New University of Lisbon
Faculty of Science and Technology
Departament of Informatics

Master's Thesis

European Master's program in Computational Logic

Constraint-based Verification of Imperative Programs

Tewodros Awgichew Beyene

Lisbon
(October, 2011)

New University of Lisbon
Faculty of Science and Technology
Departament of Informatics

Master's Thesis

Constraint-based Verification of Imperative Programs

Tewodros Awgichew Beyene (35031)

Supervisor: Professor Pedro Barahona

work presented in the context of the European Master's program in Computational Logic, as the partial requirement for obtaining Master of Science degree in Computational Logic

Lisbon
(October, 2011)

Acknowledgements

I would like to thank Professor Pedro Barahona, my supervisor, for accepting me to work on this very interesting and motivating topic with him. He has been not only providing me with invaluable ideas through out the work but also being available to help me every time I have challenges and difficulties.

Julio Marino's and Manuel Carro's lectures on rigorous software development at the Polytechnical University of Madrid (UPM) provided me the first exposure to modern software verification tools. I would like to thank both professors for first introducing me to core concepts of model checking, theorem proving, program logics, and computer aided formal verification.

I also like to thank my father Awgichew Beyene and my mother Almaz Lemma for paying all the sacrifices to make me reach where I am today. I extend my regards to all of my brothers and sisters, family members and friends who have been encouraging me, supporting me, and more importantly praying for me. I would like to give special thanks to Edengenet Mashilla, my fiancée, who has been a great help not only during my masters study but also during the three years of my life abroad. God bless you all!

FCT has been a very enjoyable place to study. I am grateful to the members of the faculty and fellow students, specially to those in the department of informatics, for enabling such an environment where both pursuing of academic goals and having fun go together.

But above all, I glorify God for His ever lasting love, forgiveness and protection upon me. I thank Jesus for providing me with such helpful professors, caring family, and supportive friends. Thank You God!

Abstract

The continuous reduction in the cost of computing ever since the first days of computers has resulted in the ubiquity of computing systems today; there is no any sphere of life in the daily routine of human beings that is not directly or indirectly influenced by computer systems anymore. But this high reliance on computers has not come without a risk to the society or a challenge to computer scientists. As many computer systems of today are safety critical, it is crucial for computer scientists to make sure that computer systems, both the hardware and software components, behave correctly under all circumstances. In this study, we are interested in techniques of program verification that are aimed at ensuring the correctness of the software component.

In this work, constraint programming techniques are used to device a program verification framework where constraint solvers play the role of typical verification tools. The programs considered are written in some subset of Java, and their specifications are written in some subset of Java Modeling Language(JML). In our framework, the program verification process has two principal steps: constraint generation and constraint solving. A program together with its specification is first parsed into a system of constraints. And then, the system of constraints is processed using constraint solvers so that the correctness of the original program is proved to hold, or not, based on the outcome of the constraint solving. The performance of our framework is compared with other well-known program verification tools using standard benchmarks, and our framework has performed quite well for most of the cases.

Keywords: Program Verification, Model Checking, Constraint Programming

Contents

List of Figures

List of Tables

1

Introduction

The continuous reduction in the cost of computing ever since the first days of computers has resulted in the ubiquity of computing systems in today's world; there is no any sphere of life in the daily routine of human beings in the 21^{st} century that is not directly or indirectly influenced by computer systems. Some of the areas where computer systems have become a very important and integral part of their existence include banking, production line control, air traffic control and transportation, etc. But this high reliance of the society on computer systems has not come without a higher risk to today's society or without a stronger challenge to today's computer scientists. Since many computer systems of today are safety critical in a sense that failure of such systems can cause a catastrophic loss to the society in terms of not only money and time but also invaluable human life, it is crucial for computer scientists to make sure that computer systems behave correctly under all circumstances. For the whole computer system to behave correctly, it must be ensured that both the hardware and software components behave correctly. However, in this study, we will be interested only in the software component of computer systems, and we deal with techniques of program verification that are aimed at ensuring the correctness of the software component. Though most modern computer systems consist of sophisticated hardware and software components, ensuring the correctness of their software component is often more challenging than that of their underlying hardware component.

In this work, constraint programming techniques are used for program verification with constraint solvers playing the role typical verification tools play in typical program verification set ups. The programs considered are written in some subset of the Java programming language, and their preconditions and postconditions are written in some subset of the Java Modeling Language(JML). The program verification process has two principal steps: constraint generation and constraint solving. A program together with its precondition and postcondition is first parsed into a system of constraints. Then, the system of constraints is processed using constraint solvers so that the correctness of the original program is proved to hold, or not, based on the outcome of the constraint solving.

The rest of this document is structured in this way: In Chapter 2, we introduce the notions of program verification and program correctness followed by a detailed discussion of some of the major program verification methods. In addition, the chapter includes a brief discussion of constraint programming and constraint solving techniques. In Chapter 3, the subsets of Java and JML considered in this work for writing the program and its specification respectively are given. The chapter starts by discussing briefly the scope of the work together with the strategy employed to verify a given program. This is followed by the discussion of how each of the constructs in Java and JML is transformed into an equivalent constraint so that a semantically equivalent system of constraints is generated for some program given in the subset of Java whose specification given in the subset of JML. In Chapter 4, the constraint solving models corresponding to the program to be verified are discussed. The efficiency of the entire program verification process is highly dependent on the efficiency of the constraint solving. In Chapter 5, the experimental results of our approach are presented in comparison with other commonly used program verification tools and benchmarks using some standard benchmark programs. In Chapter 6, we conclude by providing a summary and directions of future work.

2

Review of the State of the Art

2.1 Introduction

A computer program can be considered as a mathematical object whose properties can be formally specified so that mathematical proofs can be done on the program to check whether the program satisfies a given set of properties or not. Program verification deals with ensuring that programs satisfy the given set of properties. In this work, constraint programming techniques are used for program verification with constraint solvers playing the role typical verification tools would play in typical program verification set ups.

This chapter deals with two core concepts used in the work; Program Verification and Constraint Programming. The rest of this chapter is structured in this way: The next section of this chapter addresses program verification in general. A brief overview of program verification will be given in section 2.2 followed by a detailed discussion on correctness of a program in section 2.2.1 where the different properties that make a program correct are given. Although formal verification has enjoyed several recent successes in proving correctness of industry-scale hardware and software systems, there were mountainous obstacles to program verification in its early stage and still there are several objections that are raised against it. In section 2.2.2, some of these issues are discussed along with how program verification techniques in general have tried to answer the questions posed by the issues. In section 2.2.3, a detailed discussion for some of the most important approaches to program verification is given. The last part of the chapter discusses constraint programming in some detail whose techniques are used for program verification in this work. Section 2.3.1 provides a formal definition as well as illustration of constraint satisfaction problems (CSPs) followed by a brief coverage of basic constraint solving techniques used by constraint solvers in section 2.3.2.

2.2 Program Verification

Program verification is the use of formal techniques to ensure that programs satisfy a set of formally specified correctness properties. As computer programs are becoming part of more and more systems that we depend on for our daily lives, the need for efficient and effective program verification techniques that can be scaled up to industry level programs has been increasing. Most of the methods being commonly practiced today to ensure correctness properties for programs involve simulation of the expected properties of programs, and testing using a set of "critical" test data. There are a few serious limitations to this approach. One limitation is the fact that simulation of today's complex systems on all possible input sequences in any reasonable time is not possible. Another limitation is the fact that "critical" test data is a very vague expression that lacks formal definition to be of any practical use. Furthermore, whereas simulation and testing methods are very good in detecting well-defined types of errors, they may fail to catch elusive design faults that may make the system to behave unexpectedly only under a particular set of conditions.

In program verification methods, one models a program in a mathematical logic under some well defined theory, and formally proves that the program satisfies its desired specifications. Program verification techniques assume that the input program is syntactically correct - missing semicolons, parenthesis and the like are not the issues of program verification - and their concern will be in detecting semantic difference between the program and its specification.

2.2.1 Correctness of a Program

The correctness of a given program is specified in terms of the desirable properties it needs to satisfy. The main properties used to define correctness of a given program are given below [1]:

1. **Partial correctness :**
   A program is said to satisfy partial correctness if and only if whenever there is a result produced by the program, the result is correct with respect to the task to be solved by the program. For example, upon termination of a bubble sort program,

the input should indeed be sorted. It is called *partial* because the property is defined on the implicit condition of termination; the program is not guaranteed to terminate which means that it is not guaranteed to deliver any result at all in some situations.

2. **Termination :**
   A program is said to terminate if and only if it finishes its computation and exits normally or aborts its computation and exits abnormally after a finite amount of time. The problem of determining whether a given program will always finish execution or will keep on running forever is called the program termination problem or the uniform halting problem. Although it roots back to the era of Hilbert's decision problem before the invention of computers, and was proven not to be decidable, the program termination problem was one of the most actively researched and debated areas of theoretical computer science [2].

3. **Absence of failures :**
   Failures in a given program are caused by violations of the operational semantics governing one or more operations in the program or programming language constructs used in the program. Some common causes of program failures include division by zero, trying to access an array out of its range, and stack overflow. One desirable property of programs is the absence of any such cause of failure.

4. **Interference freedom :**
   It refers to the property that none of the simultaneously running components of a given concurrent program can modify the variables shared with another component in such a way that the change is undesirable for the other component.

5. **Deadlock freedom :**
   A concurrent program is said to be free from deadlock if and only if it does not end up in a situation where one or more of the non-terminated components of the program are waiting indefinitely for a condition becoming true.

6. **Correctness under fairness assumption :**
   Fairness usually means that a particular choice is taken sufficiently often provided that it is sufficiently often possible [3]. Sometimes it is important to use fairness assumptions on the environment the program works in. For example, we may assume that a particular scheduler in our environment never ignores some process forever, and it will eventually schedules the process. A concurrent program is said to be correct under the fairness assumption if and only if whenever a given component of of the program needs some resource, it is not the case

that it will be denied access indefinitely.

The first three properties of correct programs apply for both sequential and concurrent programs whereas the remaining three properties apply only for concurrent programs. In general, ensuring any one of these properties is more challenging for concurrent programs than that of sequential programs. Since this work deals with techniques for verification of partial correctness for a subset of sequential Java programs, this review of the state of the art for program verification will be more focused on approaches to verification of partial correctness for sequential programs.

2.2.2 Earlier Issues in Program Verification

Although formal verification has enjoyed several recent successes in proving correctness of industry-scale hardware and software systems, there were mountainous obstacles to program verification in its early stage and still there are several objections that are raised against it.

The first challenge is based on the fact that many formal theories are undecidable or they need algorithms with super-exponential time bounds to be decidable. This leads to the conclusion that verification techniques that are based on mechanical theorem proving on such theories may keep on running forever.

The second challenge regards the specification of the desired properties of programs. To verify that a program has a certain property, the property must be specified using some formal language. Here an important question arises: can this formal specification of the program be any simpler than the program itself? If not, verification may end up verifying whether a complicated object representing the program is consistent or not, in some complicated technical sense, with another complicated object representing the set of specifications. This can not lead the verification process to correctness and reliability, but instead to a multiplication of the possibility for errors. A more fruitful answer for this challenge was the emergence of specification languages that are solely designed and used for the purpose of clearly specifying the result of computations in programs. These specifications languages are completely different from the programming ones because they are not required to be efficiently executable (or even executable at all).

The third challenge is an important one raised by Richard De Millo, Richard Lipton, and Alan Perlis in their article[4]. The article argues that program verifications are to programming as "imaginary formal demonstrations" are to mathematics, and they have no role in practice. The authors also quote "..if it requires 27 equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?" In other words, this comment of suspicion on program verification is based on a questionable view of what, if any, the role of formal methods can be in mathematics and computer science.

2.2.3 Program Verification Methods

Formal verification of programs entails a mathematical proof showing that the program satisfies its desired set of properties which should be formally specified. This requires some method for mathematically modeling the program and deriving its desired set of properties as theorems. The main difference among the various formal verification approaches comes from the choice of the mathematical formalism used in the process of modeling a program and deriving its set of desired properties.

2.2.3.1 1. Program Logics - Hoare Logic

Specifying the semantics of a program in terms of the effects of its instructions on the states of the underlying machine is called an operational approach to modeling the program, and the resulting semantics is called operational semantics [5]. Operational semantics form the basis for program verification using methods such as model checking, theorem proving, etc. However, the need to reason about the underlying machine in the operational approach makes it cumbersome to prove correctness of a program. Program logics on the other hand focus on simplifying program verification by factoring out the details of the machine executing the program from the verification process. The goal in program logics is to deal with the program text itself as a mathematical object [6, 7].

In program logics, each instruction of the program is considered as performing a transformation of predicates. For some sequence of instructions I in a given programming language, the axiomatic semantics of the programming language are specified by a collection of formulas of the form $\{P\}I\{Q\}$, where P and Q are first order predicates over the program variables. Such a formula can be read as: "If P holds for the state of the machine when the program is poised to execute I, then, after the execution of I, Q holds." Predicates P and Q are called the precondition and postcondition for I respectively.

For example, if I is a single instruction specifying an assignment statement $x := a$, then its axiomatic semantics is given by the following schema, also known as the $Axiom\ of\ Assignment$.

Axiom of Assignment:

$\{P\}x := a\{Q\}$ holds if Q is obtained by replacing every occurrence of the variable x in P by a.
Hoare [103] provides five such schema, and an inference rule often called $rule\ of\ composition$ to specify the semantics of a simple programming language.
Rule of Composition:
Infer $\{P\}\langle i_1 : i_2\rangle\{Q\}$ from $\{P\}i_1\{R\}$ and $\{R\}i_2\{Q\}$. Here $\langle i_1 : i_2\rangle$ represents the sequential execution of the instructions i_1 and i_2. Another rule that allows generalization and use of logical implication is given below.
Rule of Implication:
Infer $\{P\}i\{Q\}$ from $\{R_1\}i\{R_2\}$, $P \implies R_1$, and $R_2 \implies Q$.

Hoare logic, which is an instance of program logics, is defined as a proof system that consists of first-order logic, together with Hoare axioms and inference rules to deal with program correctness. In order to use Hoare logic to reason about program correctness, assertions need to be mapped to predicates on the program variables, and instructions of the program are considered as transformation of such predicates. The semantics of a programming language specified by describing the effect of executing instructions on assertions about states (rather than states themselves) is known as axiomatic semantics. Hoare logic (or in general program logics) is a proof system over the axiomatic semantics of the programming language. Given P and Q as the precondition and postcondition of the program Π respectively, proving correctness consists of deriving the formula $\{P\}\Pi\{Q\}$ as a theorem.
To do program verification, one annotates the program with assertions at certain locations that corresponds to the entry and exit of the basic blocks of the program such as loop tests, and the entry and exit points of the program itself. These annotated program points are called cutpoints. The entry point of the program is annotated with the precondition, and the exit point is annotated with the postcondition. One then shows that if the program control is in an annotated state satisfying the corresponding assertion, then the next annotated state will also satisfy its assertion.
Let us consider the simple one loop program in figure 2.1: it has 2 variables X and Y, and loops 5 times incrementing X and decrementing Y in each iteration. The cutpoints for the program are at counters 1 (program entry), 3 (loop test), and 7 (termination). The assertions associated with each cutpoint are shown to the right. The precondition P is assumed to be universally true, and the postcondition says that the variable X has the value 5.

Now we need to show that every time the control reaches a cutpoint, the assertion holds. Let us analyze the cutpoints given by the program counter values 1 and

```
1.    X:=0;                                     {T}
2.    Y:=5;
3.    if  (Y<1)  goto 7;                        {(X+Y)=5}
4.    X:=X+1;
5.    Y:=Y-1;
6.    goto  3;
7.    HALT                                      {X=5}
```

Figure 2.1: A simple one loop program

3. To show that given the assertion at 1, after executing counters 1 and 2, the assertion at 3 holds (which can be represented as 1→ 3), we need to prove the formula $\{T\}\langle X := 0; Y := 5\rangle\{(X+Y) = 5\}$ as a theorem. The formula is called proof obligation. By applying the axioms of assignment, rule of composition, and implication rule, the simplified proof obligation $T \implies (0+10) = 10$ is obtained. From this example, it can be seen that by applying Hoare axioms, a formula that is free from constructs of the programming language is obtained. Such a formula is called verification condition. The complete correctness proof of the program given above consists of generating such verification conditions for each of the execution paths 1→ 3, 3→ 3, and 3→ 7, and showing they are logical truths. In practice, the verification conditions can be more complicated formulas with a need to do non-trivial proofs.

Doing program verification using axiomatic semantics requires two tools:

- a verification condition generator(VCG) that takes an annotated program as an input and generates the verification conditions
- a theorem prover that proves the verification conditions

A possible pitfall of this approach is the fact that it depends on two trusted tools, namely a VCG and a theorem prover. Nevertheless, program logics and axiomatic semantics have been commonly used both in program verification theory and its application [8, 9, 10]. The principal benefit of using this approach is to abstract out details of the underlying machine from the program.

2.2.3.2 2. Theorem Proving

Theorem proving is one of the most well studied approaches to program verification. The approach is based on the use of computer programs called theorem provers to construct and check derivations of theorems about the mathematical object which is the topic of interest in some formal logic. Any such formal logic consists of a formal language to express formulas about the object, a set of formulas called axioms that can be interpreted as self-evident truths about the object, and a set of validity preserving inference rules for deriving new formulas about the object from the existing ones. The logic has to make sure that the set of formulas representing axioms are valid, and the set of inference rules are validity preserving which means that the application of one or more inference rules of the logic to the axioms must result in a valid formula. A formula resulting from applying one or more inference rules to the axioms of the logic is called a theorem. A sequence of one or more formulas such that each formula is either an axiom or result of applying an inference rule to some other formula that come before this formula in the sequence is called a derivation or deduction. In the sense of theorem proving, verification means the process of showing the existence of at least one derivation for some formula of interest in the logic of the theorem prover.

Being the base for the theorem proving approach, there has been a number of theorem provers being in use today. Some of the most popular ones include HOL, Coq, ACL, Isabelle, NuPrl, PVS, TPS and Leo. The logics these theorem provers are designed to work with range among various domains; there are theorem provers for first-order logic, higher-order logic, set theory, etc. There are also significant variations among theorem provers depending on the level of automation and the need of interaction with a user. Some like HOL needs more interaction with trained users while others like PVS can work fine with just a little need of interaction with the user. Theorem provers like TPS and Leo require no interaction with a trained user and hence belong to a group of theorem provers called Automatic Theorem Provers (ATPs). Despite these diversity of features, one common feature of all theorem provers is that they support logics that are very expressive. This expressivity has allowed the applications of theorem provers to proof well known theorems in different mathematical domains. The best example is that Nqthm theorem prover has been able to mechanically verify Godel's incompleteness theorem [11]. But, this expressivity does not come without a cost; as it was pointed out in section 2.2.2, any such sufficiently expressive logic that is consistent must be undecidable. This means that there can not be an automatic procedure for determining if there is a derivation for some formula in a given logic, and therefore, the successful use of theorem proving for deriving nontrivial theorems typically requires interaction with a trained user. Any attempt to do theorem proving without a need of interaction between the theorem prover and the trained user requires trading of expressivity for automation like the case for the ATPs.

Nevertheless, theorem provers remain as one of the most important players in the area of formal verification. Three of the most important things theorem provers are generally known to do are:

- Given a formal logic, the theorem prover can mechanically verify if a certain sequence of formulas corresponds to a valid derivation in the formal logic. This is done by checking if each formula in the derivation is either an axiom in the formal logic or a result of applying an inference rule from the formal logic on the previous formula of the sequence.

- When a heuristics for proof search is provided by the user, the theorem prover can practically assist the user in the construction of a proof by implementing the heuristics. Such heuristics of proof search include generalizing the formula for applying mathematical induction, using appropriate instantiation of previously proven theorems, judicious application of term rewriting, and so on.
- If the formula to be verified as a theorem can be expressed in some well-identified decidable subset of the formal logic, then the theorem prover can make use of decision procedures to determine if the formula is a theorem or not without a need to interact with a trained user. For example, the ACL2 theorem prover that has decision procedures for deciding linear inequalities over rationals [12].

Although there have been major successes in approaches aimed at automating the search of proofs, undecidability still poses strong challenge on the automation. Therefore, having a substantial interaction between the theorem prover and a trained user is inevitable during the construction of nontrivial derivation for a formula using theorem proving. In the interaction, the user is responsible for providing an outline for the derivation of the formula required to be proved, and the theorem prover is responsible for deciding if a formal proof can be devised from the outline that can be used for deriving the formula.

The approach to verify correctness of programs using theorem provers is exactly the same as the approach to prove the correctness of any other mathematical statement in a formal logic. The desired properties of the program are specified as formulas in the logic of the theorem prover, and an attempt to derive the formulas is made from the logic using the inference rules of the logic. But the size of the formulas that need to be manipulated in order to verify programs could be extremely larger than those manipulated to prove typical mathematical statements.

The main goal in formal verification research has been to automate proofs of correctness as much as possible which does not match with the fact that theorem provers in practice need interaction with trained users to perform their activities. Nevertheless, theorem provers continue to be very useful players in formal verification for the following 3 important reasons:

- In some cases, sufficiently expressive logic is needed simply to specify the desired correctness properties of the program, and theorem proving is the only method one can depend on for proving such properties.

- Sometimes even when specifying the desired correctness properties is possible with some decidable logic, for example, when one wants to reason about a finite-state system, theorem proving has the advantage of being both succinct and general.

- A very practical reason to use theorem proving is that fact that theorem provers provide a substantial degree of control of the derivation process of complex theorems for the user. This can be exploited by the user in different ways, for example, by proving key intermediate lemmas that assist the theorem prover in its proof search.

2.2.3.3 3. Model Checking and Bounded Model Checking

In theorem proving, automation is traded for the expressivity of the formal logic used in the given theorem prover. But this does not change the fact that automation is the ultimate target of formal verification, whether for a program or a more complex computing system, and therefore, enabling automated verification, if possible, is a key consideration. Given a desirable property to be verified, automated verification methods make use of a decidable formalisms to represent the property as a formula under the formalism being used, and a decision procedure, an algorithm that terminates with the correct yes/no answer for some given decision problem, to prove the truth or falsity of the formula.

Model Checking is one such method that was introduced in 1981 by Clarke and Emerson [13], and independently by Queille and Sifakis shortly after [14]. It is an automatic technique for verifying finite state reactive systems. Specifications are expressed in a propositional temporal logic, and the reactive system is modeled as a state-transition graph. The model checking algorithm implements an efficient search procedure to determine automatically if the specifications are satisfied by the state transition graph. In the context of program verification, it determines if the model of a given program satisfies its specification. The model of the program consists of all the possible states the program can be at any one point during its execution, and transitions that describe how a program evolves from one of its possible states to another. Therefore, states of the program and the possible transitions among them together provide the building bricks for modeling the program. A state is an evaluation of the program counter, the values of all program variables, and the configurations of the stack and the heap. The desired property of the program is specified as a logical formula. The model checking algorithm checks whether the desired correctness property is satisfied by the reachable states of the program by exhaustively exploring each of these states. If the desired property holds in all states of the program, the algorithm terminates with the answer true; otherwise if it does not hold in any of the reachable states of the program, the model checking algorithm computes a counterexample, an execution trace leading to a state of the program in which the property does not hold. This procedure is guaranteed to terminate if the state space is finite. The ability of model checking algorithms to compute counterexamples is considered as a key feature, and by some even as a reason for its acceptance as a formal verification techniques [15]. Although the direct application of model checking techniques to implementation level code can significantly increase the computational requirements for a verification, the promise of this approach is that it can eliminate the need for expert model builders and can place the power of automated verification techniques where it belongs: in the hands of programmers [16].

Model checking tools verify partial specifications that are usually classified as safety or liveness properties. Safety properties describe the unreachability of bad states, such as when null pointer is dereferenced, stack overflow has occurred, API usage contracts, like the order of function calls, are not respected, etc. Liveness properties on the other hand describe that something good eventually happens such as the condition that requests must be served eventually, a program must eventually terminate, etc. Like program logics, model checkers take specifications of a program given in the form of preconditions and postconditions. However, unlike the case in program logics where different specifications are given for the properties that should hold at different points in the program, specifications in model checkers are defined for the entire program.

The fact that model checking algorithms are based on exhaustive examination of reachable states has raised a very critical issue in model checking called state-space explosion: the state-space of a program is exponential on various parameters of the program of which the most important ones are number of variables and the width of the data-types. The state-space can even be infinite if there exist function calls and dynamic memory allocations in the program. Concurrency worsens the problem even more due to the different thread schedules that must be considered which are exponential on the number of statements in the program. Despite the possibility of ending in unmanageably huge state-space, model checking algorithms generate sets of states to be analyzed by using instructions in the program, and store them to ensure that they are visited no more than once.

Model checking algorithms are divided into two principal categories depending on their methods for representing states: Explicit-state model checking algorithms and

Symbolic model checking algorithms. Explicit-state model checking algorithms use an explicit representation of the system's global state graph, usually given by a state transition function. Symbolic model checking algorithms use a symbolic representation for the state set, usually based on binary decision diagrams [17].

Explicit state model checking methods need to explicitly represent the program as a state transition graph by recursively generating successors of states starting from the initial state. The graph may be constructed in a depth-first, breadth-first, or using some heuristic. Every time a new state is generated, it is checked for a property violation on the fly, so that errors can be detected without a need to build the entire graph. Explored states are stored in a hash table to avoid recomputing their successors when some state that has been already explored is regenerated. The generated states are compressed before storage so that memory usage is optimized. If the available memory is insufficient, lossy compression methods can be used. This may be risky because it may lead to some error states being missed. In practice, with state spaces containing close to a billion states, and hash tables of several hundred megabytes, the probability of missing a state can be less than 0.1% [18]. An important method of pruning state space exploration for concurrent programs is called partial order reduction [19]. The order in which instructions in different threads are executed may not make a difference for proving some properties. Transitions whose interleavings do not affect the property can be grouped into classes. A model checker only needs to generate one representative of each class while constructing the state graph. In the best case, partial order reduction can reduce the state space to be explored by a factor that grows exponentially in the number of threads. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure, and property validation amounts to a partial or complete exploration of the state space.

Symbolic model checking methods, which are based on manipulation of boolean formulas, represent sets of states unlike the explicit state ones that enumerate individual states. The most well known symbolic representations are boolean decision diagrams(BDDs) [20] and propositional logic [21] for finite sets, and finite automata [22] for infinite sets. A BDD is obtained from a boolean decision tree by maximally sharing nodes and eliminating redundant nodes. For a fixed variable ordering, BDDs are well suited for model checking since they allow boolean functional equivalence (essential in symbolic model checking) to be checked efficiently. Although BDDs can represent well system with excess of 10^{20} states, the memory requirement for storing and manipulating BDDs is very huge since the set of states represented by boolean functions grows exponentially. The issues in using finite automata for infinite state space are comparable with that of BDDs for finite state space [23]. Symbolic representations such as propositional logic formulas are more memory efficient, at the cost of computation time.

Symbolic techniques work well for proving correctness and handling state-space explosion due to program variables and data types. Explicit state techniques are well suited to error detection and handling concurrency. A general approach to counter the issue of state-space explosion that is not specific to proving correctness or detecting errors unlike the two techniques above is abstraction. Since a program can, in general, be represented by an infinite-state model, existing tools do not directly check programs against specifications. Instead, a conservative finite state abstraction of the program is first generated. In this approach, the state-space explosion is prevented by analyzing a sound abstraction of the program that consists of smaller state space. Such an abstraction of the program used to be manually constructed, but due to advancements in the corresponding tools recently, the construction can now be done automatically. A framework known as CounterExample Guided Abstraction Refinement (CEGAR) [24] iteratively create a more precise abstractions of the program until the desired properties are proven or a real counterexample is generated. Chaki, et al. [24] summarizes the CEGAR process as follows:

- Model Creation : A model is computed using the control flow graph (CFG) of the program in combination with an abstraction method called predicate abstraction [25], [26]. Properties such as the equivalence of predicates are decided with the help of a theorem prover.

- Verification : Verify that the abstraction conforms to the specification. If this is the case, the verification is successful. Otherwise, obtain a possibly spurious counterexample and go to the next step

- Validation: Check whether the counterexample extracted in the verification step is valid. If this is the case, then we have found an actual bug and the verification terminates unsuccessfully. Otherwise construct an explanation for the spuriousness of the counterexample and proceed to the next step

- Refinement: Use the spurious counterexample from the previous step to construct an improved set of predicates. Return to the first (model creation) step to extract a more precise model using the new set of predicates instead of the old one. The new predicate set is constructed in such a way as to guarantee that all spurious counterexamples encountered so far will not appear in any future iteration of this loop.

The explicit state, symbolic as well as abstraction based model checking approaches discussed above differ from each other on how they view and try to tackle the state-space explosion. But in general, there are two approaches to program verification using model checking. The first approach requires transforming the implementation level specification of the program (the code) systematically into a language that some given verification tool can recognize. The program is rewritten in the syntax of the given verification tool. An example is the first Java Pathfinder [27] which targets the SPIN model checker [28]. This approach requires building a sort of parser that can read and transform the implementation level specifications of the program (the actual code) into detailed verification models that can be verified by a model checker. For the transformation to be done accurately, the parser needs to be able to interpret the semantic content of the program, and transform it into equivalent representations in the verification model. The second approach involves the design of verification tools that can take the program written in a certain programming language, and do the verification reasoning on the program itself without a need for transformation. The verification tool is designed specifically to handle a given programming language, and separate tools are needed to handle each and every language. Examples for this approach include the second Java Pathfinder tool [29] and Blast tool [30]. This approach requires a verifier that can make accurate decisions on the validity of a program execution after reasoning on the program itself. The major challenge in this approach will be the construction of a full fledged verification tool for any formally defined programming language. The challenge will be exacerbated for a programming language that was not designed initially with the intention of making it amenable for verification tools.

In conclusion, the following facts have paved the way for the acceptance and high success of model checking techniques in formal verifications of computing systems:

- Model checking is automatic
- Its approaches are not mainly based on doing proofs
- Model checking algorithms run fast
- When verification fails, counterexamples are generated, and
- It works well with Partial specifications

Actually, the introduction of symbolic model checking is considered as the ground breaking achievement for the subsequent success of model checking in solving industry-scale verification problems. The combination of symbolic model checking with BDDs enabled the representation of programs whose number of states is in excess of 10^{20}

as discussed earlier in this section. The integration of abstraction techniques into this process attracted a range of interest in model checking from the industry. This was followed by a significant number of realistic systems, mostly of hardware nature, could be verified using these model checking methods, which eventually resulted in acceptance and adoption of the methods in various areas of the computing industry in general. The principal limitation of these methods is the large memory requirement of the boolean functions used to represent the set of states. This limitation had motivated Biere et al. to propose a technique called Bounded Model Checking(BMC) in 1999 [21] which was based on SAT techniques rather than BBDs. This technique is a form of Model Checking that performs a depth-bound exploration of the state-space which implies that the need to examine the entire state space is relaxed. BMC explores program behavior exhaustively, but only up to a certain depth limit. If the incorrect properties of the program are exhibited only in states that are beyond this depth limit, then BMC will not catch this bugs, and may "verify" the program as correct.

Therefore, it is important of the BMC algorithm to explore "deep enough" so that there is a guarantee that all the behaviors of the program are represented and evaluated, and exploring further deep will only get states that have already been explored. The depth limit that provides such a guarantee for the BMC algorithm is called a completeness threshold [31]. However, since finding the smallest such threshold is as hard as model checking itself, in practice, BMC based methods usually try to compute and make use of an approximate value for the completeness threshold. The approximation can be done either through syntactic analysis or by using iterative algorithms. The first option uses the high-level worst-case execution time (WCET) to approximate the depth-bound. Since this time is given by a bound on the maximum number of loop-iterations, the number of loop-iterations is computed first by simply using syntactic analysis of the loop structures, and then the WCET time is determined from the number of loop iterations. If the number of loops can not be extracted from the loop structures, iterative algorithms will be applied to approximate the bound. In the iterative algorithm approach, an initial guess of the bound on the number of loop iterations is made, and then the loop is unrolled up to the bound with the assumption that some conditions called unwinding assertions hold. If the conditions are violated, a new higher bound is guessed, and the algorithm proceeds recursively until it reaches a point where the unwinding assertions are satisfied. This method is very useful when loops in a given program have run-time bound.

BMC is the best technique to find bugs that are not deep inside the state-space. The very useful feature of generating a counterexample trace when a bug is found is one of the best characteristics it takes from model checking techniques. On the down side, BMC is not generally complete, and completeness can be guaranteed only for programs whose loops are not too deep.

There are a number of tools that implement BMC for program verification. BMC was applied for the first time as a novel formal verification approach for equivalence checking of small, assembly-language routines for digital signal processors (DSP) by Currie et al. [32]. One of its first implementations of BMC for C programs is CBMC [33] developed at Carnegie Mellon University.

In this work, constraint programming techniques are applied to the bounded model checking approach to get an efficient program verification framework. Therefore, we will first discuss the main features of constraint programming in the next subsection.

2.3 Constraint Programming

Constraint programming is one of the most exciting developments in programming languages of the last two decades. It has now become one of the most suitable methods for modeling and solving optimization problems that involve complex relationship among entities of the problem, and combinatorial search. This is due to the fact that constraint programming is based on strong theoretical foundation. Unlike traditional programming languages, for instance object oriented languages, that provide little support for specifying relationships among the programmer defined objects, such relationships among programmer-defined objects form the base of constraint programming. This fact is attracting widespread commercial interests as many critical problems like job scheduling, timetabling and routing can be efficiently solved using constraint programming.

A constraint is a restriction on the space of possibilities for some choice; it can be considered as a piece of knowledge that filters out the options that are not legitimate to be chosen, and hence narrowing down the size of the space. Formulating problems in terms of constraints has proven useful for modeling fundamental cognitive activities such as vision, language comprehension, default reasoning, diagnosis, scheduling, and temporal and spatial reasoning, as well as having applications for engineering tasks, biological modeling, and electronic commerce [34].

2.3.1 Constraint Satisfaction Problem(CSP)

A CSP in general consists of three main components:

- A set of *variables* which are objects that can take some value.

- A set of *domains* for each variable in the CSP. The set of possible value for each variable is called its domain.
- A set of *constraints* which are rules that impose limitation on the values that a variable or a combination of variables may be assigned.

Therefore, a CSP can be defined as a model of some problem that consists of variables, their domains, and constraints. A common example to illustrate a CSP is the $n-Queens$ problem which is aimed at placing n queens on an $n \times n$ chessboard such that there is no possibility of attack between any two of the n queens. Two queens are said to attack each other if they are put on the same row or the same column or the same diagonal of the chessboard. One way of modeling this problem as a CSP is as follow: there are n variables $\{x_1,...,x_n\}$ for each column of the chessboard, the domains for each variable x_i will be $D_i = \{1,...,n\}$, and the constraint on each pair of columns is that the two queens must not share a row or a diagonal.

Formally, a CSP can be defined as the triple $\langle X,D,C\rangle$, where X is a finite set of variables $X = \{x_1,...,x_n\}$, with respective domains $D = \{D_1,...,D_n\}$ which list the possible values for each variable $D_i = \{v_1,...,v_k\}$, and a set of constraints $C = \{C_1,...,C_t\}$. A constraint C_i can be viewed as a relation R_i defined on the set of variables $S_i \subseteq X$ such that R_i denotes the simultaneous legal value assignments of all variables in S_i. Thus, the constraint C_i can be formally defined as the pair $\langle S_i,R_i\rangle$; S_i is called the scope of the constraint. A solution of the CSP is an n-tuple $\langle V_1,...,V_n\rangle$ where each $V_i \in D_i$ corresponds to the value assigned to each variable $x_i \in X$, and the assignment satisfies all constraints in C simultaneously.

For example, the $n-Queen$ problem for the case of 4 queens can be modeled as a CSP using finite domains as follows: There are four variables $X = \{x_1,x_2,x_3,x_4\}$, with domain $D_i = \{1,2,3,4\}$ for each of the variables. There are six constraints that avoids any two queens from attacking each other; $C_1 = R_{1,2}$, $C_2 = R_{1,3}$, $C_3 = R_{1,4}$, $C_4 = R_{2,3}$, $C_5 = R_{2,4}$, and $C_6 = R_{3,4}$. Here, $R_{i,j}$ is a relation mapping each possible value of queen i with a possible value of queen j simultaneously. The six constraints are given below by extension:

- $R_{1,2}$ = {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
- $R_{1,3}$ = {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}
- $R_{1,4}$ = {(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3)}
- $R_{2,3}$ = {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
- $R_{2,4}$ = {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}

- $R_{3,4}$ = {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}

Constraint programming is enabled by embedding constraints in some host language. The first host languages used were logic programming languages such as Prolog, which explains the reason for the field to be initially called constraint logic programming. In the language of Prolog, the domains of variables were represented as a set of Herbrand terms and constraints were formed as equalities between Herbrand terms. A CSP that consist of such constraints is solved by using the unification facility of the Prolog host language to unify the Herbrand terms. For example, the constraint $stud(Id, Name) = stud(35031, tewodros)$ is solved through unification by assigning Id to 35031 and Name to *tewodros*. Constraint logic programming has been extended to constraints over other domains among which the most important ones are boolean constraints, real linear constraints and finite domain constraints.

2.3.1.1 Boolean Constraints

A constraint is called boolean if each variable in the constraint has a domain $D = \{0, 1\}$. Such constraints are particularly useful for modeling digital circuits, and boolean constraint solver can be used for verification, design, optimization etc. of such circuits.

2.3.1.2 Finite Domain Constraints

All variables in such constraints get associated with some finite domain, either explicitly declared by the program, or implicitly imposed by the finite-domain constraint solver. By finite domain, we mean any set that can be mapped into a subset of integers. Therefore, only integers and domain variables are allowed in finite domain constraints.

Finite-domain constraint solvers mainly deal with two classes of constraints called primitive constraints and global constraints. All other types of constraints are automatically translated to conjunctions of primitive and global constraints, and then solved. Classes of primitive constraints defined by the solver include:

- **Membership constraints:** Examples include $X\ in\ 1..5$ which constraints the variable X to have the set $\{1, 2, 3, 4, 5\}$ as its domain, and $domain(L, 1, 5)$ which constraints a list of variables L such that each variables in L has the set $\{1, 2, 3, 4, 5\}$ as its domain.

- **Arithmetic constraints:** Examples include $X + Y\# = 5$ which constraints X and Y to take values that can be summed up only to 5, and $X\# > Y$ which constraints the value taken by X to be always greater than the value taken by Y.

- **Reified constraints:** Instead of merely posting constraints, it is often useful to reflect its truth value into a boolean variable B, so that the constraint is posted if B is set to 1, the negation of the constraint is posted if B is set to 0, B is set to 1 if the constraint becomes entailed, and B is set to 0 if the constraint becomes disentailed. This mechanism is known as reification. A reified constraint is written as *Cons*# <=> *B* where *Cons* is the constraint to be reified and *B* is a boolean variable.

- **Propositional constraints:** are complex constraints formed by combining individual constraints using propositional combinators. The main propositional combinators include #/\, #\/, # =>, and # <=> which play roles similar to logical conjunction, disjunction, implication and bi-implication respectively. For example, given constraints C_1 and C_2, the propositional constraint $C_1\#/\backslash C_2$ ($C_1\#\backslash/C_2$) is satisfied if and only if both(either) of C_1 and C_2 are satisfied.
  Another propositional constraint $C_1\# => C_2$ valuates to true if C_1 is false or C_2 is true. An important property of this constraint is that if C_1 is true (resp. C_2 is false), then C_2 should necessarily be true (resp. C_1 should necessarily be false). This can be used to specify constraints that are needed only under some condition. For this reason, such constraints are also called conditional constraints.

Some of the most important global(combinatorial) constraints defined by the solver include all_different, element, global_cardinality, etc. For example, $all\_different([X,Y,Z])$ constraints any variable in the list to take a unique value which is different from any other variable in the list. Another global constraint, $element(X,L,Y)$, constraints Y to be the X^{th} element of the list L given some list L.

New user-defined primitive constraints can be added to the solver by writing so-called indexicals whereas new user-defined global constraints can be also written in Sicstus Prolog by means of a programming interface. A detailed explanation of the various finite-domain constraints defined by the solver together with how to add user defined constraints is given in [35].

2.3.1.3 Real Linear Constraints

Such constraints have variables that can take any real value. Unlike finite domains, real domains are continuous and infinite. The solver called clp(R) is bundled into many Prolog implementations as a library package which is used to solve real constraints. In addition to all the common arithmetic constraints, clp(R) adopts a simplex like approach to solve a number of linear equations over real-valued variables, to cover the lazy treatment of nonlinear equations, to feature a decision algorithm for linear inequalities that detects implied equations, to remove redundancies, to perform projections (quantifier elimination), to allow for linear dis-equations, and to provide linear optimization [35].

2.3.2 Constraint Solving Approaches

Given a CSP $\langle X, D, C \rangle$, there are different ways of getting a value for each variable in X from its respective domain in D that satisfies all constraints in C. A framework that has utilities defined in it for modeling a given problem as a CSP, and that produces the solution to the CSP if there is at least one or that tells the absence of a solution is called a constraint solver. There are two broad classes of constraint solvers: complete solvers and incomplete solvers.

2.3.2.1 Complete Solvers

Complete solvers implement decision procedures that take a given CSP and produces a solved form of the problem. Examples of complete solvers include CLP(R) and CLP(B) that are used for solving real linear constraints and boolean constraints respectively. Although solving real linear constraints in itself is not possible in polynomial time, since CLP(R) implements simplex algorithms, solving linear constraints is quite efficient. However, for CLP(B), the fact that the underlying representation of boolean functions is based on Boolean Decision Diagrams results in exponential time being required for solving constraints.

2.3.2.2 Incomplete Solvers

The most interesting and fundamental concept in constraint solving that drives incomplete solvers is called constraint propagation. Solvers that implement propagation techniques are based on the observation that if the domain of any variable in some CSP is empty, the CSP is unsatisfiable. These solvers try to transform a given CSP into an equivalent CSP whose variables have a reduced domain. If any of the domains in the reduced CSP becomes empty, the reduced CSP, and hence the original CSP are said to be unsatisfiable since both CSPs are equivalent. The solvers work by considering each constraint of the CSP one by one, and they use the information about the domain of each variable in the constraint to eliminate values from domains of the other variables. These procedures alone may not succeed in getting a solution, as the case is often, and hence enumeration of variables can be also needed. Therefore, incomplete solvers interleave propagation and enumeration to obtain a solution or to infer the absence of any solution. Example includes CLP(FD) which is used to solve constraints over finite domains. Boolean constraints can also modeled here as special case of finite domain constraints with each variable having the domain $D = \{0,1\}$. Since propagation may be of no use in the worst case scenario, eventhough propagation has a polynomial time complexity, the CLP(FD) solver has an exponential time complexity on the size of the domains.

There are different levels of consistency criteria that can be achieved by the constraint propagation algorithm. The most important ones include node consistency, arc consistency, and bound consistency.

A CSP is node-consistent if there does not exist a value in the domain of any one of its variables that violates a unary constraint in the CSP. This criterion is of course very trivial but it is very important when it is considered in the context of an execution model that incrementally computes solution from partial solutions.A more demanding consistency criterion is arc-consistency. To be considered for arc-consistency, a CSP must first be node-consistent. In addition, for every pair of variables $\langle X,Y\rangle$, for every constraint C_{xy} defined over variables X and Y, and for each value V_x in the domain of X, there must exist some value V_y in the domain of Y that supports V_x. For example, the CSP $\langle\{X,Y\},\{1..5,1..5\},\{X+Y>7\}\rangle$ is not arc-consistent because there is no support in the domain of Y when X takes 1 or 2 that satisfies the constraint $X+Y>7$. The same holds for Y also. An arc-consistent CSP which is equivalent to the original CSP is obtained by reducing domains of X and Y from $\{1,2,3,4,5\}$ to $\{3,4,5\}$.

Another type of consistency criteria is called bounds consistency defined on numeric constraints which are arithmetic constraints of equalities or inequalities. For example, the CSP $\langle\{X,Y\},\{1..10,1..10\},\{X>Y\}\rangle$ in not bound-consistent because there are some bound values in the domain of both variables that can never be part of any solution as they do not have any matching value in the other variable to satisfy the given constraint. If X takes the value 1, then there is no any matching value in Y that can satisfy the constraint $X>Y$. Therefore 1 should not be in the domain of X. Similarly, there is no matching value for X when Y takes the value 10. Likewise, 10 should not be in the domain of Y. A bound-consistent equivalent CSP will be $\langle\{X,Y\},\{2..10,1..9\},\{X>Y\}\rangle$. Another example can be the CSP $\langle\{X,Y\},\{3..10,1..8\},\{X=Y\}\rangle$. There is no any matching values for Y when X takes either of 9 or 10 because we have an equality constraint $X=Y$. Similarly should Y take either 1 or 2, there is no matching value in X that satisfies the given constraint. A bound-consistent equivalent CSP in this case will be $\langle\{X,Y\},\{3..8,3..8\},\{X=Y\}\rangle$.

Algorithms that impose arc-consistency are polynomial on the number of variables, whereas algorithms that impose bounds-consistency are linear on the size of domains of variables. Global constraints have specialized propagation algorithms that exploit the semantics of the constraints to obtain a much faster propagation, and hence a much faster solving of the constraints.

3

Constraints Model Generation

In this work, an efficient way of verifying a program with respect to its specification is studied. The input program is written in some subset of Java, and its specifications (precondition and postcondition) are written in some subset of JML. The only datatypes allowed in the program are integers and array of integers. The program is assumed to be functional and sequential, and it is assumed not to make any function calling. The desired property of the program we are interested to prove is partial correctness with respect to the its specification.

In General, a Bounded Model Checking based approach is used, and in particular, the first approach of Model Checking that requires transformation of the program into the language of the verification tool is applied. Since constraint solvers are used as verification tools in our work, a parser is required that can generate a semantically equivalent system of constraints for the input Java program and its specification in JML. The system of constraints is formed as the union of the constraints corresponding to the program, its precondition, and the negation of its postcondition. The system of constraints should be modeled in such a way that the constraint solver can solve it as efficiently as possible. The constraint solver then takes this system of constraints and tries to solve it. If it succeeds in at least one scenario, then the solution is returned as a bug of the program. If it fails in every possible scenario, the program is assumed to be correct.

3.1 Introduction

For the verification of a given program via constraints programming as a tool, there must be a way of transforming the original program into a system of constraints on which the actual verification process can be done using constraint solvers. The success of the verification process depends not only on how efficiently the solver can solve the constrains system but also on how accurately the original program can be transformed into the system of constraint. The accuracy of the transformation is very crucial because a significant difference between the original program and the corresponding system of constraints will make any judgment that has been made about the original program based on the system of constraints unacceptable. Therefore, the transformation should always keep the semantics and logic of the original program, and the only significant difference allowed is the representation used. In this study, the original program is written in Java and its corresponding constraint system uses Sicstus Prolog syntax. The responsibility of keeping the semantics and logic of the original program in the resulting system of constraints falls on the parser which is the component doing the task of transforming the Java program into a constraint logic program in Sicstus Prolog.

The parser reads a program written in Java along with its preconditions and postconditions written in the Java Modeling Language(JML), parses each of the Java structures it gets on the way into Sicstus Prolog structures, and generates a system of constraints equivalent to the original program it has read. This requires the parser to be capable of reading, recognizing and parsing all types of Java structures. However, since this work does not focus on building a complete parser to convert Java programs to their equivalent constraint systems, this parser handles only programs written is some subset of the Java programming language. The parser also handles preconditions and postconditions written only in some subset of JML. These subsets of Java and JML the parser can handle are discussed in detail in section 3.2. The parser reads structures from the Java or JML subset it can handle, and generates the corresponding constraints in Prolog. The issue of transforming Java structures from the original program into these constraints will be discussed in detail in section 3.3 where the equivalent constraints generated by the parser for each and every structure in the Java and JML subsets are shown.

3.2 The subset of Java language handled

The original program which is the input to the parser can be considered as consisting of two main parts; the first part is the specification of the preconditions and postconditions of the program written in JML, and the second part is the actual program written in Java. Although the parser does the translation of both parts together, the languages used to program or specify these two parts are totally different, and the subsets of each of these languages that the parser can handle are also different. Therefore, the parser considers the original program as a JML code, which specifies the preconditions and postconditions of the program, followed by the actual Java program as shown in the grammar below.

```
Original_Program --> JML_Code, Program
```

3.2.1 Subset of Java language

The subset of Java language the parser can handle contains only basic and simple structures of the language that are sufficient to handle the sample programs used in this study but the subset can be scaled up in case of any need to incorporate more structures.

The parser assumes the input program to be a function that has a return type, a name which is an identifier, a list of arguments which are the input parameters of the function, and contains a code block followed by a return statement between opening and closing parentheses.

```
Program --> Type, Identifier, '(',List_Of_Arguments,')',
            '{',  Code_Block,  Return_Statement, '}'
```

A code block is a sequence of statements where each statement can be a declaration, an assignment, an if_else statement, a while loop or any of these followed by a code block.

```
Code_Block --> Declaration | Declaration, ';', Code_Block |
               Assignment | Assignment, ';', Code_Block |
               If_Else | If_Else, Code_Block |
               WhileLoop | WhileLoop, Code_Block
```

Since we are considering only programs with variables of type integer or one-dimensional array of integers, a type is either the keyword *int* or *int*[], and all declarations are statements of the form type followed by an identifier. An assignment is a statement which consists of the equality sign =, and an identifier and an expression to the left and right hand side of the sign respectively.

```
Type -->  int | int,'[', ']'
Declaration --> Type, Identifier
Assignment --> Identifier, '=', Expression
```

The definition of an expression is given recursively as a number, an identifier, or any two expressions combined by one of the arithmetic operators in $\{+, -, *, /\}$. In Java, an array variable of type integer can have additional expressions for different purpose; for example given an array variable *arr*, its i^{th} element can be given as the expression *arr*[*i*], and its length can be given as the expression *arr.length*. Since this two types of expressions are used repeatedly in the sample programs considered in this study, it has been inevitable for the parser to include them in the base definition of expressions, in addition to numbers and identifiers. The full definition of an expression is given below.

```
Expression --> Number | Identifier |
               Identifier, '.', length |
               Identifier, '[', Expression, ']' |
               Expression Binary_Op Expression
Binary_Op --> '+' | '-' | '*' | '/'
```

An if_else statement consists of the keyword if followed by a condition, a code block in between two parentheses and an optional else part. The else part starts by the keyword *else* followed by a code block in between two parentheses.

```
If_Else --> if, Condition, '{', Code_Block, '}' |
            if, Condition, '{', Code_Block, '}', else,
            '{', Code_Block, '}'
```

A while-loop is a statement that starts with the keyword *while* followed by a condition and a code-block in between two parentheses.

```
WhileLoop --> while, Condition, '{', Code_Block, '}'
```

A condition is a unit-condition, negation of another condition or any two conditions combined using a boolean operator. A unit-condition consists of two expressions combined together by a comparison operator.

```
Condition --> UnitCondition | '!', Condition |
              Condition, Boolean_Op, Condition
```

```
UnitCondition --> Expression, Comparison_Op, Expression
Comparison_Op --> '==' | '!=' | '>' | '>=' | '<' | '<='
Boolean_Op --> '&&' | '||'
```

The complete grammar for the subset of Java language the parser can handle is shown in figure 3.1. In addition to the main structures whose definition is given above, the grammar below contains definitions of basic structures like *Identifier*, *Number* and *Return_Statement* that are used as the building blocks in the definitions above.

3.2.2 Subset of JML

Like the subset of the Java language considered above, the subset of JML that can be handled by the parser contains only basic structures of the modeling language that are used in the sample programs considered in this study. In case of any need to handle more structures, the subset can be extended by providing the definitions of the additional structures to the parser.
A JML code consists of the precondition followed by the postcondition.

```
JML_Code -->  '/*', Precondition, Postcondition,'*/'
```

A precondition starts by the character @ followed by the keyword *requires*, and then a JML-condition will complete the definition.

```
Precondition --> '@', requires, JMLCondition
```

A postcondition has the character @ and the keyword *ensures* at the beginning followed by a postcondition block. A postcondition block is defined as a sequence of one or more postcondition statements. Each postcondition statement starts by the character @ followed by two JML-conditions which are separated by the symbol ==> showing the implication relation between the two conditions.

```
Postcondition --> '@', ensures, Postcondition_Block
Postcondition_Block --> Postcondition_Statement |
                        Postcondition_Statement,  '&&',
                        Postcondition_Block
Postcondition_Statement --> '@', JMLCondition, '==>', JMLCondition
```

A JML condition can be a normal condition like the one defined in the previous subsection for the subset of Java, or can also be defined using a for loop as shown below.

```
Program --> Type, Identifier, ’(’,List_Of_Arguments,’)’,
            ’{’,  Code_Block,  Return_Statement, ’}’
Identifier --> Letter | Letter, Followers
Followers --> Number | Number, Followers | Letter |
              Letter, Followers
Letter --> a |  b | ....|  ’A’  |  ’B’ | ....| ’Z’
Number --> Positive_Number | Negative_Number
Positive_Number --> Digit | Digit, Positive_Number
Negative_Number --> ’-’, Positive_Number
Digit --> 0 | 1 | ..... | 9
List_Of_Arguments --> Declaration |
                      Declaration, ’,’, List_Of_Arguments
Code_Block --> Declaration | Declaration, ’;’, Code_Block |
               Assignment | Assignment, ’;’, Code_Block |
               If_Else | If_Else, Code_Block |
               WhileLoop | WhileLoop, Code_Block
Type -->  int | int, ’[’, ’]’
Declaration --> Type, Identifier
Assignment --> Identifier, ’=’, Expression
If_Else --> if, Condition, ’{’, Code_Block, ’}’ |
            if, Condition, ’{’, Code_Block, ’}’, else,
            ’{’, Code_Block, ’}’
Condition --> UnitCondition | ’!’, Condition |
              Condition, Boolean_Op, Condition
UnitCondition --> Expression, Comparison_Op, Expression
Comparison_Op --> ’==’ | ’!=’ | ’>’ | ’>=’ | ’<’ | ’<=’
Boolean_Op --> ’&&’ | ’||’
Expression --> Number | Identifier |
               Identifier, ’.’, length |
               Identifier, ’[’, Expression, ’]’ |
               Expression Binary_Op Expression
Binary_Op --> ’+’ | ’-’ | ’*’ | ’/’
WhileLoop --> while, Condition, ’{’, Code_Block, ’}’
Return_Statement --> return, Identifier
```

Figure 3.1: Subset of Java

```
JMLCondition --> Condition |
                 forall, Declaration, ’(’,Condition, ’&&’,
                 Condition, ’)’, ’;’, Condition
```

The complete grammar for the subset of JML the parser can handle is shown in fig 3.2.

```
JML_Code -->  ’/*’, Precondition, Postcondition, ’*/’
Precondition --> ’@’, requires, JMLCondition
JMLCondition --> Condition |
                 forall, Declaration, ’(’, Condition, ’&&’,
                 Condition, ’)’, ’;’, Condition
Postcondition --> ’@’, ensures, Postcondition_Block
Postcondition_Block --> Postcondition_Statement |
                        Postcondition_Statement, ’&&’,
                        Postcondition_Block
Postcondition_Statement --> ’@’, JMLCondition, ’==>’, JMLCondition
```

Figure 3.2: Subset of JML

3.2.3 An example program

The program below computes the sum of all even numbers less than or equal to a given integer number. The program requires its input not to be a negative number and in turn it ensures its output to have some value depending on the value of the input and whether this input is even or odd. The program along with its precondition and postconditions written in the subset of Java language and JML subset the parser is made to recognize is shown in figure 3.3.

Therefore, in order for some program to be transformed into the constraint system using the constraint solvers that can do the verification process, the program must be written using only the constructs and syntax that can be recognized by the parser.

3.3 Input program to constraints model transformation

3.3.1 An important consideration: versioning

A problem that will be uncovered when trying to translate a program in procedural languages like Java to a constraint system in a declarative language like Prolog is how

```
/*@ requires (N>=0);
  @ ensures @ (N mod 2== 0)==>(/result==(((N*N)+(2*N))/4)) &&
  @ (N mod 2==1)==>(/result==(((N*N)-1)/4))
*/
int sumOfEven(int N) {
  int i; int sum;
  i=0;
  sum=0;
  while(i<=N){
    if(i mod 2 ==0) {sum=sum+i;}
    i=i+1;}
  return sum;
  }
```

Figure 3.3: An example program

to represent procedural language's variables, which have state, with declarative language's variables that are stateless. For example, the statement $X = X + 1$ is a valid assignment statement in Java that change the state of the variable X. This statement takes the current value of X, adds the value 1 to it, and assigns the sum back to the same variable X. In procedural languages like Java, no matter what a variable contains, it is possible to assign a new value to it as far as the data type is valid. But the same statement $X = X + 1$ will always fail in Prolog. This is because Prolog tries to unify both occurrences of X with the same value but it will never succeed in finding any such value that can be added to one and still remains the same! One way to solve this problem is to replace the statement $X = X + 1$ with another statement $Y = X + 1$, and using the variable Y in the place of other subsequent occurrences of X. In the implementation of this parser, the concept of introducing versions for each occurrence of each variable in the original program, which play a role like that of state in procedural language variables, is applied.

The concept of versioning variables works like this: every time a parser reads a declaration of a new variable, it instantiates the version of the variable to its initial value 0. The version of a variable will be updated every time the parser comes across an assignment statement where some expression is assigned to the variable. After the assignment the current version of the variable will be the new version. The name of the variable in the constraint system the parser is going to generate will be the name the parser reads from the program qualified by the current value of its version at the end. Assume the parser has just read the declarations of variables X, Y and Z. At this moment, the current version of these three variables is 0 by definition. The Java expression $X+Y$ will be represented as $X0+Y0$ in the resulting constraint system. But the Java assignment statement $Z = X+Y$ will be represented as $Z1 = X0+Y0$ or the assignment statement $X = X+1$ that we considered above will be represented as $X1 = X0+1$ in the resulting constraint system which clearly solves the problem that was discussed above. Since the assignment statements have updated the current versions of X and Z to 1, another assignment statement $Z = X+Y$ will be represented as $Z2 = X1+Y0$ in the constraint system. After declaration, Z has been assigned twice which causes its current version to be 2, X has been assigned only once which causes in its current version to be 1, and Y was not assigned at all which causes its current version to remain 0.

```
int X; int Y; int Z;
Z=X+Y; ------>  Z1=X0+Y0
X=X+1; ------> X1=X0+1
Z=X+Y; ------> Z2=X1+Y0
```

However, introducing versions for variables may cause some confusion due to unmatched update of variable versions in the different paths of the program when there is branching in the program. This occurs often when parsing if_else statements and while loops. Let us consider the sample code given in figure 3.4. Since transformation of programs to constraint systems is not yet discussed we use a Java if_else statement focusing on the assignments inside the if and the else code blocks. The condition $X > Y$ will be represented as $X0 > Y0$, the assignment $X = X+1$ as $X1 = X0+1$, and the assignment $Y = Y+1$ as $Y1 = Y0+1$. But how to represent $Z = X+Y$? Simply adding $Z1 = X1+Y1$ after the if_else statement alone, as shown in figure 3.5, is wrong since either $X1$ or $Y1$ will be invalid depending on the evaluation of the condition. In order to match changes of versions in different branches of a conditional statement, the easiest way is to add a matching assignment statement. In our example, adding $Y1 = Y0$ in the if block and $X1 = X0$ in the else block, as shown in figure 3.6, guarantees that no matter what the condition is, it is safe to represent the assignment $Z = X+Y$ as $Z1 = X1+Y1$ in the constraints system being generated. This is true because if the condition is true variable X will be assigned a new value and its version will be updated like before but what is new here is the added assignment $Y1 = Y0$ will cause the update in the version of the variable Y parallel to that of $Y1 = Y0+1$ in the else block. Similarly, if the condition is false, the assignment $X1 = X0$ in the else block will do the matching of the version of variable X with that of the assignment $X1 = X0+1$ in the if block.

```
int X; int Y; int Z;
if(X>Y) {X=X+1} else {Y=Y+1};
Z=X+Y;
```

Figure 3.4: A sample code illustrating variable versions

```
if(X0>Y0){X1=X0+1} else{Y1=Y0+1}
Z1=X1+Y1
```

Figure 3.5: Transformation with wrong version of variables

```
if(X0>Y0){X1=X0+1,Y1=Y0} else{Y1=Y0+1,X1=X0}
Z1=X1+Y1
```

Figure 3.6: Transformation with correct version of variables

Although it adds computational complexity to the parser, the idea of having version for each occurrence of every variable in the original program has enabled simple transformation of assignment statements from the original Java program into the constraint system which is based on Sicstus Prolog syntax.

3.3.2 Program to constraints transformation

The parser transforms the Java language constructs it reads from the original program into finite domain constraints so that they can be solved using the CLPFD library of Sicstus Prolog. The transformation for each of the main structures in the Java language subset the parser can handle is given below.

3.3.2.1 Declaration

When the parser reads a declaration like *int x* or $int[]$ *tab*, an initial version of 0 is instantiated for the variable, and a corresponding declaration of the variable will be made in the syntax of Sicstus Prolog CLPFD library which states the domain of the variable to be in the range -65635 to 65635.

For example, when the declaration *int x* is read, the parser adds the corresponding domain declaration statement $\_x0\ in\ -65635..65635$ in the constraints system. Similarly, when the declaration $int[]$ *tab* is read, since *tab* is an array variable, the parser makes a domain declaration $domain(\_tab0, -65636, 65635)$ which implies that $\_tab0$ is a list variable that corresponds to the integer array *tab* in Java.

It can be seen that the original variable is not only postfixed by its version but also prefixed by an underscore. This is the technique used by the parser to ensure that a given identifier is a valid variable in Prolog; by adding the underscore sign, any identifier whether it starts by lower case letter or upper case letter is guaranteed to be a valid Prolog variable.

3.3.2.2 Expression

Transforming Java expressions into their equivalent Prolog expressions is trivial when the Java expression consists of only integer variables or constants, and there is no array sub-expression inside the expression. For example, the parser transforms the expressions x, $x+2$, and 5 into $\_x0$, $\_x0+2$ and 5 respectively assuming the current version of x is 0.

But transforming expressions which contain some array sub-expressions is not straight forward due to the representational difference between Java and Prolog. Given an array variable *arr*, we have already defined in section 3.2 two important expressions on array variables which are *arr.length* and $arr[i]$ holding the length and the ith element of the array respectively. When the expression *arr.length* is read, with the assumption that *arr* is already declared as a list $\_arr0$, the parser adds the statement $length(\_arr0,\_arr0Len0)$ to the constraint system and the newly created variable $\_arr0Len0$ will play a role in the constraint system equivalent to *arr.length* in the Java program. The parser has its own ways of generating unique fresh variables when required. For example, when the parser reads the Java expression $arr.length+x$, it adds the statements $length(\_arr0,\_arr0Len0)$ and $\_arr0Len0+\_x0$ into the constraints system.

When the expression $arr[i]$ is read, with the assumption of $\_i0$ is the current copy for variable i used as the index of the array, the parser adds the statement $\_i1\#=\_i0+1$ and $element(\_i1,\_arr0,\_arr0Elem0)$ to the constraint system, and the newly created variable $\_arr0Elem0$ will play a role in the constraint system equivalent to $arr[i]$ in the Java program. Arrays in Java start from 0 for the first element where as the constraint *element* starts from 1 for the first element of an array. The assignment $\_i1\#=\_i0+1$ is added to match this semantic difference. For example, when the parser reads the Java expression $arr[i]+x$, like the case above, it adds the statements $\_i1\#=\_i0+1$, $element(\_i1,\_arr0,\_arr0Elem0)$ and $\_arr0Elem0+\_x0$ into the constraints system.

3.3.2.3 Assignment

In section 3.3.1, it was discussed that every time an assignment is done to some variable, the version of that variable is updated. Therefore, an assignment not only requires the parser to make the accurate representation of the assignment statement but also to keep track of changes in the versions of variables that appear at the left hand side of the assignment operator. When the parser comes across an assignment statement there are two cases depending on the left hand side of the assignment operator. The trivial case is when the left hand side is a simple integer variable. For example, $x=x+1$ will be transformed into $\_x1\#=\_x0+1$.

The case is non-trivial when the left hand side is an indexed array variable. For example, when the parser comes across the assignment $arr[i] = 5$, with the assumption that *arr* is already declared as a list *_arr*0, it needs to create the next version of the variable, which should be a list of similar size for this example, say *_arr*1 then it should copy all elements of *_arr*0 to *_arr*1 except the i^{th} element; 5 should be assigned to the the i^{th} element of *_tab*1. In Sicstus Prolog syntax, the parser transforms the assignment $arr[i] = 5$ into the set of statements given below and adds to the constraints system.

```
_i1#=_i0+1,
same_length(_tab0,_tab1,_tab0Len0),
(for(Count0,1,_tab0Len0) do
element(Count0,_tab0,_tab0Elem0),
element(Count0,_tab1,_tab1Elem0),
 (Count0#=_i1)#=>_tab1Elem0#=5,
#\(Count0#=_i1)#=>_tab1Elem0#=_tab0Elem0)
```

3.3.2.4 Condition

Conditions are one of the most frequently encountered structures for the parser since they form part of if_else statements, while loops, preconditions and postconditions. Generally, when the parser reads a condition, it first creates a new boolean variable using its automatic variable generation facility, reifies the condition to the newly created boolean variable, and uses the boolean variable instead of the actual condition. For example, when the parser reads the condition $x > 5$, it creates a new boolean variable say *_bool*0, does reification of this variable and the condition using the reified constraint *_bool*0# <=> *_x*1# > 5 assuming the current version of the variable *x* is 1, and uses *_bool*0 in the place of the condition $x > 5$.

At the start of this chapter, it was said that the main objective of the parser is to transform Java programs into their equivalent constraints' system in such a way that efficient solving of the generated constraints' system is facilitated. The parser has a number of techniques to achieve its efficiency objective. One such technique is applied when a new condition is read. Whenever the parser reads a condition, before creating a new variable and reifying the condition with the variable, it checks whether the condition (or the negation of the condition) has already been read by the parser sometime before this moment or not. If the condition has been already read by the parser before, it means that the condition has been already reified with some boolean variable. Therefore, the parser can simply make use of this variable without a need to create a new variable, and also without a need to do any reification. Similarly, if the negation of the condition has been already read by the parser before, it means that the negation of the condition has been already reified with some boolean variable. Therefore, the parser can simply make use of the negation of this variable without a need to create a new variable, and also without a need to do any reification.

For example, assume that after reading the condition $x > 5$ and doing the reification to $\_bool0$ above, the parser reads additional conditions $y > 10$, $x > 5$, and $x <= 5$ one after the other in some program. The condition $y > 10$ has not been already read by the parser before, so the parser creates a new boolean variable say $\_bool1$ and does the reification using the reified constraint $\_bool1\# <=> \_y1\# > 10$ with the assumption that the current version of variable y is 1. For the condition $x > 5$, since the parser has already read an instance of this condition before, it will just use the boolean variable which was previously reified to this condition by the parser which is $\_bool0$. For the condition $x <= 5$, since the parser has already read its negation which is $x > 5$ and has already done a reification between the negation of the condition and the boolean variable $\_bool0$, the negation of the boolean which is $\#\backslash\_bool0$ will be used to represent the condition $x <= 5$.

3.3.2.5 If_Else Statement

When an if_else statement is read, the parser transforms it into a conjunction of two logical implication statements under the syntax of Sicstus Prolog CLPFD library where the first logical implication represents the if block, and the second logical implication represents the else block of the if_else statement. The left hand side of the implication that represents the if block will be the condition of the if_else statement where as the left hand side of the implication that represents the else block will be the negation of the condition of the if_else statement. This is illustrated by the sample if_else statement code given in figure 3.7 along with its corresponding constraint generated by the parser. It can be seen that the constraint generated is a conjunction of the first implication ($\_bool0\# => \_y2\# = 5$) representing the if block, and the second implication ($\#\backslash\_bool0\# => \_y2\# = 10$) representing the else block. The first implication has $\_bool0$ on its left hand side where as the second implication has ($\#\backslash\_bool0$) on its left hand side.

```
if(x>5){y=5;}  else{y=10;}

_bool0#<=>_x1#>5,
_bool0#=>_y2#=5,
 (#\_bool0)#=>_y2#=10
```

Figure 3.7: If_Else statement transformation

The else part is optional in an if_else statement as it was specified in section 3.2, and it is common to have an if_else statement with an if block only i.e without the optional else block. Even when the if_else statement consists of only the if part in the original Java program, it is usually the case that assignment statements generated by the parser to handle version matching of variable will be added to the else part in the constraint system. This can be illustrated by the sample Java code and its corresponding constraint given in figure 3.8 (It is assumed that the versions of both variables x and y were 1 when the parser read the statement). It is obvious that the first implication, $(\_x1\# > 5\# => \_y2\# = 5)$ represents the actual statement given in the Java program. But the second part, $(\#\backslash(\_x1\# > 5)\# => \_y2\# = \_y1)$, is generated and added by the parser to match the changes done by the if part so that the current version of the variable will be the same whether the if path or the else path is followed.

```
if(x>5){y=5;}

_bool0#<=>_x1#>5,
_bool0#=>_y2#=5,
(#\_bool0)#=>_y2#=_y1
```

Figure 3.8: If_Else statement without the else part

Sometimes more changes to the version of variables could be done in the else part of the if_else statement which requires addition of version matching assignments in the if part. Given below is a sample if_else statement and its corresponding constraint generated by the parser with the assumption that the version of all variables was 1 when the statement was read by the parser.

```
if(x>5){y=5;} else{y=10; z=10;}

_bool0#<=>_x1#>5,
_bool0#=>(_y2#=5#/\_z2#=_z1),
(#\_bool0)#=>(_y2#=10#/\_z2#=10)
```

Figure 3.9: If_Else statement with more changes in the else part

It can be seen that the variable z is changed only in the if block, which requires a version matching statement for the variable to be added to the if block. Therefore, the first implication that represents the if part of the statement has the assignment $\_z2\# = \_z1$ playing the role of matching the versions of the variable in the if block and else block. It is also common that some variables will be changed in the if block but not in the else and some others will be changed in the else block but not in the if block. In this case, version matching assignments need to be added to both of the implications representing the if block and else block of the if_else statement. This is illustrated by the example code given in figure 3.10 along with its corresponding constraint generated by the parser.

```
if(x>5){y=5; m=5;} else{y=10; z=10;}

_bool0#<=>_x1#>5,
_bool0#=>(_y2#=5#/\_m2#=5#/\_z2#=_z1),
(#\_bool0)#=>(_y2#=10#/\_z2#=10#/\_m2#=_m1)
```

Figure 3.10: If_Else statement with changes in both of the If and the Else parts

It is common to have nested if_else statements in the original program. When an if_else statement is nested in another one, it means that either or both of the if and else blocks of the statement have another if_else statement inside. Normally, the parser does not do anything special to handle this case, and it simply treats the block with the nested if_else statement like any other block. This can be illustrated by the nested if_else statement and its corresponding constraint shown in figure 3.11. As it is given below, the nested if_else statement $if(x > 10)\{z = 0;\}else\{z = 5;\}$ itself is transformed into the constraint $(\_bool1\# => \_z2\# = 0\#/\backslash(\#\backslash\_bool1)\# => \_z2\# = 5)$. This constraint is treated simply as an element of the if block for the outer if_else statement like the other statement $\_y2\# = 5$.

```
if(x>5){y=5; if(x>10){z=0;}  else{z=5;}}
else{y=10; z=10;}

_bool0#<=>_x1#>5,
_bool1#<=>_x1#>10,
_bool0#=>(_y2#=5#/\(_bool1#=>_z2#=0#/\(#\_bool1)#=>_z2#=5)),
(#\_bool0)#=>(_y2#=10#/\_z2#=10)
```

Figure 3.11: Transformation of nested If_Else statement

A more efficient and elegant way of representing a nested if_else statements is by giving it a special treatment from the rest of statements in the block in such a way that by just using the condition or the negation of condition of the outer if_else statement that it is nested in, the nested statement can be described independent of the rest of the block. This can be illustrated by the sample code shown in figure 3.12 together with its corresponding constraints generated by the parser. The parser can do the transformation into either of these representations.

```
if(x>5){y=5; if(x>10){z=0;} else{z=5;}}
else{y=10; if(x>0){z=-5;} else{z=-10;}}

_bool0#<=>_x1#>5,
_bool1#<=>_x1#>10,
_bool2#<=>_x1#>0,
_bool0#=>_y2#=5,
_bool0#/\_bool1#=>_z2#=0,
_bool0#/\(#\_bool1)#=>_z2#=5,
(#\_bool0)#=>_y2#=10,
(#\_bool0)#/\_bool2#=>_z2#=-5,
(#\_bool0)#/\(#\_bool2)#=>_z2#=-10
```

Figure 3.12: More efficient nested If_Else representation

3.3.2.6 While Loop

Transformation of a while loop into a constraint needs the application of a bound so that the loop is guaranteed to terminate after a fixed number of iterations in the constraints system. Therefore, unlike the case in the original Java program where the decision to continue in the loop or to break the loop is made depending on some condition, the constraint corresponding to the while loop iterates some fixed number of times, which is the bound given before hand, independent of the condition of looping given in the original program. But what is being done inside the loop during each iteration depends on the evaluation of the condition. In order to implement this concept, there are certain things the parser does whenever it reads a while loop. The first thing it does is to identify the variables which appear at the left hand side of any assignment operation in the loop. This is very important because a new version for each of these variables must be created every time the loop is executed. For each of these variables, a list is created whose first element is the current value of the variable and whose size is equal to the number of iterations we want the loop to run for or simply the bound of the loop. Additional variables the parser should identify are the ones that are only used in the loop but do not appear on the left hand side of any assignment. Since there is no any need to change the version of any of these variables, the parser does not need to create a list of versions of these variables; just the current version is enough for the loop. For example, let us exemplify what the parser does when it reads a while loop with the code fragment given in figure 3.13.

The parser identifies that there are three variables in the loop: *x*, *i*, and *sum*, out of which *x* and *sum* appear at the left hand side of some assignment where as *i* does not appear at the left hand side of any assignment. Therefore the parser creates a list containing copies of *x* and *sum* variables to be used during each iteration of the

```
int x; int i; int sum; int fsum;
x=0; i=1; sum=0;
while(x<10)
{
       x=x+i;
       sum=sum+x;
}
fsum=sum;
```

Figure 3.13: An example code with a while loop

loop. Assuming a maximum bound of 20, the parser creates a list of size 20 for each of these variables in the constraint system by adding the statements $length(\_x0List, 20)$ and $length(\_sum0List, 20)$. In addition, the current values of both of these variables must be assigned to the first elements of the corresponding lists. The parser enforces this assignment by adding the assignments $\_x0List = [\_x0|\_]$ and $\_sum0List = [\_sum0|\_]$ to the constraint system. There is no need to create copies of variable i since its value is not subject to change in the body of the loop. So, the parser will have the two lists $\_xoList$ and $\_sum0List$ containing all the copies of variables x and sum respectively each of which are going to be used during each iteration of the loop, and $\_i0$ for the variable i which remains unchanged during any iteration of the loop.

Once the parser has identified these two sets of variables, it implements the while loop as a predicate that has three sets of arguments: the lists created by the parser for each variable changed in the body of the loop, the variables that are not changed in the loop, and new variables corresponding to the variables subject to change in the loop. Each of these new variables is used to hold the value of the last copy of the variable that is subject to change. For the example above, the only additional variables required are the ones for the last copies of the variables subject to change. In order for these variables to be used as the next versions of the variables, the parser gives the name of the next versions of the original variables for these new variables. i.e for variables x and sum, whose current versions before starting the loop was both 0, their next version will be with version 1 for both which are $\_x1$ and $\_sum1$. Once these three sets of variables are identified the parser calls the predicate $whileloop0$ which represents the actual body of the loop by adding $whileloop0(\_x0List, \_sum0List, \_i0, \_x1, \_sum1)$ into the constraint system.

The next thing the parser does is to implement the predicate that plays the actual role of the while loop. The main feature of this predicate is that it takes the first element of each list for the variables subject to change in the loop, computes the new values for each of these variables as the predicates executes, assigns the new values of each variable to the second element of the list representing the variable, and finally calls itself recursively using the same arguments except for the lists representing the variable subject to change. In the recursive call, the predicate uses the tails of each of the input lists that represent the variables subject to change in the loop. What the predicate does during its execution depends on the evaluation of the condition. If the condition evaluates to true, then it does what ever is there in the body of the loop, but if the conditions evaluates to false the predicate does version matching operations by just assigning old versions of variables to the new ones. The implementation of the *whileloop*0 predicate from the example in figure 3.13 along with the complete transformation of the loop done by the parser is shown in figure 3.14.

```
length(_x0List,20),
length(_sum0List,20),
whileloop0(_x0List,_sum0List,_i0,_x1,_sum1),
fsum#=_sum1.

whileloop0([_x0],[_sum0],_,_x0,_sum0).
whileloop0([_x0,_x1|_x0ListTail],[_sum0,_sum1|_sum0ListTail],
_i0,_xFinal,_sumFinal):-
   (_x0#<10) #=> (_x1#=_x0+_i0 #/\ _sum1#=_sum0+_x1),
   #\(_x0#<10) #=> (_x1#=x0 #/\ _sum1#=_sum0),
   whileloop0([_x1|_x0ListTail],[_sum1|_sum0ListTail],
   _i0,_xFinal,_sumFinal).
```

Figure 3.14: Constraint system representing a while loop

In the input Java program, it could be the case that one or more nested while loops exist. Like the case for nested if_else statements, the parser has its own way of handling nested while loops. As it was discussed above, a while loop is transformed into a predicate consisting of mainly a set of logical implication statements such that each statement in the body of the loop forms the right hand side of some implication statement, and the condition of the loop forms the left hand side of every implication statement. What the parser does to transforms a nested loop into the set of predominantly implication statements is similar with that of a simple loop, but unlike the case for a simple loop where the left hand side of the implication statement consists of only the condition of the loop, during transformation of a nested while loop, the left hand side of each implication statement is the conjunction of all the condition of the current loop and all the loops the current loop is nested in.

Let us illustrate this transformation logic applied by the parser for nested while loops using the sample program given in figure 3.15. The program has three while loops; the first (outer most) loop is a simple loop since it is not nested in any other loop, the second loop is a nested loop since it forms part of the body of the first loop, and the third (inner most) loop is nested in two levels since it forms part of the body of the second loop which is a nested loop itself.

```
int sumOfAllComb(int P, int Q, int R)
{
  int sum; sum=0;
  int i; i=0;
  while(i<=P) {
     int j;  j=0;
     while(j<=Q){
        int k; k=0;
        while(k<=R) {
           sum=(((sum+i)+j)+k);
           k=(k+1); }
        j=(j+1); }
     i=(i+1); }
  return sum;
}
```

Figure 3.15: Sample program with nested while loops

Since the focus here is to see how the parser handles nesting of while loops, we start by discussing how the transformation is done to the inner most loop. The predicate *whileLoop*0 in figure 3.16 is the resulting predicate of transforming the inner most loop. Like in any predicate resulted from transforming a while loop by the parser, *whileLoop*0 consists of mainly logical implication statements in addition to some reification definitions of conditions. An important point to notice here is that although the only condition of the inner most loop in the Java program was $k <= R$ which is represented as $\_k1\# =< \_R0$, the left hand side of each logical implication in the predicate consists of not only this condition from the inner most loop which is reified as $\_bool2$ but also conditions from all the loops the current loop is nested in which are $\_bool1$ and $\_bool0$. It can be seen that $\_bool0$ and $\_bool1$ which are the boolean variables reifying the conditions $i <= P$ and $j <= Q$ from the original Java program are the conditions of the while loops that the inner most loop is nested in, and hence they take part in the inner most loop in conjunction with the condition of the inner most loop itself.

Let us consider the case for the second while loop in the sample program above which is neither the inner most nor the outer most loop; it is nested by the first (outer most) loop, and it also nests the third (inner most) loop. Since the inner most loop does not have any effect on the left hand side of the logical implication statements in the corresponding predicate of the second loop, the left hand side of all implication statements for the second loop will have the conjunction of the conditions of the first(outer most) and second loops. The predicate *whilLoop*1 in figure 3.17 is the transformation

```
whileLoop0([_k2],[_sum2],_P0,_Q0,_R0,_i1,_j1,_k2,_sum2).

whileLoop0([_k1,_k2|_kTail],[_sum1,_sum2|_sumTail],_P0,_Q0,_R0,
_i1,_j1,_kSol,_sumSol):-
_exBool0#<=>_i1#=<_P0,
_exBool1#<=>_j1#=<_Q0,
_exBool2#<=>_k1#=<_R0,
((_exBool0#/\_exBool1)#/\_exBool2)#=>_sum2#=(((_sum1+_i1)+_j1)+_k1),
((_exBool0#/\_exBool1)#/\_exBool2)#=>_k2#=(_k1+1),
((_exBool0#/\_exBool1)#/\(#\_exBool2))#=>_sum2#=_sum1,
((_exBool0#/\_exBool1)#/\(#\_exBool2))#=>_k2#=_k1,
whileLoop0([_k2|_kTail],[_sum2|_sumTail],_P0,_Q0,_R0,_i1,_j1,_kSol,_sumSol).
```

Figure 3.16: Representation of the inner most loop

```
whileLoop1([_j2],[_sum2],_P0,_Q0,_R0,_i1,_j2,_sum2).

whileLoop1([_j1,_j2|_jTail],[_sum1,_sum2|_sumTail],_P0,_Q0,_R0,
_i1,_jSol,_sumSol):-
_exBool0#<=>_i1#=<_P0,
_exBool1#<=>_j1#=<_Q0,
(_exBool0#/\_exBool1)#=>_k1#=0,
length(_k1List,20),
_k1List=[_k1|_],
length(_sum1List,20),
_sum1List=[_sum1|_],
whileLoop0(_k1List,_sum1List,_P0,_Q0,_R0,_i1,_j1,_k2,_sum2),
(_exBool0#/\_exBool1)#=>_j2#=(_j1+1),
(_exBool0#/\(#\_exBool1))#=>_sum2#=_sum1,
(_exBool0#/\(#\_exBool1))#=>_j2#=_j1,
whileLoop1([_j2|_jTail],[_sum2|_sumTail],_P0,_Q0,_R0,_i1,_jSol,_sumSol).
```

Figure 3.17: Representation of the second inner loop

of the second loop by the parser.

The case is trivial for the first (outer most) loop. Since this loop is not nested in any other loop, the parser transforms the loop into a predicate that contains logical implications with the left hand side of each implications has only the condition of the loop. This is exactly the same as what the parser does when there is no nesting of loops. It is important to notice here that the parser is sensitive towards loops a given loop in nested in, but is not sensitive towards loops that are nested in the given loop. This is to say that the treatment of the parser towards a while loop differs depending on whether the loop is nested in another loop or not but the treatment does not differ depending on whether the loop nests another loop or not. The complete output of the parser after transforming the sample Java program is given in figure 3.18.

```
sumOfAllComb(_P0,_Q0,_R0,_sum2):-
_P0 in -65635..65635,
_Q0 in -65635..65635,
_R0 in -65635..65635,
_sum1#=0, _i1#=0,
length(_i1List,20), _i1List=[_i1|_],
length(_sum1List,20), _sum1List=[_sum1|_],
whileLoop2(_i1List,_sum1List,_P0,_Q0,_R0,_i2,_sum2).

% Definition of predicates representing loops
whileLoop2([_i2],[_sum2],_P0,_Q0,_R0,_i2,_sum2).
whileLoop2([_i1,_i2|_iTail],[_sum1,_sum2|_sumTail],_P0,_Q0,_R0,
_iSol,_sumSol):-
_exBool0#<=>_i1#=<_P0, _exBool0#=>_j1#=0,
length(_j1List,20), _j1List=[_j1|_],
length(_sum1List,20), _sum1List=[_sum1|_],
whileLoop1(_j1List,_sum1List,_P0,_Q0,_R0,_i1,_j2,_sum2),
_exBool0#=>_i2#=(_i1+1),
(#\_exBool0)#=>_sum2#=_sum1,
(#\_exBool0)#=>_i2#=_i1,
whileLoop2([_i2|_iTail],[_sum2|_sumTail],_P0,_Q0,_R0,_iSol,_sumSol).

whileLoop1([_j2],[_sum2],_P0,_Q0,_R0,_i1,_j2,_sum2).
whileLoop1([_j1,_j2|_jTail],[_sum1,_sum2|_sumTail],_P0,_Q0,_R0,_i1,
_jSol,_sumSol):-
_exBool0#<=>_i1#=<_P0, _exBool1#<=>_j1#=<_Q0,
(_exBool0#/\_exBool1)#=>_k1#=0,
length(_k1List,20), _k1List=[_k1|_],
length(_sum1List,20), _sum1List=[_sum1|_],
whileLoop0(_k1List,_sum1List,_P0,_Q0,_R0,_i1,_j1,_k2,_sum2),
(_exBool0#/\_exBool1)#=>_j2#=(_j1+1),
(_exBool0#/\(#\_exBool1))#=>_sum2#=_sum1,
(_exBool0#/\(#\_exBool1))#=>_j2#=_j1,
whileLoop1([_j2|_jTail],[_sum2|_sumTail],_P0,_Q0,_R0,_i1,_jSol,_sumSol).

whileLoop0([_k2],[_sum2],_P0,_Q0,_R0,_i1,_j1,_k2,_sum2).
whileLoop0([_k1,_k2|_kTail],[_sum1,_sum2|_sumTail],_P0,_Q0,_R0,_i1,_j1,
_kSol,_sumSol):-
_exBool0#<=>_i1#=<_P0, _exBool1#<=>_j1#=<_Q0,
_exBool2#<=>_k1#=<_R0,
((_exBool0#/\_exBool1)#/\_exBool2)#=>_sum2#=(((_sum1+_i1)+_j1)+_k1),
((_exBool0#/\_exBool1)#/\_exBool2)#=>_k2#=(_k1+1),
((_exBool0#/\_exBool1)#/\(#\_exBool2))#=>_sum2#=_sum1,
((_exBool0#/\_exBool1)#/\(#\_exBool2))#=>_k2#=_k1,
whileLoop0([_k2|_kTail],[_sum2|_sumTail],_P0,_Q0,_R0,_i1,_j1,_kSol,_sumSol).
```

Figure 3.18: Constraints' model corresponding to the sample while loop program

3.3.2.7 Code Block

As already described in section 3.2, a code block is a sequence of one or more statements where each statement can be a declaration, an assignment, an if_else statement or a while loop. When the parser reads a generic code block $S_1;S_2;S_3$, it transforms the code block into the sequence of statements C_1,C_2,C_3 or the conjunction C_1 #/\ C_2 #/\ C_3 where C_i refers to the constraint corresponding to the statement S_i of the original program.

3.3.3 JML code to constraints transformation

JML-conditions are at the heart of the definition of both the preconditions and the postconditions as it was discussed in section 3.2. Since a JML-condition is either a condition or a sequence of conditions that are combined using the logical operators && or ||, the transformation of JML-conditions in the Java program to finite domain constraints or real constraints is the same like that of transformation of conditions in the main program. For preconditions, since they are defined in terms of JML-conditions alone, the transformation is trivial. For example, let us assume the precondition of some program is given as @ *requires* $x > 5$ && $x < 10$. This precondition will be transformed into the finite domain constraint _*x*0# > 5#/_*x*0# < 10. Optionally, the conditions can be also reified to some boolean variables which will be subsequently used instead of the conditions in the transformed precondition. i.e. _*bool*1# <=> _*x*0# > 5, _*bool*2# <=> _*x*0# < 10, _*bool*1#/_*bool*2.

Similarly, the technique of transforming postconditions of a program into the constraints system is trivial but since what is needed in the constraint system is the negation of the postconditions, some tweaking must be done to change the postcondition block which is a sequence of postcondition statements into their negation. From the section 3.2, the postcondition block is a conjunction of postcondition statements which can be represented as $A_1 \wedge A_2 \wedge \ldots \wedge A_n$, and its negation will be $\neg A_1 \vee \neg A_2 \ldots.. \vee \neg A_n$ where $\neg A_i$ represents the negation of each of the the postcondition statements. But postcondition statements are given as $C1 \rightarrow C2$ which implies that their negation will be $C_1 \wedge \neg C_2$. For example, if the parser reads a postcondition @ *ensures* @ $(x > 5)$ ==> (/*result* == 1)&& @ $(x <= 5)$ ==> (/*result* == 2), it will transform the postcondition into the finite domain constraint (_*x*0# > 5#/\(#*y*1# = 1))#\/(_*x*0# =< 5#/\(#*y*1# = 2)) assuming that y is the variable returned by the program and its final version is 1. Optionally, the conditions can be also reified to some boolean variables and the boolean variables can be used instead of the conditions in the negation of the postcondition statement generated by the parser as shown below.

```
@ensures @(x>5)==>(/result==1)&& @(x<=5)==>(/result==2)
```

will be translated into

```
(_x0#>5#/\(#\y1#=1))#\/(_x0#=<5#/\(#\y1#=2))
```

or into

```
_bool1#<=> _x0#>5,
_bool2#<=> y1#=1,
_bool3#<=> _x0#=<5,
_bool4#<=>y1#=2,
(_bool1#/\(#\_bool2))#\/(_bool3#/\(#\_bool4))
```

Figure 3.19: Representation of a simple JML specification

JML specifications that involve quantifiers on an array are transformed into constraints on a list and each element of the list is accessed by using the *element*/3 constraint. Figure 3.20 shows how a JML precondition specified using a quantifier on an array is transformed into its corresponding constraint.

```
@ requires (forall int i; i>=0 && i<arr.length-1; arr[i+1]>=arr[i])

length(_arr0,_arr0Len),
_min0#=0, _max0#= (_arr0Len-1)-1,
(for(_i0,_min0,_max0) do _i1#=_i0+1,
element(_i0,_arr0,_i0arr0Elem), element(_i1,_arr0,_i1arr0Elem),
 _i1arr0Elem#>_i0arr0Elem)
```

Figure 3.20: Example for transformation of a JML specification with quantifier

This way of transformation is good enough if we are always dealing with array of integers. However, since in this work other data types, specifically reals, are used as it will be discussed in the next chapter, a more complex way of dealing with quanitified specifications is needed. The parser handles well quantified specifications over integers but for reals a manual transformation is required.

3.3.4 Example of program to constraints transformation

The constraint system generated after transforming the sample program given in figure 3.3 is shown in figure 3.21. The maximum bound the parser assumes for the while loop is 20 in this transformation.

```
sumOfEven(_N0,_sum1):-
_N0 in -65635..65635,
_i0 in -65635..65635,
_sum0 in -65635..65635,
_i0#=0,
_sum0#=0,
length(_i0List,20),
length(_sum0List,20),
_i0List=[_i0|_],
_sum0List=[_sum0|_],
whileLoop0(_i0List,_sum0List,_N0,_i1,_sum1),
% precondition
_N0#>=0,
%postcondition
_N0m2#=(_N0 mod 2),
(_N0m2#=0#/\(#\_sum1#=(((N*N)+(2*N))/4)))#\/(_N0m2#=1#/\
(#\_sum1#=(((N*N)-1)/4))).

%Definition of whileloop0
whileLoop0([_i0],[_sum0],_,_i0,_sum0).
whileLoop0([_i0,_i1|_i0ListTail],[_sum0,_sum1|_sum0ListTail],_N0,
_iFinal,_sumFinal):-
_bool0#<=>_i0#=<_N0,
_i0m2#=(_i0 mod 2),
_bool1#<=>(_i0m2#=0),
(_bool0 #=>
(_bool1#=>_sum1#= _sum0+1) #/\ ((#\_bool1) #=> _sum1#=_sum0) #/\
 _i1#=(_i0+1)) #/\
((#\_bool0) #=>  _sum1#=_sum0 #/\ _i1#=_i0),
labeling([],[_N0,_bool0,_bool1]),
whileLoop0([_i1|_i0ListTail],[_sum1|_sum0ListTail],_N0,
_iFinal,_sumFinal).
```

Figure 3.21: Constraints' model corresponding to the example program

4

Constraint Solving Models

4.1 Introduction

Once the original program is transformed into a constraint system, the constraint solving process, which is equivalent to verification of the original program, is done on the newly generated constraint system. Therefore, the efficiency of the verification process boils down to how efficiently the generated constraint system can be solved which in turn depends on the constraint solver(s) being used. In this study, two constraint solving models are considered; a Finite Domain model that uses a finite domain solver alone to solve the system of constraints, and a Hybrid model that uses a combination of a finite domain solver and a real linear solver to solve the system of constraints. In each of these models, the constraints to be solved must be given in the syntax that is valid for the solver(s) used in the model. Since the finite domain solver and the real linear solver differ in their syntax, the parser is able to do the transformation of the original program into the system of constraints in such a way that each constraint in the system is given in the syntax of the solver it is intended to be solved in. The system of constraints generated from the input Java program in the previous chapter belongs to the finite domain model class since the use of finite domain solvers alone is enough to successfully solve the constraint system. Finite domain models are dealt in detail in section 4.2. A hybrid model - as it may be guessed from the name - makes use of both solvers of type finite domain and real domain; some constructs of the original program are transformed into finite domain constraints, while others are transformed into real constraints. Since all the constraints generated in the way discussed in the previous chapter are of type finite domain, the parser need to have additional techniques for generating real constraints for the Java constructs that are needed to be represented as real constraints. Hybrid models together with how the parser generates real constraints which form part of the system of constraints in a hybrid model are dealt with in section 4.3.

4.2 Finite Domain Model

In finite domain model, the original program is transformed into a system of finite domain constraints all of which can be solved by a finite domain solver. In this work, a finite domain solver for Sicstus Prolog called CLPFD is used to solve the constraints in a finite domain model. The constraint system generated by the parser given in the previous chapter consists entirely of finite domain constraints, and hence it can be solved with the use of a finite domain solver alone. For this reason, the default model of constraint solving is finite domain model.

One main feature of this model is that basic constructs in the original program such as unit conditiond and assignments, as well as high level constructs in the program such as more complex conditions, if_else statements, while-loops, etc are all represented in the system of constraint as finite domain constraints. Another main feature is that the standard labeling procedure defined by the solver is used to label variables in the system of constraints. There are two types of variables in the system of constraints subject to labeling during the constraint solving process. The first type consists of boolean variables created by the parser during the constraint generation to represent all the decision structures in the original Java program such as a conditions of if_else statements and while loops. Such decision structures are represented in terms of constraints reified to those boolean variables as it was discussed in section 3.3.2. The second type consists of those variables introduced into the system of constraints each of which corresponds to each variable in the original program itself.

The code given in figure 4.1 is the constraint system generated in the previous chapter for the $sumOfEven$ program. It can be taken as an example for the finite domain model since it is the default constraint generated by the parser. The first observation from this code is that labeling for the main predicate $sumOfEven$, and the predicate representing the while loop $whileLoop0$ is done separately. If a variable subject to labeling is being used in nested loops, doing the labeling in the inner most loop makes the constraint system to get solved faster than doing the labeling in the outer loops. Another

```
sumOfEven(_N0,_sum1):-
_N0 in -65635..65635,
_i0 in -65635..65635,
_sum0 in -65635..65635,
_i0#=0,
_sum0#=0,
length(_i0List,20), _i0List=[_i0|_],
length(_sum0List,20), _sum0List=[_sum0|_],
whileLoop0(_i0List,_sum0List,_N0,_i1,_sum1),
% precondition
_N0#>=0,
%postcondition
_N0m2#=(_N0 mod 2),
(_N0m2#=0#/\(#\_sum1#=(((N*N)+(2*N))/4)))#\/
(_N0m2#=1#/\(#\_sum1#=(((N*N)-1)/4))).

%Definition of whileloop0
whileLoop0([_i0,_i1|_i0ListTail],[_sum0,_sum1|_sum0ListTail],
_N0,_iFinal,_sumFinal):-
_bool0#<=>_i0#=<_N0,
_i0m2#=(_i0 mod 2),
_bool1#<=>(_i0m2#=0),
_bool0#/\_bool1#=>_sum1#= _sum0+1,
_bool0#/\(#\_bool1) #=> _sum1#=_sum0,
_bool0 #=>_i1#=(_i0+1),
(#\_bool0)#=> _sum1#=_sum0 #/\ _i1#=_i0,
labeling([],[_N0,_bool0,_bool1]),
whileLoop0([_i1|_i0ListTail],[_sum1|_sum0ListTail],_N0,_iFinal,_sumFinal).
whileLoop0([_i0],[_sum0],_,_i0,_sum0).
```

Figure 4.1: A finite domain model example

observation is that since there is no variable to be labeled in the main predicate no labeling is done. But inside the *whileLoop*0 predicate, three variables are labeled. The first variable, $\_N0$ represents the input N to the original program where as the boolean variables $\_bool0$ and $\_bool1$ reified to the constraints $\_i0\# =< \_N0$ and $\_i0m2\# = 0$ respectively refers to the conditions of the while loop and the if_else statement inside the while loop of the original Java program.

The finite domain model does not need any special labeling algorithm or reification technique. It only makes use of the facilities available in the CLPFD library of the Sicstus Prolog. However, irrespective of the solver in use, the worst case time complexity required to check satisfiability of a given system of constraints is polynomial on the number of variables being labeled in the model and the size of the domain of each of these variables. For example, to just prove the unsatisfiability of the CSP $\langle\{X,Y\},\{0..1000,0..1000\},\{X > Y, X < Y\}\rangle$, it takes the finite domain solver hundreds of steps narrowing the domains of both variables while trying to keep bound-consistency of the CSP using the given constraints as propagators. The entire process of narrowing the domains until proving of unsatisfiability is shown below:

Domain of X	Domain of Y	Narrowing constraint
0..1000	0..1000	X>Y
1..1000	0..999	X<Y
1..998	2..999	X>Y
3..998	2..997	X<Y
...	...	...
495..500	494..499	X<Y
495..498	496..499	X>Y
497..498	496..497	X<Y
{}	496..497	

Figure 4.2: Narrowing of domains to keep bounds consistency

In the first narrowing step, narrowing of the original domain 0..1000 for both variables is done using the constraint $X > Y$. Since there is no any value in the domain of Y for X taking 0 such that $X > Y$ holds, 0 is left out of the domain of X. Similarly, since there is no any value in the domain of X for Y taking 1000 such that $X > Y$ holds, 1000 is left out of the domain of Y. The second narrowing step will be done on the reduced domains, 1..1000 for X and 0..999 for Y, using the constraint $X < Y$. By continuously narrowing the domains of both variables using the two constraints alternatively, we reach a point where the domain of X is reduced to empty set implying that the CSP is unsatisfiable.

If a simple CSP with two variables of very small domain takes around 500 steps to reach unsatisfiability, then it is easy to imagine how exponentially the time complexity grows for CSPs that have constraints with more number of variables such as $X+Y \geq Z+W$, and specially if the variables have larger domains, say $-100000..100000$ instead of 0..1000.

4.3 Hybrid Model

A hybrid constraint solving model makes use of both finite domain solver and the real solver to efficiently solve the system of constraints generated by the parser. The system of constraints intended to be solved in a hybrid model by itself consists of both finite domain constraints, which need the use of the finite domain solver to be solved, and real constraints, which need a real solver to be solved. The logic in hybrid model is that rather than transforming every construct from the original program into a finite domain constraint, like that of finite domain model, some constructs from the original program are transformed into real constraints to take advantage of the fact that solving real linear constraints is not dependent on the size of the domains of variables, and some other constructs are still transformed into finite domain constraints. Therefore, the hybrid model adds at the top of the finite domain model an efficient way of solving the system of constraints by representing some important constructs from the original program as real constraints.

4.3.1 Extending a Finite Domain Model into a Hybrid Model

An important point to see here will be which of the constructs from the original program are transformed into finite domain constraints, and which are transformed into real constraints. Since the transformation of the original program into a system of constraints, specifically in the finite domain model, was discussed in chapter 3, we can consider the hybrid model as an extension of the finite domain model where some constructs from the original program are transformed into real constraints rather than finite domain constraints. This model extends the finite domain model in 3 main ways:

4.3.1.1 Declaration

Domain declarations such as $\_x0\ in\ -65635..65635$ and $domain(\_tab0, -65635, 65635)$ that are added to the constraint system by the parser when it reads type declarations such as $int\ x$ and $int[]\ tab$ are no more needed in the system of constraints for the hybrid model. This is because every variable read from the program is not considered as a finite domain variable anymore; it will be considered as a real variable, and basic constructs from the program that use such variables are transformed into real constraints.

4.3.1.2 Unit Condition

The first basic constructs from the original program that make use of variables are unit conditions. Unlike the case for the finite domain model where unit conditions are transformed into finite domain constraints, unit conditions in the hybrid model are transformed into real constraints since variables from the program are represented as real variables. For example, the unit condition $x <= 10$ is transformed into the real constraint $\_x0 =< 10$.

However, reifying a real constraint with some boolean variables and making use of the boolean variable instead of the constraint itself in places where using the constraint is needed is no more possible unlike the case for the finite domain model. In other words, reification is not possible for the real constraints representing the unit conditions. Hence another technique should be devised to form a relation between the real constraint and some boolean variable created by the parser so that the constraint (the negation of the constraint) is posted when the corresponding boolean variable has a value 1 (0 respectively). Such boolean variables that are used in the constraint system in the place of real constraints representing some unit conditions are called external booleans. The relation between a given constraint and its external boolean is shown by forming an order pair (b,C) where b is a constant that can be easily associated with some variable B which actually is used instead of the real constraint C in the constraint system. A list of such order pairs is kept in the system of constraints for all real constraints used in the system. This list is very important in solving the system of constraints because it is the only way of identifying which constraint to launch whenever some of the external booleans are assigned with some value. For example, for the constraint $\_x0 =< 10$ given above, the parser can create an external boolean say $\_exBool0$ to be used instead of the constraint itself in the system of constraints, and this relation between the boolean variable and the constraint is shown by keeping the order pair $(exBool0, \_x0 =< 10)$ in the system of constraints. It can be assumed how easily the constant $exBool0$ and the variable $\_exBool0$ can be associated to each other during constraint solving.

More complex conditions that consists of more than one unit conditions are still represented as finite domain constraints over the external booleans that represent the unit conditions. These finite domain constraints then are reified to boolean variables called internal booleans so that they can be used instead of the constraints themselves in the system of constraints. For example, let us see how the condition $(x <= 10 \&\& x >= 0)$ is transformed into a system of constraints in the hybrid model by comparing the equivalent in the finite domain model. The representation for the finite domain model is given as:

```
_bool0#<=>_x0#=<10,
_bool1#<=>_x0#>=0,
_bool2#<=>_bool0#/\_bool1,
```

The boolean variable $\_bool2$ will play the role of the condition in the rest of the system of constraints. In the hybrid model, the two unit conditions $x <= 10$ and $x >= 0$ are represented as real constraints $\_x0 =< 10$ and $\_x0 >= 0$ associated with two external boolean variables $\_exBool0$ and $\_exBool1$ respectively, the whole condition is represented as a finite domain constraint $\_exBool0\#/\backslash\_exBool1$ which is the conjunction of the two external booleans representing the two unit conditions, and this finite domain constraint is reified to an internal boolean variable $\_inBool0$. In addition, a list of order pairs is used to denote the relation between external booleans and the corresponding constraints. The complete representation of the example condition in the hybrid model is given as:

```
_inBool0#<=>_exBool0#/\_exBool1,
[(exBool0,_x0=<10),(exBool1,_x0>=0)]
```

4.3.1.3 Assignment

An assignment gets similar treatment like unit conditions; it will be transformed into a real constraint, and the constraint is associated to an external boolean that plays the role of the assignment in the rest of the system of constraints. Then an order pair is used to denote the association between the constraint and the external boolean. A list of such order pairs is formed if there are more than one such associations in the system of constraint to aid the process of solving constraints later. For example, let us consider how the assignment in the statement $if(x>5)\{y=1;\}$ is handled by the hybrid model by comparing it to how the same statement is handled by the finite domain model. The finite domain model represents the statement as:

```
_bool0#<=>_x0>5,
_bool1#<=>_y0#=1,
_bool0#=>_bool1
```

In the hybrid model, the statement is represented as:

```
_inBool0#<=>_exBool0,
_inBool0#=>_exBool1,
[(exBool0,_x0>5),(exBool1,_y0=1)],
```

Apart from the way variables, unit conditions and assignments are represented, the hybrid model represents all other constructs from the original program in the same manner like that of the finite domain model.

4.3.2 Labeling Algorithm

Like that of the finite domain model, labeling of variables is required in the hybrid model as well. However, there are differences of labeling in the two models regarding the variables that are subject to labeling and how the labeling is done to the variables. Since variables in the system of constraints representing program variables are of type real, there is no need to label these variables. Therefore, the only variables left for labeling are the decision boolean variables created during the generation of the constraints by the parser to represent decision constructs such as unit-conditions and conditions, as well as assignments. As discussed in the previous section, there are two types of such decision variables in the hybrid model: external booleans which are associated to some real constraints, and internal booleans that are reified with some finite domain constraints. Actually, only the internal booleans are labeled using the known labeling procedure provided by the CLPFD library. But at every labeling of all these internal booleans, the external booleans are checked if they have acquired some value, possibly 0 or 1. If an external boolean has acquired a value 1, the corresponding real constraint is posted, or if an external boolean has acquired a value 0, the negation of the corresponding real constraint is posted.

Solving the system of constraints in the hybrid model consists of labeling all internal booleans followed by checking the values of all the external booleans, and posting the necessary real constraints to the system for the external booleans with some value assigned. A special labeling algorithm is defined to efficiently carry out these activities during solving constraints for the hybrid model. The complete implementation of this algorithm specified in the syntax of Sicstus Prolog is shown in figure 4.3.

```
doLabeling(InBools,ExBoolVarsPair,ExBoolsConsPair):-
domain(InBools,0,1),
(foreach((_,ExBools2El),ExBoolVarsPair) do ExBools2El in 0..1),
my_label(InBools,ExBoolVarsPair,ExBoolsConsPair).

my_label(Ibs, ExBoolVarsPair, ExBoolsConsPair):-
labeling([],Ibs), my_lab(ExBoolVarsPair, ExBoolsConsPair).

my_lab([],_).
my_lab([(N,V)|Vs],[(N,C)|Cs]):-
indomain(V),apply_v(V,C), my_lab(Vs,Cs).

apply_v(1,Rel):- {Rel}.
apply_v(0,Rel):- neg(Rel,Neg),{Neg}.

neg(A  =  B, A =\= B).
neg(A =\= B, A  =  B).
neg(A  >  B, A =<  B).
neg(A =<  B, A  >  B).
neg(A  <  B, A >=  B).
neg(A >=  B, A  <  B).
```

Figure 4.3: Sicstus Prolog implementation of the labeling algorithm

4.3.3 An Example for the Hybrid Model

The system of constraints generated in finite domain model for the sample program given in figure 3.3 was shown in figure 3.21. The system of constraints generated in hybrid model for the same program is given in figure 4.4.

```
sumOfEven(_N0,_sum1):-
_exBool0,
_exBool1,
length(_i0List,20),
length(_sum0List,20),
_i0List=[_i0|_],
_sum0List=[_sum0|_],
whileLoop0(_i0List,_sum0List,_N0,_i1,_sum1),
% precondition
_exBool9,
%postcondition
_exBool10,
(_exBool11#/\(#\_exBool12))#\/(_exBool13#/\(#\_exBool14)),
InBools = [],
ExBools =  [(exBool0,_exBool0),(exBool1,_exBool1),
(exBool9,_exBool9),(exBool10,_exBool10),(exBool11,_exBool11),
(exBool12,_exBool12),(exBool13,_exBool13),(exBool14,_exBool14)],
ExCons = [(exBool0,_i0=0),(exBool1,_sum0=0),(exBool9,_N0>=0),
(exBool10,_N0m2=(_N0 mod 2)),(exBool11,_N0m2=0),
(exBool12,_sum1=(((N*N)+(2*N))/4)),(exBool13,_N0m2=1),
(exBool14,_sum1=(((N*N)-1)/4))],
doLabeling(InBools, ExBools,ExCons ).

%Definition of whileloop0
whileLoop0([_i0],[_sum0],_,_i0,_sum0).
whileLoop0([_i0,_i1|_i0ListTail],[_sum0,_sum1|_sum0ListTail],
_N0,_iFinal,_sumFinal):-
_inBool0#<=>_exBool2,
_exBool3,
_inBool1#<=>_exBool4,
(_inBool0 #=>
(_inBool1#=>_exBool5)#/\ ((#\_bool1)#=>_exBool6)#/\_exBool7)#/\
((#\_inBool0) #=>  _exBool6 #/\ _exBool8),
InBools0 = [_inBool0,_inBool1],
ExBools0 = [(exBool2,_exBool2),(exBool3,_exBool3),(exBool4,_exBool4),
(exBool5,_exBool5),(exBool6,_exBool6),(exBool7,_exBool7),
(_exBool8,_exBool8)],
ExCons0 = [(exBool2,_i0=<_N0),(exBool3,_i0m2=(_i0 mod 2)),
(exBool4,_i0m2=0),(exBool5,_sum1=_sum0+_i0),(exBool6,_sum1=_sum0),
(exBool7,_i1=_i0+1),(_exBool8,_i1=_i0)],
doLabeling(InBools0,ExBools0,ExCons0),
whileLoop0([_i1|_i0ListTail],[_sum1|_sum0ListTail],_N0,
_iFinal,_sumFinal).
```

Figure 4.4: Representation of the example program in the hybrid model

The while loop is assumed to have a maximum bound of 20 similar to the assumption that was done for the finite domain model. The *doLabeling* predicate calls the special labeling algorithm that is given in figure 4.3.

5

Experimental Results

5.1 Introduction

In this work, the hybrid model for solving constraints is proposed that involves combining a finite domain solver and a real solver to efficiently solve the system of constraints generated by the parser for some program to be verified in an effort to verify the program. In order to evaluate the actual performance of the model, it is important to make a comparison between the performance results achieved by the model and some other available and commonly used frameworks of program verification. In this chapter, we present how the hybrid model has performed with respect to other common program verification frameworks and tools. The performance of all of the frameworks involved in the comparison, including the one suggested by this work, is given in terms of the time it takes each framework to verify some standard benchmarks for program verification.

In section 5.2, a brief introduction of the other frameworks involved in the performance comparison will be discussed followed by another brief introduction of the benchmarks used for the comparison in section 5.3. Finally, the actual performances of each of the framework is presented for each of the benchmarks is section 5.4.

5.2 Frameworks Considered for Comparison

The performance of the hybrid model is compared with six other frameworks and tools developed for verifying programs. These frameworks and tools are developed by different people and they target programs written in different languages. The verification mechanisms employed by the frameworks and tools also vary in general.

5.2.1 ESC/Java

ESC/Java (Extended Static Checker for Java) [36] is a program verification tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations during compile time. ESC/-Java uses extended static checking, which is a collective name referring to a set of techniques for statically checking if the program satisfies its desired correctness properties. Extended static checking techniques often employ an automated theorem prover as a verification method, and ESC/Java used the Simplify theorem prover.

5.2.2 CBMC

CBMC [37] is a Bounded Model Checker for ANSI-C and C++ programs. It first transforms the program into a control-flow graph (CFG), and with the assumption that desired properties of the program are given in the form of assertions, follows paths through the CFG to each of these assertions, and build formulas that corresponds to each of such paths. It then uses SAT solvers to check if the formulas built are satisfiable. CBMC allows verifying array bounds (buffer over-flows), pointer safety, exceptions and user-specified assertions.

5.2.3 BLAST

BLAST(Berkeley Lazy Abstraction Software Verification Tool) [38] is an automatic verification tool for C programs which is based on Model Checking approach. The goal of BLAST is to check if a given program satisfies its temporal safety properties or not. Blast constructs, explores, and refines abstractions of the program state space, also called abstract model of the program, based on lazy predicate abstraction and interpolation-based predicate discovery. Then this abstract model is model checked for safety properties.

5.2.4 EUREKA

EUREKA [39] is a symbolic model checker for Linear Programs with arrays, i.e. programs where variables and array elements range over a numeric domain, and expressions involve linear combinations of variables and array elements. Desired correctness properties are specified by addind them in the form of assertions to the program, and the EUREKA tool checks if the assertions are reachable, and hence the correctness properties hold. The fragment of C programming language handled by the tool support a number of features such as arbitrarily nested loops, non-determinism, etc. EUREKA interprets the counterexample guided abstraction refinement (CEGAR) paradigm in a novel way by using array indexes instead of predicates. A defining feature of EUREKA is that it abstracts the program with respect to a family of sets of array indexes; the abstraction is a Linear Program (without arrays), and refinement searches for new array indexes.

5.2.5 WHY

Why is a software verification platform that takes annotated programs written in a very simple imperative programming language of its own, generates verification conditions, and sends them to existing provers (proof assistants such as Coq, PVS, HOL 4, etc or decision procedures such as Simplify, Yices, etc) to prove that the generated verification conditions hold. Since it has tools for translating Java and C programs into Why programs, Why can handle programs written in both of these languages [40]. The Why tool is capable of declaring logical models, which are types, functions, predicates, axioms and lemmas, that can be used in programs and annotations. The fact that Why tool supports a number of existing provers allows verification conditions to be discharged by several provers independently.

5.2.6 CPBPV

CPBPV (Constraint Programming framework for Bounded Program Verification) [41] is a constraint programming framework for bounded program verification that uses constraint stores to represent the specification and the program, and explores execution paths non-deterministically to verify the conformity of a program with its specification. An input program whose precondition and postcondition are given is said to be partially correct if each constraint store produced from the program and the precondition implies the postcondition. CPBPV does not explore spurious execution paths as it incrementally prunes execution paths early by detecting that the constraint store is not consistent. This framework uses the rich language of constraint programming to express the constraint store for the program.

CPBPV is the closest one to our framework since it is based on bounded-model checking approach, and it applies techniques of constraint programming. However, CPBPV uses finite domain constraints to express the original program, and finite domain solvers to solve the system of constraints corresponding to the program. Our framework, on the other hand, uses both finite domain constraints and real linear constraints to express the original program, and uses a hybrid solver, which combines both finite domain solvers and real linear solvers, to make efficient constraint solving which results in efficient verification of the program.

5.3 Benchmark Programs Used

5.3.1 Triangle Classification

The tritype program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible paths: only 10 paths correspond to actual inputs because of complex conditional statements in the program. The program takes three positive integers as inputs (the triangle sides) and returns 2 if the inputs correspond to an isoscele triangle, 3 if they correspond to an equilateral triangle, 1 if they correspond to some other triangle, and 4 otherwise. The complete program written in Java and JML is shown in figure 5.1.

5.3.2 Binary Search

The second benchmark is a binary search program which determines if a value *x* is found in a sorted array *tab* or not. If it is found, the program returns the *index* of *x*, otherwise, it simply returns -1. The tritype program has a number of execution paths but it neither has array variables not while loops. The binary search program has an array variable and a while loop which makes it more complex than the tritype program, and hence it is useful to show how program verification tools can cope up with this added difficulties. The complete binary search program together with its specification is given in figure 5.2.

```
/*@ requires (i >= 0 && j >= 0 && k >= 0);
@ ensures
 (((i+j)<=k||(j+k)<= i||(i+k)<= j)==>(\result==4))
 &&((!((i+j)<=k||(j+k)<= i||(i+k)<=j)&&(i==j&&j==k))==>(\result==3))
 &&((!((i+j)<= k||(j+k)<= i||(i+k)<=j)&&!(i==j&&j==k)&&
(i==j||j==k||i==k))==>(\result==2))
 &&((!((i+j)<=k||(j+k)<= i||(i+k)<= j)&&!(i==j&&j==k)&&
!(i==j||j==k||i==k))==>(\result==1));
*/
int tritype (int i, int j, int k)
{
int trityp;
if (i == 0 || j == 0 || k == 0) {trityp = 4;}
else {
   trityp = 0;
   if (i == j) {trityp = trityp + 1;}
   if (i == k) {trityp = trityp + 2;}
   if (j == k) {trityp = trityp + 3;}
   if (trityp == 0) {
       if ((i+j) <= k || (j+k) <= i || (i+k) <= j) {trityp = 4;}
        else {trityp = 1;}
    }
   else {
            if (trityp > 3) {trityp = 3;}
            else {
                 if (trityp == 1 && (i+j) > k) {trityp = 2;}
                 else {
                     if (trityp == 2 && (i+k) > j) {trityp = 2;}
                                                   % Error trityp==1
                     else {
                         if (trityp == 3 && (j+k) > i) {trityp = 2;}
                         else {trityp = 4;}
                              }}}}}
  return trityp;
}
```

Figure 5.1: Triangle Classification

5.3.3 Bubble Sort with Initial Condition

The third benchmark is a program that performs a bubble sort of an array *tab* which contains integers from 0 to $t-1$ in decreasing order where t is the length of the array *tab*. This program has more complexity level than both the binary search and the tritype program for two main reasons. The first reason is that this program not only needs to read and use the values in the array variable but also needs to assign new values at some positions in the array. The second reason is that there is a nested while loop in this program which further adds complexity to the program.

```
/*@ requires(\forall int i;(i>=0&& i<tab.length-1);
     tab[i]<=tab[i+1]);
   @ ensures
   @ ((\result==-1)==>(\forall int i; (i>=0&& i<tab.length);
     tab[i]!=x))
   @ && ((\result!=-1)==>(tab[\result]==x));
   @*/
int binarySearch (int[] tab, int x)
{
   int index = -1;
   int m=0;
   int l=0;
   int u=tab.length-1;
   while (index==-1 && l<=u) {
      m=(l+u)/2;
      if (tab[m]==x) {index=m;}
      else {
         if (tab[m]>x) {u=m-1;}
         else {l=m+1;} % Error u=m-1
      }
  }
  return index;
}
```

Figure 5.2: Binary Search

The complete bubble sort program together with its specification is shown in figure 5.3.

```
/* @ requires(\forall int i;0<= i&& i<tab.length;
     tab[i]=tab.length-1-i);
   @ ensures(\forall int i;0<= i&& i<tab.length-1;
     tab[i]<=tab[i+1]);
*/
int[] tri(int[] tab)
{
   int i=0;
   while (i<tab.length-1){
      int j=0;
      while (j < tab.length-i-1) {
         while (j < tab.length-i-1) {
            if (tab[j]>tab[j+1]) {
               int aux = tab[j]; tab[j]= tab[j+1]; tab[j+1] = aux;}
             j=j+1;}
         i=i+1;}
      }
   return tab;
}
```

Figure 5.3: Bubble Sort with Initial Condition

5.4 Comparative Results

Most of the results given in this work for the frameworks and tools that are briefly introduced in section 5.2 are taken from [42]. Although we have tried to cross check these results by running tests on our machine for CPBPV, ESC/Java, CBMC, and BLAST, we have succeeded only in running CBMC; the tool for CPBPV was not available in the web, and we could not succeed in running ESC/Java and BLAST on our machine although they are freely available. The results taken from [42] were produced on a machine with Intel Pentium(R) M @1.86 GHz processor, and 1.5GB of RAM. The machine that we used to run the tests for our framework and CBMC tool has Intel Pentium(R) dual core CPU @2.00GHz processor, and 4GB of RAM.

Before going directly to the experimental results, we introduce four terms that are used to describe some special cases of performances in [42]. These terms are: UNABLE which means that the framework is unable to validate the program, TIMEOUT which means that it takes more than 6000 seconds to validate the program, NOT_FOUND which means that it is unable to find an error when an incorrect program is given, and finally FALSE_ERROR which means that it finds an "error" in a correct program. For CBMC, we use the name *CBMC* 4.0 to differentiate the results we have obtained on our machine from the results we have taken from [42] for which we simply use *CBMC*. In addition, we use the abbreviations HM and FDM for the Hybrid Model and the Finite Domain Model of our framework respectively.

5.4.1 Triangle Classification

The BLAST tool is unable to verify the given triangle classification program due to its internal working logic, but it is able to verify a simpler version of the program whose results are given here. The experimental results for both correct and erroronous versions of the program are given in tables 5.1 and 5.2 respectively.

Tool	Hybrid Model	Finite Domain Model	CBMC 4.0	CPBPV
Time(sec)	0.037	2.543	0.197	0.287
Tool	ESC/Java	CBMC	Why	BLAST
Time(sec)	1.828	0.820	8.850	0.716

Table 5.1: triangle classification without an error

For both cases, the hybrid model of our framework has taken shorter time than the other tools to complete the verification task. However, the finite domain model

Tool	Hybrid Model	Finite Domain Model	CBMC 4.0	CPBPV
Time(sec)	0.003	0.468	0.056	0.056
Tool	ESC/Java	CBMC	Why	BLAST
Time(sec)	1.853	NOT_FOUND	NOT_FOUND	0.452

Table 5.2: triangle classification with an error

performance is only better than the Why tool for the case without an error, and the Why and CBMC tools for the case with error. The reason for this performance of the finite domain model is the existence of many comparisons that involve variables in the tritype program that need a number of steps to solve the resulting constraints system as illustrated in figure 4.2.

5.4.2 Binary Search

The expermental results for the binary search program are given for arrays of length 8, 16, 32, 64, 128 and 256. Since there is a while loop in the program, all frameworks except the CPBPV require additional information in the form of loop invariant or loop unfolding bound. ESC/Java, CBMC, and the hybrid model require an overestimate of the number of loop unfoldings whereas the Why framework requires an invariant to work. For the correct version, ESC/Java returns a false error whereas BLAST is unable to do the verification for array of any length. The Why tool returns UNABLE if no invariant is given, but if an invariant is given, it achieves an efficient performance by verifying the program in 11.18 seconds irrespective of the size of the array. However, for the erroneous version, it always returns NOT_FOUND. The experimental results for both correct and erroneous versions of the binary search program for the rest of the tools are given in tables 5.3 and 5.4 respectively.

Tool	HM	FDM	CBMC 4.0	CPBPV	CBMC
Time(sec) for length 8	0.033	Time out	31.049	1.081	1.370
Time(sec) for length 16	0.198	Time out	467.821	1.690	1.430
Time(sec) for length 32	1.536	Time out	Time out	4.043	Time out
Time(sec) for length 64	23.634	Time out	Time out	17.009	Time out
Time(sec) for length 128	503.307	Time out	Time out	136.800	Time out
Time(sec) for length 256	Time out	Time out	Time out	1731.696	Time out

Table 5.3: binary search without an error

The finite domain model is too inefficient for the case without error like the case for the triangle classification program(and for the same reason); there are comparisons

Tool	HM	FDM	CBMC 4.0	CPBPV	CBMC	ESC/Java
Time(sec) for length 8	0.005	0.000	0.167	0.027	1.380	1.210
Time(sec) for length 16	0.008	0.006	0.449	0.037	1.690	1.347
Time(sec) for length 32	0.047	0.006	1.052	0.064	7.620	1.792
Time(sec) for length 64	0.258	0.006	3.802	0.115	27.050	1.886
Time(sec) for length 128	2.003	0.009	9.831	0.241	189.200	1.964

Table 5.4: binary search with an error

with each element of the array variable that make the model to rely on enumeration which makes it so slow. However, since the error can be uncovered in the first enumeration of variables, the finite domain model performance was the best for the case with an error. For the remaining tools, in both cases, the hybrid model has performed better than the other tools for an array size of up to 32. However, when the size of the array is getting larger than 32, the hybrid model's performance declines. The reason is that the while loop is represented as a predicate whose number of execution depends on the size of the array and there is no smart way of handling the execution when the loop conditions do not hold any more.

5.4.3 Bubble Sort

The experimental result for bubble sort given in [42] does not include the results of BLAST and Why frameworks, and also the results given are only for the correct version of the program. We have given the experimental results of the rest of the frameworks for the correct version of the program in table 5.5.

Tool	HM	FDM	CBMC 4.0	CPBPV	ESC/Java	CBMC	EUREKA
Time(sec) length 8	0.063	0.053	0.192	0.031	3.778	1.110	91.000
Time(sec) length 16	0.367	0.345	0.955	0.032	Unable	2.010	Unable
Time(sec) length 32	2.628	2.364	5.385	Unable	Unable	6.100	Unable
Time(sec) length 64	21.450	Unable	36.148	Unable	Unable	37.650	Unable

Table 5.5: buble sort without an error

For this benchmark, the finite domain model of our framework performs as good as the hybrid model for array size of 8, 16 and 32. The reason is that the program does not deal with any variable i.e. the input array comprises integers from 0 to N-1 in the reverse order where N is the length of the array, and hence no labeling is involved. The hybrid model is slightly delayed since the special labeling that would be useful in the case of variables adds extra overhead and makes the constraint solving a little slower when the program does not deal with any variable but only with integer constants.

The finite domain model performs better than all the tools except CPBPV for array sizes of 8 and 16, and its performance is the best for array size 32. However, for array size of 64, it runs out of memory due to the need to create a new array every time there is an array assignment. The hybrid model's performance is the best for array of size 64.

6

Conclusions

6.1 Summary

In general, the aim of this thesis has been to show that an efficient program verification framework can be devised by using constraint solvers as a tool, and to demonstrate this by devising such verification framework for programs written in some subset of Java and whose specification is given in some subset of JML.

We began in Chapter 1 by discussing how high reliance of daily life routines of human beings on computer systems demands a highly reliable software system, and how critical it is to ensure program correctness. This is followed by a brief description of the technique applied in this work to do program verification more efficiently. In Chapter 2, we introduce the notions of program verification and program correctness followed by a detailed discussion of some of the major program verification methods. This is then followed by a detailed discussion of constraint programming and constraint solving techniques. Chapter 3 starts by a brief discussion on the scope of the work regarding what class of Java programs are handled together with the strategy employed to verify a given program. Then, the subsets of Java and JML considered in the work for writing the program and its specification are given respectively. This is followed by the discussion of how each of the constructs in Java and JML is transformed into an equivalent constraint so that a semantically equivalent system of constraints is generated for some program given in the subset of Java whose specification given in the subset of JML. In Chapter 4, the constraint solving models considered in this

work to enable efficient way of solving the system of constraints generated for the program to be verified are given. The efficiency of the entire program verification process is highly dependent on the efficiency of the constraint solving. In Chapter 5, the experimental results of our approach are presented in comparison with other commonly used program verification tools and benchmarks using some standard benchmark programs.
The primary contributions of this thesis can be summarized as follows.

- A survey of some of the most important software verification methods for general purpose programming languages.
- A specific purpose parser for transforming a given program into system of constraints. Although the parser can handle only some simple subset of Java, it can be easily extended by adding more Java constructs to its language.
- An efficient program verification framework for verifying programs written in some subset of Java which is the main contribution of the work.

6.2 Future Work

This work focused on verification of programs written in a subset of Java language whose specifications are given in a subset of JML. The general directions for future work will be to extend the framework by incorporating more Java and JML constructs so that it can eventually be used to verify a wider range of Java programs. One possible extension is to allow more data types such as float, double, multidimensional arrays, etc to be used in programs. Another extension can be enabling the use of non-linear expressions in the program. In this work, when the parser reads expressions $x >= y$ and $x < y$, it automatically recognizes that they are opposite of one another which is very important for efficient solving of the resulting constraint system. However, the parser can not automatically conclude $\neg(p \vee q)$ and $\neg p \wedge \neg q$ are opposite of one another. This is one more area for future work which can make the constraint solving even more efficient. In the previous section, it was mentioned that the current logics of representing while-loops and array assignments were responsible for some inefficient performance of the framework. Another area of future work, therefore, is an efficient way of representing while-loops and array assignments. Quantified JML specifications are important areas of future work as we do not have a complete way of generating constraints automatically for such specifications.

Bibliography

[1] Krzysztof R. Apt, Frank de Boer, and Ernst-Rdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[2] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Commun. ACM*, 54:88–98, May 2011.

[3] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. Technical report, Austin, TX, USA, 1988.

[4] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22:271–280, May 1979.

[5] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[6] Edsger W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In *Proceedings of the international conference on Reliable software*, pages 2–2.13, New York, NY, USA, 1975. ACM.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *communications of the ACM*, 12(10):576–580, 1969.

[8] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. pages 103–122. Springer-Verlag, 2001.

[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java, 2002.

[10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *SIGPLAN Not.*, 37:58–70, January 2002.

[11] Natarajan Shankar. *Metamathematics: Machines, and Goedel's proof*. Cambridge University Press, 1994.

[12] P. Manolios M. Kaufmann and J S. Moore. Acl2 case studies. In *Computer-Aided Reasoning*. Kluwer Academic Publishers, June, 2000.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[14] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[15] Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Birthday ...*, pages 208–224, 2003.

[16] Gerard J. Holzmann. Trends in software verification. In *In: Proceedings of the Formal Methods Europe Conference*, 2003.

[17] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Int. J. Softw. Tools Technol. Transf.*, 5:247–267, March 2004.

[18] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[19] Patrice Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem, 1995.

[20] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[21] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[22] Yonit Kesten, Oded Maler, M. Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich ssertional languages. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 424–435, London, UK, 1997. Springer-Verlag.

[23] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10 20 states and beyond, 1990.

[24] Sagar Chaki, Edmund Clarke, and Alex Groce. Modular verification of software components in c. In *IEEE Transactions on Software Engineering*, pages 385–395, 2003.

[25] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. pages 72–83. Springer-Verlag, 1997.

[26] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[27] S. Park W. Visser G. Brat, K. Havelund. Java pathfinder - a 2nd generation of a java model checker. In *Proceedings of Workshop on Advances in Verification*, 2000.

[28] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[29] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder, 1998.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[31] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking and Abstract Interpretation*, pages 298–309, 2003.

[32] David W. Currie, Billerica Ma, Alan J. Hu, Sreeranga Rajan, Masahiro Fujita, and Sunnyvale Ca. Automatic formal verification of dsp software, 2000.

[33] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking, 2003.

[34] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, New York, NY, USA, September 2003.

[35] Mats Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, Kista, Sweden, September 2010.

[36] KindSoftware. Esc/java2 summary. http://kind.ucd.ie/products/opensource/ESCJava2/. Accessed August 12, 2011.

[37] Carnegie Mellon. Bounded model checking for ansi-c. http://www.cprover.org/cbmc/. Accessed August 12, 2011.

[38] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[39] Alessandro Armando, Massimo Benerecetti, Dario Carotenuto, Jacopo Mantovani, and Pasquale Spica. The eureka tool for software model checking. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 541–542, New York, USA, 2007.

[40] Jean christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *In CAV '07*, pages 173–177, 2007.

[41] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. *Computing Research Repository*, pages 327–341, 2008.

[42] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Comparison between cpbpv, esc/java, cbmc, blast, eureka and why for bounded program verification. *Computing Research Repository*, abs/0808.1, 2008.

A

System of constraints for the benchmark programs

A.1 Tritype - Hybrid Model

```
ver_tritype([A,B,C,Ty]):-
Dummy=Dummy,
statistics(runtime,[_,_]),
tritypeHM(A,B,C,Ty),
nl,write([A,B,C,Ty]),
statistics(runtime,[_,T]), write('runtime (in ms) '= T),nl.

ver_tritype(_):-
     write('programa verified.................'),
 statistics(runtime,[_,T]), write('runtime (in ms) '= T),nl.

% System of constraints for the tritype program ********
tritypeHM(_i0,_j0,_k0,_trityp5):-
(_inBool0#<=>((_exBool0#\/_exBool1)#\/_exBool2)),
_inBool0#=>_exBool3#/\_exBool26#/\_exBool7#/\
_exBool10#/\_exBool13,
(#\_inBool0)#=>_exBool4,
(_inBool1#<=>_exBool5),
(#\_inBool0#/\_inBool1)#=>_exBool6,
(#\_inBool0#/\(#\_inBool1))#=>_exBool7,
(_inBool2#<=>_exBool8),
(#\_inBool0#/\_inBool2)#=>_exBool9,
(#\_inBool0#/\(#\_inBool2))#=>_exBool10,
(_inBool3#<=>_exBool11),
(#\_inBool0#/\_inBool3)#=>_exBool12,
(#\_inBool0#/\(#\_inBool3))#=>_exBool13,
(_inBool4#<=>_exBool14),
(_inBool5#<=>((_exBool15#\/_exBool16)#\/_exBool17)),
(#\_inBool0#/\(_inBool4#/\_inBool5))#=>_exBool18,
(#\_inBool0#/\(_inBool4#/\(#\_inBool5)))#=>_exBool19,
(_inBool6#<=>_exBool20),
(#\_inBool0#/\(#\_inBool4#/\_inBool6))#=>_exBool21,
(_inBool7#<=>(_exBool22#/\(#\_exBool15))),
(#\_inBool0#/\(#\_inBool4#/\(#\_inBool6#/\_inBool7)))#=>_exBool23,
(_inBool8#<=>(_exBool24#/\(#\_exBool17))),
(#\_inBool0#/\(#\_inBool4#/\(#\_inBool6#/\(#\_inBool7#/\_inBool8))))
#=>_exBool23, % Error _exBool19, Correct _exBool23
(_inBool9#<=>(_exBool25#/\(#\_exBool16))),
(#\_inBool0#/\(#\_inBool4#/\(#\_inBool6#/\(#\_inBool7#/\
(#\_inBool8#/\_inBool9)))))#=>_exBool23,
(#\_inBool0#/\(#\_inBool4#/\(#\_inBool6#/\(#\_inBool7#/\
(#\_inBool8#/\(#\_inBool9))))))#=>_exBool18,

% Precondition
((_exBool27#/\_exBool28)#/\_exBool29),

% Postconditions
(((_exBool15#\/_exBool16)#\/_exBool17)#/\(#\_exBool18))#\/
(((#\((_exBool15#\/_exBool16)#\/_exBool17))#/\
(_exBool5#/\_exBool11))#/\(#\_exBool21))#\/
((((#\((_exBool15#\/_exBool16)#\/_exBool17))#/\
(#\(_exBool5#/\_exBool11)))#/\((_exBool5#\/_exBool11)#\/_exBool8))
#/\(#\_exBool23))#\/((((#\((_exBool15#\/_exBool16)#\/_exBool17))
#/\(#\(_exBool5#/\_exBool11)))#/\(#\((_exBool5#\/_exBool11)#\/
_exBool8)))#/\(#\_exBool19)),

ExBoolCons =
[(exBool0,_i0=0),(exBool1,_j0=0),(exBool2,_k0=0),(exBool3,_trityp1=4),
(exBool4,_trityp1=0),(exBool5,_i0=_j0),(exBool6,_trityp2=(_trityp1+1)),
(exBool7,_trityp2=_trityp1),(exBool8,_i0=_k0),(exBool9,_trityp3=
(_trityp2+2)),(exBool10,_trityp3=_trityp2),(exBool11,_j0=_k0),
(exBool12,_trityp4=(_trityp3+3)),(exBool13,_trityp4=_trityp3),
(exBool14,_trityp4=0),(exBool15,(_i0+_j0)=<_k0),
(exBool16,(_j0+_k0)=<_i0),(exBool17,(_i0+_k0)=<_j0),
(exBool18,_trityp5=4),(exBool19,_trityp5=1),(exBool20,_trityp4>3),
(exBool21,_trityp5=3),(exBool22,_trityp4=1),(exBool23,_trityp5=2),
(exBool24,_trityp4=2),(exBool25,_trityp4=3),(exBool26,_trityp5=
_trityp1),(exBool27,_i0>=0),(exBool28,_j0>=0),(exBool29,_k0>=0)],

InBools = [],
ExBoolVars =
 [(exBool0,_exBool0),(exBool1,_exBool1),(exBool2,_exBool2),(exBool3,
_exBool3),(exBool4,_exBool4),(exBool5,_exBool5),(exBool6,_exBool6),
(exBool7,_exBool7),(exBool8,_exBool8),(exBool9,_exBool9),(exBool10,
_exBool10),(exBool11,_exBool11),(exBool12,_exBool12),(exBool13,
_exBool13),(exBool14,_exBool14),(exBool15,_exBool15),(exBool16,
_exBool16),(exBool17,_exBool17),(exBool18,_exBool18),(exBool19,
_exBool19),(exBool20,_exBool20),(exBool21,_exBool21),(exBool22,
_exBool22),(exBool23,_exBool23),(exBool24,_exBool24),(exBool25,
_exBool25),(exBool26,_exBool26),(exBool27,_exBool27),(exBool28,
_exBool28),(exBool29,_exBool29)],

doLabeling(InBools,ExBoolVars,ExBoolCons).
```

A.2 Tritype - Finite Domain Model

```
ver_tritype([A,B,C,Ty]):-
Dummy=Dummy,
statistics(runtime,[_,_]),
tritypeFDM(A,B,C,Ty),
nl,write([A,B,C,Ty]),
statistics(runtime,[_,T]), write('runtime (in ms) '= T),nl.

ver_tritype(_):-
     write('programa verified................'),
 statistics(runtime,[_,T]), write('runtime (in ms) '= T),nl.

% System of constraints for the tritype program ********
tritypeFDM(_i0,_j0,_k0,_trityp5):-
domain([_i0,_j0,_k0],-65635,65635),
domain([_trityp0,_trityp1,_trityp2,_trityp3,_trityp4,_trityp5],0,6),
B0#<=>(B01#\/B02#\/B03),
B01#<=>(_i0#=0),
B02#<=>(_j0#=0),
B03#<=>(_k0#=0),
B0#=>_trityp1#=4,
B0#=>_trityp5#=_trityp1,
(#\B0)#=>_trityp1#=0,
B1#<=>(_i0#=_j0),
((#\B0)#/\B1)#=>_trityp2#=(_trityp1+1),
((#\B0)#/\(#\B1))#=>_trityp2#=_trityp1,
B2#<=>(_i0#=_k0),
((#\B0)#/\B2)#=>_trityp3#=(_trityp2+2),
((#\B0)#/\(#\B2))#=>_trityp3#=_trityp2,
B3#<=>(_j0#=_k0),
((#\B0)#/\B3)#=>_trityp4#=(_trityp3+3),
((#\B0)#/\(#\B3))#=>_trityp4#=_trityp3,
B4#<=>(_trityp4#=0),
B5#<=>((B11#\/B12)#\/B13),
B11#<=>((_i0+_j0)#=<_k0),
B12#<=>((_i0+_k0)#=<_j0),
B13#<=>((_j0+_k0)#=<_i0),
((#\B0)#/\(B4#/\B5))#=>_trityp5#=4,
((#\B0)#/\(B4#/\(#\B5)))#=>_trityp5#=1,
B6#<=>(_trityp4#>3),
((#\B0)#/\((#\B4)#/\B6))#=>_trityp5#=3,
B7#<=>((_trityp4#=1)#/\(#\B11)),
((#\B0)#/\((#\B4)#/\((#\B6)#/\B7)))#=>_trityp5#=2,
B8#<=>((_trityp4#=2)#/\(#\B12)),
(#\B0#/\(#\B4#/\(#\B6#/\(#\B7#/\B8))))#=>_trityp5#=2,
% Error _trityp5#=1, Correct _trityp5#=2
B9#<=>((_trityp4#=3)#/\(#\B13)),
(#\B0#/\(#\B4#/\(#\B6#/\(#\B7#/\(#\B8#/\B9)))))#=>_trityp5#=2,
(#\B0#/\(#\B4#/\(#\B6#/\(#\B7#/\(#\B8#/\(#\B9))))))#=>_trityp5#=4,

%Precondition
(((_i0#>=0)#/\(_j0#>=0))#/\(_k0#>=0)),

%Postconditions
(B5#/\(#\(_trityp5#=4)))#\/
(((#\B0)#/\(#\B5)#/\((B1#/\B2)#/\B3))#/\(#\(_trityp5#=3)))#\/
((((#\B0)#/\(#\B5)#/\(#\((B1#/\B2)#/\B3)))#/\((B1#\/B2)#\/B3))#/\
(#\(_trityp5#=2)))#\/((((#\B0)#/\(#\B5)#/\(#\((B1#/\B2)#/\B3)))#/\
(#\((B1#\/B2)#\/B3)))#/\(#\(_trityp5#=1))),

labeling([],[B0,B01,B02,B03,B1,B2,B3,B4,B5,B6,B9,B7,B8,B11,
B12,B13,_i0,_j0,_k0]).
```

A.3 Binary Search - Hybrid Model

```
:- use_module(commonLib).
ver_bin([N,L,X,I]):-
statistics(runtime,[_,_]),
length(L,N),
pre_condition(L),
bsHM(N,L,X,I),
neg_pos_condition(I,X,L),
statistics(runtime,[_,T]),
write('     runtime (in ms) ': T).
ver_bin(_):-
write('programa verified................'),
statistics(runtime,[_,T]), write('     runtime (in ms) ': T).

pre_condition(L):- incr(L).
incr([_]).
incr([A,B|T]):- {A =< B},incr([B|T]).
```

```
neg_pos_condition(-1,X,L):- !,nth0(_,L,X1),{X1=X}.
neg_pos_condition(K,X,L):- {K >= 0}, nth0(K,L,Y),{X =\= Y}.

% System of constraints for the binary search program ********
bsHM(_tab0Len,_tab0,_x0,_index2):-
Ln is (ceiling(log(_tab0Len+1)/log(2))+1),
_index1#=(-1),
_m1#=0,
_l1#=0,
length(_tab0,_tab0Len),
_u1#=(_tab0Len-1),
length(_index1List,Ln),
_index1List=[_index1|_],
length(_l1List,Ln),
_l1List=[_l1|_],
length(_m1List,Ln),
_m1List=[_m1|_],
length(_u1List,Ln),
_u1List=[_u1|_],
whileLoopHM(_index1List,_l1List,_m1List,_u1List,_tab0,_x0,
_index2,_l2,_m2,_u2).

% Definition of predicates representing loops ***************
whileLoopHM([_index1,_index2|_indexTail],[_l1,_l2|_lTail],
[_m1,_m2|_mTail],[_u1,_u2|_uTail],_tab0,_x0,_indexSol,_lSol,_mSol,
_uSol):-
(_inBool0#<=>(_index1#=(-1)#/\_l1#=<_u1)),
(#\_inBool0)#=>_l2#=_l1#/\_u2#=_u1#/\_index2#=_index1#/\
_m2#=_m1,
_inBool0#=>_m2#=((_l1+_u1)/2),
nth0(_m2,_tab0,_tab0Elemm2),
(_inBool0#/\_exBool6)#=>_index2#=_m2#/\_l2#=_l1#/\_u2#=_u1,
(_inBool0#/\(#\_exBool6#/\_exBool8))#=>_u2#=(_m2-1)#/\_l2#=_l1,
(_inBool0#/\(#\_exBool6#/\(#\_exBool8)))#=>
_l2#=(_m2+1)#/\_u2#=_u1, % Correct
%(_inBool0#/\(#\_exBool6#/\_exBool8))#=>
%_u2#=(_m2-1)#/\_l2#=_l1, % Error
(_inBool0#/\(#\_exBool6))#=>_index2#=_index1,
ExBoolCons0 = [(exBool6,_tab0Elemm2=_x0),(exBool8,_tab0Elemm2>_x0)],
ExBoolVars0 = [(exBool6,_exBool6),(exBool8,_exBool8)],
InBools0 = [],
doLabeling(InBools0,ExBoolVars0,ExBoolCons0),
whileLoopHM([_index2|_indexTail],[_l2|_lTail],[_m2|_mTail],
[_u2|_uTail],_tab0,_x0,_indexSol,_lSol,_mSol,_uSol).
whileLoopHM([_index2],[_l2],[_m2],[_u2],_tab0,_x0,_index2,_l2,
_m2,_u2).
```

A.4 Binary Search - Finite Domain Model

```
:- use_module(commonLib).
ver_bin([N,L,X,I]):-
statistics(runtime,[_,_]),
length(L,N),
pre_condition(L),
bsFDM(N,L,X,I),
neg_pos_condition(I,X,L),
statistics(runtime,[_,T]),
write('      runtime (in ms) ': T).
ver_bin(_):-
write('programa verified................'),
statistics(runtime,[_,T]), write('      runtime (in ms) ': T).

pre_condition(L):- incr(L).
incr([_]).
incr([A,B|T]):- {A =< B}, incr([B|T]).
neg_pos_condition4(-1,X,L):- !,element(_,L,X).
neg_pos_condition4(_,X,L):- \+element(_,L,X).

% System of constraints for the binary search program ********
bsFDM(_tab0Len,_tab0,_x0,_index2):-
Ln is (ceiling(log(_tab0Len+1)/log(2))+1),
domain([_x0|_tab0],-65635,65635),
_index1#=(-1),
_m1#=0,
_l1#=0,
length(_tab0,_tab0Len),
_u1#=(_tab0Len-1),
length(_index1List,Ln),
_index1List=[_index1|_],
length(_l1List,Ln),
_l1List=[_l1|_],
length(_m1List,Ln),
_m1List=[_m1|_],
length(_u1List,Ln),
_u1List=[_u1|_],
labeling([],[_x0|_tab0]),
whileLoopFDM(_index1List,_l1List,_m1List,_u1List,_tab0,_x0,_index2,_l2,
_m2,_u2).

% Definition of predicates representing loops **********
whileLoopFDM([_index1,_index2|_indexTail],[_l1,_l2|_lTail],[_m1,_m2|_mTail],[_u1,_u
_tab0,_x0,_indexSol,_lSol,_mSol,_uSol):-
(_inBool0#<=>(_index1#=(-1)#/\_l1#=<_u1)),
(#\_inBool0)#=>_l2#=_l1#/\_u2#=_u1#/\_index2#=_index1#/\_m2#=_m1,
_inBool0#=>_m2#=((_l1+_u1)/2),
nth0(_m2,_tab0,_tab0Elemm2),
(_exBool6#<=>_tab0Elemm2#=_x0),
(_inBool0#/\_exBool6)#=>_index2#=_m2#/\_l2#=_l1#/\_u2#=_u1,
(_exBool8#<=>_tab0Elemm2#>_x0),
(_inBool0#/\(#\_exBool6#/\_exBool8))#=>_u2#=(_m2-1)#/\_l2#=_l1,
(_inBool0#/\(#\_exBool6#/\(#\_exBool8)))#=>_l2#=(_m2+1)#/\_u2#=_u1,%Correct
%(_inBool0#/\(#\_exBool6#/\_exBool8))#=>_u2#=(_m2-1)#/\_l2#=_l1,% Error
(_inBool0#/\(#\_exBool6))#=>_index2#=_index1,
labeling([],[_inBool0,_exBool6,_exBool8]),
whileLoopFDM([_index2|_indexTail],[_l2|_lTail],[_m2|_mTail],[_u2|_uTail],
_tab0,_x0,_indexSol,_lSol,_mSol,_uSol).
whileLoopFDM([_index2],[_l2],[_m2],[_u2],_tab0,_x0,_index2,_l2,_m2,_u2).
```

A.5 Buble Sort - Hybrid Model

```
:- use_module(commonLib).
ver_bub([N,T2,LBound]):-
statistics(runtime,[_,_]),
N1 is N-1,
pre_condition(N1,T1),
bubHM(T1,T2,LBound),
neg_pos_condition(T2),
statistics(runtime,[_,T]), write('     runtime (in ms) ': T).
ver_bub(_):-
write('programa verified................'),
statistics(runtime,[_,T]), write('     runtime (in ms) ': T).

pre_condition(N,[H|T]):- N>0, N1 is N-1, {H=N}, pre_condition(N1,T).
pre_condition(0,[0.0]).
neg_pos_condition([A,B|_]):- {A>B},!.
neg_pos_condition([A,B|T]):- {A=<B},neg_pos_condition([B|T]).

% System of constraints for buble sort
bubHM(_tab0,_tab2,LBound):-
_i1#=0,
length(_i1List,LBound), _i1List=[_i1|_],
length(_tab0List,LBound), _tab0List=[_tab0|_],
write(input:_tab0),nl,
whileLoopBubHM1(LBound,_i1List,_tab0List,_i2,_tab2),
write(sorted:_tab2),nl,!.

% Definition of predicates representing loops ****************
whileLoopBubHM1(LBound,[_i1,_i2|_iTail],[_tab0,_tab2|_tabTail],
_iSol,_tabSol):-
length(_tab0,_tab0Len),
length(_tab2,_tab0Len),
_exBool1#<=>_i1#<(_tab0Len-1),
_j1#=0,
length(_j1List,LBound), _j1List=[_j1|_],
length(_tab0List2,LBound), _tab0List2=[_tab0|_],
whileLoopBubHM2(_j1List,_tab0List2,_i1,_j2,_tab2),
_exBool1#=>_i2#=(_i1+1),
_tab0LenLoop#=_tab0Len-1,
(for(I,0,_tab0LenLoop),param(_tab0,_tab2,_exBool1) do
nth0(I,_tab0,_tab0ElemI),nth0(I,_tab2,_tab2ElemI),
(#\_exBool1)#=>_exBool12, % _tab2ElemI#=_tab0ElemI,
doLabeling([],[(exBool12,_exBool12)],[(_exBool12,_tab2ElemI=_tab0ElemI)])),
(#\_exBool1)#=>_i2#=_i1,
whileLoopBubHM1(LBound,[_i2|_iTail],[_tab2|_tabTail],_iSol,_tabSol).
whileLoopBubHM1(_,[_i2],[_tab2],_i2,_tab2).

% Definition of predicates representing loops ****************
whileLoopBubHM2([_j1,_j2|_jTail],[_tab0,_tab2|_tabTail],_i1,_jSol,
_tabSol):-
length(_tab0,_tab0Len),
_tab1LenLoop#=_tab0Len-1,
same_length(_tab0,_tab2,_tab0Len),
_exBool1#<=>_i1#<(_tab0Len-1),
_exBool3#<=>_j1#<((_tab0Len-_i1)-1),
(_exBool1#/\_exBool3)#=>_j2#=(_j1+1),
(#\_exBool1)#\/(#\_exBool3)#=>_j2#=_j1,
nth0(_j1,_tab0,_tab0Elemj1),
nth0(_j2,_tab0,_tab0Elemk1),
doLabeling([],[(exBool5,_exBool5),(exBool11,_exBool11)],
[(_exBool5,_tab0Elemj1>_tab0Elemk1),(_exBool11,_aux1=_tab0Elemj1)]),
(_exBool1#/\_exBool3#/\_exBool5)#=> _exBool11,
(for(I1,0,_tab1LenLoop),
param(_tab0,_tab2,_exBool1,_exBool3,_exBool5,_j1,_tab0Elemk1,_j2,_aux1) do
nth0(I1,_tab0,_tab0ElemI1),nth0(I1,_tab2,_tab2ElemI1),
(#\_exBool1)#\/(#\_exBool3)#=>_exBool6,
(_exBool1#/\_exBool3#/\_exBool5#/\(I1#=_j1))#=>_exBool7,
(_exBool1#/\_exBool3#/\_exBool5#/\(I1#=_j2))#=>_exBool8,
(_exBool1#/\_exBool3#/\_exBool5#/\(I1#\=_j1)#/\(I1#\=_j2))#=>_exBool9,
(_exBool1#/\_exBool3#/\(#\_exBool5))#=>_exBool10,
doLabeling([],[(exBool6,_exBool6),(exBool7,_exBool7),(exBool8,_exBool8),
(exBool9,_exBool9),(exBool10,_exBool10)],[(_exBool6,
_tab2ElemI1=_tab0ElemI1),(_exBool7,_tab2ElemI1=_tab0Elemk1),
(_exBool8,_tab2ElemI1=_aux1),(_exBool9,_tab2ElemI1=_tab0ElemI1),
(_exBool10,_tab2ElemI1=_tab0ElemI1)])),
whileLoopBubHM2([_j2|_jTail],[_tab2|_tabTail],_i1,_jSol,_tabSol).
whileLoopBubHM2([_j2],[_tab2],_i1,_j2,_tab2).
```

A.6 Buble Sort - Finite Domain Model

```
:- use_module(commonLib).
ver_bub([N,T2,LBound]):-
statistics(runtime,[_,_]),
N1 is N-1,
pre_condition(N1,T1),
bubFDM(T1,T2,LBound),
neg_pos_condition(T2),
statistics(runtime,[_,T]), write('     runtime (in ms) ': T).
ver_bub(_):-
write('programa verified................'),
statistics(runtime,[_,T]), write('     runtime (in ms) ': T).

pre_condition(N1,L):- (for(I,0,N1),foreach(I,L1) do true),  reverse(L1,L).

neg_pos_condition([A,B|_]):- A#>B,!.
neg_pos_condition([A,B|T]):- A#=<B,neg_pos_condition([B|T]).

% System of constraints for buble sort
bubFDM(_tab0,_tab2,LBound):-
_i1#=0,
length(_i1List,LBound),
_i1List=[_i1|_],
length(_tab0List,LBound),
_tab0List=[_tab0|_],
whileLoopBub0(LBound,_i1List,_tab0List,_i2,_tab2).

% Definition of predicates representing loops *****************
whileLoopBub0(LBound,[_i1,_i2|_iTail],[_tab0,_tab2|_tabTail],
_iSol,_tabSol):-
length(_tab0,_tab0Len),
_exBool1#<=>_i1#<(_tab0Len-1),
_j1#=0,
length(_j1List,LBound),
_j1List=[_j1|_],
length(_tab0List,LBound),
_tab0List=[_tab0|_],
whileLoopBub1(_j1List,_tab0List,_i1,_j2,_tab2),
_exBool1#=>_i2#=(_i1+1),
same_length(_tab0,_tab2,_tab0Len),
_tab0LenLoop#=_tab0Len-1,
(for(I,0,_tab0LenLoop),param(_tab0,_tab2,_exBool1) do
nth0(I,_tab0,_tab0ElemI),nth0(I,_tab2,_tab2ElemI),
(#\_exBool1)#=>_tab2ElemI#=_tab0ElemI),
(#\_exBool1)#=>_i2#=_i1,
whileLoopBub0(LBound,[_i2|_iTail],[_tab2|_tabTail],_iSol,_tabSol).
whileLoopBub0(_,[_i2],[_tab2],_i2,_tab2).

% Definition of predicates representing loops *****************
whileLoopBub1([_j1,_j2|_jTail],[_tab0,_tab2|_tabTail],_i1,_jSol,
_tabSol):-
length(_tab0,_tab0Len),
_tab1LenLoop#=_tab0Len-1,
same_length(_tab0,_tab2,_tab0Len),
_exBool1#<=>_i1#<(_tab0Len-1),
_exBool3#<=>_j1#<((_tab0Len-_i1)-1),
(for(I0,0,_tab1LenLoop),param(_tab0,_tab2,_exBool1,_exBool3) do
nth0(I0,_tab0,_tab0ElemI0),nth0(I0,_tab2,_tab2ElemI0),
(#\_exBool1)#\/(#\_exBool3)#=>_tab2ElemI0#=_tab0ElemI0),
(#\_exBool1)#\/(#\_exBool3)#=>_j2#=_j1,
(_exBool1#/\_exBool3)#=>_j2#=(_j1+1),
nth0(_j1,_tab0,_tab0Elemj1),
nth0(_j2,_tab0,_tab0Elemk1),
_exBool5#<=>_tab0Elemj1#>_tab0Elemk1,
(_exBool1#/\_exBool3#/\_exBool5)#=>_aux1#=_tab0Elemj1,
(for(I1,0,_tab1LenLoop),param(_tab0,_tab2,_exBool1,_exBool3,
_exBool5,_j1,_tab0Elemk1,_j2,_aux1) do
nth0(I1,_tab0,_tab0ElemI1),nth0(I1,_tab2,_tab2ElemI1),
(_exBool1#/\_exBool3#/\_exBool5#/\(I1#=_j1))#=>
_tab2ElemI1#=_tab0Elemk1,
(_exBool1#/\_exBool3#/\_exBool5#/\(I1#=_j2))#=>
_tab2ElemI1#=_aux1,
(_exBool1#/\_exBool3#/\_exBool5#/\(I1#\=_j1)#/\(I1#\=_j2))#=>
_tab2ElemI1#=_tab0ElemI1),
(for(I3,0,_tab1LenLoop),
param(_tab0,_tab2,_exBool1,_exBool3,_exBool5) do
nth0(I3,_tab0,_tab0ElemI3),nth0(I3,_tab2,_tab2ElemI3),
(_exBool1#/\_exBool3#/\(#\_exBool5))#=>_tab2ElemI3#=_tab0ElemI3),
whileLoopBub1([_j2|_jTail],[_tab2|_tabTail],_i1,_jSol,_tabSol).
whileLoopBub1([_j2],[_tab2],_i1,_j2,_tab2).
```