 ≤x≤-2 ∧ size≥1 ∧ pc=a) ∨ false ∨ (2≤y≤ x+1 ∧ size≥1 ∧ pc=b)
pre a1
pre a2, a3, a4, a5
pre a1, a2, a4, a5
pre a3
I3: x>y ∨ (y≤x≤-1 ∧ size≥1 ∧ pc=a) ∨ (1≤y≤x ∧ size≥1 ∧ pc=b) ∨ (y-1 ≤x≤-2 ∧ size≥1 ∧ pc=a) ∨ (2≤y≤x+1 ∧ size≥1 ∧ pc=b) ∨ (y-2 ≤x≤-3 ∧ size≥ 1 ∧ pc=a) ∨ false ∨ (3≤y≤x+2 ∧ size≥1 ∧ pc=b)

Figure 4.10: The result of the first three iterations of computing $EF(x > y)$ using the Marking heuristic for the Action Language specification given in Figure 4.6. 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 is enclosed in a rounded-corner box.

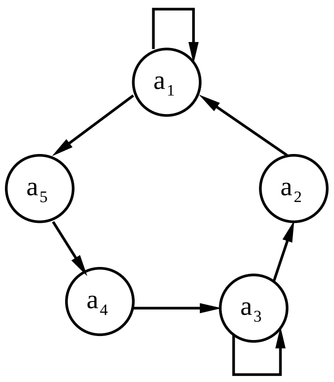


Figure 4.11: The dependency graph for the Action Language specification in Figure 4.6. A directed edge between the nodes a_i and a_j means that the constraints generated as a result of the pre-condition computation using the action a_i may enable the action a_j for computing the pre-condition computation.

iteration, 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 state with a marked state. Although this reduces the effectiveness of the marking heuristic, we think that this will improve the overall performance. The reason is that merging the matching constraints during simplification will create new opportunities for simplification.

The 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 when using the pre-condition computations. It can be adapted for the post-condition computation in a similar way.

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:

$$Pre(Pre(true, r_i), r_j) \neq false$$

The dependency graph, in a way, describes all the feasible interleaving of the atomic transitions. Figure 4.11 shows the dependency graph for the Action Language specification given in Figure 4.6. During the fixpoint computations, which 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 $s_2 = Pre(s_1, a_1)$ where s_1 and s_2 represent states, a_1 represents the atomic transition that corresponds to the action `a1` given in Figure 4.6. 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 it performs the computation, otherwise it skips the computation. For instance, the pre-condition computation on s_2 using the transition a_3 can be skipped as a_3 is not an element of the set $\{a_1, a_5\}$. Figure 4.12 shows the results of the first three iterations for computing $EF(x > y)$ for the specification given in Figure 4.6. 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

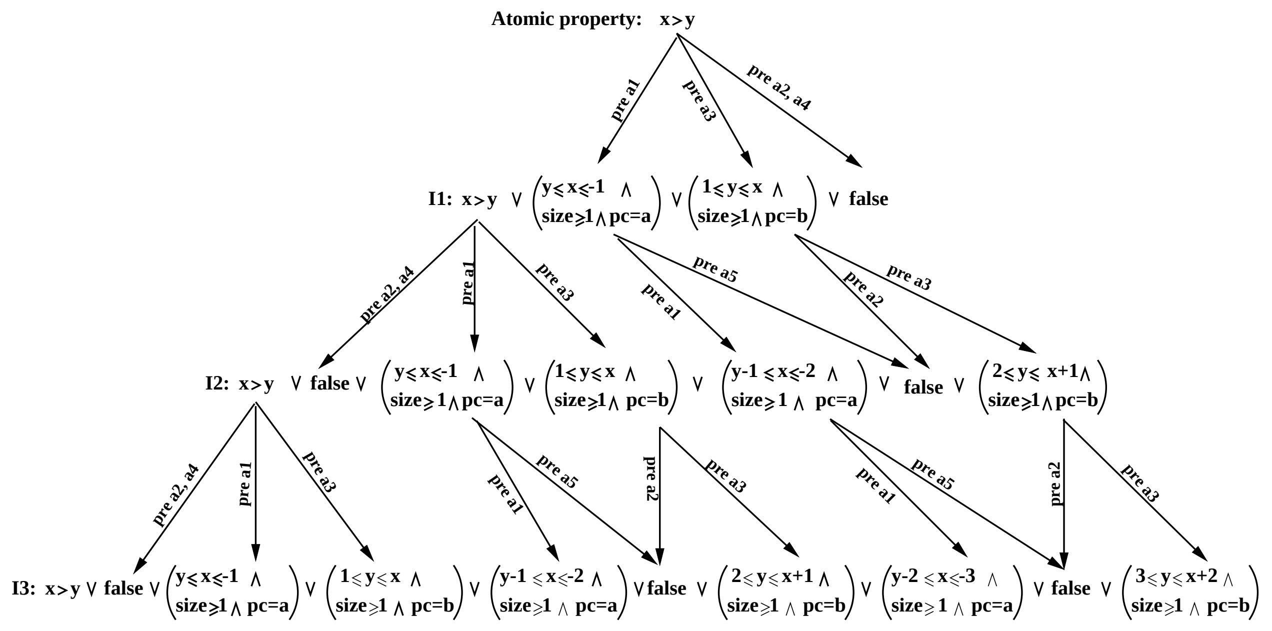

Atomic property: x>y
pre a1
pre a3
pre a2, a4
I1: x>y ∨ (y⩽x⩽-1 ∧ size⩾1∧pc=a) ∨ (1⩽y⩽x ∧ size⩾1∧pc=b) ∨ false
pre a2, a4
pre a1
pre a3
pre a1
pre a5
pre a2
pre a3
I2: x>y ∨ false ∨ (y⩽x⩽-1 ∧ size⩾1∧pc=a) ∨ (1⩽y⩽x ∧ size⩾1∧ pc=b) ∨ (y-1 ⩽x⩽-2 ∧ size⩾1 ∧ pc=a) ∨ false ∨ (2⩽y⩽ x+1∧ size⩾1∧pc=b)
pre a2, a4
pre a1
pre a3
pre a1
pre a5
pre a2
pre a3
pre a1
pre a5
pre a2
pre a3
I3: x>y ∨ false ∨ (y⩽x⩽-1 ∧ size⩾1∧ pc=a) ∨ (1⩽y⩽x ∧ size⩾1 ∧ pc=b) ∨ (y-1 ⩽x⩽-2 ∧ size⩾1 ∧ pc=a) ∨false ∨ (2⩽y⩽x+1∧ size⩾1 ∧ pc=b) ∨ (y-2 ⩽x⩽-3 ∧ size⩾1 ∧ pc=a) ∨ false ∨ (3⩽y⩽x+2 ∧ size⩾1 ∧ pc=b)

Figure 4.12: The results of the first three iterations of computing $EF(x > y)$ using the Dependency heuristic for the Action Language specification given in Figure 4.6. 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 $1 \leq i \leq 5$.

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 $y \leq x \leq -1 \land size \geq 1 \land pc = a$ 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, the Action Language Verifier uses this information during the simplification of the results of the fixpoint iterations (see Section 3.3). Two disjuncts are compared for equivalence during the simplification phase only if their enable backward sets are the same.

Comparison of the Marking and the Dependency Heuristics If we compare Figures 4.10 and 4.12 with Figure 4.7, 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 Dependence 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 the Action Language Verifier is computing a least fixpoint.

4.3 Parameterized Verification

In this section we will present the adaptation of an automated abstraction technique called *counting abstraction* [31] to the parameterized verification of specifications in the Action Language. Using counting abstraction one 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. Counting abstraction preserves the CTL properties that do not involve the local states of the processes [31].

For this abstraction technique to work we need the local states of the submodules to be finite. For example, if a submodule has a local variable that is an unbounded integer, we cannot directly use the counting abstraction. Note that, in the general case, 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 is the Cartesian product of the domains of the local variables of that submodule. In the Action Language one can specify a module instantiation to be parameterized by appending the '*' character, 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 4.3 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

	`Departing`	`Departing*`
S	$\text{pc} = \text{parked} \lor \text{pc} = \text{depFlow} \lor \text{pc} = \text{takeOff}$	$\text{parkedC} \geq 0 \land \text{depFlowC} \geq 0 \land \text{takeOffC} \geq 0 \land \text{parkedC} + \text{depFlowC} + \text{takeOffC} = C$
I	$\text{pc} = \text{parked}$	$\text{parkedC} = \text{C} \land \text{depFlowC} = 0 \land \text{takeOffC} = 0$
R		
r_1	$\text{pc} = \text{parked} \land \text{numRW16L} = 0 \land \text{numC3} + \text{numC4} + \text{numC5} + \text{numC6} + \text{numC7} + \text{numC8} = 0 \land \text{pc}' = \text{takeOff} \land \text{numRW16L}' = \text{numRW16L} + 1$	$\text{parkedC} > 0 \land \text{numRW16L} = 0 \land \text{numC3} + \text{numC4} + \text{numC5} + \text{numC6} + \text{numC7} + \text{numC8} = 0 \land \text{takeOffC}' = \text{takeOffC} + 1 \land \text{numRW16L}' = \text{numRW16L} + 1 \land \text{parkedC}' = \text{parkedC} - 1 \land \text{depFlowC}' = \text{depFlowC}$
r_2	$\text{pc} = \text{takeOff} \land \text{pc}' = \text{depFlow} \land \text{numRW16L}' = \text{numRW16L} - 1$	$\text{takeOffC} > 0 \land \text{depFlowC}' = \text{depFlowC} + 1 \land \text{numRW16L}' = \text{numRW16L} - 1 \land \text{takeOffC}' = \text{takeOffC} - 1 \land \text{parkedC}' = \text{parkedC}$

Table 4.3: Transition system information for an instantiation of `Departing` module in Figure 2.2 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.

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`).

The algorithm for parameterization of a module of an Action Language specification using counting abstraction is given in Figure 4.13. The algorithm accepts

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

Figure 4.13: The algorithm for applying counting abstraction to a module m of an Action Language specification.

a module name, m, as the input and generates $L+1$ number of auxiliary integer variables, where L denotes the size of the local state space of m. One of the auxiliary variables is a parameterized constant. It is used to denote the arbitrary number of processes. Each of the remaining L auxiliary variables denote the number of processes in a local state. We call the latter the *counters*. We distinguish two counters that correspond to two different local states by assigning an integer value 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 4.13 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. It changes the initial formula by calling the

```
COUNTABS_STATES($m$,$f$) : composite formula
$m$: module name
$f$, $l$, $s$, $resultDis$, $result$: composite formula
$index$: integer
let $V_s$ be the list of all the boolean variables other than $m$'s local boolean variables
let $V_l$ be the list of $m$'s local boolean variables
$result \leftarrow false$
for each composite atom $d = \bigwedge_{t \in T} d_t$ of $f$ do
    $resultDis \leftarrow false$
    for each minterm $e$ of $d_{bool}$ do
        $l \leftarrow \exists\ V_s.e$
        $s \leftarrow \exists\ V_l.e$
        let $index$ denote an integer value that uniquely represents $l$
        $resultDis \leftarrow resultDis \ \vee\ (s \ \wedge\ counter_{index} > 0 \ \wedge\ \bigwedge_{0 \leq i < L, i \neq index} counter_i = 0)$
    $result \leftarrow result \ \vee\ \bigwedge_{t \in T, t \neq bool} d_t \wedge\ resultDis$
return $result$
```

Figure 4.14: The algorithm for applying counting abstraction to a formula f.

algorithm COUNTABS_STATES given in Figure 4.14, whereas it changes the transition formula of each action by calling the algorithm COUNTABS_TRANSITIONS given in Figure 4.15.

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,

[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.

```
COUNTABS_TRANSITIONS($m$,$a$): composite formula
$m$: module name
$a$: action name
$index$:integer
$l$, $s$, $result$: composite formula
$result \leftarrow false$
let $V_s$ denote the list of all boolean variables other than $m$'s local boolean variables
let $V_{snext}$ denote the list of all next state boolean variables other than $m$'s local next state boolean variables
let $V_l$ denote the list of $m$'s local boolean variables
let $V_{lnext}$ denote the list of $m$'s local next state boolean variables
let $Act(a) = \bigwedge_{t \in T} r_t$
for each minterm $e$ of $r_{bool}$ do
    $l_d \leftarrow \exists\ V_{lnext}.\exists\ V_{snext}.\exists\ V_s.e$
    $l_r \leftarrow \exists\ V_l.\exists\ V_{snext}.\exists\ V_s.e$
    $s \leftarrow \exists\ V_{lnext}.\exists\ V_l.e$
    let $index_d$ denote an integer value that uniquely represents $l_d$
    let $index_r$ denote an integer value that uniquely represents $l_r$
    if $index_d = index_r$ then
        $resultAbs \leftarrow counter_{index_d} > 0\ \wedge\ counter'_{index_d} = counter_{index_d}$
    else
        $resultAbs \leftarrow counter_{index_d} > 0\ \wedge\ counter'_{index_d} = counter_{index_d} - 1\ \wedge$
        $counter'_{index_r} = counter_{index_r} + 1$
    for each $0 \leq i < L \wedge i \neq index_d \wedge i \neq index_r$ do
        $resultAbs \leftarrow resultAbs \wedge counter'_i = counter_i$
    $result \leftarrow result\ \vee\ resultAbs\ \wedge\ s$
return $\bigwedge_{t \in T, t \neq bool} r_t\ \wedge\ result$
```

Figure 4.15: The algorithm for applying counting abstraction to an action a of module m.

the algorithm extracts a local state l and generates a constraint that states that the counter that corresponds to l is greater than zero and the other counters are all equal to zero. It conjuncts this constraint with the non-local part of the minterm. It performs this for all minterms and gets the disjunction of the resulting constraints. Finally, it conjuncts the generated constraint with the non-boolean parts of the composite atom.

The algorithm COUNTABS_TRANSITIONS takes 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 at which the local transition is enabled by existentially quantifying the next state variables 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. 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. 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. In both cases the constraint states that the other counters keep their value in the next state. It conjuncts the constraint with the non-local part of the minterm. It performs this for all minterms and gets the disjunction of the resulting constraints. Finally, it conjuncts the generated constraint with the non-boolean parts of the action a.

4.4 Counter-Example Generation

An important feature of model checkers is their ability to generate counter-example behaviors. the Action Language Verifier 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 $\neg\phi$. A *witness* for a CTL property is a path that starts from the initial states and demonstrates that the property is satisfied [25]. 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.

When asked to generate a counter-example, the Action Language Verifier negates the input property ϕ, converts it to $\{\mathrm{EX}, \mathrm{EG}, \mathrm{EU}\}$ 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 $s \Rightarrow I \cap \neg\phi$. 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 path for $\neg\phi$ 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 4.16 shows the algorithm for computing the witness path for a given CTL formula in {EX,EU,EG} basis. The algorithm first reverses the results of fixpoint iterations for the input CTL formula. The reason is that fixpoint com-

```
GEN_WITNESS($s$, $\phi$)
$s$, $s_1$: Symbolic, $\phi$: CtlFormula, $witness$, $iterR$: List
**if** $\phi.isAtomic()$ **then return**
$iterR \leftarrow \phi.iterates.reverse()$
$witness.add(s)$
**case** $\phi = \mathrm{EX}\phi_1$: $s_1 \leftarrow s.post(R)$ $witness.add(s_1.conjunction(iterR.get(1)))$
  GEN_WITNESS($witness.get(1)$,$\phi_1$)
**case** $\phi = \phi_1 \mathrm{EU} \phi_2$:
  **if** $\neg s.isSubsumed(iterR.getLast())$ **then**
    **while** $0 < i < iterR.size() \wedge$
    $\neg witness.get(i).isSubsumed(iterR.getLast())$ **do**
          $s_1 \leftarrow witness.get(i-1).post(R)$ $witness.add(s_1.conjunction(iterR.get(i)))$
  **for** $0 \leq i < witness.size()$ **do**
      **if** $i = witness.size() - 1$ **then**
        GEN_WITNESS($witness.get(i)$,$\phi_2$)
      **else** GEN_WITNESS($witness.get(i)$,$\phi_1$)
**case** $\phi = \mathrm{EG}\phi_1$: $cycleNotReached \leftarrow true$
  **for** $1 \leq i \leq MAX\_ITER$ **and** $cycleNotReached$ **do**
      $s_1 \leftarrow witness.get(i-1).post(R)$
      **if** $\neg s_1.isSatisfiable()$ **then break** $witness.add(s_1.conjunction(iterR.get(i)))$
        **for** $0 \leq j < i$ **do**
            **if** $witness.get(i).isSubsumed(witness.get(j))$ **then**
              $cycleNotReached \leftarrow false$ **break**
  **for** $0 \leq i < witness.size()$ **do** GEN_WITNESS($witness.get(i)$,$\phi_1$)
```

Figure 4.16: The algorithm for generating witness for the CTL formula ϕ in $\{$EX,EU,EG$\}$ basis starting from state s.

putation is propagated backwards starting from the inner CTL formula, whereas witness computation is propagated 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 $\mathit{Post}(s, R)$ and conjuncts the result with the next one of the reversed results of the fixpoint iterations, which denotes the state that satisfy ϕ. The conjunction is saved as the next state in the witness path and used to generate a witness path for ϕ.

To generate a witness for the property EU(ϕ_1, ϕ_2) starting from a state s that satisfies EU(ϕ_1, ϕ_2), the algorithm starts with the result of the last iteration of the fixpoint. Since that corresponds to the states that satisfy EU(ϕ_1, ϕ_2), it is guaranteed that s is in it. If s satisfies ϕ_2 then the algorithm stops. Otherwise, $\mathit{Post}(s, R)$ is conjuncted 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 picks 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.

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, $\mathit{Post}(s, R)$ is computed and conjuncted with the states that satisfy EGϕ. The algorithm picks a state that satisfy 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 we print the 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 our search. When it reaches that bound it prints the path computed so far as the prefix of a witness path.

The algorithm in Figure 4.16 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, the Action Language Verifier 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 $\{\mathrm{EX}, \mathrm{EG}, \mathrm{EU}\}$ basis for the CTL and we try to verify the property $\mathrm{AG}p$. 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 $\mathrm{AG}p$. If we can show that $I \Rightarrow (\mathrm{AG}p)^-$ then we are done. However, if $I \not\Rightarrow (\mathrm{AG}p)^-$ we cannot use our computations for $(EF\neg p)^+$ to generate a counter-example. Since $(EF\neg p)^+$ 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 $(EF\neg p)^-$ we are sure that it is a valid counter-example. Because of this issue the Action Language Verifier works in two phases; 1) the verification phase and 2) the falsification phase. In the verification phase it does not try to generate a counter-example. If it cannot prove the property in the verification phase then it recomputes the fixpoints and tries to generate a counter-example.

Problem	Transition Relation Size					
Instance	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 4.4: 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 the arbitrary number of departing and arriving airplanes.

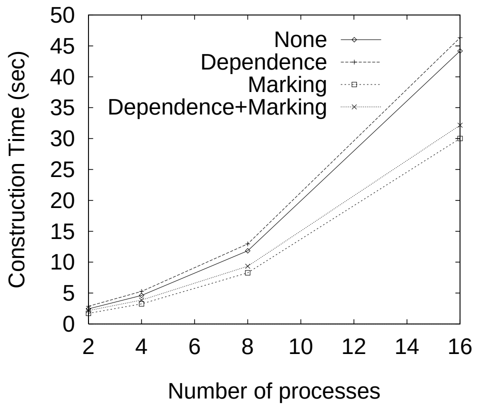
As explained above, these phases will use different types of approximations if approximate fixpoints are being used. Either of these cases can be skipped by the user using the input flags of the the Action Language Verifier.

4.5 Experiments

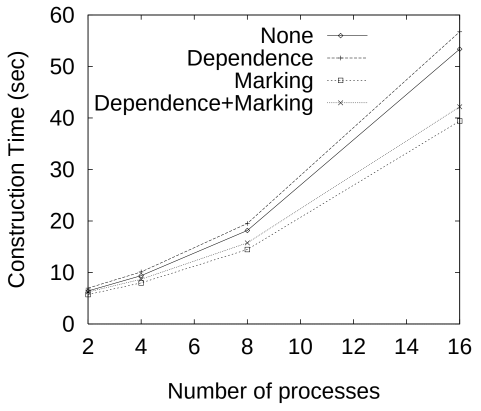
This section presents the experimental results that are obtained using the Airport Ground Control case study.

Table 4.4 shows the size of the transition systems used in the experiments. 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 arbitrary (parameterized.

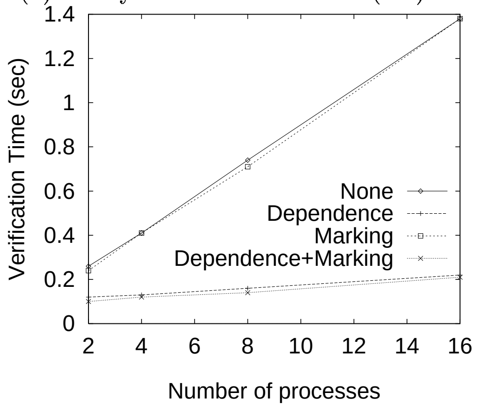
Figure 4.17 shows performance of the Dependence and the Marking heuristics for the verification of the safety and liveness properties in terms of construction time, verification time, and memory usage. The verifier used the widening and the approximate reachable states heuristics for this experiment. Results show that



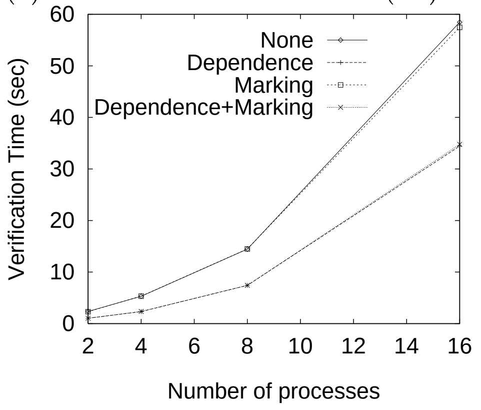
(a) Safety Construction Time (sec)



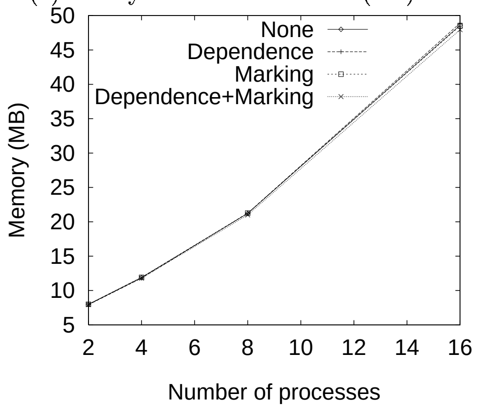
(b) Liveness Construction Time (sec)



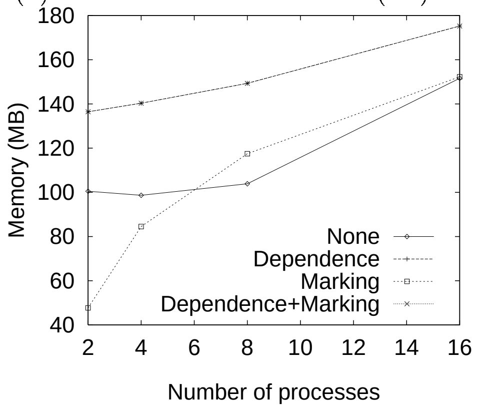
(c) Safety Verification Time (sec)



(d) Liveness Verification Time (sec)



(e) Safety Memory (MB)



(f) Liveness Memory (MB)

Figure 4.17: Comparison of the Dependence and the Marking Heuristics

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 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 3.3) 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.

Our experimental results indicate that the verification results for the fully parameterized case, where both the number of the arriving and the number of the departing airplanes are arbitrary, become 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. This is due to the symmetry reduction obtained as a result of the counting abstraction.

We have changed the specification in Figure 2.2 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 2.1. The Action Language Verifier checked this

```
  UNABLE TO VERIFY !EF(numRW16L=0 and numC3+numC4+numC5+numC6+numC7
+numC8 > 0 and EX(!numRW16L=0)
  THE FORMULA EF(numRW16L=0 and numC3+numC4+numC5+numC6+numC7+
numC8>0 and EX(!numRW16L=0)
  IS WITNESSED BY THE FOLLOWING PATH
  PATH STATE 0 (Departing.pc=parkedD and Arriving.pc=arFlow and
numRW16R=0 and numRW16L=0 and numC3=0 and numC4=0 and numC5=0 and
numC6=0 and numC7=0 and numC8=0 and numB2A=0 and numB7A=0 and
numB9A=0 and numB10A=0 and numB11A=0)
  PATH STATE 1 (Departing.pc=parkedD and Arriving.pc=touchDown and
numRW16R=1 and numRW16L=0 and numC3=0 and numC4=0 and numC5=0 and
numC6=0 and numC7=0 and numC8=0 and numB2A=0 and numB7A=0 and
numB9A=0 and numB10A=0 and numB11A=0)
  PATH STATE 2 (Departing.pc=parkedD and Arriving.pc=taxiTo16LC3
and numRW16R=0 and numRW16L=0 and numC3=1 and numC4=0 and numC5=0
and numC6=0 and numC7=0 and numC8=0 and numB2A=0 and numB7A=0 and
numB9A=0 and numB10A=0 and numB11A=0)
  THE FORMULA numRW16L=0 and numC3+numC4+numC5+numC6+numC7+numC8>0
and EX(!numRW16L=0)
  IS SATISFIED BY THE STATE
(Departing.pc=parkedD and Arriving.pc=taxiTo16LC3 and numRW16R=0
and numRW16L=0 and numC3=1 and numC4=0 and numC5=0 and numC6=0 and
numC7=0 and numC8=0 and numB2A=0 and numB7A=0 and numB9A=0 and
numB10A=0 and numB11A=0)
  AND THE FORMULA EX(!numRW16L=0)
  IS WITNESSED BY THE FOLLOWING PATH
  PATH STATE 0 (Departing.pc=parkedD and Arriving.pc=taxiTo16LC3
and numRW16R=0 and numRW16L=0 and numC3=1 and numC4=0 and numC5=0
and numC6=0 and numC7=0 and numC8=0 and numB2A=0 and numB7A=0 and
numB9A=0 and numB10A=0 and numB11A=0)
  PATH STATE 1 (Departing.pc=takeOff and Arriving.pc=taxiTo16LC3
and numRW16R=0 and numRW16L=1 and numC3=1 and numC4=0 and numC5=0
and numC6=0 and numC7=0 and numC8=0 and numB2A=0 and numB7A=0 and
numB9A=0 and numB10A=0 and numB11A=0)
```

Figure 4.18: A counter-example path as output by the Action Language Verifier.

erroneous specification for property P3, which corresponds to rule 3, and falsified the property by providing a counter-example path as shown in Figure 4.18. The counter-example path given in Figure 4.18 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 given in Figure 4.18, 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 (`PATH STATE 0`). Then the arriving airplane lands and transitions to the `touchDown` state (`PATH STATE 1`). Having landed, the arriving airplane selects exit C3 and starts taxiing on it by transitioning into `taxiTo16LC3` state (`PATH STATE 2`). During this transition note that 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 (`PATH STATE 2` of the sub-witness path 1). Then the departing airplane starts the takeoff and transitions into the `takeOff` state (`PATH STATE 1`). This violates the property since exit C3 is occupied while the departing airplane is taking off.

4.6 Related Work

SMV [58] and NuSMV [24] are symbolic model checkers based on BDDs. SPIN [51] is an explicit-state model checker. These model checkers are bound to the verification of finite state systems. To analyze infinite-state spaces (defined on the integer domain) with these model checkers, one needs to first generate an abstraction of the original specification. On the other hand, with the Action Language Verifier the only thing a user is required to do is to select among various heuristic options for infinite-state verification and feed these selections as flags to the Action Language Verifier. Then, the Action Language Verifier applies the selected abstraction heuristics automatically. Additionally, the counter-example paths that are generated by the Action Language Verifier 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.

Hytech [49] is an infinite-state symbolic model checker for hybrid systems. It uses linear arithmetic representation on reals as the underlying symbolic representation. It uses heuristics for infinite-state verification similar to those used in the Action Language Verifier. Although the Action Language Verifier currently uses linear arithmetic representation on integers, by integrating the linear arithmetic representation for reals to the Composite Symbolic Library it can easily be extended to model check software specifications with reals.

Murϕ [38] is an explicit-state model checker that uses symmetry reduction techniques [52] to combat the state space explosion problem. It reduces the state space significantly for systems with identical components and verifies some data-independent systems with unbounded state space. The counting abstraction technique used by the Action Language Verifier also makes use of symmetries in systems with identical components. However, the Action Language Verifier uses these symmetries to generate a parameterized version of the system instead of reducing the state space. Murϕ [53] analyzes parameterized systems using a less precise version of the counting abstraction technique, hence, regarding parameterized systems Murϕ can only verify ACTL properties, whereas the Action Language Verifier can verify full CTL properties.

[32] uses counting abstraction to verify safety properties of parameterized cache coherence protocols. Our work differs from [32] in the following ways: 1) The Action Language Verifier can *automatically* translate an Action Language specification to its parameterized version, whereas [32] 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 [32] 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 [32] is specialized for the verification of safety properties of parameterized cache coherence protocols.

Chapter 5

Automated Concurrency Controller Synthesis

Writing a concurrent program is an error prone task. A concurrent programmer has to keep track of not only the possible values of the variables of the program, but also the states of its concurrent processes. Failing to use concurrency constructs such as semaphores or monitors correctly results in the run-time errors such as deadlocks and the violation of the safety properties. The conventional validation techniques such as testing become ineffective for concurrent programs, since the state space of a concurrent program increases exponentially both with the number of variables and the number of concurrent processes in it.

A monitor is a programming language construct introduced to ease the difficult task of concurrent programming [50]. Structured programming languages help programmers in keeping track of the states of the program variables by providing abstractions such as procedures and associated scoping rules to localize variable access. Monitors are a similar mechanism for structuring concurrent programs, they provide scoping rules for concurrency. Even though monitors provide a better abstraction for concurrent programming compared to other constructs such as semaphores, they are still error prone. Coordinating wait and signal operations on several condition variables among multiple processes can be very challenging even for the implementation of simple algorithms.

In this chapter we propose a new approach for developing reliable concurrent programs. The first step of our approach starts with a formal specification of the concurrency control component of the program rather than its implementation. We use monitors as the underlying concurrency control primitive and specify them in the Action Language. The second step of our approach uses the Action Language Verifier to check temporal correctness properties of the monitor specifications. The third step of our approach consists of automated code generation for synthesizing monitors from the Action Language specifications. Our goal is not generating complete programs, rather, we are proposing a modular approach for generating the concurrency control component of a program that manipulates shared resources.

We use a case study on airport ground traffic control (see Section 2.1) to show the effectiveness and the scalability of our technique. We have presented the verification results for this case study in Section 4.5. The Action Language Verifier verifies all the correctness properties of the specification for this case study and our code generation tool automatically generates an optimized (in terms of the context switch overhead that would be incurred in a multi threaded application) Java class.

This chapter is organized as follows. Section 5.1 explains monitors as a concurrency control mechanism, the way they are implemented in Java, and how one can use the Action Language to specify behavior of monitors. Section 5.2 presents the automated and property-preserving synthesis of Java code for monitors that are specified in the Action Language and verified using the Action Language Verifier. Finally, Section 5.3 discusses the related work.

5.1 Concurrency Control with Monitors

A monitor is a synchronization primitive that is used to control access to a shared resource by multiple concurrent processes. A monitor consists of a set of variables and procedures with the following rules: 1) The variables in a monitor can only be accessed through the procedures of the monitor. 2) No two processes can execute procedures of the monitor at the same time. We can view the second rule as the monitor having a mutual exclusion lock. Only the process that has the monitor lock can be active in the monitor. Any process that calls a monitor procedure has to acquire the monitor lock before executing the procedure and release it after it exits. This synchronization is provided implicitly by the monitor semantics, hence, the programmer does not have to explicitly write the acquire lock and release lock operations.

Monitors provide additional synchronization operations among processes based on condition variables. Two operations on condition variables are defined: *wait* and *signal*. A process that performs a wait operation on a condition variable sleeps and releases the monitor lock. It can only be awakened by a signal operation on the same condition variable. A waiting process that has been awakened has to re-acquire the monitor lock before it resumes operation. If there are no waiting processes, then signal operation is ignored (and forgotten, i.e., it does not affect processes that execute a wait later on). Wait and signal operations can be implemented using one wait queue per condition variable. When a process executes the wait operation on a condition variable it enters the corresponding wait queue. A signal operation on a condition variable removes one process from the corresponding wait queue and resumes its operation (after re-acquiring the monitor lock). In *signal and continue* semantics for the signal operation, the signaling process keeps the monitor lock until it exits or waits. Different semantics and additional operations have also been used for signaling such as *signal and wait* semantics and *signalAll* operation [4].

Typically condition variables are used to execute a set of statements only after a guard condition becomes true. To achieve this, a condition variable is created that corresponds to the guard condition. The process that will execute the guarded statements tests the guard condition and calls the wait on the corresponding condition variable if the guard condition is false. Each process that executes a statement that can change the truth value of the guard condition signals this to the processes that are waiting on the corresponding condition variable.

The state of a monitor is represented by its variables. The set of states that are safe for a monitor can be expressed as a monitor invariant [4]. The monitor invariant is expected to hold when no process is accessing the monitor (i.e., it is not guaranteed to hold when a process is active within a monitor procedure).

5.1.1 Monitors in Java

Java is an object-oriented programming language that supports concurrent programming via threads and monitors. Each Java object has a mutual exclusion lock. A monitor in Java is implemented using the object locks and the `synchronized` keyword. A block of statements can be declared to be synchronized using the lock of an object `o` as `synchronized(o) { ... }`. This block can only be executed after the lock for the object `o` is acquired. Methods can also be declared to be `synchronized`, which is equivalent to enclosing the method within a synchronized block using the object `this`, i.e., `synchronized(this) { ... }`. A monitor object in Java is created by declaring a class with private variables that correspond to the shared variables of the monitor. Then each monitor procedure is declared as a synchronized method to meet the mutual exclusion requirement.

Wait and signal operations are implemented as `wait` and `notify` methods in Java. However, in Java, each object has only one wait queue. This means that when there is a notify call, any waiting process in the monitor can wake up. If there is more than one condition that processes can be waiting for, awakened processes have to recheck the conditions that they have been waiting for. Note that, if a process that was waiting for a different condition is awakened, then the notify call is lost. This can be prevented by using the `notifyAll` method, which wakes up all the waiting processes.

Using a single wait queue and `notifyAll` method one can safely implement monitors in Java. However, such an implementation will not be very efficient. To get better efficiency, one can use other objects (declared as members of the monitor class) as condition variables together with the synchronized blocks on those objects. Since each object has a lock and an associated wait queue, this makes it possible to put processes waiting on different conditions to different queues. However, this implies that there will be more than one lock used in the monitor. (In addition to the monitor lock there will be one lock per condition variable.) The use of multiple locks in Java monitor classes is prone to deadlocks and errors [56].

5.1.2 Specifying Monitors in the Action Language

Although monitors provide a higher level of abstraction for concurrent programs compared to mechanisms such as semaphores, they can still be tedious and difficult to implement. We argue that the Action Language can be used to specify monitors in a higher level of abstraction. Monitor specifications in the Action Language do not rely on condition variables. Since in the Action Language an action is executed only when its guard evaluates to true, we do not need conditional waits.

Figure 2.2 (in Section 2.2) shows the monitor specification for the Airport Ground Traffic Control (see Section 2.1) without specifying the details about the implementation of the monitor. It is a high level specification compared to a monitor implementation, in the sense that, it does not introduce condition variables and waiting and signaling operations, which are error prone.

We give a general template for specifying monitors in the Action Language in Figure 5.1. It consists of a main module m and a list of submodules $m_1, \ldots, m_n$. The variables of the main module (denoted $\textsc{var}(m)$) define the shared variables of the monitor specification. Currently, available variable types in the Action Language are boolean, enumerated and integer. This restriction comes from the symbolic manipulation capabilities of the Action Language Verifier (which can be extended as we discuss in [69]). We also allow the declaration of parameterized constants. For example, a declaration such as `parameterized integer size` would mean that `size` is an unspecified integer constant, i.e., when a specification with such a constant is verified it is verified for all possible values of `size`.

Each submodule m_i corresponds to a process type, i.e., each instantiation of a submodule corresponds to a process. Each submodule m_i has a set of local

```
module $m()$
    integer $i_1$, $i_2$, ...; boolean $b_1$, $b_2$, ...; enumerated $e_1$, { $val_1$, $val_2$, ... };
     ...
    parameterized integer $p_1$, $p_2$, ...;
    restrict: $restrictCondition$;
    initial: $initialCondition$;
    module $m_1()$
        integer ...
        boolean ...
        enumerated ...
        restrict: ...
        initial: ....
        $a_1$: ...;
        $a_2$: ...;
        ...
        $a_{n_1}$: ...;
        $m_1$: $a_1$ | $a_2$ | ... | $a_{n_1}$;
    endmodule
    ...
    module $m_n()$ ... endmodule
    $m$: $m_1()$ | $m_1()$ | ... | $m_2()$ | $m_2()$ ...
    spec: $monitorInvariant$
endmodule
```

Figure 5.1: A Monitor Template in the Action Language

variables ($\textsc{var}(m_i)$) and atomic actions ($\textsc{act}(m_i)$). Note that, in a monitor specification our goal is to model only the behavior of a process that is relevant to the properties of the monitor. Therefore, local variables $\textsc{var}(m_i)$ of a submodule should only include the variables that are relevant to the correctness of the monitor. The transition relation of a submodule is defined as the asynchronous execution of its atomic actions.

5.2 Synthesis of Monitors

In the monitor specification given in Figure 2.2, the shared variables such as `numRW16R` and `numC3` represent the resources that will be shared among multiple processes. Submodules `Arriving` and `Departing` specify the type of processes that will share these resources. Our goal is to generate a monitor class in Java from monitor specifications such as the one given in Figure 2.2. First, we will declare the shared variables of the monitor specification (for example, `numRW16R` and `numC3` in Figure 2.2) as private fields of the monitor class. Hence, these variables will only be accessible to the methods of the monitor class.

We will not try to automatically generate code for the threads that will use the monitor. This would go against the modularization principle provided by the monitors. Rather, we will leave the assumption that the threads behave according to their specification as a proof obligation. In general, a submodule in a monitor specification (Figure 5.1) should specify the most general behavior of the corresponding thread, or, equivalently, it should specify the minimum requirements for the corresponding thread for the monitor to execute correctly. Since the specifications about the local behavior of the threads are generally straightforward (such as an `Arriving` process should not execute `exitRW3` action before executing `reqLand`) we think that it would not be too difficult for the concurrent programmer to take the responsibility for meeting these specifications.

We will generate a monitor method corresponding to each action of each submodule in the monitor specification. Consider the action:

```
  exitRW3: pc=touchDown and numC3=0 and numC3'=numC3+1 and
  numRW16R'=numRW16R-1 and pc'=taxiTo16L3;
```

We are not interested in the expressions on local variables `pc` of the submodules `Arriving` and `Departing`. As we discussed above, we are only generating code for the monitor class that only has access to the shared variables. For the action `exitRW3` removing the expressions on the local variables leaves us with the expression

```
  exitRW3: numC3=0 and numC3'=numC3+1 and numRW16R'=numRW16R-1;
```

To implement this action as a monitor method we first have to check the guard condition `numC3=0` and then update `numRW16R and numC3`. However, if the guard condition does not hold, we should wait until a process signals that the condition might have changed. A straightforward translation of this action to a monitor method would be

```
public synchronized void exitRW3() {
  while (!(numC3==0))
    wait();
  numC3=numC3+1;
  numRW16R=numRW16R-1;
  notifyAll();
}
```

The reason we call the `notifyAll` method at the end is to wakeup processes that might be waiting on a condition related to variable `numRW16R` or `numC3`, which have just been updated by this action. Also note that the `wait` method is inside a while loop to make sure that the guard still holds after the thread wakes up. In the above translation, we used the `synchronized` keyword to establish atomicity. Note that atomicity in Java is established only with respect to other methods or blocks that are also synchronized. So for this approach to work we have to make sure that shared variables are not modified by any part of the program that is not synchronized. We can establish this by declaring shared variables as private variables in the monitor class and making sure that all the methods of the monitor class are synchronized.

Using this straightforward approach, we can translate a monitor specification (based on the template given in Figure 5.1) to a Java monitor class using the following rules: 1) Create a monitor class with a private variable for each shared variable of the specification. 2) For each action in each submodule, create a synchronized method in the monitor class. 3) In the method for action a start with a while loop that checks if $d_s(a)$ is true and waits if it is not. Then, put a set of assignments to update the variables according to the constraint in $r_s(a)$. After the assignments, call `notifyAll` method and exit. We will call this translation *single-lock implementation* of the monitor since it uses only *this* lock of the monitor class.

5.2.1 Specific Notification Pattern

The *single-lock implementation* described above is correct but it is inefficient [20, 59]. If we implement the airport ground traffic control monitor using the above scheme an `exitRW3` action would awaken all the airplane threads that are sleeping. However, the departing airplane threads should be awakened only when the number of airplanes on the runway 16L or one of the taxiways in C3-C8 changes (when one of the variables `numRW16L`, `numC3`, `numC4`, `numC5`, `numC6`, `numC7`, and `numC8` become zero) and they do not need to be awakened on an update to status of the runway 16R (when `numRW16R` is updated) or on entrance of an airplane into the taxiway C3 (when `numC3` is incremented). Using different condition variables for each guard condition improves the performance by awakening only related threads and eliminating the overhead incurred by the context switch for threads that do not need to be awakened. In [20] using separate objects to wait and signal for separate conditions is described as a Java design pattern called *specific notification pattern*.

Figure 5.2 shows a fragment of the Java monitor that is automatically generated by our code generator from the Action Language specification of the airport ground traffic control monitor given in Figure 2.2 using the specific notification pattern. The method for action `exitRW3` calls `Guard_exitRW3` method in a while loop till it returns true. If it returns false it waits on the condition variable `exitRW3Cond`. Any action that can change the guard for `exitRW3` from false to true notifies the threads that are waiting on condition variable `exitRW3Cond` using `exitRW3Cond`. If the guard `(numC3==0)` is true then `Guard_exitRW3` method decrements the number of airplanes using the runway 16R (`numRW16R=numRW16R-1`) and increments the number of airplanes using the taxiway C3 atomically and returns true. Since executing `exitRW3` can only change the action `reqLand`'s guard from false to true, only threads that are waiting on the condition variable `reqLandCond` are notified before method `exitRW3` returns.

The action `leave` does not have a guard, i.e., its execution does not depend on the state of the shared variables of the monitor. Hence, the method for the action `leave` does not need to wait to decrement the number of airplanes on the runway 16L (`numRW16L=numRW16L-1`). After updating `numRW16L`, however, it notifies the

```
public class Airport{
  private int numC3;
  private int numRW16L;
  private int numRW16R;
  private Object exitRW3Cond;
  private Object reqTakeOffCond;
  private Object reqLandCond;
  private Object crossRW3Cond;
  . . .
  public Airport(){
    numC3=0;
    numRW16L=0;
    numRW16R=0;
    . . .
    exitRW3Cond=new Object();
    reqTakeOffCond=new Object();
    reqLandCond=new Object();
    crossRW3Cond=new Object();
    . . .
  }
  private synchronized
    boolean Guard_reqLand(){
    if (numRW16R==0){
       numRW16R=numRW16R+1;
       return true;
    }
    else return false;
  }
  public void reqLand(){
    synchronized(reqLandCond){
     while(!Guard_reqLand())
      try{ reqLandCond.wait();}
      catch(      }
       InterruptedExceptione){}
    }
  }
```

```
   private synchronized
     boolean Guard_exitRW3(){
     if (numC3==0){
        numC3=numC3+1;
        numRW16R=numRW16R-1;
        return true;
     } else return false;
   }

  public void exitRW3(){
    synchronized(exitRW3Cond){
     while(!Guard_exitRW3())
      try{ exitRW3Cond.wait();}
      catch(
       InterruptedExceptione){}

   }
   synchronized(reqLandCond){
    reqLandCond.notify();
   }
  }
  public void leave(){
    synchronized(this){
      numRW16L=numRW16L-1;
     }
    synchronized(crossRW3Cond){
     crossRW3Cond.notify();
    }
    synchronized(reqTakeOffCond){
     reqTakeOffCond.notify();
    }
     // other notifications. . .

     . . .

 }
```

Figure 5.2: Airport Class Using the specific notification pattern

threads waiting on the condition variables `crossRW3Cond` and `reqTakeOffCond`.

We will give an algorithm for generating Java code from the monitor specifications in the Action Language using the specific notification pattern below. We will assume that each action expression is in the form:

$$\textsc{exp}(a) \equiv d_l(a) \ \wedge \ r_l(a) \ \wedge \ d_s(a) \ \wedge \ r_s(a)$$

where $d_l(a)$ is an expression on the unprimed local variables of module m_i ($\textsc{var}(m_i)$), $r_l(a)$ is an expression on the primed and the unprimed local variables of m_i, $d_s(a)$ is an expression on the unprimed shared variables ($\textsc{var}(m)$), and $r_s(a)$ is an expression on the primed and the unprimed shared variables. Since we are not interested in the local states of the processes, we will only use $d_s(a)$ and $r_s(a)$ in the code generation for the monitor methods. Let $guard_a$ denote a Java expression equivalent to $d_s(a)$. We will also assume that $r_s(a)$ can be written in the form $r_s(a) \equiv \bigwedge_{v \in \text{RVAR}(a)} v' = e_v$ where e_v is an expression on the domain variables in $\textsc{var}(m)$. Let $assign_a$ denote a set of assignments in Java that correspond to $r_s(a)$.

To use the specific notification pattern in translating the Action Language monitor specifications to Java monitors we need to associate the guard of each action with a lock specific to that action. Let a be an action with a guard, $guard_a$, and let $cond_a$ be the condition variable associated with a. The thread that calls the method that corresponds to action a will wait on $cond_a$ when $guard_a$ evaluates to false. Any thread that calls a method that corresponds to another action, b, that can change the truth value of $guard_a$ from false to true will notify $cond_a$. Hence, after an action execution only the threads that are relevant to the updates performed by that action will be awakened.

The algorithm given in Figure 5.3 generates the information about the syn-

```
1 for each action $a$ do
2     if $d_s(a) \neq true$ then
3         mark $a$ as guarded
4         create condition variable $cond_a$
5     else mark $a$ as unguarded
6     for each action $b$ s.t. $a \neq b$ do
7         if $Post(\neg d_s(b), \textsc{exp}(a)) \cap d_s(b) \neq \emptyset$ then
8             add $cond_b$ to notification list of $a$
```

Figure 5.3: Extracting Synchronization Information

chronization dependencies among different actions needed in the implementation of the specific notification pattern. For each action a in each submodule m_i the algorithm decides whether action a is *guarded* or *unguarded* by checking the expression $d_s(a)$. If $d_s(a)$ is true (meaning that there is no guard) then the action is marked as unguarded. Otherwise, it is marked as guarded and a condition variable, $cond_a$, is created for action a. Execution of an unguarded action does not depend on the shared variables, hence, it does not need to wait on any condition variable. Next, the algorithm finds all the actions that should be notified after the action a is executed. We can determine this information by checking for each action $b \neq a$, if executing the action a when $d_s(b)$ is false can result in a state where $d_s(b)$ is true. If this is possible, then the condition variable created for the action b, $cond_b$, is added to the notification list of the action a, which holds the condition variables that must be notified after the action a is executed.

Figure 5.4 shows the translation of guarded and unguarded actions to Java [59]. For each guarded action a a specific notification lock, $cond_a$ is declared and

```
private Object $cond_a$ = new Object();
public void $Guarded\_Wait_a$() {
    synchronized($cond_a$) {
        while (!$Guarded\_Execute_a$()) {
            try { $cond_a$.wait();}
            catch(InterruptedException e) {}
        }
    }
}

private synchronized boolean $Guarded\_Execute_a$() {
    if ($guard_a$) {$assign_a$;return true;}
    else return false;
}
```

(a)

```
public void $Execute_a$() {synchronized(this) {$assign_a$;}}
```

(b)

Figure 5.4: The translation of (a) guarded and (b) unguarded actions

one private method and one public method is generated. The private method $Guarded\_Execute_a$ is synchronized on *this* object. If the guard of the action a is true then this method executes assignments in $assign_a$ and returns true. Otherwise, it returns false. The method $Guarded\_Wait_a$ first gets the lock for $cond_a$. Then it runs a while loop till $Guarded\_Execute_a$ method returns true. In the body of the while loop it waits on $cond_a$ till it is notified by some thread that performs an update that can change the truth value of the method $guard_a$ and, therefore, $Guarded\_Execute_a$. For each unguarded action a a single public method $Execute_a$

is produced. This method first acquires the lock for `this` object. Then executes the assignments $assign_a$ of the corresponding action. Before exiting the public methods $Guarded\_Wait_a$ and $Execute_a$, synchronized($cond_b$) { $cond_b$.notifyAll(); } is executed for each action b in the notification list of the action a (note that this is not shown in Figure 5.4).

The automatically generated Java monitor class should preserve the verified properties of the Action Language specification. This can be shown in two steps: 1) Showing that the verified properties are preserved by the *single-lock implementation* of the Action Language specification. 2) Showing the equivalence between the single-lock implementation and the specific notification pattern implementation. The proof of correctness of specific notification pattern (step 2) is given in [59]. The algorithm we give in Figure 5.3 extracts the necessary information in order to generate a Java monitor class that correctly implements the specific notification pattern.

Below, we will give a set of assumptions under which the monitor invariants that are verified on an the Action Language specification of a monitor are preserved by its single-lock implementation as a Java monitor class.

1. *Initial Condition:* The set of program states immediately after the constructors of the monitor and the threads are executed satisfy the initial expression of the Action Language specification.

2. *Atomicity:* The observable states of the monitor are defined as the program states where *this* lock of the monitor is available, i.e., the states where no thread is active in the monitor.

3. *Thread Behavior:* The local behavior of the threads are correct with respect to the monitor specification.

4. *Scheduling:* If there exists an enabled action then an enabled action will be executed.

Assuming that the above conditions hold we claim that the observable states of the single-lock implementation of the Action Language monitor specification satisfy the monitor invariants verified by the Action Language Verifier.

5.3 Related Work

Recently, there have been several attempts at adopting model checking to the verification of concurrent programs [21, 48]. These approaches translate a concurrent Java program to a finite model and then check it using the available model checking tools. Hence, they rely on the ability of model checkers to cope with the state space explosion problem. However, to date, model checkers are not powerful enough to check implementations of concurrent programs. Hence, most of the recent work on the verification of concurrent programs have been on efficient model construction from concurrent programs [21, 48, 35]. Static analysis guided abstractions can be useful in model construction. For example, abstractions can reduce the state-space either by mapping variables to smaller domains [35] or by eliminating some of the variables that do not affect the correctness of the specification [21]. Slicing can be used to guide the abstractions. Using the property to be verified a slicing criterion can be extracted, and program parts that do not affect the correctness of the specification can be eliminated using program slicing [40]. Other techniques such as shape analysis [26] and predicate abstraction [5] have also been used for the efficient model construction.

Our approach provides a different direction for creating reliable concurrent programs. It has several advantages: 1) It avoids the implementation details in the program that do not relate to the property to be verified. 2) There is no model construction problem since the specification language used has a model checker associated with it. 3) By pushing the verification to an earlier stage in software development (to the specification phase rather than the implementation phase) it reduces the cost of fixing bugs. However, our approach is unlikely to scale to the generation of complete programs. This would require the specification language to be more expressive and would introduce a model construction problem at the specification stage. Hence we focus on synthesizing concurrency control components that are correct by construction and can be integrated to a concurrent program safely. Another aspect of our approach that is different from the previous work is the fact that we are using an infinite state model checker rather then finite state techniques. Using our infinite state model checker we can verify properties of specifications with unbounded integer variables and an arbitrary number of threads.

While this work was under review, independently, Deng et al. also presented an approach that combines specification, synthesis and verification for concurrent programs [36]. One crucial difference between our approach and the approach presented in [36] is apparent in the (otherwise remarkably similar) titles. In our approach automated verification is performed on the specification, not on the implementation. Hence, our approach shields the automated verification tool from the implementation details.

Chapter 6

Verifying Concurrent Linked Lists

The implementation of dynamic data structures is a tricky concept since a small error can cause unexpected aliasing between the pointer variables. Concurrency increases the possibility of such errors even further. Therefore, automated verification of concurrent programs that use dynamic data structures is highly desirable. This chapter presents a technique for the automated verification of concurrent linked lists such as doubly linked lists with multiple concurrent producer and consumer processes. Our technique can verify linked list implementations for structural invariant properties, which can also relate the structure to the integer variables in the implementation, for example, to a variable that denotes the size of the linked list.

Our technique uses the Composite Symbolic Representation framework. We provide a symbolic representation for encoding the state space of the heap. The main challenge for symbolically representing the heap configurations is to provide a bounded representation for an unbounded state space due to dynamic memory allocation and at the same time to keep the precision of the representation as high as possible. To make our symbolic representation for the heap bounded we use the *summarization patterns*, which is a user provided pattern specification about the shape of the linked list. The summarization patterns are specified in terms of a restricted class of graph grammars that is expressive enough to specify the pattern that the correct implementation of a linked list should match. The idea is to summarize a set of nodes that are connected with each other according to the edge relations specified by the summarization pattern by a single abstract node, which we call a *summary node*. However, summarizing a set of concrete nodes with a single abstract node means that we lose the information on the number of nodes that are summarized. In order to alleviate this imprecision, we introduce an integer counter, which we call the *summary count*, for each summary node. Note that using the summary counts makes the representation unbounded due to the integer domain. However, we show that the number of topologies for the heap configurations become bounded for a correct implementation. The unboundedness of the representation due to the integer summary counts is handled by the widening heuristic for the integer domains, which is presented in Chapter 4.2.

This chapter is organized as follows. Section 6.1 presents a concurrent linked list specification. Section 6.2 introduces the pattern-based representation, explains the pattern-based summarization and reports the boundedness results, and presents the algorithms for manipulation of the pattern-based representation. Section 6.3 explains the integration of the pattern-based shape analysis with the Composite Symbolic Library. Section 6.4 reports our experimental results. Finally, Section 6.5 summarizes the related work and compares our approach with the existing approaches for shape analysis.

6.1 An Example Concurrent Linked List

In this section we present a circular doubly linked list implementation that involves concurrent enqueue and dequeue operations. Figure 6.1 shows the Java implementation for the linked list. The `Node` class is used to implement a node in the linked list and the `Queue` class implements the circular doubly linked list. Each node in the linked list has three field selectors: `next`, `prev`, and `data`. The `next` field connects a node to the succeeding node in the linked list. The `prev` field connects a node to the preceding node in the linked list. The `data` field points to a unique data object. Since the linked list is circular, the `next` field of the last node points to the first node and the `prev` field of the first node points to the last node. The `head` field of the `Queue` class designates the first node in the linked list, whereas the `count` field of the `Queue` class denotes the number of nodes in the linked list.

Figures 6.2, 6.3, and 6.4 show the Action Language specification that models the linked list implementation given in Figure 6.1. We have extended the Action Language with a new type called *heap*. Heap type is similar to a restricted version of structure types in C where all the fields are pointers to the structure type that they belong. We call variables of the heap type *heap variables*. The heap type in Figure 6.2 models the `Node` class in Figure 6.1. The heap variable `head` models the `head` field of the `Queue` class. In Java each object is associated with a synchronization lock, which needs to be acquired in order to execute a synchronized block. The boolean variable `lock` models the synchronization lock. The `Enqueue` and the `Dequeue` modules model the `enqueue` and the `dequeue` methods of the `Queue` class, respectively. Asynchronous composition of instantiation of the `Enqueue` and the `Dequeue` modules model concurrently running threads that call

```
public class Node {
  public Node next;
  public Node prev;
  public Object data;
  public Node(Object o) {
    data=o;
    next=null;
    prev=null;
  }
}
public class Queue {
  private Node head;
  private int count;
  public Queue() {
      head=null;
      count=0;
  }
  public synchronized
  void enqueue(Object d) {
    Node temp1, temp2;
    temp1=new Node(d);
    temp1.next=temp1;
    temp1.prev=temp1;
    if (count==0) head=temp1;
    else {
       temp2=head.prev;
       temp1.next=head;
       temp1.prev=temp2;
       temp2.next=temp1;
       head.prev=temp1;
    }
    count++;
  }
  public synchronized
  Object dequeue() {
    Node temp1, temp2;
    if (count>0) {
       temp1=head.next;
       if (temp1==head) {
          temp2=head;
          head=null;
       }
       else {
          temp2=head.prev;
          temp1.prev=temp2;
          temp2.next=temp1;
          temp2=head;
          head.next=null;
          head.prev=null;
          head=temp1;
       }
       count--;
       return temp2.data;
    }
    return null;
  }
}
```

Figure 6.1: A Java class that implements a queue using a circular doubly linked list with concurrent enqueue and dequeue operations

```
module main()
  heap head {next,prev,data};
  boolean lock;
  integer count;

  initial: count=0 and lock and head=null;
  module Enqueue()
    ...
  endmodule
  module Dequeue()
    ...
  endmodule
  main: Enqueue() | Dequeue();
endmodule
```

Figure 6.2: An Action Language specification for the circular doubly linked list implementation given in Figure 6.1

the `enqueue` and the `dequeue` methods, respectively.

```
  module Enqueue()
    heap temp1, temp2 {next,prev,data};
    enumerated pc {p1, p2, p3, p4, p5, p6, p7, p8, p9};
    e1: pc=p1 and lock and !lock' and temp1'=new and pc'=p2;
    e2: pc=p2 and temp1'.next=temp1 and pc'=p3;
    e3: pc=p3 and temp1'.prev=temp1 and pc'=p4;
    e4: pc=p4 and temp1'.data=new and pc'=p5;
    e5: pc=p5 and count=0 and head'=temp1 and count'=count+1
        and lock'=true and pc'=p1;
    e6: pc=p5 and count!=0 and temp2'=head.prev and pc'=p6;
    e7: pc=p6 and temp1'.next=head and pc'=p7;
    e8: pc=p7 and temp1'.prev=temp2 and pc:=p8;
    e9: pc=p8 and temp2'.next=temp1 and pc:=p9;
    e10: pc=p9 and head'.prev=temp1 and count'=count+1 and
         lock'=true and pc'=p1;
    Enqueue: e1 | e2 | e3 | e4 | e5 | e6 | e7 | e8 | e9 | e10;
  endmodule
```

Figure 6.3: Specification of the `enqueue` method of the `Queue` class in the Action Language

Given the specification of a concurrent linked list as shown in Figure 6.2, we

```
module Dequeue()
  heap temp1, temp2 {next,prev,data};
  enumerated pc {p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11,
     p12, p13, p14, p15};
  d1: pc=p1 and lock and count>0 and !lock' and
      temp1'=head.next and  pc'=p10;
  d2: pc=p10 and temp1=head and head'=null and lock' and
      count'=count-1 and pc'=p1;
  d3: pc=p10 and temp1!=head and temp2'=head.prev and pc'=p11;
  d4: pc=p11 and temp1.prev'=temp2 and pc'=p12;
  d5: pc=p12 and temp2'.next=temp1 and pc'=p13;
  d6: pc=p13 and head'.next=null and pc'=p14;
  d7: pc=p14 and head'.prev=null and pc'=p15;
  d8: pc=p15 and head'=temp1 and lock' and count'=count-1 and
      pc'=p1;
  Dequeue: d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8;
endmodule
```

Figure 6.4: Specification of the `dequeue` method of the `Queue` class in the Action Language

would like to be able to reason about its correctness in terms of the shape related invariant properties of the data structure. Note that both the Java implementation (Figure 6.1) and the Action Language specification (Figures 6.3 and 6.4) for the circular doubly linked list keeps an integer variable (*count*) and checks conditions that depend on its value, e.g., $count > 0$. One way for handling such conditions is to ignore them completely by turning them into nondeterministic choice operations. However, we are interested in verifying the correctness properties of the concurrent linked list specifications such that the conditions on the integer variables would be handled as precisely as possible. Section 6.3 explains how our technique can achieve this.

Figure 6.5 shows the part of the reachable states for the `Enqueue` module given in Figure 6.3. The dots denotes the fact that the reachable states is infinite, hence, the shown sequence will continue indefinitely where each state in the sequence

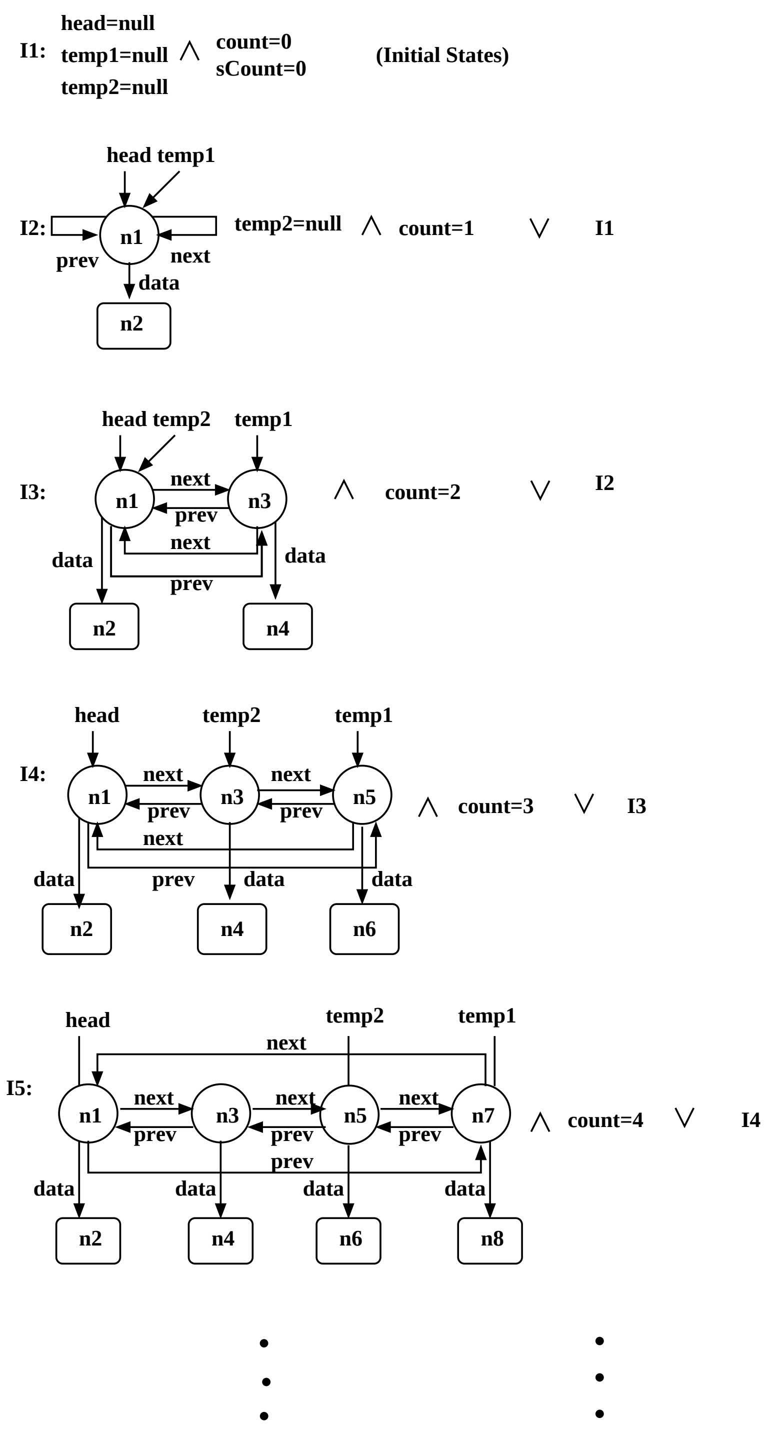


Figure 6.5: Part of the reachable states for the `Enqueue` module given in Figure 6.3 when `pc=p1` and `lock=true`.

will differ from the preceding one by one more node in the linked list and `count` being one greater. In order to reason about the correctness properties of this specification we need to approximate the infinite reachable states to a finite but precise abstraction. Section 6.2 explores how the unbounded growth in the heap can be made bounded and Section 6.3 explores approximating the unbounded integer domain.

Provided that we are able to provide an approximate and finite reachable state for the specification given in Figure 6.2, we are interested in verifying the correctness of the following invariant properties:

1. If the linked list is empty, i.e., $head = null$, then the value of $count$ is zero.
2. If both the $next$ and the $prev$ fields of the first node of the linked list point to itself then the value of $count$ is one.
3. If the $next$ field of the first node points to itself then its $prev$ field also points to itself.
4. If both the $next$ and the $prev$ fields of the first node of the linked list point to the same node other than itself then the value of $count$ is two.

Note that each of the invariants are expected to hold when the mutual exclusion lock is available in the specification given in Figure 6.2, i.e., when the value of $lock$ is true.

In the following sections, we explain our technique for automatically verifying the shape related invariant properties of concurrent linked lists.

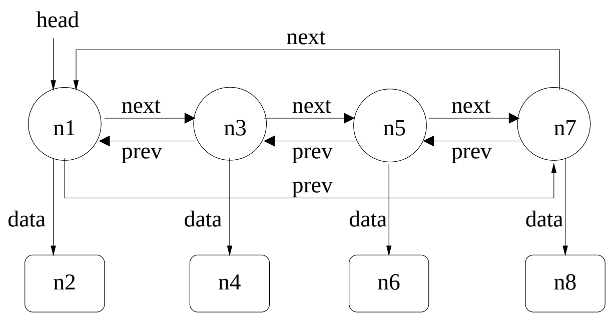


Figure 6.6: A circular doubly linked linked list with four nodes.

6.2 Pattern-based Representation

We focus on the linked lists that conform to a user-specified pattern. For instance, the circular doubly linked list whose instance with four nodes is shown in Figure 6.6 conforms to a pattern that can be informally described as “Every node in the linked list points to the succeeding node with its *next* field, points to the preceding node with its *prev* field, and points to a distinct location with its *data* field”. The importance of being able to associate a pattern with a linked list becomes obvious when considering the fact that linked lists can grow arbitrarily large and hence requires a special encoding for a finite representation. Using a pattern specification one can use a compact representation in which some nodes represent themselves and some nodes actually represent a set of nodes all of which satisfies the pattern and forms a connected graph. We call the latter type of nodes *summary node*s. Such a representation would also be precise since any manipulation of the linked list that requires a node that is represented by a summary node to be materialized, i.e., to be represented by itself, will be performed precisely. This is simply because of the availability of the pattern information, i.e., we know how the set of nodes represented by a summary node are connected to each other.

Now, we would like to formalize the pattern-based representation concept. Since we use a pattern to create summary nodes, we use the term *summarization pattern*. A *summarization pattern* is a four-tuple $P = (L, Sel, N_{var}, PR)$ where L is the non-terminal symbol, Sel is the set of selectors, N_{var} is the set of node variables, and PR is a graph grammar production rule in the following form:

$$\begin{aligned} L\ v_{lhs}x_1x_2\ldots x_u \rightarrow y_1.sel_1 &= t_1, y_2.sel_2 = t_2, \ldots, y_m.sel_m = t_m, \\ & L\ v_{rhs}x_1x_2\ldots x_u \end{aligned}$$

where $N_{var} = \{y_1, y_2, \ldots, y_m\} \cup \{t_1, t_2, \ldots, t_m\}$ and $v_{lhs} \in N_{var}$ and $v_{rhs} \in N_{var}$ and $v_{lhs} \neq v_{rhs}$ and $x_i \in N_{var}$ and $x_i \neq v_{lhs}$ and $x_i \neq v_{rhs}$ for all $1 \leq i \leq u$, and $sel_j \in Sel$ for all $1 \leq j \leq m$. v_{lhs} is called the *lhs-only variable* and v_{rhs} is called the *rhs-only variable*. We call $S_{lhs} = \{v_{lhs}, x_1, x_2, \ldots, x_u\}$ the lhs-variables, $S_{rhs} = \{v_{rhs}, x_1, x_2, \ldots, x_u\}$ the rhs-variables, and $y_1.sel_1 = t_1, y_2.sel_2 = t_2, \ldots, y_m.sel_m = t_m$ the matching constraints of the production rule PR. For the circular doubly linked list given in Figure 6.6, the pattern can be specified as $Lxy \rightarrow x.next = z, z.prev = x, x.data = y\ Lzy$. Here $Sel = \{prev, next\}$, $N_{var} = \{x, y, z\}$, $v_{lhs} = x$, $v_{rhs} = z$, $S_{lhs} = \{x, y\}$, and $S_{rhs} = \{z, y\}$.

We represent instances of a linked list using the *shape graph*s. Given a set of heap variables V and a set of field selectors Sel, a shape graph is a six-tuple $SG = (P, N, E, PT, SM, \Theta)$. P is the summarization pattern as defined above. N is the set of nodes. E denotes the set of edges in the form (n_1, sel, n_2) where $n_1, n_2 \in N$, $sel \in Sel$, and n_1 is the source node and n_2 is the destination node. $PT : V \rightarrow N \cup \{null, undef\}$ maps each heap variable $v \in V$ to $undef$ if it is not initialized, $null$ if it is set to null, or to the node that it points to. $SM \subset N$ is the set of summary nodes. $\Theta : SM \rightarrow N$ maps each summary node to a *tail node*, which will be explained later. Note that whenever $SM = \emptyset$, SG represents a single instance of the linked list and whenever $SM \neq \emptyset$, SG represents a set of instances of the linked list. Figure 6.6 shows the shape graph that represents an instance of the circular doubly linked list specified by the Action Language specification given in Figures 6.2, 6.3, and 6.4. Here

$$\begin{aligned}Sel = \{prev, next, data\},\ V = \{head, temp_1^E, temp_2^E, temp_1^D, temp_2^D\}, \\ N = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}, \\ E = \{(n_1, next, n_3), (n_3, next, n_5), (n_5, next, n_7), (n_7, next, n_1), (n_1, prev, n_7), \\ (n_7, prev, n_5), (n_5, prev, n_3), (n_3, prev, n_1), (n_1, data, n_2), (n_3, data, n_4), \\ (n_5, data, n_6), (n_7, data, n_8)\},\ PT(head) = n_1, \\ PT(temp_1^D) = PT(temp_1^E) = PT(temp_2^D) = PT(temp_2^E) = null,\ SM = \emptyset.\end{aligned}$$

A *one-step match function* $f \in F_{P,SG} = N_{var} \rightarrow N$ is defined as a one-to-one function that maps each node variable in N_{var} to a node in the shape graph such that $\forall 1 \leq i \leq m, (f(y_i), sel_i, f(t_i)) \in E$ holds. We further restrict the production rule of P as follows: instantiation of each node variable by a one-step match function f is reachable by the instantiation of the lhs-only variable, i.e., $\forall n_v \in N_{var}, \exists(n_1, n_2, \ldots, n_k), \forall 1 \leq i \leq k, k \geq 1, n_i \in N, n_1 = f(v_{lhs}) \wedge n_k = f(n_v) \wedge \forall 1 \leq j \leq i-1, \exists sel_m \in Sel, (n_j, sel_m, n_{j+1}) \in E$. Given the production rule $Lxy \rightarrow x.next = z, z.prev = x, x.data = y\ Lzy$ and the shape graph given in Figure 6.6, two example one-step match functions f_1 and f_2 are defined as $f_1(x) = n_3$, $f_1(y) = n_4$, $f_1(z) = n_5$, $f_2(x) = n_5$, $f_2(y) = n_6$, and $f_2(z) = n_7$.

Given a shape graph $SG = (P, N, E, PT, SM, \Theta)$, where $SM = \emptyset$, a *matching set* between nodes n and m, $M(SG, n, m)$, is not empty if there exists a sequence of one-step match functions $f_1, f_2, \ldots, f_k$, where $k \geq 1$, that satisfy the following constraints:

1. $f_1(v_{lhs}) = n$,
2. $f_k(v_{rhs}) = m$,
3. $\forall j, 1 \leq j < k, \forall i, 1 \leq i \leq u, f_j(z_i) = f_{j+1}(x_i)$,
4. $\forall v \in N_{var}, v \notin S_{lhs} \cup S_{rhs}, \forall i, j, 1 \leq i, j \leq k, i \neq j \rightarrow f_i(v) \neq f_j(v)$,
5. $M(SG, n, m) = \{f_j(v) \mid 1 \leq j \leq k \wedge v \in N_{var} \wedge v \notin S_{rhs}\}$, otherwise $M(SG, n, m) = \emptyset$.

Note that $M(SG, n, m)$ includes all the nodes that are in the range of the one-step match functions other than m. The matching set that is defined by the one-step match function f_1 is $\{n_3, n_4\}$, whereas the matching set that is defined by the one-step match functions f_1 and f_2 defined above is $\{n_3, n_4, n_5, n_6\}$.

Given a shape graph $SG = (P, N, E, PT, SM, \Theta)$, where $SM = \emptyset$, a *maximal matching set* between nodes n and m, $M_{max}(SG, n, m)$, is a matching set between nodes n and m that satisfies the following constraints:

1. For any matching set $M(SG, n', m')$ for any pair of nodes n' and m', $M_{max}(SG, n, m) \not\subset M(SG, n', m')$, i.e., $M_{max}(SG, n, m)$ is not a proper subset of any other $M(SG, n', m')$.
2. No node in $M_{max}(SG, n, m)$ is pointed by a heap variable, i.e., $\forall v \in V$, $PT(v) \notin M_{max}(SG, n, m)$.
3. No node in $M_{max}(SG, n, m)$ has more than one incoming edge with the same selector, i.e., $\forall n_1 \in M_{max}(SG, n, m), \neg\exists n_2, n_3, n_2 \neq n_3 \wedge (n_2, sel, n_1) \in N \wedge (n_3, sel, n_1) \in N$.

4. If the nodes in $M_{max}(SG, n, m)$ has incoming edges from the nodes that are not in $M_{max}(SG, n, m)$ then either n is the destination of such an edge or m is the source of such an edge but not both, i.e., $\forall(n_1, sel, n_2) \in E, (n_1, n_2 \in M_{max}(SG, n, m) \;\vee\; (n_2 = n \wedge n_1 \neq m) \;\vee\; (n_2 \neq n \wedge n_1 = m))$.

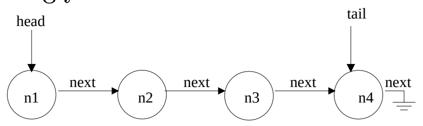
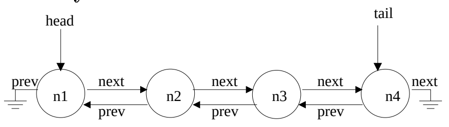
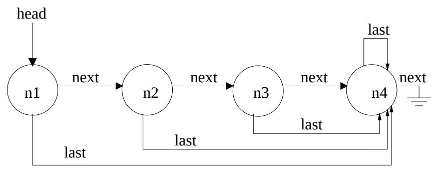
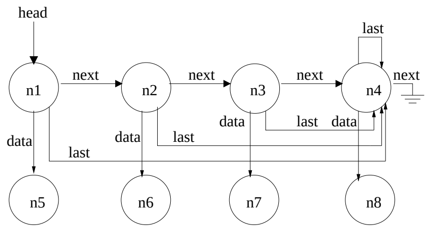
5. If the nodes in $M_{max}(SG, n, m)$ has outgoing edges to the nodes that are not in $M_{max}(SG, n, m)$ then either n is the source of such an edge or m is the destination of such an edge, i.e., $\forall(n_1, sel, n_2) \in E, (n_1, n_2 \in M_{max}(SG, n, m) \vee n_1 = n \;\vee\; n_2 = m)$.

6. For the sequence of one-step match functions $f_1, f_2, \ldots, f_k$, that corresponds to $M_{max}(SG, n, m)$, $k \geq 2$.

Node n and m are called the *entry point* and the *exit point* of the maximal matching set, respectively. Note that there may not be a maximal matching set between nodes n and m. The matching set that is defined by the one-step match function f_1 defined above is not a maximal matching set, however, the matching set that is defined by the one-step matching functions f_1 and f_2 is a maximal matching set, where n_3 is the entry point and n_7 is the exit point. Table 6.1 shows the production rules of the patterns corresponding to several linked lists along with their maximal matching sets.

Theorem 6.1 *Given a shape graph $SG = (P, N, E, PT, SM, \Theta)$, where $SM = \emptyset$, there does not exists a node of SG that can be mapped to two different maximal matching sets. (for the proof see the Appendix)*

Table 6.1: Summarization patterns for concurrent linked lists.

Summarization Pattern	Linked List
$L\ x \rightarrow x.next = y,\ L\ y$ $M_{max}(n_2, n_4) = \{n_2, n_3\}$	Singly linked list  head tail n1 next n2 next n3 next n4 next
$L\ x \rightarrow x.next = y,\ y.prev = x,\ L\ y$ $M_{max}(n_2, n_4) = \{n_2, n_3\}$	Doubly linked list  head tail prev n1 next prev n2 next prev n3 next prev n4 next
$L\ x\ z \rightarrow x.next = y,\ x.last = z,\ L\ y\ z$ $M_{max}(n_2, n_4) = \{n_2, n_3\}$	Linked list with connection to the last element  head last n1 next n2 next n3 next n4 next last last last
$L\ x\ z \rightarrow x.next = y,\ x.last = z,$ $x.data = w,\ L\ y\ z$ $M_{max}(n_2, n_4) = \{n_2, n_6, n_3, n_7\}$	Linked list with data and connection to the last element  head last n1 next n2 next n3 next n4 next data last data last data last data n5 n6 n7 n8

6.2.1 Pattern-based summarization

After having defined the basic concepts about the summarization patterns, in this subsection we present the *summarization* operation, which transforms a shape graph SG without summary nodes into a minimal shape graph SG' with summary nodes. Each summary node in SG' represents a unique maximal matching set of SG. Each summary node sm is associated with a *tail node*, which is the exit point, m, of the corresponding maximal matching set $M_{max}(SG, n, m)$. $\Theta : N \times N$ is a relation between the summary nodes and their tail nodes. Use of tail nodes will become clear in Section 6.2.2.

SG' is minimal in the sense that for each maximal set of SG there exists a corresponding summary node in SG'. The property of the summarization operation to generate minimal shape graphs is crucial for the boundedness of the representation.

Figure 6.7 shows algorithm SUMMARIZE, which implements the summarization operation. It takes a shape graph SG with no summary nodes as input and returns a minimal shape graph with summary nodes. Algorithm SUMMARIZE traverses the input graph starting from each node that is pointed by a heap variable. It calls algorithm FINDMAXIMALSETS, which is given in Figure 6.8, to find the set of all maximal matching sets. Algorithm FINDMAXIMALSETS takes as input a shape graph SG, a node m, and the set of nodes visited so far, $visited$. If m has not been visited yet, it tries each selector sel in Sel to reach every node n that has an incoming edge from m. If n has not been visited yet then the algorithm tries to find a maximal matching set whose entry point is n (lines 11-19). It finds a maximal matching set by iteratively finding a sequence of one-step matching functions. In every iteration it calls algorithm ISMAXIMAL which is given in Figure 6.9, to check whether the nodes that are mapped by the one-step matching functions can indeed be summarized. Algorithm ISMAXIMAL checks the constraints 2-4 of maximal matching sets. Constraint 1 of maximal matching sets is ensured by traversing the graph starting from each heap variable, and constraints 5 and 6 of maximal matching sets are checked at lines 17-18. Each maximal set that is found is stored in variable $result$, which denotes the sets of maximal matching sets found so far. All the nodes in a maximal matching set are also stored in $visited$. When the algorithm discovers that there is no maximal matching set with n being its entry point or the exit point of the maximal set is reached,

```
SUMMARIZE($SG = (P, N, E, PT, SM, \Theta)$): shape graph
$SG$: shape graph, $visited$: set of nodes, $maximalSets$:set of maximal sets
$visited \leftarrow \emptyset$, $maximalSets \leftarrow \emptyset$
for each $v \in V$ do
    if $PT(v) \neq null$ then
        $maximalSets \leftarrow maximalSets \cup$ FINDMAXIMALSETS$(SG, PT(v), visited)$
$N' \leftarrow N$
$E' \leftarrow E$
$PT' \leftarrow PT$
$SM' \leftarrow \emptyset$
for each $m_i \in maximalSets$ do
    let $m_i = \langle maxSet, entry, exit \rangle$
    $N' \leftarrow N' \setminus maxSet_i$
    $N' \leftarrow N' \cup \{sm_i\}$
    $SM' \leftarrow SM' \cup \{sm_i\}$
    $E' \leftarrow E' \setminus \{(n_1, sel, n_2) \mid (n_1, sel, n_2) \in E \wedge n_1 \in maxSet_i \wedge n_2 \in maxSet_i\}$
    $E' \leftarrow E' \cup \{(n_1, sel, sm_i) \mid (n_1, sel, n_2) \in E \wedge n_2 \in maxSet_i \wedge n_1 \notin maxSet_i\}$
    $E' \leftarrow E' \cup \{(sm_i, sel, n_2) \mid (n_1, sel, n_2) \in E \wedge n_1 \in maxSet_i \wedge n_2 \notin maxSet_i\}$
    $PT' \leftarrow PT$
    $\Theta' = \Theta[exit/sm_i]$
return $(N', E', PT', SM', \Theta')$
```

Figure 6.7: The algorithm for summarizing a given shape graph SG.

```
FINDMAXIMALSETS($SG = (P, N, E, PT, SM, \Theta)$, $m$, $visited$): set of maximal sets
$SG$: shape graph, $m$: node, $visited$: set of nodes, $result$: set of maximal sets
$result \leftarrow emptyset$
if $m \notin visited$ then
  $visited \leftarrow visited \cup \{m\}$
  for each $sel \in Sel$ do
      let $(m, sel, n) \in E$
      if $n \notin visited \wedge n \neq null$ then
        $maxSet \leftarrow \emptyset$
        $n_{lhs} \leftarrow n$
        while there exists a one-step match function $f$ where $f(v_{lhs}) = n_{lhs}$ do
              $visited \leftarrow visited \cup \{f(n_v) \mid n_v \in N_{var}\} \setminus \{v_{rhs}\}$
              if ISMAXIMAL($SG, n, maxSet, f$) then
                $maxSet \leftarrow maxSet \cup \{f(n_v) \mid n_v \in N_{var}\} \setminus \{f(v_{rhs})\}$
              else break
              $n_{lhs} \leftarrow f(v_{rhs})$
        if $\neg\exists m_1, m_2, m_1 \notin maxSet \wedge m_2 \in maxSet, (m_2, sel, m_1) \in E$
          $\wedge m_2 \neq n \wedge m_1 \neq f(v_{rhs}) \wedge |maxSet| \geq 2$ then
          $result \leftarrow result \cup \{\langle maxSet, n, f(v_{rhs})\rangle\}$
        $result \leftarrow result \cup$ FINDMAXIMALSETS($SG, n_{lhs}, visited$)
return $result$
```

Figure 6.8: The algorithm for finding the maximal matching sets of a shape graph SG reachable form node m.

```
ISMAXIMAL($SG = (P, N, E, PT, SM, \Theta)$, $e$, $maxSet$, $f$): boolean
$SG$: shape graph, $e$: node, $maxSet$: set of nodes, $f$: one-step match function
for each $n_v \in N_{var} \setminus \{v_{rhs}\}$ do
    if $\exists v \in V, PT(v) = f(n_v)$ then
        return $false$
    if $\exists n_1, n_2, sel, n_1 \neq n_2, (n_1, sel, f(n_v)) \in E \wedge (n_2, sel, f(n_v)) \in E$ then
        return $false$
    if $\exists (n, sel, f(n_v)) \in E, n \notin maxSet$ then
        if $f(n_v) \neq e \wedge n \neq f(v_{rhs})$ then
            return $false$
return $true$
```

Figure 6.9: The algorithm for checking the constraints 2-4 of maximal matching sets for the given candidate set $maxSet$.

algorithm FINDMAXIMALSETS is called on node n_{lhs}. n_{lhs} is equal to either n or the node that n_{rhs} was mapped by the last one-step match function. After all the maximal matching sets are found, algorithm SUMMARIZE creates a minimal shape graph where each maximal matching set of SG is represented by a summary node. Figure 6.10 shows the output of algorithm SUMMARIZE on input shape graph given in Figure 6.6.

Since algorithm SUMMARIZE finds all maximal matching sets and Theorem 6.2 implies that the returned shape graph is unique, the returned shape graph is guaranteed to be minimal.

Below we claim that the pattern-based representation is a conditionally bounded representation. First we need to give some definitions.

Definition 6.1 *A shape graph SG is minimal iff* SUMMARIZE$(SG) = SG$.

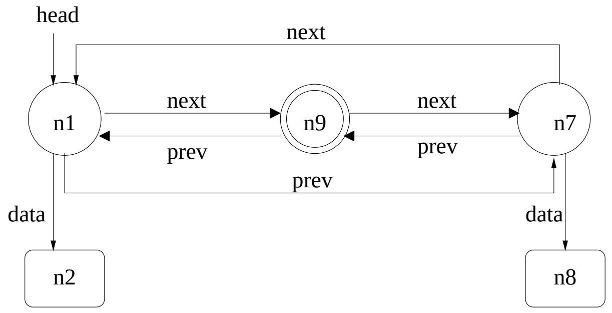


Figure 6.10: The output of the summarization operation on the shape graph of Figure 6.6

Definition 6.2 *A shape graph* $SG = (P, N, E, PT, SM, \Theta)$ *is* k-well formed *iff* SG *is minimal and the number of non-summary nodes is* k*, i.e.,* $|N \setminus SM| = k$.

Theorem 6.2 *The number of nodes in a k-well formed shape graph* $SG = (P, N, E, PT, SM, \Theta)$ *is bounded by* $k \times (|Sel| + 1)$*. (for the proof see the Appendix)*

Corollary 6.1 *Given a linked list* D *and a summarization pattern* P*, let* S *denote the set of minimal shape graphs representing all the instances of* D*. If there exists a constant* n *such that every* s *in* S *is a* k_s*-well formed shape graph based on pattern* P*, where* $k_s \leq n$*, then* S *can be represented by a finite set of minimal shape graphs. Hence, linked list* D *can be encoded by a bounded pattern-based symbolic representation using* P*.*

6.2.2 Manipulation of pattern-based representation

In this subsection, given the pattern-based representation of a heap configuration and a heap formula we explain how to compute the post-condition. Table 6.2 shows the set of heap formulas that can appear in the guard and update part of an action in an Action Language specification.

Table 6.2: List of the heap formulas that can appear in *guard* and *update* parts of a specification (id_1 and id_2 are heap variables and sel is a field selector)

Guard Formulas		Update Formulas	
$id_1 = id_2$	$id_1.sel = id_2$	$id_1' = id_2$	$id_1' = id_2.sel$
$id_1'.sel = id_2.sel$	$id_1 \neq id_2$	$id_1'.sel = id_2$	$id_1'.sel_1 = id_2.sel_2$
$id_1.sel \neq id_2$	$id_1.sel_1 \neq id_2.sel_2$	$id_1' = null$	$id_1'.sel = null$
$id_1 = null$	$id_1.sel = null$	$id_1' = new$	$id_1'.sel = new$

Before presenting the algorithms for each type of update formula, we would like to introduce a key operation that enables precise updates: *materialization operation*. To motivate the use of materialization operation , let us assume that the update formula $temp1' = head.next$ will be performed on the shape graph given in Figure 6.10. Since *head.next* refers to a summary node, n_9, we cannot simply make *temp*1 point to n_9. The reason is that n_9, does not represent a single node but a set of nodes that form a maximal matching set M and *temp*1, being a heap variable, can only point to a single node. Therefore, in order to perform the update $temp1' = head.next$ on the shape graph given in Figure 6.10 nodes in a subset M' of M are made explicit and the summary node n_9 becomes a representation of the maximal matching set $M \setminus M'$. The edge relations for the nodes in M' are constructed according to the summarization pattern. For instance, after materializing the shape graph given in Figure 6.10 the shape graph in Figure 6.6 can be obtained. Below we present three type of materialization operations and the update algorithms.

Materialization Operation

Since the materialization operation involves the materialization of nodes that are represented by a summary node, we need to have two types of materialization operations: i) One type of materialization operation assumes that it is dealing

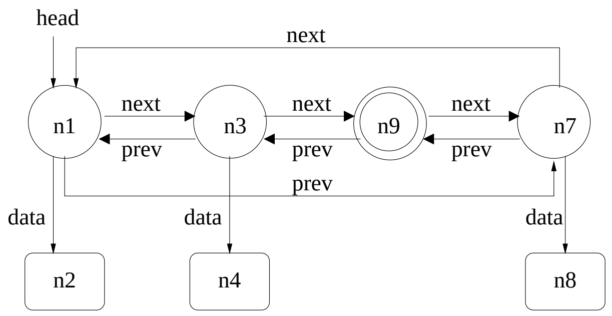


Figure 6.11: The output of the type-2 left-materialization operation on the shape graph of Figure 6.10

with the corner case: the number of one-step match functions for the maximal matching set that corresponds to the materialized summary node was 2. We call this *type-1 materialization* operation. In this type of materialization operation, since all the nodes that are represented by a summary node are materialized, the summary node disappears. Figure 6.6 shows the shape graph obtained after performing the materialization operation on the shape graph given in Figure 6.10. ii) Another type of materialization operation assumes that the number of one-step match functions for the maximal matching set that corresponds to the materialized summary node was greater than 2. This type is further divided into two categories depending on whether the materialized nodes will be to the left of the summary nodes or to the right of the summary nodes, which we call *type-2 left-materialization* and *type-2 right-materialization* operations, respectively. Figure 6.11 and Figure 6.12 shows the shape graph obtained after performing type-2 left-materialization operation and type-2 right-materialization operation on the shape graph given in Figure 6.10, respectively.

Below we give the formal definitions of the three materialization operations. Given a shape graph $SG = (P, N, E, PT, SM, \Theta)$ and a summary node $sm \in SM$,

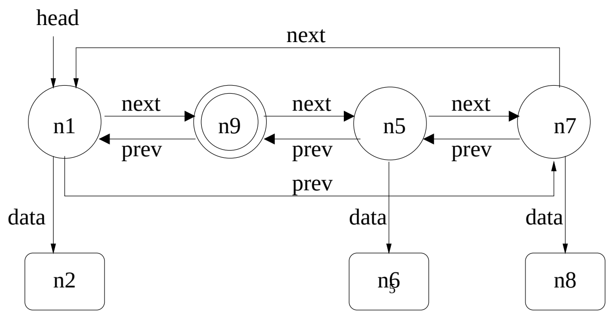


Figure 6.12: The output of the type-2 right-materialization operation on the shape graph of Figure 6.10

type-1 materialization operation generates a new shape graph SG' by materializing sm, i.e., $SG' = (P, N', E', PT', SM', \Theta') = \textsc{type-1m}(SG, sm)$, such that

- $N' = (N \setminus \{sm\}) \cup m_1 \cup m_2$, where $m_1 \neq m_2 \wedge |m_1 \cap m_2| = |S_{rhs}|$,
- $SM' = SM \setminus \{sm\}$,
- $$\begin{aligned} E' \;=\; & (E \setminus \{(n_1, sel, n_2) \mid n_1 = sm \vee n_2 = sm\}) \cup \\ & \{(n_1, sel, n_2) \mid n_1, n_2 \in m_1 \wedge \phi_1(n_1).sel = \phi_1(n_2)\} \cup \\ & \{(n_1, sel, n_2) \mid n_1, n_2 \in m_2 \wedge \phi_2(n_1).sel = \phi_2(n_2)\} \cup \\ & \{(n_1, sel, n_2) \mid (n_1, sel, sm) \in E \wedge n_1 = \Theta(sm) \wedge \\ & \phi_2(n_1).sel = \phi_2(n_2)\} \cup \\ & \{(n_1, sel, n_2) \mid (n_1, sel, sm) \in E \wedge n_1 \neq \Theta(sm) \wedge \\ & \phi_1(n_2) = v_{lhs}\} \cup \\ & \{(n_1, sel, n_2) \mid (sm, sel, n_2) \in E \wedge n_2 = \Theta(sm) \wedge \\ & \phi_2(n_1).sel = \phi_2(n_2)\} \cup \\ & \{(n_1, sel, n_2) \mid (sm, sel, n_2) \in E \wedge n_2 \neq \Theta(sm) \wedge \phi_1(n_1) = v_{lhs}\} \end{aligned}$$

where functions $\phi_1 : m_1 \rightarrow N_{var}$ and $\phi_2 : m_2 \rightarrow N_{var}$ are one-to-one functions such that $\forall i, 1 \leq i \leq u, \exists n \in m_1 \cap m_2, \phi_1(n) = z_i \wedge \phi_2(n) = x_i \wedge \Theta(sm) \in m_2 \wedge \phi_2(\Theta(sm)) = v_{rhs}$ and $\forall y \in N_{var} \setminus S_{rhs}, \forall n \in m_1, \phi_1(n) = y \Rightarrow n \notin N \wedge \forall n \in m_2, \phi_2(n) = y \Rightarrow n \notin N$.

- $PT' = PT$,
- $\forall n \in SM, n \neq sm, \Theta'(n) = \Theta(n)$.

Sets m_1 and m_2 include new nodes generated based on the summarization pattern P. The nodes in $m_1 \cup m_2$ form a maximal matching set

in SG', i.e., $M_{max}(SG', n, m) = m_1 \cup m_2$ where the length of the sequence of one-step match functions that corresponds to $M_{max}(SG', n, m)$ is two. The edge set E' is obtained from E by i) removing the incoming edges to sm and outgoing edges from sm, ii) adding new edges between the nodes in m_1 according to summarization pattern P, iii) adding new edges between the nodes in m_2 according to summarization pattern P, iv) adding new edges from the nodes not in $m_1 \cup m_2$ to the nodes in $m_1 \cup m_2$, which concretizes the edges from non-summary nodes to sm in SG, and v) adding new edges from the nodes in $m_1 \cup m_2$ to nodes that are not in $m_1 \cup m_2$, which concretizes the edges from sm to non-summary nodes in SG. Since sm is removed from the set of summary nodes, the tail node for sm is no longer defined.

Given a shape graph $SG = (P, N, E, PT, SM, \Theta)$ and a summary node $sm \in SM$, type-2 left-materialization operation generates a new shape graph SG' by materializing sm from the left, i.e., $SG' = (P, N', E', PT', SM', \Theta') = \text{TYPE-2LEFTM}(SG, sm)$, such that

- $N' = N \cup m$, where $|m| = |N_{var}| \wedge sm \in m$,

- $SM' = SM$,

- $$\begin{aligned} E' &= (E \setminus \{(n_1, sel, n_2) \mid (n_1 = sm \wedge n_2 \neq \Theta(sm)) \vee (n_2 = sm \wedge \\ &\quad n_1 \neq \Theta(sm))\}) \cup \{(n_1, sel, n_2) \mid n_1, n_2 \in m \wedge \phi(n_1).sel = \phi(n_2)\} \cup \\ &\quad \{(n_1, sel, n_2) \mid (n_1, sel, sm) \in E \wedge n_1 \neq \Theta(sm) \wedge \\ &\quad \phi(n_1) = v_{lhs}\} \cup \\ &\quad \{(\hat{n_1}, sel, \hat{n_2}) \mid (sm, sel, \hat{n_2}) \in \hat{E} \wedge \\ &\quad (\exists y_i.sel = y_j, y_j \in S_{lhs} \wedge y_j \in S_{rhs} \wedge n_1 = \phi(y_i)) \vee \\ &\quad (\neg\exists y_i.sel = y_j, y_j \in S_{lhs} \wedge y_j \in S_{rhs} \wedge n_2 \neq \Theta(sm) \wedge \\ &\quad \phi(n_1) = v_{lhs})\} \end{aligned}$$

  where function $\phi : m \to N_{var}$ is a one-to-one function such that $\phi(sm) = v_{rhs}$ and $\forall y \in N_{var} \setminus S_{rhs}, \forall n \in m, \phi(n) = y \Rightarrow n \notin N$.

- $PT' = PT$.

- $\Theta' = \Theta$.

Set m includes new nodes generated based on the summarization pattern P. The nodes in m form a matching set in SG', i.e., $M(SG', n, m) = m$ where the length of the sequence of one-step match functions that corresponds to $M(SG', n, m)$ is one. The edge set E' is obtained from E by i) removing the incoming edges to sm whose source node is not the tail node of sm and outgoing edges from sm whose destination node is not the tail node of sm, ii) adding new edges between the nodes in m according to summarization pattern P, iii) adding new edges from the nodes not in m to the nodes in m, which concretizes some of the edges from non-summary nodes to sm in SG, and iv) adding new edges from the nodes in m to nodes that are not in m, which concretizes some of the edges from sm to non-summary nodes in SG.

Given a shape graph $SG = (P, N, E, PT, SM, \Theta)$ and a summary node $sm \in SM$, type-2 right-materialization operation generates a new shape graph SG' by materializing sm from the right, i.e., $SG' = (P, N', E', PT', SM', \Theta') = \textsc{type-2RightM}(SG, sm)$, such that

- $N' = N \cup m$, where $|m| = |N_{var}| \wedge \Theta(sm) \in m$,
- $SM' = SM$,
- $$\begin{aligned} E' \;=\; & (E \setminus \{(n_1, sel, n_2) \mid (n_1 = sm \wedge n_2 = \Theta(sm)) \vee (n_2 = sm \wedge \\ & n_1 = \Theta(sm))\}) \cup \{(n_1, sel, n_2) \mid n_1, n_2 \in m \wedge \phi(n_1).sel = \phi(n_2)\} \cup \\ & \{(n_2, sel, sm) \mid (n_1, sel, sm) \in E \wedge n_1 = \Theta(sm) \wedge \\ & \phi(n_2) = v_{lhs}\} \cup \\ & \{(sm, sel, n_1) \mid (sm, sel, n_2) \in E \wedge n_2 = \Theta(sm) \wedge \\ & \phi(n_1) = v_{lhs}\} \cup \\ & \{(n_1, sel, n_2) \mid (sm, sel, n_2) \in E \wedge \exists y_i.sel = y_j, y_j \in S_{lhs} \wedge y_j \in S_{rhs} \\ & \wedge n_1 = \phi(y_i)\} \end{aligned}$$

  where function $\phi : m \rightarrow N_{var}$ is a one-to-one function such that $\phi(\Theta(sm)) = v_{rhs}$ and $\forall y \in N_{var} \setminus S_{rhs}, \forall n \in m, \phi(n) = y \Rightarrow n \notin N$.
- $PT' = PT$.
- $\forall n \in SM, n \neq sm, \Theta'(n) = \Theta(n) \wedge \exists n \in m, \phi(n) = v_{lhs} \wedge \Theta'(sm) = n.$

Set m includes new nodes generated based on the summarization pattern P. The nodes in m form a matching set in SG', i.e., $M(SG', n, m) = m$ where the length of the sequence of one-step match functions that corresponds to $M(SG, n, m)$ is one. The edge set E' is obtained from E by i) removing the incoming edges to sm whose source node is the tail node of sm and the outgoing edges from sm whose destination node is the tail node of sm, ii) adding new edges between the nodes in m according to the summarization pattern P, iii) adding new edges from the nodes not in m to the nodes in m, which concretizes some of the edges from non-summary nodes to sm in SG, and iv) adding new edges from the nodes in m to nodes that are not in m, which concretizes some of the edges from sm to non-summary nodes in SG. The tail node of sm becomes the node, which is mapped to the lhs-only variable by the function ϕ.

Update Algorithms

In this section, we present the post-condition computation algorithm and the update algorithms for each heap update formula.

Algorithm *Post*, which is given in Figure 6.13, computes the post-condition of a heap update formula on a given shape graph. If the heap update formula is in the form $x' = null$ then it calls algorithm SETNULL, which is given in Figure 6.14. Algorithm SETNULL sets x to $null$ and summarizes the resulting shape graph. If the heap update formula is in the form $x'.sel = y$ then it calls algorithm SETFIELD, which is given in Figure 6.15. Algorithm SETFIELD calls algorithm SETFIELDNULL, which is given in Figure 6.16, to set the sel field of x to $null$. Then sets the sel field of x to y. Note that the summarization is handled inside algorithm SETFIELDNULL. If the heap update formula is in the form $x' = y.sel$ then it calls algorithm SETTOFIELD, which is given in Figure 6.17. Depending on

```
$Post(SG, r)$: set of shape graphs
$SG$: shape graph, $r$: heap update formula
**case** $(x' = null)$:
    SETNULL($SG$,$x$);
    **return** $\{SG\}$;
**case** $(x'.sel = y)$:
    SETFIELD($SG$,$x$,$sel$,$y$);
    **return** $\{SG\}$;
**case** $(x' = y.sel)$:
    **return** SETTOFIELD($SG$,$x$,$y$,$sel$)
**case** $(x' = y)$:
    SET($SG$,$x$,$y$)
    **return** $\{SG\}$;
**case** $(x' = new)$:
    SETNEW($SG$,$x$)
    **return** $\{SG\}$;
**case** $(x'.sel_1 = y.sel_2)$:
    SETFIELDTOFIELD($SG$, $x$, $sel_1$,$y$,$sel_2$);
    **return** $\{SG\}$;
**case** $(x'.sel = null)$:
    SETFIELDNULL($SG$,$x$,$sel$);
    **return** $\{SG\}$;
**case** $(x'.sel = new)$:
    SETFIELDNEW($SG$,$x$,$sel$);
    **return** $\{SG\}$;
```

Figure 6.13: The algorithm for computing the post-condition of a heap configuration represented by a shape graph SG with respect to a heap update formula r

whether the node pointed at by y via the *sel* field is a summary node, algorithm SETTOFIELD either performs materialization or simply performs the update. In the case of a summary node two types of materialization operations are performed so that all possible summary count values are considered, i.e., summary count is equal to two or greater than two. For the case that summary count is equal to two it performs TYPE-1Moperation. For the case that summary count is greater than two, if y points to the tail node of the summary node then it performs TYPE-2RIGHTM, otherwise performs TYPE-2LEFTM. It sets x to the node pointed to by y after the materialization is performed. Finally, it performs the summarization operation on each resulting shape graph. If the heap update formula is in the form $x' = new$, then it calls algorithm SETNEW, which is given in Figure 6.19. Algorithm SETNEW creates a new node, sets x to n, and performs the summarization. If the heap update formula is in the form $x'_sel_1 = y'.sel_2$ then it calls algorithm SETFIELDTOFIELD, which performs materialization similar to algorithm SETTOFIELDexcept that it sets the sel_1 field of x to the node pointed at by the sel_2 field of y. If the heap update formula is in the form $x'.sel = null$ then it calls algorithm SETFIELDNULL, which is given in Figure 6.16. Algorithm SETFIELDNULL sets the *sel* field of x to *null* and performs summarization. Finally, if the heap update formula is in the form $x'.sel = new$ then it calls algorithm SETFIELDNEW, which is given in Figure 6.21. Algorithm SETFIELDNEW creates a new node and sets the *sel* field of x to that new node and then performs summarization.

6.2.3 Improving the precision

The pattern-based representation for heap configurations is a compact and partly an imprecise representation. Although the pattern information makes it

```
SETNULL($SG$, $x$) : shape graph
$SG$: shape graph, $x$: heap variable
$SG.PT(x) \leftarrow null$
return SUMMARIZE($SG$)
```

Figure 6.14: The algorithm for setting a heap variable to null

```
SETFIELD($SG$, $x$, $sel$, $y$): shape graph
$SG$: shape graph, $sel$: selector, $x$, $y$: heap variable
$SG_1 \leftarrow$ SETFIELDNULL($SG, x, sel$)
let $SG_1.PT(x) = n \wedge SG_1.PT(y) = m$
$SG_1.E \leftarrow SG_1.E \cup \{(n, sel, m)\}$
return $SG_1$
```

Figure 6.15: The algorithm for setting a field of a heap variable to a heap variable.

```
SETFIELDNULL($SG$, $x$, $sel$): shape graph
$SG$: shape graph, $x$: heap variable, $sel$: selector
let $SG.PT(x) = n$
$SG.PT \leftarrow SG.PT \cup \{(n, sel, null)\} \setminus \{(n, sel, x) | (n, sel, x) \in SG.E\}$
return SUMMARIZE($SG$)
```

Figure 6.16: The algorithm for setting a field of a heap variable to null.

```
SETTOFIELD($SG = (P, N, E, PT, SM, \Theta)$, $x$, $y$, $sel$): set of shape graphs
$SG$: shape graph, $x$, $y$: heap variable, $sel$: selector
let $(PT(y), sel, n) \in E$
if $n \in SM$ then
   $SG_1 \leftarrow$ TYPE-1M$(SG, n)$
   let $(SG_1.PT(y), sel, m) \in SG_1.E$
   $SG_1.PT(x) \leftarrow m$
   $SG_1 \leftarrow$ SUMMARIZE$(SG_1)$
   let $(n, t) \in \Theta$
   if $PT(y) = t$ then
      $SG_2 \leftarrow$ TYPE-2RIGHTM$(SG)$
   else
         $SG_2 \leftarrow$ TYPE-2LEFTM$(SG)$
   let $(SG_2.PT(y), sel, u) \in SG_2.E$
   $SG_2.PT(x) \leftarrow u$
   $SG_2 \leftarrow$ SUMMARIZE$(SG_2)$
   return $\{SG_1, SG_2\}$
else
      $SG_1 \leftarrow SG$
      $SG_1.PT(x) \leftarrow n$
      $SG_1 \leftarrow$ SUMMARIZE$(SG_1)$
      return $\{SG_1\}$
```

Figure 6.17: The algorithm for setting a heap variable to a field of a heap variable.

```
SET($SG$, $x$, $y$): shape graph
$SG$: shape graph, $x$, $y$: heap variable
$SG.PT(x) \leftarrow SG.PT(y)$
return SUMMARIZE($SG$)
```

Figure 6.18: The algorithm for setting a heap variable to a heap variable.

```
SETNEW($SG$, $x$)
$SG$: shape graph, $x$: heap variable
let $n \notin SG.N$
$SG.N \leftarrow SG.N \cup \{n\}$
$SG.PT(x) \leftarrow n$
return SUMMARIZE($SG$)
```

Figure 6.19: The algorithm for setting a heap variable to a new node.

```
SETFIELDTOFIELD($SG$,$x$,$sel_1$,$y$,$sel_2$): shape graph
$SG$: shape graph, $x$, $y$: heap variable, $sel_1$, $sel_2$: selector
let $(PT(y), sel_2, n) \in E$
if $n \in SM$ then
  $SG_1 \leftarrow$ TYPE-1M$(SG, n)$
  let $SG_1.PT(x) = s \wedge (SG_1.PT(y), sel_2, m) \in SG_1.E$
  $SG_1.E \leftarrow SG_1.E \cup \{(s, sel_1, m)\} \setminus \{(s, sel_1, x) | (s, sel_1, x) \in SG_1.E$
  $SG_1 \leftarrow$ SUMMARIZE$(SG_1)$
  let $(n, t) \in \Theta$
  if $PT(y) = t$ then
    $SG_2 \leftarrow$ TYPE-2RIGHTM$(SG)$
  else
      $SG_2 \leftarrow$ TYPE-2LEFTM$(SG)$
  let $SG_2.PT(x) = v \wedge (SG_2.PT(y), sel, u) \in SG_2.E$
  $SG_2.E \leftarrow SG_2.E \cup \{(v, sel_1, u)\} \setminus \{(v, sel_1, x) | (v, sel_1, x) \in SG_2.E\}$
  $SG_2 \leftarrow$ SUMMARIZE$(SG_2)$
  return $\{SG_1, SG_2\}$
else
    $SG_1 \leftarrow SG$
    let $SG_1.PT(x) = y$
    $SG_1.E \leftarrow SG_1.E \cup \{(y, sel_1, n)\} \setminus \{(y, sel_1, x) | (y, sel_1, x) \in SG_1.E\}$
    $SG_1 \leftarrow$ SUMMARIZE$(SG_1)$
    return $\{SG_1\}$
```

Figure 6.20: The algorithm for setting a field of a heap variable to a field of a heap variable.

```
SETFIELDNEW($SG$, $x$, $sel$): shape graph
$SG$: shape graph, $x$: heap variable, $sel$: selector
$SG_1 \leftarrow$ SETFIELDNULL($SG, x, sel$)
let $n \notin SG_1.N \wedge SG_1.PT(x) = m$
$SG_1.E \leftarrow SG_1.E \cup \{(m, sel, n)\}$
return $SG_1$
```

Figure 6.21: The algorithm for setting a field of a heap variable to a new node.

possible to keep the shape information about the summarized nodes, we lose the information about the number of nodes that are matched to a particular summary node. We can overcome this imprecision of the pattern-based representation by associating each summary node with a *summary count*, which keeps the number of nodes represented by that summary node.

We introduce the *extended shape graph*s, which are shape graphs with summary counts:

Definition 6.3 *An extended shape graph is an 8-tuple $ESG = (P, N, E, PT, SM, \Theta, V_{sc}, SC)$, where there exists a shape graph $SG = (P, N, E, PT, SM, \Theta)$, V_{sc} is a set of integer variables denoting the summary counts, and $SC : SM \to V_{sc}$, which is called the* summary counter*, is a function that maps each summary node to a summary count. SG is called the topology of the extended shape graph ESG.*

A consequence of using the summary counts is that the pattern-based representation becomes a precise representation hence it is no longer a bounded representation, i.e., the summary counts can take arbitrarily large values. Corollary

6.1 is no longer valid for the version of the pattern-based representation with summary counts, i.e., we may not have a finite set of shape graphs to encode all the instances of a linked list even though it has the property that any of its instance can have maximal matching sets with a bounded size. However, Corollary 6.2 implies a useful result for the pattern-based representation with summary counts:

Corollary 6.2 *Given a linked list D and a summarization pattern P, let S denote the set of minimal extended shape graphs representing all the instances of D. If there exists a constant n such that topology of every s in S is a k_s-well formed shape graph based on pattern P, where $k_s \leq n$, then projection of S on the topology can be represented by a finite set of minimal shape graphs.*

Corollary 6.2 states that as long as the summary counts can be abstracted by a bounded representation the pattern-based representation may become a bounded representation. We will further explore this in Section 6.3.

6.3 Integrating Shape Analysis to Composite Representation

We have extended the Composite Symbolic Library with the pattern-based extended shape graph representation for symbolically encoding heap configurations. After this extension a state S that is encoded using the composite representation can be defined as

$$S = \bigvee_{1 \leq i \leq n} S_{i,bool} \wedge S_{i,int} \wedge S_{i,heap}$$

where $S_{i,bool}$, $S_{i,int}$, and $S_{i,heap}$ denote states in the boolean domain, the integer domain, and the heap domain, respectively. Similarly, a transition relation R

using the composite representation is defined as

$$R = \bigvee_{1 \le j \le m} R_{j,bool} \wedge R_{j,int} \wedge R_{j,heap}$$

where $R_{i,bool}$, $R_{i,int}$, and $R_{i,heap}$ denote transition relations in the boolean domain, the integer domain, and the heap domain, respectively. Given S and R, *Post*(S,R) is defined as

$$\begin{aligned} Post(S,R) \quad = \quad & \bigvee_{1 \le i \le n} \bigvee_{1 \le j \le m} Post(S_{i,bool}, S_{i,bool}) \wedge Post(S_{i,heap}, R_{j,heap}) \wedge \\ & Post(S_{i,int}, R_{j,int} \wedge IC(S_{i,heap}, R_{j,heap})) \end{aligned}$$

where IC is a function that generates the *interference constraint*, which is a linear arithmetic update formula that defines the next state values of the summary counts used in the extended shape graphs. The interference constraints are passed to linear arithmetic constraint representation in the composite framework and manipulated inside that representation. This has two main advantages: 1) Linear arithmetic representation, which is implemented by class `IntSym` in the Composite Symbolic Library, already provides heuristics for approximate fixpoint computations that can be used to guarantee convergence of the shape analysis for linear linked lists that have finite topology. 2) Handling the linear constraints on the integer variables of the linked list specification together with the linear constraints on the summary counts provide a more precise shape analysis that can also reason about the properties that involve both the integer variables and the structure of the heap.

Table 6.3 lists the interference constraints that are generated after the summarization, type-1 materialization and type-2 materialization operations. None denotes the case where neither summarization nor materialization is performed. k_{sm} denotes the number of nodes summarized by summary node sm, whereas

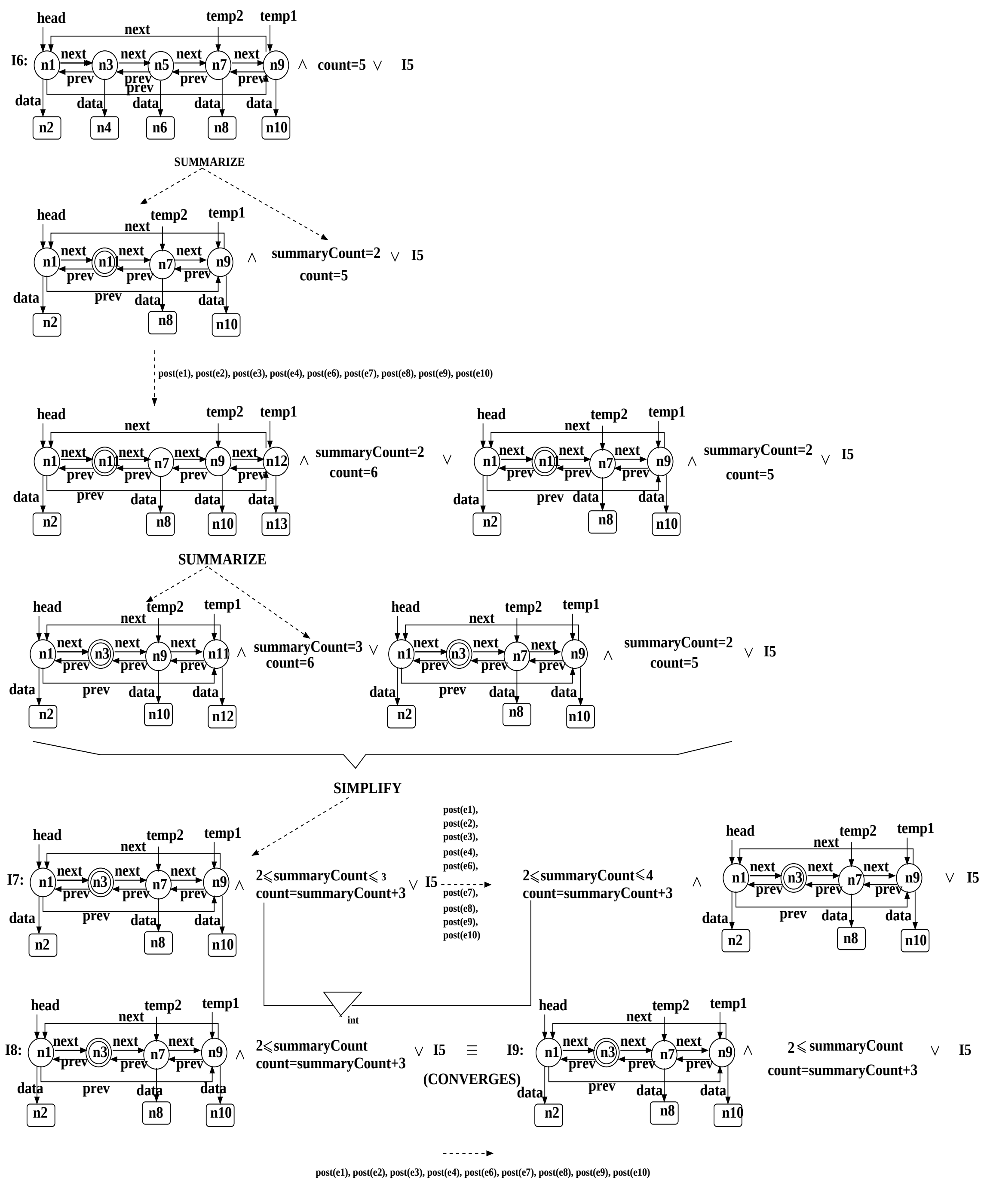
Operation	Interference Constraint
Summarization	$\bigwedge_{sm \in SG.SM} SC(sm)' = k_{sm}$
Type-1 Materialization	$SC(sm) = 2 * I \ \wedge \ SC(sm)' = 0$
Type-2 Left(Right)-Materialization	$SC(sm) > 2 * I \wedge$ $SC(sm)' = SC(sm) - 2 * I$
None	$\bigwedge_{sm \in SG.SM} SC(sm)' = SC(sm)$

Table 6.3: The operations and the corresponding interference constraints.

I is a constant and denotes the number of nodes each one-step match function matches for the pattern that the representation is based on.

Our technique for verifying invariant properties for linked lists works as follows. Using the post condition computation (*Post*) defined above, it computes an upper approximation to the set of reachable states, $RS \equiv \mu x \ . \ I \ \vee \ Post(x, R)$. Since computing the reachable states involves a least fixpoint, to compute an upper approximation for the fixpoint we use the widening operator defined in Section 4.2. Note that the widening operator is defined over linear arithmetic constraints and it not only approximates the reachable states for the integer variables of the linked list specification but also the state space for the heap since it is also applied over the linear arithmetic constraints involving the summary counts via interference constraints. After RS^+ is computed, it is checked whether all the states in RS^+ satisfies the invariant. If so then the property is verified, otherwise either the property is violated or the approximation is too coarse for the analysis. At this point one can either use truncated fixpoint computations (see Section 4.2) or ask the Action Language Verifier to generate a counter-example.

Figure 6.22 shows an example of the approximate reachable state computation for the `Enqueue` module given in Figure 6.3. I_5, I_6, I_7, I_8, and I_9 denote the result of the fixpoint iterations. The value of I_5 was given in Figure 6.5. Figure 6.22 shows only the reachable states when `pc=p1` and `lock=true`. In order to obtain



I6:
head
temp2
temp1
next
prev
data
n1
n2
n3
n4
n5
n6
n7
n8
n9
n10
n11
n12
n13
∧ count=5 ∨ I5
SUMMARIZE
summaryCount=2 count=5
post(e1), post(e2), post(e3), post(e4), post(e6), post(e7), post(e8), post(e9), post(e10)
summaryCount=2 count=6
summaryCount=3 count=6
SIMPLIFY
I7:
2⩽summaryCount⩽3 count=summaryCount+3
2⩽summaryCount⩽4 count=summaryCount+3
int
I8:
2⩽summaryCount count=summaryCount+3
≡
(CONVERGES)
I9:

Figure 6.22: The approximate reachable state computation for the `Enqueue` module given in Figure 6.3 when `pc=p1` and `lock=true`. I_5, I_6, I_7, I_8, and I_9 denote the result of the fixpoint iterations. I_5 is shown in Figure 6.5.

I_{j+1} from I_j, one needs to perform the post-condition computation on the actions `e1`, `e2`, `e3`, `e4`, `e6`, `e7`, `e8`, `e9`, and `e10`. Note that action `e5` is not enabled on the states represented by I_5-I_9. After the post-condition computation the Action Language Verifier performs summarization on the shape graphs, simplification on the disjuncts of the result of the same iteration, and widening on the pair of disjuncts from the results of the consecutive iterates that have subsumption relation between them. The linear arithmetic constraint $2 \leq summaryCount \wedge count = summaryCount + 3$ generated by the widening operation does not change in the following iteration, I_9. Since as a result of the summarization operation, all the generated shape graphs are isomorphic to the shape graphs generated in the previous iterations, the computation converges after computing I_9, i.e., $I_8 = I_9$.

6.3.1 Encoding Shape Graphs with BDDs

We encode the shape graphs symbolically using BDDs. We represent the nodes in the shape graph with minterms on a set of boolean variables (conjunctions of boolean variables or their negations). For example, the nodes of the shape graph in Fig. 6.6 can be represented using three boolean variables b_1, b_2, b_3, where the minterm $\neg b_1 \wedge \neg b_2 \wedge \neg b_3$ encodes node $n1$, the minterm $b_1 \wedge \neg b_2 \wedge \neg b_3$ encodes node $n2$, and so on. Then, function PT corresponds to a function from the heap variables to boolean logic formulas on the boolean variables b_1, b_2, and b_3. For example, $PT(head)$ is the boolean logic formula $\neg b_1 \wedge \neg b_2 \wedge \neg b_3$ for the shape graph in Fig. 6.6 based on the encoding given above.

Each selector defines a binary relation on N, the set of nodes in the shape graph. To encode binary relations defined by the selectors, we duplicate the boolean variables used to encode the nodes in the heap. For example for the above example, we add three new boolean variables b'_1, b'_2, b'_3. Now, the binary relation on N defined by the selector *data* can be represented as a boolean logic formula on variables b_1, b_2, b_3, b'_1, b'_2, and b'_3. The formula that corresponds to the edge between nodes n_1 and n_2 via selector data in the shape graph in Fig. 6.6 based on the encodings of the nodes given above is:

$$\neg b_1 \wedge \neg b_2 \wedge \neg b_3 \wedge b'_1 \wedge \neg b'_2 \wedge \neg b'_3$$

This type of encoding is essentially the same encoding used in BDD based model checking to encode the transition relations. Note that *null* corresponds to the boolean value $false$. Boolean value *true* on the other hand is used to encode the configurations where the selector for a node is not initialized.

To keep the size of the BDD encoding small we introduce boolean variables only when they are needed. In our encoding, a shape graph with the set of nodes N will be encoded with $2 \times \log_2(|N|)$ boolean variables. Since new nodes can be added to the heap using the *new* keyword, we introduce new boolean variables to the encoding dynamically if necessary.

6.4 Experiments

We have experimented with our verification technique that uses summarization patterns on eight different concurrent linked list specifications: the singly linked list (*singly*), the circular singly linked lists (*singlycircular*), the doubly linked list (*doubly*), the circular doubly linked list (*doublycircular*) and the circular and noncircular linked lists with an extra selector connecting each element directly to the last element, which yields four versions depending on whether there is data selector (*datalast* and *datalastcircular*) or not (*last* and *lastcircular*). Table 6.1

shows the summarization pattern corresponding to each of these linked lists. Table 6.5 shows the invariants that are verified for each linked list. The results of our experiments are shown in Table 6.4. Each item in Table 6.4 is the average of the result of verification of the specification for each of the invariants. We verified each specification with varying number of producer P and consumer C processes running concurrently. We also verified each specification with two different versions. In the first version, HC, we used heap formulas, e.g., $head = null$, in the guards of the actions, and in the second version, IC, we replaced heap formula with an integer formula, e.g., $count = 0$, in the guard of each action. For the results of the experiments, we reported the transition system construction time CT in seconds, verification time VT in seconds, and the memory usage M in Mbytes. We obtained the experimental results on a SUN ULTRA 10 workstation with 768 Mbytes of memory, running SunOs 5.7. Our results show that most of the time is spent during the verification phase. In most of the cases the verification with the version where the heap formula in the guard are replaced with the integer formula is comparable to the version where there is a heap formula in the guard.

6.5 Related Work

In [65] we presented a technique for verifying invariants of singly linked lists that involve both heap variables and integer variables. The technique presented in [65] is specialized for singly linked lists. This work extends the techniques presented in [65] by introducing the summarization patterns. The summarization patterns enable verification of linked lists such as doubly linked lists.

In [41] shape invariants are described using context-free graph grammars. Each update on a data structure is represented by a transformer that describes a single-

Table 6.4: Verification results for the concurrent linked list specifications.

Specification	Number of Processes	HC			IC		
		CT	VT	M	CT	VT	M
singly	1P-1C	0.29	2.54	24.10	0.26	2.64	23.82
	2P-2C	0.50	3.73	33.98	0.46	3.70	33.55
	4P-4C	1.02	5.88	54.80	0.97	6.45	53.68
doubly	1P-1C	0.31	4.40	32.97	0.31	4.49	32.78
	2P-2C	0.59	6.39	50.77	0.61	6.46	50.09
	4P-4C	1.52	15.38	88.56	1.57	15.52	87.04
last	1P-1C	0.30	9.63	44.10	0.27	9.63	43.98
	2P-2C	0.51	14.53	66.82	0.49	14.36	66.49
	4P-4C	1.20	32.98	115.42	1.16	33.00	114.71
datalast	1P-1C	0.32	17.95	58.32	0.32	18.39	58.23
	2P-2C	0.58	33.79	91.43	0.57	33.42	91.37
	4P-4C	1.37	69.81	162.28	1.37	70.49	161.60
singlycircular	1P-1C	0.32	20.44	68.53	0.31	18.76	55.21
	2P-2C	0.50	40.89	112.07	0.47	31.45	86.07
	4P-4C	0.95	85.76	202.04	0.95	62.35	150.33
doublycircular	1P-1C	0.37	86.60	115.93	0.40	69.31	91.26
	2P-2C	0.68	155.01	192.14	0.66	112.45	149.77
	4P-4C	1.50	371.62	351.30	1.48	246.76	269.76
lastcircular	1P-1C	0.35	46.41	75.88	0.34	39.82	82.79
	2P-2C	0.55	80.29	114.9	0.57	66.77	135.04
	4P-4C	1.10	148.77	200.35	1.14	150.31	244.36
datalastcircular	1P-1C	0.39	49.97	79.70	0.41	52.08	79.70
	2P-2C	0.69	81.21	127.53	0.69	82.29	127.31
	4P-4C	1.43	171.73	227.53	1.45	171.98	227.33

step rewriting. An algorithm that employs multiset rewriting techniques is proposed for checking the preservation of the shape invariant by each transformer in the program. This approach only enables checking the shape invariants and cannot handle checking the properties that should hold when the shape invariants are temporarily violated. Our technique, on the other hand, can verify properties about the state of the heap even when the shape invariants are temporarily violated.

[37] presents a representation for alias analysis for recursive data structures that is based on symbolic access paths. Symbolic access paths can be parame-

Table 6.5: Safety properties of the concurrent linked list specifications.

Specification	Verified Invariants
singly, *doubly*, *singlycircular*, *doublycircular*	$(count = 0 \wedge lock) \Rightarrow (head = null \wedge tail = null)$ $(head = null \wedge tail = null) \Rightarrow count = 0$ $head \neq null \Rightarrow count > 0$ $(head = tail \wedge head \neq null \wedge lock) \Rightarrow count = 1$ $(head \neq tail \wedge head \neq null \wedge lock) \Rightarrow count \geq 2$
singlycircular, *doublycircular*	$(head.next = tail \wedge head.next = head \wedge head \neq null \wedge lock) \Rightarrow count = 1$ $(head \neq null \wedge head.next = tail \wedge head.next \neq head \wedge lock) \Rightarrow count = 2$ $(head \neq null \wedge lock) \Rightarrow tail.next = head$ $(lock \wedge head.next \neq head) \Rightarrow tail.next \neq tail$
doublycircular	$(head.next = tail \wedge head.next \neq null \wedge head \neq tail \wedge lock) \Rightarrow$ $(count = 2 \wedge tail.prev = head \wedge tail.next = head \wedge head.prev = tail)$ $(head.next \neq null \wedge head.next = tail \wedge lock) \Rightarrow tail.prev = head$
last, *datalast*	$(count = 1 \wedge lock) \Rightarrow (head.next = null \wedge head.last = head)$ $(count = 2 \wedge lock) \Leftrightarrow (head.next = head.last)$ $(head.next \neq head.last \wedge lock) \Rightarrow count > 2$
lastcircular, *datalastcircular*	$(head.next = head \wedge head.last = head \wedge head.next \neq null \wedge lock) \Rightarrow$ $count = 1$ $(head.next = head.last \wedge head.next \neq null \wedge lock) \Rightarrow$ $(count = 1 \vee count = 2)$

terized on integer variables, which enables 1) encoding of all possible aliasing of two variables in a compact way 2) making use of widening operators to guarantee termination of the analysis. Our approach is similar to that of [37] since there exists an integer formula associated with every summary node in our shape graph representation and for termination of the fixpoint computation we also make use of the widening operator defined for linear arithmetic constraints. However, in [37] the integer formula only states a relation between the integer variables of two aliased symbolic access paths, whereas in our approach the summary counts can be related to the integer variables in the specification (e.g., variable *count* representing the number of nodes in the data structure) that has nothing to the with the symbolic representation of the heap. [37] only keeps aliasing information for recursive data structures that may have more than one field. On the other hand, our approach keeps the precise shape of the heap for linked lists so that programs with destructive updates can be verified.

[60] presents an approach for abstracting the heap state using shape graphs. The abstraction is achieved by representing a set of indistinguishable nodes by a single abstract node, which is called the summary node. As a result, a finite representation for a set of heap configurations is obtained. [61] extends the work in [60] to a more generalized framework that uses 3-valued logic. In this framework nodes of the shape graph represent equivalence classes defined by the truth values of the unary predicates. [39], [57] and [64] use the shape analysis framework presented in [61], for checking memory errors for programs that manipulate linked lists, for verifying properties of sorting algorithms that manipulate linked lists, for verifying invariants of concurrent programs that manipulate linked lists, respectively. Our approach has three main differences compared to the work on shape analysis described above: 1) Our summarization algorithm is parameterized with respect to the summarization patterns, and each summary node represents a set of nodes that match the summarization pattern. 2) We use the summary counts and the arithmetic constraints on the summary counts to propagate the information on the number of nodes represented by a summary node. This enables us to verify properties that relate the integer variables and the heap variables automatically. 3) We use a disjunctive composite symbolic representation that enables us to combine different type-specific symbolic representations such as BDDs, polyhedra, and the shape graphs.

Chapter 7

Conclusions and Future Work

In this dissertation we have presented a symbolic model checking tool, the Action Language Verifier, that can be used to verify the correctness of concurrent software specifications. The Action Language Verifier can analyze infinite-state transition systems using the infinite-state verification heuristics. As a result it can analyze systems with unbounded data domains and systems with an arbitrary number of concurrent processes. Although for some cases the analysis may remain inconclusive, our experience with various concurrent software specifications shows that one can verify or falsify many useful properties using the Action Language Verifier. Among the various concurrent system specifications is a case study on the airport ground traffic control.

The capability of the Action Language Verifier to analyze software specifications depends on the existence of suitable symbolic representations for encoding the state space of these specifications. We have built the Composite Symbolic Library, which is the symbolic manipulator used by the Action Language Verifier, to create an extensible framework under which different symbolic representations can be combined. The Composite Symbolic Library proved to be a useful tool: 1)

We could increase the range of software specifications that the Action Language Verifier can analyze by only extending the Composite Symbolic Library with new symbolic representations. 2) The Action Language Verifier dynamically chooses the symbolic representations it is going to use: if there is only one data type then it becomes a symbolic model checker based on the corresponding symbolic representation, otherwise, it uses the composite representation. This is due to the fact that the verification procedures are polymorphic regarding the symbolic representations. This feature helped us to implement new heuristics without any change to the implementation of the basic symbolic representations. 3) The composite approach used in the Composite Symbolic Library both inspired and enabled the development of a novel technique for analyzing invariant properties of concurrent linked list specifications. 4) The Composite Symbolic Library served as an experimental framework, specifically it has been used to compare polyhedra-based and automata-based linear arithmetic constraint representations.

We have proposed an approach for developing reliable concurrency controllers. Our approach consists of three steps: 1) The specification of a concurrency controller in the Action Language. 2) Automated verification of the specification for temporal correctness properties using the Action Language Verifier. 3) Automated synthesis of an efficient and correct-by-construction implementation of the concurrency controller in Java using the symbolic manipulation techniques provided by the Composite Symbolic Library. Considering the fact that model checking would not scale to whole program analysis, we think that this work serves as a good example of the cost-effective usage of the Action Language Verifier for reliable software development.

7.1 Future Work

The research problems that we are interested in exploring as future work can be summarized as follows:

Symbolic representations: Programming languages employ various data and control abstractions. Applicability of infinite-state symbolic model checking to software systems depends on the existence of symbolic representations that correspond to such abstractions. We would like to extend the Composite Symbolic Library with new symbolic representations for the abstractions such as arrays, strings, and unbounded channels.

Powerful Abstractions: Software may involve unbounded state spaces due to dynamic memory allocation, dynamic process creation, and so on. We need powerful abstraction techniques that will be precise enough to reason about the system while making the analysis feasible. There are two main approaches for employment of abstraction techniques in this regard. One approach reduces the state space of a system to a finite abstraction and performs the analysis on this reduced state space. Another approach performs the analysis on the concrete/original state space, however it computes an approximation to the result of the analysis. While the superiority of one approach over the other is controversial, we would like to explore ways of automatically choosing the right approach for a given system by analyzing the property to be verified.

Compositional Reasoning: Reasoning about correctness of large software systems requires employing compositional reasoning techniques, which infer the correctness of the whole system from the correctness of its components. One such technique is the assume-guarantee style of reasoning for concurrent systems, which involves significant human guidance for finding the environment assumptions/component guarantees. We think that we need techniques that will automatically infer environment assumptions by analyzing the behavior of individual components and the guarantees expected from them. Recently, there have been similar efforts at NASA Ames ASE Group [44] for automatically generating environment assumptions for invariant properties. We are interested to extend their approach to more general properties including liveness properties.

Specification languages: Currently, there are two main approaches for model checking software. The first one is to analyze a system on the source-code level. This requires employing various abstraction techniques in order to make the analysis feasible. The second one is to design an abstract model of the software system and specify it using the input specification language of a model checker. The problem with the former is that it is a kind of reverse engineering and the verification is employed in the very late stages of the software development process. The problem with the latter is that one would like to be able to automatically generate an implementation of the verified model. However, the fact that specification languages of model checkers are very low level compared to modern object oriented languages complicates the issue. We would like to explore ways for reducing the gap between the specification languages and modern programming languages.

APPENDIX

A-1 Action Language Specifications

```
module main()
  // number of airplanes occupying the runways
  integer numRW16R, numRW16L;
  // number of arriving airplanes occupying the taxiways C[3-8]
  integer numC3, numC4, numC5, numC6, numC7, numC8;
  // number of arriving airplanes occupying the taxiways
  // B[2,7,9,10,11]A
  integer numB2A, numB7A, numB9A, numB10A, numB11A;
  // runways are initialized as available
  initial: numRW16R=0 and numRW16L=0;
  // taxiways are initialized as available
  initial: numC3=0 and numC4=0 and numC5=0 and numC6=0 and numC7=0
           and numC8=0 and numB2A=0 and numB7A=0 and numB9A=0 and
           numB10A=0 and numB11A=0;
  // the state space is restricted to nonnegative values of the
  // integer variables
  restrict: numRW16R>=0 and numRW16L>=0 and numC3>=0 and numC4>=0
            and numC5>=0 and numC6>=0 and numC7>=0 and numC8>=0
            and numB2A>=0 and numB7A>=0 and numB9A>=0 and
            numB10A>=0 and numB11A>=0;
  module Departing() ... endmodule
  module Arriving() ... endmodule
  main: Arriving() | Departing();
  spec: AG(numRW16R<=1 and numRW16L<=1) //P1
  spec: AG(numC3>=0 and numC3<=1) // P2
  spec: AG((numRW16L=0 and numC3+numC4+numC5+numC6+numC7+numC8=0)
        => AX(numRW16L=0)) //P3
endmodule
```

Figure A.1: The Action Language Specification of the Airport Ground Traffic Control Case Study. The modules `Departing` and `Arriving` are given in Figures A.2 and A.3, respectively.

```
module Departing()
   // encodes the state of a departing airplane
   enumerated pc {parked, takeOff, depFlow};

   // a departing airplane is initially in arrival parked state
   initial: pc=parked;

   reqTakeOff : pc=parked and numRW16L=0 and numC3+numC4+numC5+
                numC6+numC7+numC8=0 and  pc'=takeOff and
                numRW16L'=numRW16L + 1;

   leave      : pc=takeOff and pc'=depFlow and
                numRW16L'=numRW16L - 1;

   Departing: reqTakeOff | leave ;

   // P4
   spec: AG(pc=parked => AF(pc=depFlow))
endmodule
```

Figure A.2: The module `Departing` of the Action Language Specification given in Figure A.1

A-2 Proofs for the Pattern-Based Representation

Proof of Theorem 6.1: We use proof by contradiction. Let us assume that node n can be mapped to both $M_1 = M_{max}(SG, P, n_1, m_1)$ and $M_2 = M_{max}(SG, P, n_2, m_2)$ where M_1 and M_2 are both defined and $M_1 \neq M_2$, i.e, $n \in M_1 \wedge n \in M_2 \wedge (n_1 \neq n_2 \vee m_1 \neq m_2)$. Since n_1 and n_2 are the entry points of the maximal sets M_1 and M_2, respectively n is reachable from both n_1 and n_2. We need to consider two main sub cases: 1) $n_1 \neq n_2$ and 2) $m_1 \neq m_2$.

1. $n_1 \neq n_2$: We should further consider two sub cases:

   (a) $n \neq n_1$ and $n \neq n_2$. There exists two nodes $n' \in M_1$ and $n'' \in M_2$

such that n is reachable from both n' and n'', however neither n' is in M_1 nor n'' is in M_2. However, this is not possible since this contradicts with constraint 4 for both M_1 and M_2.

(b) $n = n_1$ and $n \neq n_2$. There are two cases: 1) $M_1 \not\subset M_2$. This is not possible since it contradicts with constraint 5. 2) $M_1 \subset M_2$. This contradicts with constraint 1.

2. $m_1 \neq m_2$: This requires that $n_1 \neq n_2$, which is proved above.

Before giving the proof of Theorem 6.2 we think that presenting some definitions and lemmas are in order.

Definition A.1 *Given a shape graph $SG = (N, E, PT)$, a heap variable $v \in V$, and a node $s \in N$, s is reachable from v, $Reachable_{SG}(v, s) = true$, iff $\exists t \in N, PT(v) = t \wedge \exists (s_1, s_2, ..., s_k), k \geq 1 \wedge t = s_1 \wedge \forall i,\ 1 \leq i < k, (s_i, s_{i+1}) \in E \wedge s_k = s$.*

Definition A.2 *A given shape graph $SG = (N, E, PT)$ is k-well-formed with respect to a given summarization pattern $P = (L, Sel, N_{var}, PR)$, $WF_k(SG, P) = true$, where $k \in Nat$, iff $|N_{nwf}| = k$ where $N_{nwf} = \{n \mid n \in N \wedge \neg\exists s, t \in N, n \in M_{max}(P, SG, s, t)\}$.*

Lemma A.1 *In an abstract shape graph $\hat{SG} = (\hat{N}, \hat{E}, \hat{PT}, SM, \Theta)$ each summary node $sm \in SM$ has at least one incoming edge from a non-summary node.*

Proof: *From the definition of a shape graph, each node in a shape graph is reachable from a heap variable. From the definition of summary nodes, the nodes that are mapped to a summary node cannot be directly pointed by a heap variable. Therefore, for a summary node to be reachable from a heap variable it must have an incoming edge from a node that is reachable by that heap variable. It is not* *possible for a summary node to have incoming edges only from other summary nodes. Let us assume that the reverse is true. Then we have to consider two cases:*

1. *A summary node has an incoming edge from a single summary node and no incoming edge from a non-summary node. This case is not possible since summary nodes represent maximal matching sets.*

2. *A summary node sm has incoming edges from multiple summary nodes, e.g., n_1 and $n2$, and no incoming edge from a non-summary node. This requires sm to have multiple incoming edges with the same selector. This is because sm is tail of both n_1 and n_2. The type of edges between a summary node and a tail is defined by the summarization pattern. Therefore, sm cannot have incoming edges from n_1 and n_2 with different selectors. According to the definition of maximal matching sets, no node in a maximal matching set have two incoming edges with the same selector. Therefore, sm cannot have incoming edges with the same selector.*

Since we got contradictions for both of the two cases discussed above, there exists at least one incoming edge to a summary node from a non-summary node. ■

Lemma A.2 *In a shape graph $SG = (N, E, PT)$ that is k-well-formed with respect to a given summarization pattern $P = (L, Sel, N_{var}, PR)$ the number of summary nodes in the unique abstract shape graph $\hat{SG} = (\hat{N}, \hat{E}, \hat{PT}, SM, \Theta)$ corresponding to SG can have at most $k \times |Sel|$ summary nodes, i.e., $|SM| = k \times |Sel|$.*

Proof: From Definition 6.2 the number of nodes in SG that are not mapped to a summary node in $\hat{SG}$ is k. Hence the number of non-summary nodes in $\hat{SG}$ is k. Since Lemma A.1 states that each summary node must have at least one incoming edge from a non-summary node, given that there are k non-summary nodes and each node can have $|Sel|$ many outgoing edges the number of summary nodes can be at most $k \times |Sel|$. ■

Proof of Theorem 6.2: The number of nodes in an abstract shape graph is equal to the sum of the number of non-summary nodes and the number of summary nodes. A k-well-formed shape graph is mapped to an abstract shape graph with k non-summary nodes (from Definition A.2) and at most $k \times |Sel|$ summary nodes (from Lemma A.2). Hence the number of nodes in the corresponding abstract shape graph of a k-well-formed shape graph is $\leq k \times (|Sel| + 1)$.

```
module Arriving()
  enumerated pc {arFlow, touchDown, taxiTo16LC3, taxiTo16LC4,
  taxiTo16LC5, taxiTo16LC6, taxiTo16LC7, taxiTo16LC8, taxiFr16LB2,
  taxiFr16LB7, taxiFr16LB9, taxiFr16LB10, taxiFr16LB11, parked};
  initial: pc=arFlow;
  reqLand  : pc=arFlow and numRW16R=0 and pc'=touchDown and
             numRW16R'=numRW16R+1;
  exitRW3  : pc=touchDown and numC3=0 and pc'=taxiTo16LC3 and
             numRW16R'=numRW16R-1 and numC3'=numC3+1;
  exitRW4  : pc=touchDown and numC4=0 and pc'=taxiTo16LC4 and
             numRW16R'=numRW16R-1 and numC4'=numC4+1;
  exitRW5  : pc=touchDown and numC5=0 and pc'=taxiTo16LC5 and
             numRW16R'=numRW16R-1 and numC5'=numC5+1;
  exitRW6  : pc=touchDown and numC6=0 and pc'=taxiTo16LC6 and
             numRW16R'=numRW16R-1 and numC6'=numC6+1;
  exitRW7  : pc=touchDown and numC7=0 and pc'=taxiTo16LC7 and
             numRW16R'=numRW16R-1 and numC7'=numC7+1;
  exitRW8  : pc=touchDown and numC8=0 and pc'=taxiTo16LC8 and
             numRW16R'=numRW16R-1 and numC8'=numC8+1;
  crossRW3 : pc=taxiTo16LC3 and numRW16L=0 and numB2A'=numB2A+1
             and pc'=taxiFr16LB2 and numC3'=numC3-1 and numB2A=0;
  crossRW4 : pc=taxiTo16LC4 and numRW16L=0 and numB7A'=numB7A+1
             and pc'=taxiFr16LB7 and numC4'=numC4-1 and numB7A=0;
  crossRW5 : pc=taxiTo16LC5 and numRW16L=0 and numB9A'=numB9A+1
             and pc'=taxiFr16LB9 and numC5'=numC5-1 and numB9A=0;
  crossRW6 : pc=taxiTo16LC6 and numRW16L=0 and numB10A'=numB10A+1
             and pc'=taxiFr16LB10 and numC6'=numC6-1 and numB10A=0;
  crossRW7 : pc=taxiTo16LC7 and numRW16L=0 and numB10A'=numB10A+1
             and pc'=taxiFr16LB10 and numC7'=numC7-1 and numB10A=0;
  crossRW8 : pc=taxiTo16LC8 and numRW16L=0 and numB11A'=numB11A+1
             and pc'=taxiFr16LB11 and numC8'=numC8-1 and numB11A=0;
  park2    : pc=taxiFr16LB2 and pc'=parked and numB2A'=numB2A-1;
  park7    : pc=taxiFr16LB7 and pc'=parked and numB7A'=numB7A-1;
  park9    : pc=taxiFr16LB9 and pc'=parked and numB9A'=numB9A-1;
  park10   : pc=taxiFr16LB10 and pc'=parked and numB10A'=numB10A-1;
  park11   : pc=taxiFr16LB11 and pc'=parked and numB11A'=numB11A-1;
  Arriving : reqLand | exitRW3 |...| crossRW3 |...| park2 |...;
endmodule
```

Figure A.3: The module `Arriving` of the Action Language Specification given in Figure A.1

```
module main()
  enumerated p1,p2 {think, try, cs};
  integer  a,b;
  initial: a=0 and b=0 and p1=think and p2=think;
  restrict: a>=0 and b>=0;

  module process(y1,y2,pc)
    integer y1,y2;
    enumerated pc {think, try, cs};
    a1: pc=think and y2'=y1+1 and pc'=try;
    a2: pc=try && pc'=cs && (y1=0 || y2<y1);
    a3: pc=cs and y2'=0 and pc'=think;
    process : a1 | a2 | a3;
  endmodule

  main: process(a,b,p1) | process(b,a,p2);
  spec: invariant(!(p1=cs and p2=cs))
  spec: invariant(p1=try => eventually(p1=cs))
endmodule
```

Figure A.4: The specification of Bakery Mutual Exclusion Protocol for two processes in the Action Language.

```
module main()
  enumerated p1,p2 {think, try, cs};
  integer  t,s;
  initial: t=s and p1=think and p2=think;
  restrict: t>=0 and s>=0;

  module process(pc)
    enumerated pc {think, try, cs};
    integer a;
    a1: pc=think and a'=t and t'=t+1 and pc'=try;
    a2: pc=try and s>=a and pc'=cs;
    a3: pc=cs and s'=s+1 and pc'=think;
    process : a1 | a2 | a3;
  endmodule

  main: process(p1) | process(p2);
  spec: invariant(!(p1=cs and p2=cs))
  spec: invariant(p1=try => eventually(p1=cs))
endmodule
```

Figure A.5: The specification of Ticket Mutual Exclusion Protocol for two processes in the Action Language.

```
module main()
  integer barber, chair, open;
  initial: barber=0 and chair=0 and open=0;
  restrict:  barber>=0 and chair>=0 and open>=0;

  module customerP()
    enumerated pc {s1,s2};
    initial: pc=s1;
    a1: pc=s1 and barber>0 and barber'=barber-1 and chair'=chair+1
        and pc'=s2;
    a2: pc=s2 and open>0 and open'=open-1 and pc'=s1;
    customerP: a1 | a2;
  endmodule

  module barberP()
    enumerated pc {s1,s2,s3,s4};
    initial: pc=s1;
    a1: pc=s1 and barber'=barber+1 and pc'=s2;
    a2: pc=s2 and chair>0 and chair'=chair-1 and pc'=s3;
    a3: pc=s3 and open'=open+1 and pc'=s4;
    a4: pc=s4 and open=0 and pc'=s1;
    barberP: a1 | a2 | a3 | a4;
  endmodule

  main: customerP()  | customerP()  | barberP() ;

 spec:  invariant(chair<=1)
 spec:  invariant(open<=1)
 spec:  invariant(barber<=1)
endmodule
```

Figure A.6: The specification of solution to the Sleeping Barber Problem in the Action Language.

```
module main()
  boolean ex; parameterized integer n;
  enumerated state {Idle, ServeE, InvE, GrantE, ServeS, GrantS};
  integer xNull, xWaitS, xWaitE, xShared, xExclusive;
  initial: state=Idle and !ex and xNull>=1 and xShared=0 and
           xExclusive=0 and xWaitE=0 and xWaitS=0;
  restrict: xNull>=0 and xWaitS>=0 and xWaitE>=0 and xShared>=0 and
            xExclusive>=0 and n>=1;
  module Request()
    reqS:    state=Idle and xNull>=1 and state'=ServeS and
             xNull'=xNull-1 and xWaitS'=xWaitS+1;
    reqE1:   state=Idle and xNull>=1 and state'=ServeE and
             xNull'=xNull-1 and xWaitE'=xWaitE+1;
    reqE2:   state=Idle and xShared>=1 and state'=ServeE and
             xShared'=xShared-1 and xWaitE'=xWaitE+1;
    Request: reqS | reqE1 | reqE2;
  endmodule
  module Serve()
    inv:    state=ServeS and ex and xExclusive>=1 and state'=GrantS
            and xExclusive'=xExclusive-1 and xNull'=xNull+1 and !ex';
    nonex:   state=ServeS and !ex and state'=GrantS;
    invS:    state=ServeE and state'=InvE and xNull'=xNull+xShared
             and xShared'=0;
    invE:    state=InvE and ex and xExclusive >=1 and state'=GrantE
             and xNull'=xNull+1 and xExclusive'=xExclusive-1 and !ex';
    nonexE:  state=InvE and !ex and state'=GrantE;
    Serve :  inv | nonex | invS | invE | nonexE;
  endmodule
  module Grant()
    grantS:   state=GrantS and xWaitS>=1 and state'= Idle and
              xWaitS'=xWaitS-1 and xShared'=xShared+1;
    grantE:   state=GrantE and xWaitE>=1 and state'=Idle and
              xWaitE'=xWaitE-1 and xExclusive'=xExclusive+1 and ex';
    Grant :   grantS | grantE;
  endmodule
  main: Request() | Serve() | Grant();
  spec: AG(!((xShared>=1 and xExclusive>=1) or xExclusive>=2))
endmodule
```

Figure A.7: The specification of a Cache Coherence Protocol in the Action Language.

```
module main()
  enumerated state {Idle, ServeE, GrantE, ServeS, GrantS};
  boolean ex; parameterized integer n;
  integer xNull, xWaitS, xWaitE, xShared, xExclusive;
  initial: state=Idle and !ex and xNull>=1 and xShared=0 and
           xExclusive=0 and xWaitE=0 and xWaitS=0;
  restrict: n>=1 and xNull>=0 and xWaitS>=0 and xWaitE>=0 and
            xShared>=0 and xExclusive>=0;
  module Request()
    reqS:    state=Idle and xNull>=1 and state'=ServeS and
             xNull'=xNull-1 and xWaitS'=xWaitS+1;
    reqE1:   state=Idle and xNull>=1 and state'=ServeE and
             xNull'=xNull-1 and xWaitE'=xWaitE+1;
    reqE2:   state=Idle and xShared>=1 and state'=ServeE and
             xShared'=xShared-1 and xWaitE'=xWaitE+1;
    Request: reqS | reqE1 | reqE2;
  endmodule
  module Serve()
    inv:    state=ServeS and ex and xExclusive>=1 and state'=GrantS
            and xExclusive'=xExclusive-1 and xNull'=xNull+1 and !ex';
    nonex:   state=ServeS and !ex and state'=GrantS;
    invS:    state=ServeE and xShared>=1 and state'=ServeE and
             xNull'=xNull+1 and xShared'=xShared-1;
    invE:    state=ServeE and xExclusive>=1 and state'=ServeE and
             xNull'=xNull+1 and xExclusive'=xExclusive-1 and !ex';
    nonexE:  state=ServeE and xShared=0 and xExclusive=0 and
             state'=GrantE;
    Serve :  inv | nonex | invS | invE | nonexE;
  endmodule
  module Grant()
    grantS:   state=GrantS and xWaitS>=1 and state'= Idle and
              xWaitS'=xWaitS-1 and xShared'=xShared+1;
    grantE:   state=GrantE and xWaitE>=1 and state'=Idle and
              xWaitE'=xWaitE-1 and xExclusive'=xExclusive+1 and ex';
    Grant :   grantS | grantE;
  endmodule
  main: Request() | Serve() | Grant();
  spec: AG(!((xShared>=1 and xExclusive>=1) or xExclusive>=2))
endmodule
```

Figure A.8: The specification of a Cache Coherence Protocol in the Action Language.

```
module main()
   integer nr;
   boolean busy;
   restrict: nr >=0;
   initial: nr=0 and !busy;

   module reader()
      boolean reading;
      initial: !reading;
      r_enter: !reading and !busy and nr'=nr+1 and reading';
      r_exit: reading and !reading' and nr'=nr-1;
      reader: r_enter | r_exit;
   endmodule

   module writer()
      boolean writing;
      initial: !writing;
      w_enter: !writing and nr=0 and !busy and writing' and busy';
      w_exit: writing and !writing' and !busy';
      writer: w_enter | w_exit;
   endmodule

   main: reader() | writer();

  spec: invariant(busy => nr=0)
endmodule
```

Figure A.9: The specification of a solution to the Readers-Writers Problem in the Action Language.

```
module main()
  integer count,produced, consumed;
  parameterized integer size;
  initial: count=0 and size>=1;
  restrict: size>=1;

  module producer()
    initial: produced=0;
    producer: count<size and count'=count+1 and produced'=produced+1 ;
  endmodule

  module consumer()
    initial: consumed=0;
    consumer: count>0 and count'=count-1 and consumed'=consumed+1;
  endmodule


  main: producer() | consumer();

  spec: invariant(produced-consumed=count and count<=size)
endmodule
```

Figure A.10: The specification of a solution to the Bounded-Buffer Producer-Consumer Problem in the Action Language.

```
module main()
  integer i, k;
  parameterized integer n;
  enumerated pc {init, for, entryA1, while, entryA2, entryA3, end};
  initial: pc = init;

  a1: pc=init and pc'=for and k'=1;
  a2: pc=for and pc'=entryA1 and k<=n-1;
  a3: pc=entryA1 and pc'=while and i'=k-1;
  a4: pc=while and pc'=entryA2 and i>=0;
  a5: pc=entryA2 and pc'=while and i'=i-1;
  a6: pc=while and pc'=entryA3;
  a7: pc=entryA3 and pc'=for and k'=k+1;
  a8: pc=for and pc'=end and k>=n;

  main:  a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8 ;

  spec: AG(!((pc=entryA1 and k>=n) or (pc=entryA1 and k<=-1)
          or (pc=entryA3 and i>=n-1) or (pc=entryA3 and i<=-2)
          or (pc=entryA2 and i<=-1) or (pc=entryA2 and i<=-2)
          or (pc=entryA2 and i>=n-1) or (pc=entryA2 and i>=n-1)))
endmodule
```

Figure A.11: The specification of a Insertion Sort for Array Bounds Checking in the Action Language.

```
module main()
  enumerated Office {Empty, Occupied};
  enumerated Occupants {One, Multiple};
  enumerated Light {On, Off};
  boolean enter, exit, turn_on, turn_off; integer count;
  initial : count=0 and Office=Empty;
  restrict: count >= 0
            and (enter implies !(exit or turn_on or turn_off))
            and (exit implies !(enter or turn_on or turn_off))
            and (turn_on implies !(enter or exit or turn_off))
            and (turn_off implies !(enter or exit or turn_on));

  t1 : Office=Empty and enter and Office'=Occupied and
       Occupants'=One and Light'=On and count'=count+1;
  t2 : Office=Occupied and Occupants=One and exit and Office'=Empty
       and count'=count-1;
  t3 : Office=Occupied and Occupants=One and enter and
       count'=count+1 and Occupants'=Multiple;
  t4 : Office=Occupied and Occupants=Multiple and count=2 and exit
       and Occupants'=One and count'=count-1;
  t5 : Office=Occupied and Occupants=Multiple and count>2 and exit
       and count'=count-1;
  t6 : Office=Occupied and Occupants=Multiple and enter
       and count'=count+1;
  t7 : Office=Occupied and Light=On and turn_off
       and Occupants=One and Light'=Off;
  t8 : Office=Occupied and Light=Off and turn_on and Light'=On;
  t9 : Office=Occupied and Light=Off and enter and Light'=On;
  environment: (enter implies !enter') and (exit implies !exit')
               and (turn_on implies !turn_on') and
               (turn_off implies !turn_off');

  main : (t1 | t2  | ((t3 | t4 | t5 | t6) & (t7 | t8 | t9))) &
         environment;

 spec: AG(count>1 iff Office=Occupied and Occupants=Multiple)
endmodule
```

Figure A.12: The specification of an Office Light Control System in the Action Language.

```
module main()
  enumerated Pressure {TooLow, Permitted, TooHigh};
  boolean Overridden, Block, Reset, Inject;
  parameterized integer low;
  parameterized integer permit;
  integer WaterPres;

  initial: !Block and !Reset and !Inject and !Overridden and
           Pressure=Permitted and low<=WaterPres  and
           WaterPres<permit;
  restrict: low<permit;

  p1 : Pressure=TooLow and Pressure'=Permitted and WaterPres'>=low
       and WaterPres'<permit and !(WaterPres>=low and
       WaterPres<permit);
  p2 : Pressure=Permitted and Pressure'=TooHigh and
       WaterPres'>=permit and !(WaterPres>=permit);
  p3 : Pressure=Permitted and Pressure'=TooLow and WaterPres'<low
       and !(WaterPres<low);
  p4 : Pressure=TooHigh and Pressure'=Permitted and WaterPres'>=low
       and WaterPres'<permit and !(WaterPres>=low and
       WaterPres<permit);
  o1 : Overridden' and Block' and !Block and !Reset and
       (Pressure=TooLow or Pressure=Permitted);
  o2 : !Overridden' and ( !(Pressure'=Pressure) or Reset' and
       !Reset and (Pressure=TooLow or Pressure=Permitted));
  s :  (Inject and Pressure=TooLow and !Overridden or !Inject and
       (Pressure=TooHigh or Pressure=Permitted or Pressure=TooLow
       and Overridden)) and (Inject' and Pressure'=TooLow and
       !Overridden' or !Inject' and (Pressure'=TooHigh or
       Pressure'=Permitted or Pressure'=TooLow and Overridden'));

  main : (p1 | p2 | p3 | p4) & (o1 | o2)  & s;

 spec: invariant(Inject implies Pressure=TooLow)
endmodule
```

Figure A.13: The specification of a Safety Injection System in the Action Language.

```
module main()
  heap head, tail, add {next}; boolean lock; integer count;
  initial: count=0 and lock and head=null and tail=null;
  module put()
     enumerated pc {p1, p2, p3, p4, p5};
     initial: pc=p1;
     ap1: pc=p1 and lock and  !lock' and add'=new and pc'=p2;
     ap2: pc=p2 and add'->next=null and pc'=p3;
     ap3: pc=p3 and tail=null and tail'=add and pc'=p4;
     ap4: pc=p4 and head'=add and count'=1 and lock' and pc'=p1;
     ap5: pc=p3 and tail!=null and tail'->next=add and pc'=p5;
     ap6: pc=p5 and tail'=add and count'=count+1 and lock' and
          pc'=p1;
     put: ap1 | ap2 | ap3 | ap4 | ap5 | ap6  ;
  endmodule
  module take()
     enumerated pc {p1, p2};
     initial: pc=p1;
     at1: pc=p1 and lock and head!=null and !lock' and
          head'=head->next and pc'=p2;
     at2: pc=p2 and head=null and tail'=null and lock' and
          count'=count-1 and pc'=p1;
     at3: pc=p2 and head!=null and lock' and count'=count-1
          and pc'=p1;
     take: at1 | at2 | at3 ;
  endmodule
  main: put() | take();
  spec: AG((count=0 and lock)  implies
           (head=null and tail=null))
  spec: AG((head != null and lock) implies count>0)
  spec: AG((head =tail and head != null and lock) implies
           count=1)
  spec: AG((head != tail and head != null and lock) implies
           count>=2)
endmodule
```

Figure A.14: The specification of a queue using a singly linked list in the Action Language.

```
module main()
  heap head, tail, add {next,prev}; boolean lock; integer count;
  initial: count=0 and lock and head=null and tail=null;
  module put()
     enumerated pc {p1, p2, p3, p4, p5, p6, p7};
     initial: pc=p1;
     ap1: pc=p1 and lock and  !lock' and add'=new and pc'=p2;
     ap2: pc=p2 and add'->next=null and pc'=p3;
     ap3: pc=p3 and add'->prev=null and pc'=p4;
     ap4: pc=p4 and tail=null and tail'=add and pc'=p5;
     ap5: pc=p5 and head'=add and count'=1 and lock' and pc'=p1;
     ap6: pc=p4 and tail!=null and tail'->next=add and pc'=p6;
     ap7: pc=p6 and add'->prev=tail and pc'=p7;
     ap8: pc=p7 and tail'=add and count'=count+1 and lock' and
          pc'=p1;
     put: ap1 | ap2 | ap3 | ap4 | ap5 | ap6 | ap7 | ap8;
  endmodule
  module take()
     enumerated pc {p1, p2, p3};
     initial: pc=p1;
     at1: pc=p1 and lock and head!=null and !lock' and
          head'=head->next and pc'=p2;
     at2: pc=p2 and head=null and count'=count-1 and pc'=p3;
     at3: pc=p2 and head!=null and head'->prev=null and
          count'=count-1 and pc'=p3;
     at4: pc=p3 and head=null and tail'=null and lock' and pc'=p1;
     at5: pc=p3 and head!=null and lock'=true and pc'=p1;
     take: at1 | at2 | at3 | at4 | at5;
  endmodule
  main: put() | take();

  spec: AG((count=0 and lock)  => (head=null and tail=null))
  spec: AG(head != null => count>0)
  spec: AG((head =tail and head != null and lock) => count=1 )
  spec: AG((head != tail and head != null and lock) => count>=2)
endmodule
```

Figure A.15: The specification of a queue using a doubly linked list in the Action Language.

```
module main()
  heap head, add, temp1 {next,last}; boolean lock; integer count;
  initial: count=0 and lock and head=null and add=null;
  module put()
    enumerated pc {p1, p2, p3, p4, p5, p6, p7};
    initial: pc=p1;
    ap1: pc=p1 and lock and !lock' and add'=new and pc'=p2;
    ap2: pc=p2 and add'->next=null and pc'=p3;
    ap3: pc=p3 and add'->last=add and pc'=p4;
    ap4: pc=p4 and head=null and head'=add and count'=count+1 and
         lock' and pc'=p1;
    ap5: pc=p4 and head!=null and add'->next=head
         and pc'=p5;
    ap6: pc=p5 and temp1'=head->last and pc'=p6;
    ap7: pc=p6 and add'->last=temp1 and pc'=p7;
    ap8: pc=p7 and head'=add and count'=count+1 and lock' and
         pc'=p1;
    put: ap1 | ap2 | ap3 | ap4 | ap5 | ap6 | ap7 | ap8;
  endmodule
  module take()
    enumerated pc {p1, p2, p3, p4};
    initial: pc=p1;
    at1: pc=p1 and lock and !lock' and head!=null and
         add'=head->next and pc'=p2;
    at2: pc=p2 and head'->next=null and pc'=p3;
    at3: pc=p3 and head'->last=null and pc'=p4;
    at4: pc=p4 and head'=add and count'=count-1 and lock'
         and pc'=p1;
    take : at1 | at2 | at3 | at4;
  endmodule
  main: put() | take();
  spec: AG((head->next=null and head->last=head and lock) implies
           count=1)
  spec: AG((head->next=head->last and head->next!=null and lock)
           implies count=2)
  spec: AG((head->next!=null and lock) implies head->last!=null)
endmodule
```

Figure A.16: The specification of a queue using a linked list with connection to the last element in the Action Language.

```
module main()
  heap head, add, temp1 {next,last,data};
  boolean lock; integer count;
  initial: count=0 and lock and head=null and add=null;
  module put()
    enumerated pc {p1, p2, p3, p4, p5, p6, p7, p8}; initial: pc=p1;
    ap1: pc=p1 and lock and !lock' and add'=new and pc'=p2;
    ap2: pc=p2 and add'->next=null and pc'=p3;
    ap3: pc=p3 and add'->last=add and pc'=p4;
    ap4: pc=p4 and add'->data=new and pc'=p5;
    ap5: pc=p5 and head=null and head'=add and count'=count+2 and
         lock' and pc'=p1;
    ap6: pc=p5 and head!=null and add'->next=head
         and pc'=p6;
    ap7: pc=p6 and temp1'=head->last and pc'=p7;
    ap8: pc=p7 and add'->last=temp1 and pc'=p8;
    ap9: pc=p8 and head'=add and count'=count+2 and lock' and
         pc'=p1;
    put: ap1 | ap2 | ap3 | ap4 | ap5 | ap6 | ap7 | ap8 | ap9;
  endmodule
  module take()
    enumerated pc {p1, p2, p3, p4, p5}; initial: pc=p1;
    at1: pc=p1 and lock and !lock' and head!=null and
         add'=head->next and pc'=p2;
    at2: pc=p2 and head'->next=null and pc'=p3;
    at3: pc=p3 and head'->last=null and pc'=p4;
    at4: pc=p4 and head'->data=null and pc'=p5;
    at5: pc=p5 and head'=add and count'=count-2 and lock' and
         pc'=p1;
    take : at1 | at2 | at3 | at4 | at5 ;
  endmodule
  main: put() | take();
  spec: AG((head->next=null and head->last=head and lock)
          implies count=2)
  spec: AG((head->next=head->last and head->next!=null and lock)
          implies count=4)
endmodule
```

Figure A.17: The specification of a queue using a linked list with data and connection to the last element in the Action Language.

Bibliography

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

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

[3] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[4] G. R. Andrews. *Concurrent programming, Principles and Practice.* Benjamin/Cummings Publishing Co., 1991.

[5] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI 2001*, 2001.

[6] B. Bartzis and T. Bultan. Efficient image computation in infinite state model checking. In *Proc. of the 15th International Conference on the Computer Aided Verification (CAV 2003)*, 2003.

[7] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueb, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *Proceedings of the Fifth Langley Formal Methods Workshop*, June 2000.

[8] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with bdds for automatic invariant checking. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 378–394. Springer, April 2000.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[10] G. Brat, K. Havelund, S. Park, and W. Visser. Java path finder: Second generation of a java model checker. In *Proc. of Workshop on Advances in Verification*, 2000.

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

[12] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[20] T. Cargill. Specific notification for Java thread synchronization. In *International Conference on Pattern Languages of Programming*, 1996.

[21] James C.Corbett, Matthew B.Dwyer, John Hatcliff, Shawn Laubach, Corina S.Pasarenau, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Int. Conf. on Soft. Eng. (ICSE)*, 2000.

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

[23] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327. Springer, June 1997.

[24] 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.

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

[26] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, 2000.

[27] P. J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323, May 1993.

[28] 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.

[29] 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.

[30] O. J. Dahl, E. W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, New York, 1972.

[31] 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.

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

[33] G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001.

[34] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Journal of Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[35] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software-Practice and Experience*, 29(7):577–603, 1999.

[36] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, 2002.

[37] A. Deutsch. Interprocedureal may-alis analysis for pointers: Beyond k-limiting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[38] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware deign aid. In *Proc. of IEEE International Conference on Computer Design: VLSI in computers and processors*, pages 522–525, 1992.

[39] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In Jens Palsberg, editor, *7th International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 200.

[40] Matthew B. Dwyer, John Hatcliff, and Hongjun Zheng. Slicing software for model construction. In *ACM SIGPLAN Partial Evaluation and Program Manipulation*, January 1999.

[41] P. Fradet and D. L. Metayer. Shape types. In *ACM SIGPLAN Conference on Principles of Programming Languages (POPL'97)*, 1997.

[42] X. Fu, T. Bultan, and J. Su. Formal verification of e-services and workflows. In *Proc. of the Workshop on Web Services, e-Bussiness, and the Semantic Web: Foundations, Models, Architectures, Engineering and Applications (WES 2002)*, 2002.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1994.

[44] D. Giannakopoulou, C. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, 2002.

[45] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Proceedings of computer aided verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer-Verlag, 1993.

[46] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science.* Springer-Verlag, September 1994.

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

[48] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[49] 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.

[50] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

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

[52] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2), 1996.

[53] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murphi. In *Proc. of International Conference on Computer Aided Verification*, 1996.

[54] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wo nnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.

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

[56] Doug Lea. *Concurrent Programming in Java, Design Principles and Java.* Sun Microsystems, 1999.

[57] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing And Analysis*, 2000.

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

[59] Masaaki Mizuno. A structured approach for developing concurrent programs in Java. *Information Processing Letters*, 1999.

[60] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[61] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *26th ACM Symposium on Principles of Programming Languages*, 1999.

[62] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence*, 8:315–343, 1993.

[63] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[64] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. of POPL'01*, January 2001.

[65] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *9th International Static Analysis Symposium*, 2002.

[66] T. Yavuz-Kahveci and T. Bultan. Heuristics for efficient manipulation of composite constraints. In Alessandro Armando, editor, *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCos 2002)*, volume LNAI 2309, pages 57–71. Springer, 2002.

[67] 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.

[68] 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)*, to appear 2003.

[69] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representation. In *Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, 2001.

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