University of Texas at El Paso
DigitalCommons@UTEP

Open Access Theses & Dissertations

2011-01-01

A Symbolic Approach Towards Constraint Based Software Verification

Shubhra Datta
University of Texas at El Paso, shubhra.datta@gmail.com

Follow this and additional works at: https://digitalcommons.utep.edu/open_etd

Part of the Computer Sciences Commons

Recommended Citation
Datta, Shubhra, "A Symbolic Approach Towards Constraint Based Software Verification" (2011). *Open Access Theses & Dissertations*. 2266.
https://digitalcommons.utep.edu/open_etd/2266

This is brought to you for free and open access by DigitalCommons@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of DigitalCommons@UTEP. For more information, please contact lweber@utep.edu.

A SYMBOLIC APPROACH TOWARDS CONSTRAINT BASED SOFTWARE VERIFICATION

SHUBHRA DATTA

Department of Computer Science

APPROVED:

Martine Ceberio, Chair, Ph.D.

Vladik Kreinovich, Ph.D.

Virgilio Gonzalez, Ph.D.

Benjamin Flores, Ph.D.
Dean of the Graduate School

©

Copyright

by

Shubhra Datta

2011

to

MY PARENTS

and

BIVAS

...

with love

A SYMBOLIC APPROACH TOWARDS CONSTRAINT BASED SOFTWARE VERIFICATION

by

SHUBHRA DATTA, B.Tech.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2011

Acknowledgement

First and foremost I offer my sincere gratitude to my supervisor, Dr Martine Ceberio, who has supported me throughout my thesis with her patience while allowing me the room to work in my own way. I am greatly indebted to her for my Masters degree without whose encouragement and effort this theses would be incomplete.

I also thank the members of my graduate committee for their guidance and suggestions.

I am heartily thankful to my fiancée Bivas Das for all his support and encouragement. Without his persuasion and guidance, I would not have come to study abroad in the first place.

I also want to thank my dear friends Manali Chakraborty and Amritam Sarcar. They helped me in countless ways every time I needed them.

Last but not the least, I thank my parents for supporting me throughout all my studies at UTEP, without their motivation and thoughtful decisions, I would not have been even what I am now.

And finally, I offer my thanks and regards to all of those who supported me in any respect during the completion of the thesis.

Shubhra Datta

NOTE: This thesis was submitted to my Supervising Committee on Sep 9, 2011.

Abstract

Verification and validation (V&V) are two components of the software engineering process that are critical to achieve reliability that can account for up to 50% of the cost of software development [24]. Numerous techniques ranging from formal proofs to testing methods exist to verify whether programs conform to their specifications. Recently, constraint programming techniques for V&V have emerged [18, 24]: they use the idea of proof by contradiction. They typically aim at proving that the code is inconsistent with the negation of the specification, which means that the software conforms to its specifications. Although the framework seems straightforward, the number of generated constraints can be high and the solving process tedious.

In this work, we propose ideas for improvement based on symbolic manipulation of the constraints to be solved. Our approach differs from the current approach in its way to determine the compliance of the code with respect to its specification. Instead of using numeric solvers, we designed symbolic techniques to check compliance between the code and its specification.

We analyzed how much practical the approach is if the program is correct and if the program is incorrect: can we make the verification process faster by applying our rules? CPBPV: a Constraint-Programming Framework for Bounded Program Verification [25], the work done by H. Collavizza, M. Rueher, and P. Hentenryck is the inspiration for our work.

We established that our approach is feasible, and our experimental results prove that our proposed method is a promising addition to the existing framework to eliminate some of the basic challenges associated with constraint-based software verification.

Table of Contents

Page

Acknowledgement v

Abstract vi

Table of Contents vii

List of Tables x

List of Figures xi

Chapter

1 Introduction 1

1.1 Motivation 1

1.2 What are we Proposing? 3

1.3 Thesis Outline 3

2 Preliminary Notions 4

2.1 Verification, Validation, and Testing 4

2.2 Constraints 5

2.2.1 General Definitions 5

2.2.2 Types of Constraints 6

2.2.3 Constraint Programming and Solving 7

2.3 Tree Exploration 9

2.3.1 Depth First Search (DFS) 9

2.3.2 Bidirectional Search 10

2.3.3 Weakest Precondition 11

3 Review of Related Research 12

3.1 Existing Verification Techniques 12

3.1.1 Types of Verification Techniques 12

3.1.2 Techniques for Verification 16

3.2 Related Research on Constraint Programming Techniques for Software Verification 21
3.2.1 Automatic Test Case Generation 22
3.2.2 Conformity of Specifications and Code 23
3.2.3 Constraint-Based Verification with Floating-Point Numbers 28
4 Limitations of Existing Approaches and Problem Statement 34
4.1 Problem Statement 34
4.1.1 Motivation for Our Approach 34
4.1.2 Limitations of the CPBPV Approach 35
4.2 Our Approach 36
5 Design and Implementation 39
5.1 Our Approach in a Nutshell 39
5.1.1 Inputs to Our Algorithm 39
5.1.2 Variables Used in Our Algorithm 40
5.1.3 How the Algorithm Works 40
5.2 Algorithm of Our Proposed Approach 43
5.3 Our Approach in Detail 46
5.3.1 Specific Format of the Input Program 46
5.3.2 Types of Constraints Generated 48
5.3.3 Types of Errors the Verifier can Catch 50
5.4 Examples 51
5.4.1 Absolute Difference Function 51
5.4.2 Insertion Sort Algorithm 54
5.4.3 Insertion Sort with Error 58
6 Experiments and Results 59
6.1 Experiment Setup 60
6.2 Results 61
6.2.1 Experimental Results for Goals 1 and 2 61

6.2.2 Experimental Results for Goal 3 75
7 Conclusion and Future Work 79
7.1 Conclusion 79
7.2 Future Work 80
References 82
Resources 89
Curriculum Vitae 90

List of Tables

2.1 Example of a CSP in RealPaver . . . 8

2.2 Solution to the above CSP using RealPaver . . . 8

6.1 Results for **correct insertion** sort . . . 62

6.2 Results for **correct selection** sort . . . 62

6.3 Results for **correct bubble sort** . . . 63

6.4 Results for **incorrect insertion** sort . . . 64

6.5 Results for **incorrect selection** sort . . . 65

6.6 Results for **incorrect bubble** sort . . . 65

6.7 From insertion sort . . . 67

6.8 From selection sort . . . 68

6.9 From bubble sort . . . 68

List of Figures

2.1 Depth First Search 9

2.2 Bidirectional Search 10

3.1 Types of Verification and Existing Techniques 13

3.2 Problem with negation of constraints 32

4.1 Simple example of how our algorithm works 38

5.1 STACK 41

5.2 Tree for Absolute Difference Algorithm 54

5.3 Tree for Insertion Sort Algorithm 57

6.1 **correct** sorting algorithms 63

6.2 **incorrect** sorting algorithms 66

6.3 Relation between Time Taken and Number of Constraints 69

6.4 **incorrect and correct** sorting algorithms together 71

6.5 **RealPaver solution** to the CSP generated from correct insertion sort algorithm 76

6.6 **Verifier solution** from correct insertion sort algorithm 77

Chapter 1

Introduction

1.1 Motivation

Verification and validation (V&V) are two important aspects of software life-cycle and they play a key role to determine the reliability and quality of software. Software is heavily used in critical fields like air traffic control, medical diagnostics, space shuttle missions, stock market reporting etc. The presence of bugs in the software application can cause irreparable losses. There are several such catastrophic events which are caused by software failure such as,

NASA Mars Climate Orbiter [56], a $125 million spacecraft, a key part of NASA's Mars exploration program, crashed due to a mathematical mismatch that was not caught earlier.

Northeast Blackout [41], costing $10 billion, happened due to a programming error which caused the failures occurred when multiple systems trying to access the same information.

Ariane 5 [33], a space project built with $7 billion over 10 years, intended to give Europe overwhelming supremacy in the commercial space business, exploded in less than a minute due to a small computer program trying to stuff a 64-bit number into a 16-bit space.

All these events clearly show that quality of software is of utmost importance and making sure the software meets the quality standards would have prevented these events. A report [56] says that *"Software bugs are costing the U.S. economy an estimated $59.5*

billion each year and improvements in testing could reduce this cost by about a third or $22.5 billion". Therefore it is very important to improve the V&V techniques in order to produce reliable and bug-free software.

There exist different solving approaches to address software verification. In particular, the common verification techniques are: informal, static, dynamic, symbolic, formal, and constraint techniques [1]. Since informal techniques involve human intervention and reasoning, it is error prone. Since static techniques do not require machine execution of the model, they are sometimes unable to verify the execution behavior of the model. With dynamic analysis, it sometimes becomes very complex and sometimes the software is verified only partially for a particular test case (in the case of testing). Both symbolic and formal techniques involve very complex and costly mathematical analysis. Constraint verification techniques make use of assertions which can be difficult to state and place correctly in the code.

Though the existing frameworks work reasonably well, there is still room for improvement: like cutting the cost or following a simpler approach. Recently, the CP (Constraint Programming) techniques constituted a new approach towards software verification and validation. The CP logic for V&V is reasonably simple and it is expected to be cost-effective.

CPBPV [25] is one of the most recent seminal works done in this field, proposed by H. Collavizza, M. Rueher and P. Hentenryck in 2010. This work has achieved significant performance improvement over the existing CP frameworks for V&V. The structure they follow to exploit the execution paths of the program, is very promising and if it can be successfully used in software verification, it can open a new direction in software industry. But their method has limitations: the number of generated constraints is very high, and as a result, the solution process is time consuming. In this work, we show how the computation time can be decreased.

1.2 What are we Proposing?

In the light of what we have discussed in motivation, in this work, we propose a framework similar to CPBPV in the structure but that only implements symbolic computations in inner nodes to detect any inconsistency earlier in the program.

In the rest of the sections, we give the detailed analysis of our approach, and we show that this approach indeed reduces the computation time.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

In Chapter 2, the preliminary notions are presented. These include notions related to validation, verification and testing, constraints and tree search methods that we will be referring to in this work. In Chapter 3, a review of the related work is presented. We describe existing verification techniques, in particular constraint programming techniques. We also point out the pros and cons of using constraint-based techniques for V&V. In Chapter 4, we present our contribution. We first recall our problem statement and the motivation behind our approach, along with some simple pseudo code of the algorithm we are proposing. In Chapter 5, we go into the details of our proposed approach, explaining it with examples. Our experimental results are reported and analyzed in Chapter 6. Conclusions and directions for future work are presented in Chapter 7.

Chapter 2

Preliminary Notions

In this thesis, the concepts of validation, verification, testing, constraints, weakest precondition, DFS tree traversal, bi-directional search, and formal methods are central. In this chapter, we provide background notions on each of these topics.

2.1 Verification, Validation, and Testing

The concepts of validation, verification, and testing are closely related and even sometimes used interchangingly in the literature. The concepts of validation and verification are even used outside of the programming world. In this section, we go over the meaning of these as commonly used and accepted in the programming world. Verification and validation (V&V) are sometimes used together to refer to all the activities that we perform to check that some piece of software does what it is supposed to do. We adopt the definitions for validation and verification as they are used in software engineering.

Verification refers to **proving** that a system (implementation) satisfies its specification – usually proving that the code satisfies the design specifications [1]. In other words, when we verify, we ask the question: *Are we building the product right?* [55, 58]. Some of the existing verification techniques include inspections, walkthroughs, data flow analysis, debugging, and testing. These are described in Section 3.1.

Validation refers to **checking** that the design specification satisfies the user's requirements [1]. In other words, when we validate, we ask the question: *Are we building the right product?* [55, 58]. Some of the activities carried out as part of validation are interviews and

presenting prototypes to the customer to check that the design specification of the software meets his/her need.

Testing consists in executing a program or some parts of it. While testing a piece of a program, we should design test cases according to the specification of the program at hand. When testing, the actual output is observed and compared against the expected output. We can say that the system is faulty if a test case fails to produce the expected output. However, if the system passes a test case, i.e., the output coincides with the expected output, we cannot say that the program has been fully verified, we can only say that the system works for this specific test case [58].

2.2 Constraints

2.2.1 General Definitions

Definition 1 (Constraint) *Let $D_1, \ldots, D_n$ be sets. A constraint c over variables $x_1 \in D_1, \ldots, x_n \in D_n$ is a relation between these variables.*

By definition, a relation defines a subset of the search space $D_1 \times \ldots \times D_n$. Thus c restricts the space to the sub-space of the possible combinations of values of the variables. A constraint solving problem (CSP) consists of a finite set of variables, each defined on a non-empty domain, and a finite set of constraints restricting the values of the variables[1]. More formally,

Definition 2 (Constraint solving problem) *A Constraint solving problem (CSP) is defined as a triple (C,X,D), where:*

- *C is a set of constraints;*
- *$X = \{x_1, \ldots, x_n\}$ is the set of variables bound by the constraints of C;*

[1]A CSP can in some instances be completed with a cost function that measures the quality of the assignments of the variables [52]

- *$D = D_1 \times \ldots \times D_n$ is the Cartesian product of the domains of the variables ($x_i \in D_i$); it defines the initial search space.*

Definition 3 (Solution to a constraint) *A solution of a constraint is an assignment of values to all variables, where these values are in the variables' respective domains, in such a way that the constraint is satisfied.*

Definition 4 (Solution to a CSP) *Solution of a CSP is an assignment of values to all variables, where these values are in the variables' respective domains, in such a way that all constraints are satisfied.*

2.2.2 Types of Constraints

There exist several types of constraints. These types can be categorized based on the nature of the elements defining the constraints:

- **Variables:** can have discrete or continuous domains. The constraints with discrete variables only are called discrete constraints; constraints with continuous variables only are called continuous constraints; constraints with both discrete and continuous variables can be called hybrid constraints.
- **Symbolic expressions** of the constraints make for different types of constraints; e.g., linear, non-linear, guarded constraints. These terms are described below.
- **Solutions:** can in some cases be redefined. This is the case of so-called soft constraints, which are inconsistent constraints or sets of constraints. In such cases, a more flexible definition of the solution set must be defined.
- **Solving process:** general constraints are usually solved through the typical split/filter (or branch/prune) approach. In some cases, e.g., global constraints, clever solving techniques can / should be used.

A **global constraint** is a constraint that captures a relation between a non-fixed number of variables [53]. An example is the constraint alldifferent$(x_1, \ldots, x_n)$, which specifies that the values assigned to the variables $x_1, \ldots, x_n$ must be pair-wise distinct.

We review the most common types of constraints in what follows.

- **Linear Constraints:** are of the form: a op b, where a and b are linear expressions and op$\in \{<, >, \leq, \geq, =, \neq\}$.

  Linear constraints can be used for instance, over variables that model real, integer, or interval quantities.

  Example: $6 \cdot x - y \geq 7 \cdot y + z$.

- **Guarded Constraints:** are of the form: *condition* $\rightarrow C$ where *condition* and C are regular constraints (as described in Definition 1).

  $A \rightarrow B$ is a guarded constraint which behaves in the following way:

  - B is added to the set of constraints if A holds;
  - B is disregarded if A does not hold;
  - $A \rightarrow B$ is suspended otherwise; which means, if we cannot prove either way (A holds or not).

- **Soft Constraints:** are constraints that may not be satisfiable. They are expressed as regular constraints, but solving them is different in the sense that, since solutions of the constraints may not be found, instead, best tradeoffs are sought [1].

2.2.3 Constraint Programming and Solving

Constraint programming is a programming paradigm where the problem is stated in terms of the constraints or requirements that need to be met, and a solution for these constraints is found using a general or domain specific constraint solving method [5].

A typical constraint solver proceeds by successively applying consistency techniques (filtering out non-solutions) and splitting the domains.

Table 2.1 contains an example that shows a particular CSP [53] written in the RealPaver[2] syntax. Table 2.2 shows the solution to the above CSP when solved using RealPaver.

Table 2.1: Example of a CSP in RealPaver

Variables
int x in [-1000, 10000],
int y in [-1000, 10000];
Constraints
$4 * x + 15 * y = 750$,
$5 * y + 7 * y = 33$;

Table 2.2: Solution to the above CSP using RealPaver

INITIAL BOX
x in [-1000, +10000]
y in [-1000, +10000]
OUTER BOX 1
x in [177.1874999999995, 177.1875000000005]
y in [2.75, 2.750000000000001]
precision: $9.09e^{-13}$, elapsed time: 0 ms
END OF SOLVING
Property: reliable process (no solution is lost)
Elapsed time: 0 ms

[2]RealPaver [35, 36] is a modeling language along with an interval solver for numerical constraint solving.

2.3 Tree Exploration

Since our approach is based on a tree structure of the program to verify (as we will describe later), we recall hereafter notions about tree traversal techniques that we will be using or outlining as potential future work.

2.3.1 Depth First Search (DFS)

Depth first search (DFS) [26] is an algorithm for traversing or searching a tree structure starting at the root and exploring as far as possible along each branch in depth before backtracking when reaching a dead end (leaf or other) (See Fig. 2.1).

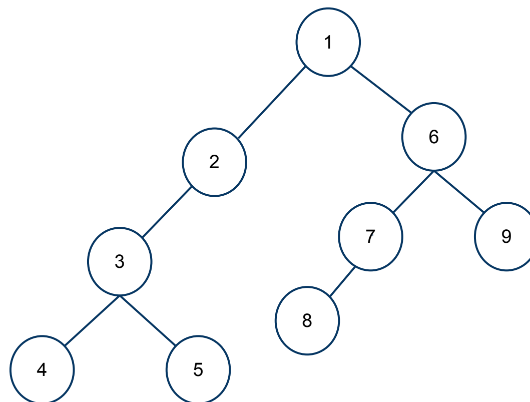

Figure 2.1: Depth First Search

Figure 2.1 shows a DFS traversal: the nodes in one branch are explored until the end of that branch has been reached. When one branch has been fully explored, the other branches that were left out during the initial exploration are considered one by one in the same fashion as the first one (in depth first). In our illustration (Figure 2.1), the nodes are numbered in the order in which they are explored.

2.3.2 Bidirectional Search

The idea behind bidirectional search [19] is to run two simultaneous searches – one forward search from the initial state (root), another search backward from the goal state (see Figure 2.2). The search stops when searches from both directions meet in the middle. Once the search is over, the path from the initial state is then concatenated with the inverse of the path from the goal state to form the complete solution path.

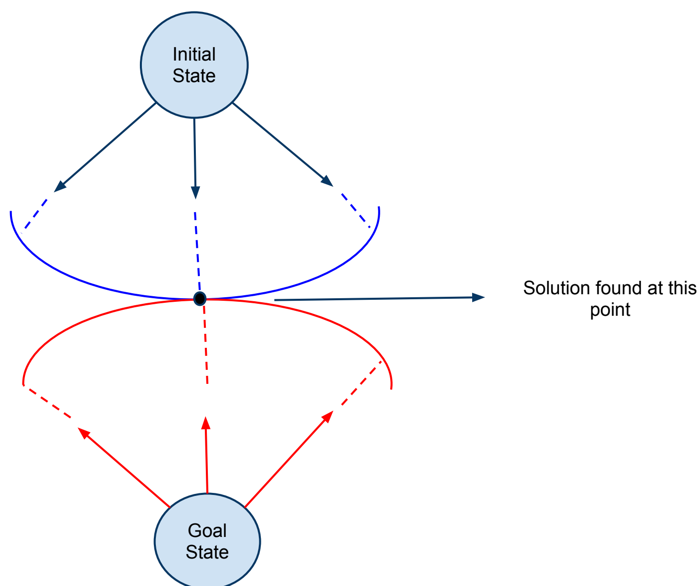

Figure 2.2: Bidirectional Search

If the tree of interest expands with a branching factor b and the distance from start to goal is d, the time complexity of bidirectional search is $O(b^{d/2})$ since each search needs only to proceed to about half the solution path. The advantage of bidirectional search is its speed. In order to implement bidirectional search, a clear goal state is required. Also, additional logic must be included to decide which search sub-tree to expand at each step of both searches (forward and backward).

We can achieve noticeable speedup and improvement using this search if the following can be guaranteed:

- concrete information regarding goal state;
- intersection between the two sub-trees coming towards each other starting respectively from the start and goal state; and
- computational storage capability.

2.3.3 Weakest Precondition

In order for bi-directional search to be integrated to the case of program verification, weakest preconditions need to be used. That is why we describe hereafter what they are about.

In [43], verification of programs based on weakest precondition strategy is proposed. Let us assume that we want to verify a program S where we know the post-conditions R but not the precondition Q. We will denote this situation by $\{?\}S\{R\}$.

There could be many arbitrary preconditions Q which are valid for the program S and the post-condition R. However, there is precisely one precondition describing the maximal set of possible initial states such that the execution of S leads to a state satisfying R. This Q is called the weakest precondition. A condition Q is weaker than P iff $P \Rightarrow Q$.

We believe we can integrate the weakest precondition approach to a half-search as in bi-directional search. This is one of the components of our future work.

Chapter 3

Review of Related Research

In this chapter, we provide a review of the related research in the area of verification, validation and testing.

3.1 Existing Verification Techniques

In this section, current approaches in software verification and validation are described. In [7], Balci presented a taxonomy for the most common techniques for verification and validation: he differentiated them based on their degree of credibility and their common characteristics. Figure 3.1 shows the main classification for verification. We now provide a brief summary of Balci's classification.

3.1.1 Types of Verification Techniques

Following are the main methods of verification:

- ***Informal techniques.*** These techniques rely mainly on human reasoning and usually involve human participation. Some of the techniques that fit in this category are the following:
  - **Audits:** this technique requires one person whose goal is to check the conformity of the software according to the established practices, standards, plans and guidelines [40, 7, 39].
  - **Inspections:** they are usually conducted by a team with predefined roles; e.g., a reader, a recorder, a designer, a moderator, an implementer, and a tester.

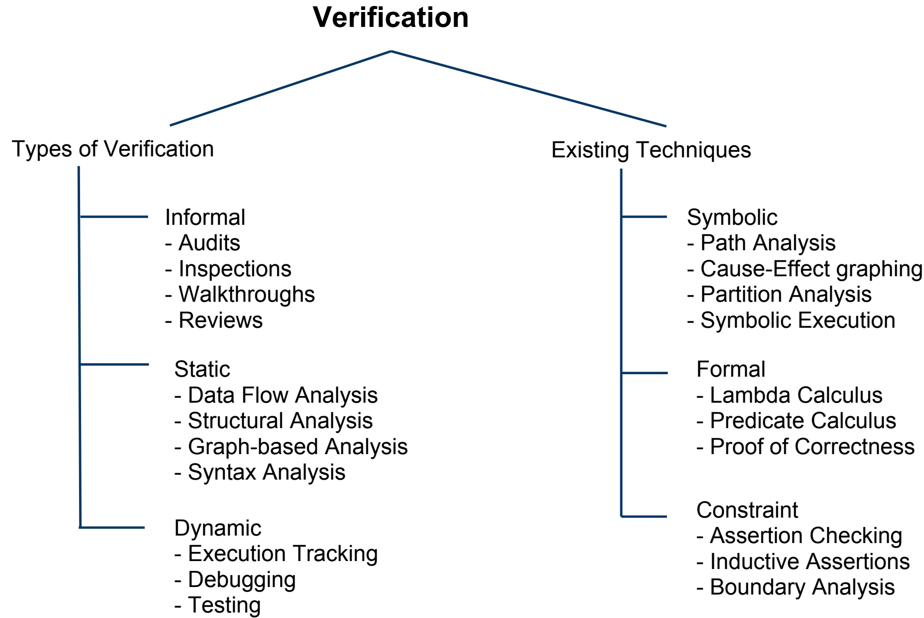

Figure 3.1: Types of Verification and Existing Techniques

Inspections go through several phases, such as overview of the model, fault finding process, fault resolution, examination of documents. They ensure that all faults are resolved [7]. Normally, inspections consist of five phases: overview, preparation, inspection, rework, and follow-up [54].

– **Walkthroughs:** they are usually conducted by a team with members who are not directly involved in the development of the product, except for the model developer who is usually included as the only person who is also involved in the development process. The aim here is to show that a certain level of quality has been reached rather than analyzing the code line by line to find faults [2, 29, 49, 50, 61].

– **Reviews:** they involve higher level techniques than inspections and walkthroughs. Like inspections, reviews are usually conducted by a team but it usually also includes managers. The objective here is to show managers and sponsors that the product or software is being developed based on the stated

specifications. In a review, models are also evaluated according to development standards, guidelines, and objectives [7].

The main disadvantages of informal techniques are that the teams in charge may overlook some aspects of the program since informal techniques require human intervention and reasoning. It is human to make mistakes or overlook some details, therefore errors in code might be missed.

- ***Static Verification.*** Static verification does not require execution of the model by machine. Examples of the techniques that fall into this category are as follows;
  - **Control Flow Analysis:** This method is useful for identifying incorrect or inefficient constructs within model representation and it examines sequences of control transfers. A graph of the model is constructed in which nodes represent conditional branches and model junctions and links represent model segments between such nodes [10]. An edge represents the junction that assumes control, whereas a node of the model graph represents a logical junction where the flow of control changes.
  - **Data Flow Analysis:** Such analysis assesses model accuracy with respect to the use of model variables [7]. This assessment is classified when variable space is allocated, accessed, and de-allocated [2]. This method is used to gather a program's data flow without actually executing it. A data flow graph is constructed to aid in the data flow analysis. The nodes of the graph represent statements and corresponding variables. The edges represent control flow. Data flow analysis can be used to detect undefined or unreferenced variables and, when aided by model instrumentation, can track minimum and maximum variable values, data dependencies, and data transformations during model execution [3].
  - **Structural Analysis:** It aims at analyzing the structure of the model or program. It usually builds a control flow graph to analyze some features about the

program, such as entry-exit points and use of unconditional branches [7]. Yucesan and Jacobson [62, 63] illustrated that modeling issues such as ambiguity of model specifications, state-accessibility, events-ordering, and execution stalling are problems for which general design techniques do not produce efficient solutions. They also showed that the problem of verifying structural properties of M&S applications (described in [63]) is difficult to solve. They proved it by applying the theory of computational complexity.

- **Cause-Effect Graphing:** Causes and effects are first identified in the system being modeled and then their representations are examined in the model specification. It lists as many causes and effects as possible. Once the cause-effect graph has been constructed, a decision table is created by tracing back through the graph to determine combinations of causes that result in each effect. The decision table is then converted into test cases with which the model is tested [51, 60, 61].
- **Syntax Analysis:** This method ensures the correctness of the program with respect to the syntax rules of the programming language. It is normally done by the compiler [7, 10].

Static verification techniques are able to make some inferences about the semantics and some aspects of the execution of the model. Also they can verify the syntax of the program. But they are unable to verify the execution behavior of the model. Besides we need to show that the static verification tools are correct [60].

- ***Dynamic Verification.*** Techniques falling in this category are based on model execution. Following are some examples of such techniques:
  - **Execution Tracing:** This technique generates trace data from the execution of the system line-by-line. The trace is then analyzed to find errors. But the main problem is that the amount of trace produced is sometimes very large and

complex. So it is very difficult to analyze [7].

- **Debugging:** This is the process of iteratively finding the errors that cause a system failure. Once the errors are found, we modify the system to correct the errors and keep debugging, until, ideally there are no errors, or more commonly, until we are satisfied with the system [7].
- **Testing:** This is one of the most commonly used V&V techniques. It first consists in creating test cases from the specification of a system and then in running the test cases. The outcome of running the test cases helps decide whether the system failed or passed the test based on the output. A test case consists of input data and an expected output. If the output does not coincide with the expected output then we can say that the system has a failure. However, if a system passes a test case then we can only say that the system works fine for that particular test case [58].

3.1.2 Techniques for Verification

In what follows, we review current state-of-the-art verification techniques:

- ***Symbolic Verification.*** In symbolic verification, symbolic inputs are fed to a model. The model is then run with these symbolic inputs. These symbolic inputs are transformed throughout the execution path by the model. The output consists of expressions that result from this transformation. Following are some of the techniques that fit in this category;
  - **Path Analysis:** We aim at testing all of the paths of a model in this technique. We often incorporate other techniques such as structural analysis and symbolic execution with this one. There can be different ways to choose test data according to the path coverage criteria that we want to achieve. Some of the coverage that we can achieve are: node coverage, multiple decision coverage, statement coverage, branch coverage and path coverage [60].

- **Cause-Effect Graphing:** This technique starts by identifying the causes and effects of the system. Then it constructs a cause-effect graph expressing the meaning of the system's specifications. After that, it builds a decision table by considering possible combinations of causes that will cause each effect. Finally, we can build test-cases by considering the decision table [7, 57].
- **Partition Analysis:** This technique consists of three main steps. First, it partitions the model into sub models. And then, it compares the specification of the elements and their intended functionality of each sub model with the implementation of the elements and their actual functionality of each sub model. Finally, it produces test data to extensively test each sub model [60].
- **Symbolic Execution:** This technique involves executing the model with symbolic values as input instead of using actual values. Then the symbolic values (inputs to the model) are transformed during the execution of the model and at the end of the execution, the output consists of the resulting expression [60].

The main disadvantage of symbolic verification techniques is the process of deriving the symbolic expressions, which can be difficult and complex. These techniques are also costly in terms of human intervention in order to obtain or even interpret the symbolic results and the expressions [60].

- ***Formal Verification.*** Formal verification is based on formal mathematical proof of correctness. Formal verification is highly relevant to practical software engineering [15]. First, it increases the understanding of the underlying nature of the program. Second, because of the proofs, engineers try to first check the consistency of the simplest implementation with requirements. This reduces errors and contributes the maintainability of less error prone programs. The most commonly known approaches to formal verification [42, 60] are briefly described below followed by some of the existing verification methods based on formal proof of correctness.

Approaches to Formal Verification

- **Lambda Calculus:** as described in [8], is a system for transforming the model into formal expressions by rewriting strings. Lambda calculus specifies rules for transforming the model into lambda calculus expressions. Using lambda calculus, the modeler can formally express the model so that mathematical proofs of correctness can be applied to it. More details on types of lambda calculus can be found at [9].

- **Predicate Calculus:** as described in [6], provides rules for manipulating predicates, which are combinations of simple relations that can be either true or false. The model can be defined in terms of predicates and manipulated using the rules of predicate calculus. Predicate transformation [30] provides a basis for verifying model correctness by formally defining the semantics of the model with a mapping, which transforms model output states to all possible model input states. This representation provides the basis for proving model correctness. Some of the works based on predicate abstraction are given in[32, 59].

- **Proof of Correctness:** as described in [6, 54], corresponds to expressing the model in a precise notation and then mathematically proving that the executed model terminates and satisfies the requirements specification with sufficient accuracy. Attaining proof of correctness may not be possible using state of the art technology. However, the advantage of realizing proof of correctness is so great that when the capability is realized, it will revolutionize the model V&V.

- **Theorem Proving:** as described in [14], theorem proving checks the general validity of a formula or whether a formula F holds in all models: $\models F$. Traditionally the logic used in theorem proving is the First-order (FOL) or Higher-order logic (HOL). Some other logics can be used and since all of them can be expressed as HOL, HOL is used much more often in theorem proving. Soundness is an essential property of a proof system: a proof system is sound when the fact that all premisses are valid indeed (semantically) guarantees that the conclusion

holds [14]. A proof system is complete when every valid sequent has a proof in the proof system. Completeness is important only for decidable or semi-decidable logics. Examples of theorem proving tools are Analytica, CProver etc. The main problem with theorem proving is the difficulty in automating the proof. Also lack of expertise, inadequate tools, and incompatiblity with current technologies make the implemention of theorem provers extremely difficult.

Formal Verification Methods

- **Hoare logic:** Hoare language used simple constructs: it has only assignment, sequencing of statements, if-then-else statements, and while loops [38]. Each of these constructs is interpreted by a proof rule. All the rules are some forms of the expression $\{P\}S\{R\}$, which means "if P is true before the execution of S then, provided S terminates, R is also true". The nature of the proof of a program S is as follows: The programmer supplies statements P and R, which supposedly describe the intended purpose of the program. Statement P defines properties of the input of the program; R defines properties of the output. Expression $\{P\}S\{R\}$ is then the hypothesis to be proven.

  There are some limitations: post-conditions are not easy to express in first-order logic. The proof of programs in Hoare logic guarantees correctness of implementation only if the program is processed by a compiler based on Hoare logic.

- **Dijkstra's approaches:**
  - Weakest preconditions: described in Section 2.3.
  - Parallel development of the program and the proof: weakest preconditions are often associated with developing a program and its proof in parallel with the proof ideas guiding the program development [31].

- **Mills's functional correctness:** Horlan Mills [47] proposed a method for

proving based on relations and functions rather than on preconditions and post-conditions. In his approach, programs or procedures are translated into functions relating outputs with inputs. This is the specification function. The goal of the proof is to demonstrate that the function computed from the program contains the specification function.

There have been many arguments about the feasibility of using formal verification. A proof has to convince its reader, which formal proofs may fail to do when the code is faulty. Regarding simultaneous development of an algorithm and a program, it can work very well for simple algorithms [6, 37] but some algorithms, like iteration schemas of numerical analysis, are very difficult to deal with. So the emphasis is on separating the proof of an algorithm from the proof of a program.

The main disadvantage of formal verification techniques is that they require a formal mathematical proof of correctness. This can be very costly human-wise and machine-wise, and it might even not be possible with current technology. The main reasons include lack of adequate tools, lack of mathematical sophistication in developers, incompatibility with current techniques [45].

Though abstract data types provide an area in which the formal approach has been a success, routine application of reasoning and data abstraction is just beginning in the industry [23], about 10-15 years after the initial research.

- ***Constraint-based Verification.*** This technique aims at verifying the software or the model based on the comparison of the assumptions made for the model to its actual behavior during the model execution. Some of the techniques that fit in this category are the following:

  - **Assertion Checking:** This technique places assertion statements in the code that should hold during the execution of the model or the program. It verifies

the model or the program by comparing the information about the model on some state with its intended behavior at that state [60].

- **Inductive Assertions:** The inductive assertions technique involves writing input-output relations for all model variables. It places these relations at the beginning and the end of each path in the model. For each path, if the assertion at the beginning holds and all the statements in the path are executed then we can prove the correctness of the model or the program provided we prove the termination of the program or model. Though this technique is close to formal verification techniques, it has been included in constraint techniques because it makes use of assertions [60].

- **Boundary Analysis:** This method usually verifies the model's behavior at boundaries of its input. Input is partitioned and test cases are generated with values inside the partition boundaries, on its boundaries and just past the boundary [60].

The main disadvantage of the above constraint-based verification techniques is that they make use of assertions which can be difficult to state and place correctly in the code. Moreover, since stating assertions is tied to the formal specification, it also inherits the disadvantage and difficulty of formal specifications [60].

3.2 Related Research on Constraint Programming Techniques for Software Verification

In this section, we go more in details over recent work done in the area of constraint-based software verification. Constraint solving for automated software verification and testing is an emerging topic. However, the extensive use of constraint programming in software verification is still being researched.

To the best of our knowledge, Constraint Programming (CP) techniques have been studied and used in the V&V of software and hardware since the late 1980s - early 1990s. In [58], Waters proposed that constraint modeling be used as another approach for system validation (1991).

In the following section, we present constraint programming techniques being used for system verification.

3.2.1 Automatic Test Case Generation

The approach taken by Gotlieb et al. [34] consists in automatically generating test data that will execute a selected point in the code. They transform the code into Static Single Assignment[1] (SSA) form and analyze control-dependencies. They then build a constraint system with this information and solve it. While solving the obtained constraint system, test data is generated in such a way that the selected point executes (if there is a feasible path that leads to the selected point). The main steps of Gotlieb et al. can be outlined as follows:

1. Translate the code into a constraint system from the SSA form and control-dependencies.
    - The resulting constraint system (CS) is a combination of constraints generated from the program and from the selected point.
2. Solve CS to generate test data for the selected point if there exists at least one feasible path leading to the selected point.

In [28], DeMillo and Offutt presented a constraint-based technique for **automatically generating test data** that causes a mutant program[2] to fail (1991). Then in 1992, Chandra and Iyengar, presented a constrained-based approach to generate test cases to

[1]In compiler design, static single assignment form (often abbreviated as SSA form or simply SSA) is a property of an intermediate representation (IR), which says that each variable is assigned exactly once [4].

[2]A mutant program is a program with a single modification to its original program.

verify the designs of machine [20]. This approach is different from the previous one in a way that they generated test data to make a mutant program fail.

Later in 1998, Michel Rueher from Université de Nice-Sophia-Antipolis, France and Arnaud Gotlieb, Bernard Botella from Dassault Electronique, France introduced a new method for automatic test data generation based on **constraint solving techniques**. They translated a procedure into a constraint system by using SSA form and control dependencies. Then they solved the constraint system and checked if there was any feasible **control flow path** to execute a selected point in the program. If such a path existed, they generated test data that went through the path and executed the selected point. In addition, they also built a prototype implementation on a restricted subset of C language constructs [34].

Related work described in [21] also showed that the constraint programming techniques can also been used to help generate relevant test data.

However, testing is not sufficient to verify the conformity between a software and its specifications. Indeed, with testing, when the code satisfies all the test cases, we can only say that the code is valid for that particular test case. As a result, the code is only partially proved correct. Therefore, we need other ways to fully verify code with respect to its specification.

3.2.2 Conformity of Specifications and Code

Here, we describe the work of Rueher et al. in proving the compliance of code with its specification.

Collavizza and Rueher's approach

In [24], Hélène Collavizza and Michel Rueher explored the capabilities of CP techniques in software verification. It is to be noted, however, that their approach handles only operations with integers, i.e., they work on discrete domains (discrete constraints).

The idea behind Collavizza's and Rueher's approach is that they transform the program and its specification into a constraint system. A program is verified if the union of the constraints derived from the program and the negation of constraints derived from the specification of the program is inconsistent, meaning the CSP does not have a solution. Let us assume that we have specification S and its implementation C: Then we solve $C \wedge \neg S$, which is in some sense similar to the process of resolution in logic. If this process yields no solution, it means that the implementation models the specification.

That is, $C \models S$ which is equivalent to $C \wedge \neg S \models \perp$, meaning C is inconsistent with S.

The **main steps** of Collavizzas and Ruehers approach [24] can be outlined as follows:

- Translate the program into a constraint system C.
- Translate the negation of the specifications into a constraint system $(\neg S)$.
- Consider the conjunction of these two constraint systems as a CSP (possibly involving guarded constraints): $C \wedge \neg S$
  - If a solution is found, it means that the program does not meet its specification and the solutions to the CSP constitute the test cases that would fail to meet the specifications.
  - If no solution is found, it means that the program meets its specification.

In this work, the authors proposed to use a SAT solver first to deal with the usual shortcomings of standard CSP solvers: this was possible only because their techniques handled only programs over discrete variables.

Example of Collavizza's and Rueher's Approach

Here we illustrate the approach presented just before with the example provided in [24]. It is shown in Algorithm 1.

When following the steps mentioned above, we get:

Algorithm 1 A method that returns the difference between two numbers

Ensure: /** result $\geq$ 0 **/

```
1: function ABSOLUTE(int i, int j)
2:     if (i < j) then
3:         return (j-i)
4:     else
5:         return (i-j)
6:     end if
7: end function
```

1. Translating the program into a constraint system:
   $i < j \rightarrow r = j - i, \neg(i < j) \rightarrow r = i - j$
2. Translating the negation of the specification into a constraint system: $r < 0$
3. Conjunction of these two constraint systems, give us the CSP:
   $\{i < j \rightarrow r = j - i, \neg(i < j) \rightarrow r = i - j, r < 0, D_i = D_j = D_r = \{0, \ldots, 65535\}\}$, where D_i, D_j, D_r are the domains of the variables$i$,j, and r respectively. 0 to 65,535 is the range of short unsigned integers.

According to the verification approach proposed in [24], after translating the program and the negation of the specifications as a CSP, the CSP is solved and if it has no solution, it means that the program is correct with respect to the specifications, otherwise it means that the program does not meet the specifications.

CPBPV: A Constraint-Programming Framework for Bounded Program Verification

The latest work of Collavizza et al. in constraint-based verification is CPBPV [25], a Constraint-Programming framework for Bounded Program Verification. The goal of CPBPV is to verify the conformity of a program with its specification. CPBPV derives a con-

straint store from the specification and the program, and explores execution paths non-deterministically. This non-determinism occurs while there is a conditional or iterative instruction and the non-deterministic execution refines the constraint store by adding constraints coming from conditions and from assignments. The input program is partially correct if each constraint so produced in the constraint store implies the post-condition. CPBPV does not explore spurious execution paths as it incrementally prunes execution paths early by detecting that the constraint store is not consistent. It is important to notice that in order to verify the conformity between a program and its specification, CPBPV requires to check (explicitly or implicitly) all executables paths.

Example of the CPBPV Approach

Following is an example of the CPBPV verifier on binary search program (see Algorithm 2) described in [25].

Assuming an array input of length 8, the initial constraint store (CS) consists of the precondition $c_{pre} \equiv \forall 0 \leq i < 7 : t^0[i] \leq t^0[i+1]$ where t^0 is an array of constraint variables capturing the input. CPBPV conducts a SSA-like renaming [27] on the fly. The rest of the steps carried out by the verifier are as follows:

1. From line 2-3, add the constraints $l^0 = 0 \wedge u^0 = 7$.

2. Since $l^0 \leq u^0$, enter loop body and add $m^0 = (l^0 + u^0)/2$, which gives $m^0 = 3$.

3. From line 6, which is a conditional statement, it generates two alternatives and both must be explored.

4. Considering the first alternative, add $t^0[3] = v^0$ to CS and from line 7 the verifier adds $result = m^0$ to CS.

5. At this point, CPBPV has explored an execution path whose final constraint store c_{final} is: $c_{pre} \wedge l^0 = 0 \wedge u^0 = 7 \wedge m^0 = (l^0 + u^0)/2 \wedge t^0[m^0] = v^0 \wedge result = m^0$.

Algorithm 2 Binary Search Program

Require: /** $\forall$ int i; i$\geq$0 and i $<$ t.length-1; t[i] $\leq$ t[i+1] **/

Ensure: /** (result != -1 $\rightarrow$ t[result] == v) && (result != -1 $\rightarrow$ $\forall$ int k; 0 $\leq$ k $<$ t.length ;t[k] ! = v) **/

```
1:  function BINARY-SEARCH(int[] t, int v)
2:      int l = 0
3:      int u = t.length-1
4:      while (l ≤ u) do
5:          int m = (l + u)/2
6:          if (t[m] == v) then
7:              return m
8:          end if
9:          if (t[m] > v) then
10:             u = m-1
11:         else
12:             l = m+1
13:         end if
14:     end while
15:     return -1
16: end function
```

6. Then it checks whether the store, c_{final}, implies the post-condition c_{post} by searching for a solution to $c_{final} \wedge \neg c_{post}$.

7. If this test fails then this execution path is consistent with the specification.

8. Then the verifier will follow the same steps (1 through 7) for other alternatives from the conditional statements that were left out at the initial stage.

Implementation Issues and Limitations

The prototype implementation of CPBPV uses a sequence of solvers(MIP, CP), where MIP is the mixed integer-programming tool ILOG CPLEX2 and CP is the constraint-programming tool Ilog JSOLVER. In order to verify the constraints, the MIP solver is called at each node of the executable paths. The CP solver is only called at the end of the executable paths when all the post conditions are considered. Moreover, the authors use a depth-first strategy to traverse the executable paths of the code.

From our observation of CPBPV, the problem associated with CPBPV is that in order to check the conformity between the code and the specification, it uses a numerical solver, which introduces noise and makes the solving process tedious. This work is nevertheless what inspired us and we proposed an alternative symbolic approach that aims at avoiding the numerical constraint solving part.

3.2.3 Constraint-Based Verification with Floating-Point Numbers

Some of the leading works done for handling constraint-based verification with floating-point numbers are described below.

The V3F Project

In 2006, Blanc et al. published their work on the verification and validation of programs with floating-point numbers (V3F) [13]. As part of the **V3F** project[3], the authors developed a constraint solver (FPCS) over floating-point numbers for the generation of test cases [13]. FPCS relies on two key techniques: interval computation [48] and a computation of the inverse projection [46].

Techniques to solve constraints over floating-point numbers on the basis of projection functions can lead to slow convergence for very common constraints like addition and subtraction constraints. In [44], the authors introduced new addition and subtraction constraints that can drastically speed up the filtering process.

Symbolic Execution of Floating-Point Computations

Symbolic execution consists of going through the execution path of the program with symbolic input data. A common strategy is to use a constraint solver over the rationals or the reals whenever path conditions contain floating-point numbers. Unfortunately, using constraint solver to handle floating-point numbers can lead to approximations and even incorrectness in the solution (described in the following section). B. Botella, A. Gotlieb and C. Michel proposed a solution towards handling the symbolic execution of floating-point computations [16] in 2006. Their approach consists of the following two steps:

- First, they translated complex expressions over floaing-point numbers into equivalent relations which are binary or ternary constraints over the floating-point numbers to capure all the semantics of the floating-point operations. They named this translation process as *normalization*. When dealing with floating-point numbers, special attention must be given to conform to actual execution of program; meaning preserving

[3]The **V3F** project is now over, but has been extended by the Constraints and Proofs (CeP) team researching on subjects such as software testing and verification, which can benefit from both constraint programming and formal proving approaches.

the order of evalution since algebric properties such as associativity or distributivity are lost while floating-point numbers are concerned. Normalization decomposes an expression in a sequence of assignments where fresh temprary variables are introduced bearing in mind that the ordering of evaluation must be preserved. E.g., let $Expr = a_1 \otimes a_2 \oplus a_3$. Resulting decomposition is $Expr = x_1 \oplus a_3 \wedge x_1 = a_1 \otimes a_2$ because $\otimes$ has higher priority over $\oplus$ and operands are evaluated from left to right.

- In the second step, they used a dedicated constraint solver over the floating-point numbers to solve the resulting constraints. They used FPCS (floating-point constraint solver described previously) to handle the resulting constraints according to the semantics of the floating-point arithmetic. The solving process is based on interval propagation [12, 11], which deals with computing the set of solutions of non-linear constraints over the reals. This technique takes advantage of interval arithmetic [48] and relational arithmetic [22] to reduce the domains of the variables. With relational arithmetic, constraints are decomposed in projection functions over intervals. For example, the constraint $c = a + b$ is decomposed into three projection functions:

  $I_c \leftarrow I_{a+b} \cap I_c$, $I_a \leftarrow I_{c-b} \cap I_a$, $I_b \leftarrow I_{c-a} \cap I_b$

  A detailed description of exact projection function for floating-point number constraints is presented in the paper [46].

A Constraint-Based Approach to Verication of Programs with Floating-Point Numbers

The work done in [13, 24] is the base of the paper [18]. In this paper, the authors proposed a way to handle floating-point numbers representing real values. Since SAT solvers, which are used for discrete constraint programming, cannot be applied for such problems, they proposed to use guarded constraints; they translated them by using the equivalence of logical implication and a disjunction. They discussed the process for solving constraints of the form $A \rightarrow B$ along with the pros and cons of it. Their proposed approach works as

follows:

- Translate the code C and the negation of the specifications S into constraints.
- Translate constraints of the form $A \rightarrow B$ into $\neg A \wedge B$.
- Transform the CSP in the form of a CNF into a DNF $CSP_1 \cup ... \cup CSP_m$.
- Solve the CSPs and consider the final solution to be the union of solutions of CSP_i.

Associated Challenges

There are several challenges associated to the method proposed in [18]:

- **Problem with negation of constraints:**

  Since guarded constraints are converted to constraints involving negation, it may create problems for CSP solvers. These interval solvers may face two kinds of problems: the risk of **false positives** and **missing solutions**. These are explained in the following example.

  Example: Let us say that we want to solve the following constraint: $A \wedge \neg B$ over the reals. We can solve this constraint with traditional interval solving techniques, returning an outer approximation of A. In this case, there are other areas that get included in the solution (see the dark grey area of the left-hand-side picture of Figure 3.2). We can also approach this problem by solving $\neg A$, obtaining an outer approximation of $\neg A$, which is in turn an inner approximation of A. In this case some parts of solution are missed out (see Figure 3.2).

- **Solving disjunction of CSPs:**

  Since current state-of-the-art continuous CSP solvers are usually based on interval computations, their outputs are blurred with noise. They are most likely in particular to return solution for CSPs that do not have solutions.

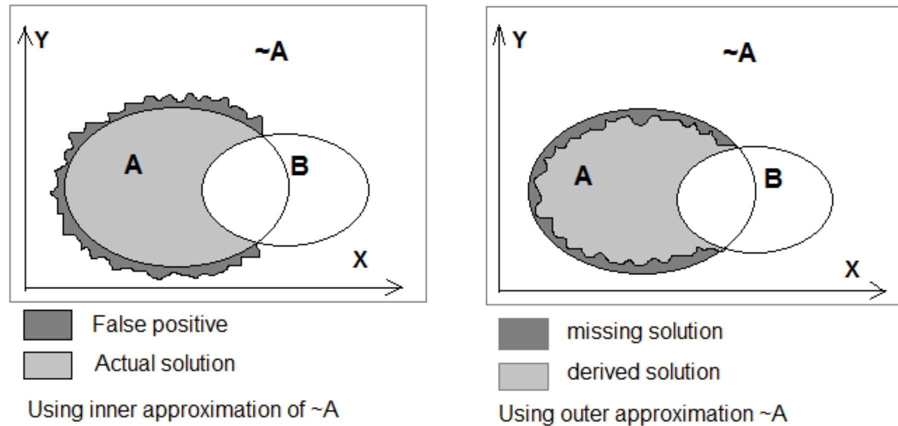

Figure 3.2: Problem with negation of constraints

In particular, in the case of disjunctions, which are not typically well handled, the blurring effect is multiplied.

- **Problems to deal with Floating point numbers:**

  - **Absorption**

    Floating-point numbers are exposed to the problem of absorption. Absorption occurs when a large floating-point number is added to a very small one. The following example (see Algo. 3) explains this problem.

    Due to limitations on number size, $10^9 + 10^{-9}$ leads to 10^9. As a result, instead of the correct path $z > x$, we get $z = x$ and thus follow the wrong path $z == x$. Such a problem must be accounted for in program verification when handling program that deal with floating-point numbers.

  - **Approximation**

    The set of all real numbers is infinite. Computers can only represent finitely many objects. Thus, real numbers are represented approximately, as rational floating-point numbers. For example, π is not a rational number, and thus, is represented in a computer only approximately.

Algorithm 3 Example with floating point numbers

```
1: function FOO()
2:     float x = 10^9, y = 10^-9, z
3:     z = x + y
4:     if (z > x) then
5:         ...
6:     end if
7:     if (z == x) then
8:         ...
9:     end if
10: end function
```

- **Poor arithmetical properties**

  Addition and multiplication on floating-point numbers are neither associative (i.e., $a+(b+c) \neq (a+b)+c$) nor distributive (i.e., $a*(b+c) \neq (a*b)+(a*c)$). However they are commutative (i.e., $b*c = c*b$).

- **Cancellation**

  Cancellation occurs when subtracting two floating-point numbers that are close to each other. Throughout the computation on floating-point numbers, rounding errors may arise and they may be accumulated, hence, it may propagate and produce significant rounding errors. Considering the example given in [13], $(10.00000000000004 - 10.0)/(10.000000000000004 - 10.0)$ results in 11.5 while it should be 10.0.

These types of peculiarities make it difficult to handle floating-point numbers while solving constraints involving them. These peculiarities can be handled by correctly designing the projection functions over floating-point intervals and by using a proper rounding mode [13].

Chapter 4

Limitations of Existing Approaches and Problem Statement

In the previous chapter, we described the ongoing research in the fields of constraint-based V&V and their associated challenges. In the context of constraint-based verification, we now highlight the problem statement and outline our proposed approach to address some of these challenges.

4.1 Problem Statement

The objective in this research work is, given a specification and its corresponding implementation, to be able to automatically verify whether the specification and the code are in compliance. In other words, we aim at designing a software that *verifies* whether an implementation meets its specification.

4.1.1 Motivation for Our Approach

As stated early in this document, verification is crucial and the lack of reliable tools for it can lead to catastrophes. Our approach is inspired by CPBPV [25]. We find CPBPV's approach interesting in the sense that constraints on a single execution path at a time are generated as the parsing of the code progresses and are checked at each conditional node. This feature relieves the verification process from having to process heavy and possibly very complex full model of the program by focusing on an execution branch at a time and by not waiting to accumulate all information from this execution path to already check the

compliance. This is the structure that we adopted as well. However, our approach differs from CPBPV in the sense that we do not provide similar node treatment. When CPBPV make calls to numerical solvers at each node, we decided to remain at the symbolic level not to introduce noise in the decision / tree pruning process. Other improvements brought by our approach are as follow:

- CPBPV creates a new variable (SSA) each time a variable is assigned a new value, which increases the constraint store size. If the old variables are only updated with their new value, the constraint store size can be better controlled by not overpopulating it.
- CPBPV calls numerical solver while we decided to only conduct simple symbolic analysis to determine the compliance between the constraints and specification, such as updating a comparison constraint with an assignment constraint, string matching.
- For sorting algorithms, CPBPV satisfies the loop invariants at each loop iteration. Since the complete set of specifications would not have been developed from any single loop iteration, the constraint store would be incomplete to check against specification at that point.

So our question is: can we decide that the constraints from the code are consistent/ (or inconsistent) with the specification without numerically solving the corresponding constraint solving problem? Can we do it efficiently[1]?

4.1.2 Limitations of the CPBPV Approach

CPBPV has been shown to be efficient when we consider discrete domains, such as in the case of integers. However, its efficiency in the case of real numbers, to best of our knowledge, has not been analyzed. The risk of running into false negatives is high. The authors of [25] also mentioned that in the CPBPV verifier, combining CP technique and other verification

[1]Our baseline for comparison will be CPBPV.

techniques can be beneficial for V&V. Moreover, at each step of the process, the constraint store keeps on increasing with a high growth rate, after which the constraint store is sent to a solver. For a very complicated program, the number of generated constraints can be very high, so the scalability of the approach is likely to be compromised.

We consider an alternative approach that aims at reducing the growth of the constraint store produced from following the execution paths of the program and at preventing noise in the process by avoiding to call a numerical solver at each node.

4.2 Our Approach

We propose a new approach where we do not use a numerical solver at each decision point (conditional/iterative statement), an approach which is more **symbolic in nature** than the usual numerical constraint solving techniques. In our approach, we try to symbolically detect the inconsistency by analyzing the constraints generated so far (either by direct implication or by derived implication from the constraint store). If there is any constraint which is inconsistent with the specification rules, then we stop exploring any more execution paths starting from that point: we cut the current branch and make a note of the point in the code as well as the depth of the tree at which the error was detected, and how many nodes have been generated so far. We then start exploring the other alternatives from the conditional statement that were left out at the initial stage. Since we are not using a numerical solver, our approach is faster in detecting inconsistencies (no false negatives). Hereafter we present a short pseudocode (see Algorithm 4) of our proposed approach with an example (see Figure 4.1).

Here S stands for the set of constraints derived from the specifications and STORE stands for the set of constraints derived from the code so far.

Algorithm 4 Pseudo code of our approach

```
1: for each line in code do
2:     if line is NOT a conditional/iterative statement then
3:         STORE = STORE + [Constraints from line]
4:     else if line is conditional/iterative statement then
5:         Verify STORE with S
6:         if STORE is consistent with S then
7:             Proceed
8:         else
9:             Stop and start exploring other alternative branches
10:        end if
11:    end if
12:    if Code End then
13:        Verify again
14:    end if
15: end for
```

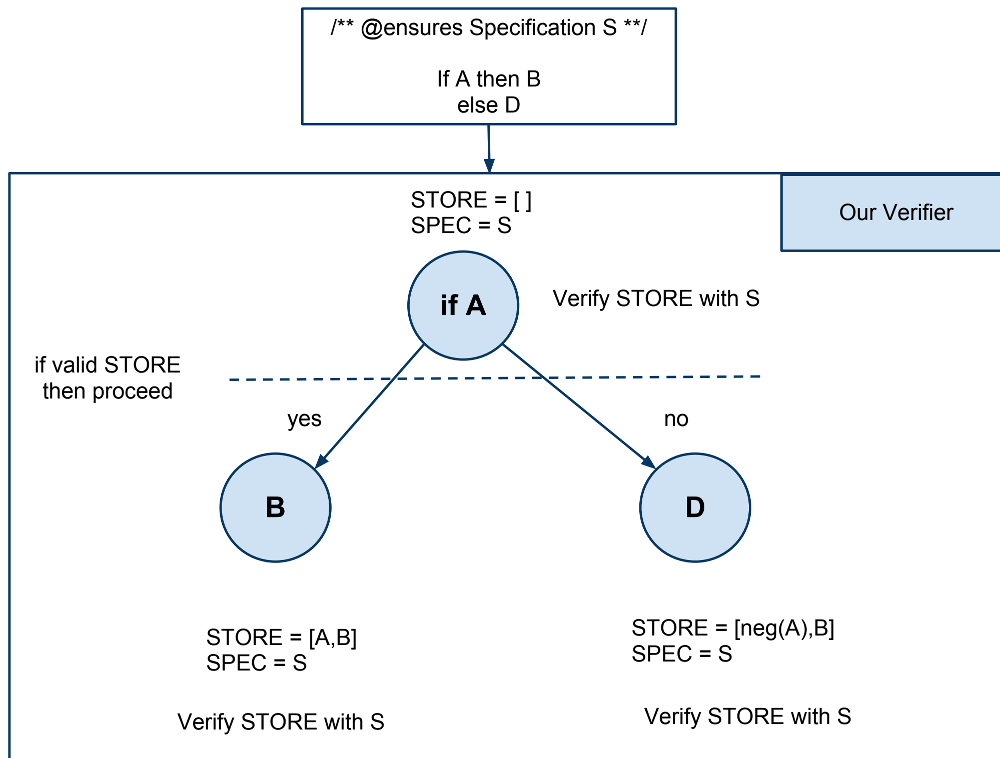

Figure 4.1: Simple example of how our algorithm works

Chapter 5

Design and Implementation

In this chapter, we present the details of what we proposed and implemented. We then illustrate our approach with examples such as *finding the absolute difference of two integer values* as well as sorting algorithms such as *insertion sort, bubble sort and selection sort.*

5.1 Our Approach in a Nutshell

Following we outline the flow and major constituents of our algorithm.

5.1.1 Inputs to Our Algorithm

Our algorithm needs the following two inputs in order to work:

- **The code**, i.e., the program that needs to be verified with respect to its specification.
- **A well-defined specification of the input program:** for example, for finding the absolute difference of two integer values, the requirement is that the difference must be greater than or equal to zero. In this case, a well-defined specification is: $result \geq 0$ where $result$ is the difference between two values. For sorting algorithm applied to array of length 3, a well-defined specification is: $a[1] \geq a[0] \wedge a[2] \geq a[1]$ where a is the array we want to sort. *Note: we assume we will have access to such well defined specification: either in JML format for instance, or in any properly defined semi-formal syntax.*

5.1.2 Variables Used in Our Algorithm

We used the following notations to describe the variables used throughout our algorithm. They are listed hereafter:

- **S:** denotes the well-defined specification of the input program with which the verifier will compare the program. As it is parsed, it already constitutes a set of constraints, and is interpreted as a conjunction. This set of constraints S is implemented as an array of strings obtained from the specification. The set $\neg S$ contains negations of all the constraints from S, and is interpreted as a disjunction.

- **STORE:** is a repository which holds the constraints generated from each line of the input program. It is implemented as an array of strings.

- **VARSTORE:** stores the variables used in the input program with their most recent values. It is implemented as a hash table where variables and their values are stored as a (key: value) pair.

- **STACK:** is used to accommodate the alternative branches from conditional statements, which are left out at the initial stage and are explored later; see Figure 5.1. Along with the negation of the *if* conditions, the object stored in STACK are additionally composed of the depth of the tree, the STORE, the VARSTORE, the line number of the code at the conditional point. Such information will be used to restore the state of the execution when backtracking to explore the*else* branches.

5.1.3 How the Algorithm Works

Our verifier starts with:

- the input program;

- its specification S;

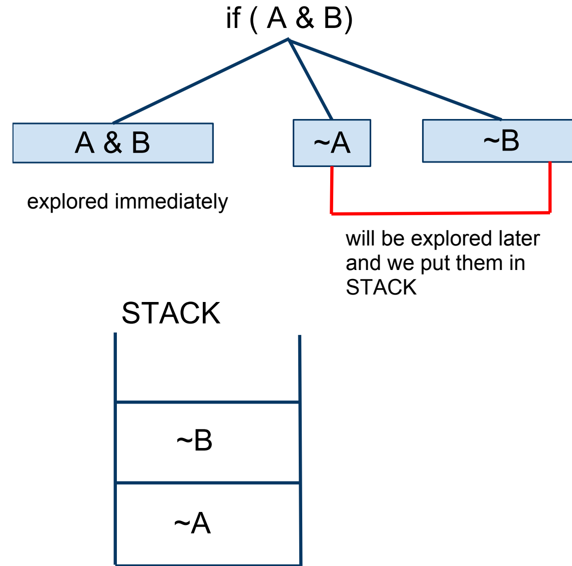

Figure 5.1: STACK

- an empty constraint store **STORE**; and
- an empty **VARSTORE**.

It starts exploring the execution paths of the program starting with the first line of the code. For each line that does not contain a conditional statement, it keeps on generating the corresponding constraints and adds them to STORE until it encounters a conditional/iterative statement (also called *decision point*). The types of constraints that are generated from various type of program statement are discussed in Section 5.3.

The main types of constraints generated by our verifier are **assignment** and **comparison** constraints which are triggered by assignment and conditional statements respectively.

Note: We want to highlight the fact that *each assignment constraint has an impact on the comparison constraints that comes later in the execution path.* Suppose we have an assignment constraint like $a[1] = val$, and then the verifier sees a comparison constraint such as $a[1] > 10$. This comparison constraint will then be changed to $val > 10$.

What the Algorithm does when there is a Conditional Statement

It first generates all possible branches from the conditional statement. For example, from a statement like $if(j > 0 \ \&\& \ a[j] > v)$, the possible branches are $(j > 0 \ \&\& \ a[j] > v)$ and $\neg(j > 0 \ \&\& \ a[j] > v)$. The true alternative, $(j > 0 \ \&\& \ a[j] > v)$, will be explored first and the other will be pushed onto the STACK.

Once the true alternative has been picked, our verifier assesses the consistency of the program traversed so far with the specifications by checking the constraints in STORE with S, which is done by function **CONSISTENT** defined later in our algorithm. This consistency check is distinct from what is done in CPBPV. They call a numerical solver at this stage to solve the CSP $STORE \wedge \neg S$, which involves a computationally tedious numerical solving of the CSPs and might incur noise and errors in solution. Instead our verifier conducts a simple symbolic analysis of the constraints in STORE with S.

Symbolic Verification Steps in Detail

Each constraint in STORE is matched with each constraint in the set $\neg S$. For example, if S consists of $r \geq 0$, then $\neg S$ is $r < 0$. If there is any constraint in STORE which is $r < 0$ or if there is any subset of constraints in STORE from which $r < 0$ can be derived, then the verifier detects that the STORE is inconsistent with S. This means that the execution path that led to this situation is an invalid implementation of the specifications and our verifier cuts the current branch from the tree and marks the branch as inconsistent. In case no constraint negates the constraint S, the verifier continues processing this branch.

In case our verifier cannot decide anything from the constraints with respect to S, then it calls a numerical solver[1] to solve the CSP $STORE \wedge \neg S$.

In case a branch turns out to be inconsistent with the specifications, or once a leaf has been reached (potentially successfully), our verifier looks into the STACK – since we

[1] Note: we reserved ourselves this option to detangle any situation in which the symbolic approach proves to be weak. However, this option was never triggered in any of the tests conducted during the research work presented in this manuscript.

do an exhaustive traversal of the search space. If the STACK is empty, this means that the verifier has completely traversed the tree, which is, that all execution paths have been explored. Otherwise, the verifier picks the alternative at the top of STACK and starts exploring the corresponding branch in the same fashion as presented in the case of the first branch.

What the Algorithm does at the Leaf Level

When the verifier reaches the last line of the input code, or when it encounters a return statement, it has reached the end of the current branch. At the leaf level, the constraints in STORE are checked against the constraints $\neg S$. In the following section, we present the layout of our algorithm.

5.2 Algorithm of Our Proposed Approach

Because of the length of our algorithm, the first part of it is described in Algorithm 5 and the second part in Algorithm 6. Function PROCESS is described in Algo. 8. Function CONSISTENT is described in Algorithm 7.

Note: The right headed triangles at the side of the algorithm denote comments.

Algorithm 5 Pseudo-code of our verifier

Require: /** The input program in specific format and a well-defined specification S **/

1: Translate the negation of the specification ($\neg S$) into a disjunctive constraint system
2: Line = first executable line of the input program
3: **while** (Line has not passed the last line of the code **do**
4:     **if** (Line is conditional statement) **then**
5:         Generate all alternative branches from the *if* condition
6:         Follow only the true alternative among those branches:
7:             ▷ i.e., add true alternative to STORE
8:         if-condition ← PROCESS(if-condition,VARSTORE) ▷ calls function PROCESS
9:         Store the other alternative into STACK ▷ stored for later exploration
10:         **if** (CONSISTENT(STORE,S)) **then** ▷ calls function CONSISTENT
11:             Line = Line + 1; break;
12:         **else**
13:             ▷ An inconsistency has been detected
14:             Record the depth at which it was found
15:             Store the erroneous branch with depth and node count into SOLUTION
16:             Look into STACK for other possible unexplored branches
17:             **if** (STACK $\neq \emptyset$) **then**
18:                 Node = top of STACK; Line = execution line of Node; break;
19:                     ▷ Resume execution starting with the Node at the top of STACK
20:             **else**
21:                 return SOLUTION ▷ empty if no inconsistency has been found
22:             **end if**
23:         **end if**

Algorithm 6 Pseudocode of our verifier (continued)

```
24:     else if (Line is an assignment statement of the form 'x = s') then
25:         y ← PROCESS(s,VARSTORE)                    ▷ calls function PROCESS
26:                                          ▷ Variable x is assigned with value y
27:         Add literal 'x : y' to VARSTORE;
28:         Line = Line + 1; break;
29:     else if (Line is a loop statement of the form 'LOOP1:') then
30:         Store the loop number and the line number for LOOP1
31:         Line = Line + 1; break;
32:     else if (Line is a goto statement of the form 'GOTO LOOP1') then
33:         Find out the line number for LOOP1
34:         Line = Line of LOOP1; break;
35:     end if
36:     if (Line has passed the last line of the code and/or a return statement) then
37:         if (CONSISTENT(STORE,S)) then            ▷ calls function CONSISTENT
38:             Execution path is Verified
39:         else
40:             Store the erroneous branch with depth and node count in SOLUTION
41:         end if
42:         Look into STACK for other possible unexplored branches
43:         if STACK ≠ ∅ then
44:             Node = top of STACK; Line = execution line of Node; break;
45:                    ▷ Resume execution starting with the Node at the top of STACK
46:         else
47:             return SOLUTION              ▷ empty if no inconsistency has been found
48:         end if
49:     end if
50: end while
```

Algorithm 7 Algorithm to check consistency

```
1: function CONSISTENT(C,S)                    ▷ C: constraint store, S: specification
2:     Apply inference rules to simplify C     ▷ i > j, r = i − j will be simplified to
   r = i − j; r > 0)
3:     for (each literal ns ∈ ¬S) and (each literal c ∈ C) do
4:         if (ns == c) then    ▷ Constraint c conforms to ns which is a literal in ¬S set
5:             return 0                        ▷ 0 indicates not consistent
6:             Break
7:         end if
8:     end for
9:     return 1                                ▷ 1 indicates consistent
10: end function
```

We now discuss the details of our approach in the following section. We then illustrate our algorithm with examples.

5.3 Our Approach in Detail

In the previous section, we outlined our algorithm. Now, we will discuss the details of its workings, such as: how our algorithm generates constraints from different types of program statements, the types of errors it can detect symbolically. We start by describing the specific format the input program must be in for our verifier to properly function.

5.3.1 Specific Format of the Input Program

The input code must be in a specific format in order for the verifier to work. For sorting algorithms for instance, the length of the array to be sorted must be specified and a well-defined specification for all of the input program must be provided as well. Moreover, the input program must be written in language C with the additional following specific

Algorithm 8 Evaluates the right-hand-side of an assignment statement

```
1: function PROCESS(s,v)                     ▷ s: string literal, v: VARSTORE
2:     if s is of form 'y' then
3:         if 'y:val(y)' ∃ in VARSTORE then
4:             return val(y)
5:         else
6:             return y
7:         end if
8:     else if s is an arithmetic expression such as 'y + 1' then
9:         if 'y:val(y)' ∃ in VARSTORE then
10:            return val(y)+1
11:        else
12:            return y + 1
13:        end if
14:    else if s is an array element of the form 'a[i]' then
15:        return 'a[val(i)]'
16:    else if s is an array element of the form 'a[i + j]' then
17:        return 'a[val(i) + j]'
18:    end if
19: end function
```

formatting requirements:

- **Function Declarations** must always be followed by the opening curly bracket.
- **IF conditions** must be in between parentheses and the statement must be followed by the opening curly bracket. E.g., $if(i < j)\{$.
- **Blocks** always end with the closing curly bracket in a separate line.
- **Loop** are changed to the following format:

  Loop1:
  $if(A)\{$
  ...
  goto Loop1;
  }

5.3.2 Types of Constraints Generated

Constraints are generated from each line of the code except for the lines that contain function definitions. The types of constraints generated from the program statements and the actions taken by the verifier for each of these statements are as follows (for a sample program please refer to algorithms in the Examples section):

- **From an assignment statement:** such as $x = y + 1$.

  The above statement assigns the value of $y+1$ to variable x. At this stage, our verifier also checks if x was declared initially or passed as a function argument; otherwise, it throws the following error: "variable x is used before declaration". If x is declared, then it updates the value of x with the value of $y + 1$ in VARSTORE.

- **From a conditional statement:** such as $if(i < l)$.

  Our verifier first checks if $(i < l)$ is valid by replacing the variables with their values, if values are available. If the condition is valid, then a constraint $(val(i) < val(l))$ is

added to STORE. If this inequality is invalid, then the algorithm comes out of the *if-block* and continues execution starting after the block. For example, if $i = 0$ and $l = 4$ then $(0 < 4)$ is valid and it is added to the STACK.

At the same time, if the values of i and l are not known, there is no way to check the validity of this inequality. In this case, the string $i < l$ is added to STORE and the negation of the constraint $i < l$, which is $i \geq l$ is added to STACK.

- **From Loop statement:** such as 'Loop1'

  Our verifier stores the loop number and the associated line number so that after each loop iteration it can restart execution for the next iteration starting from that line.

- **From Goto statement:** such as 'GOTO Loop1'

  The verifier is set to the line at which Loop1 starts so that it can continue execution from that point.

- **From Else statement:** 'Else'

  The *Else* is an alternative branch of a previous *If* statement. The verifier checks if the *If* branch is explored completely otherwise the *Else* is ignored. In case a complete execution path has been explored from the *If*, the verifier proceeds with the *Else* block. First, it restores back the STORE as it was before the *If* block, and then adds the negation of the *If* condition to STORE. This negation is retrieved from top of the STACK.

- **From End brace:** '}'

  End Brace indicates end of a block or the program body. In case it is the end of program, it means that a complete execution path has been explored. So at this point the verifier checks the consistency of the constraints in STORE, which have been generated from this execution path, with the specification of the input program.

If the execution path is a valid one, the verifier then starts looking into the unexplored branches from the STACK that were left out during the previous stage.

5.3.3 Types of Errors the Verifier can Catch

While exploring the possible execution paths of the program we try to catch possible syntax errors from the code statements. Type of errors we are able to catch:

- **Variable declaration missing:** If a variable is used in the program before it is declared, then the verifier throws this error.

- **Array index out of range:** If array index is smaller than 0 or greater than array size, then the verifier throws this error.

- **Inconsistency detection:** Following we explain the notion of inconsistency.

  No constraint in the constraint store should be inconsistent with the specification, which is the requirement the code must satisfy.

  Example

  The specification for the function that computes the absolute difference between two numbers includes $result \geq 0$. So, the negation of the specification will be $result < 0$. Now there should not be any constraint which is equal to $result < 0$ or set of constraints from which $result < 0$ can be derived, in order to be consistent with the specification.

  According to the above definition, both of the following STOREs are invalid with respect to the specification $result \geq 0$.

  - **STORE** $= [result < 0]$. This is a direct negation of the specification
  - **STORE** $= [i < j, result = i - j]$. From these constraints in STORE it can be derived that $result < 0$. Hence it negates the specification.

The example here shows that we are doing symbolic analysis of the constraints generated from the code statements rather than directly solving the constraints as in other CP techniques. According to [17] symbolic execution is a classical program testing technique which evaluates a selected control flow path with symbolic input data. The logic that we follow and its symbolic nature are illustrated with examples in the following section.

5.4 Examples

In this section, we illustrate our approach with a few examples. We start with the simple case of the **absolute difference function**: the simple nature of this function makes it easier to explain the symbolic nature of our algorithm. We then show how our verifier works in the more complex case of **insertion sort**: in particular we illustrate its workings on a correct implementation of insertion sort and on an incorrect one.

5.4.1 Absolute Difference Function

Algorithm 9 A method that returns the absolute difference between two numbers

Ensure: /** $result \geq 0$ **/

```
1: function ABSOLUTE(int i, int j)
2:     if (i > j) then
3:         r = i − j
4:     else
5:         r = j − i
6:     end if
7:     return r
8: end function
```

Let us consider the program provided in Algorithm 9, which returns the absolute dif-

ference between two numbers. In this case, it is clear that the returned result should be non negative. So we define the specification as $(r \geq 0)$, where r is the result.

Steps

Our verifier will proceed as follows while parsing the code line by line:

1. From parsing the **function definition** at line 1, function parameters i, j will be saved to VARSTORE.

2. Steps for the **if statement**:

   (a) The verifier checks if variables i and j are declared.

   (b) It then detects if the constraint generated so far, which is empty in this case, is consistent with the specification, $r \geq 0$. It is consistent in this case.

   (c) It goes on to evaluate the expression $i > j$ to see if it is a valid condition. If no numeric value is available for i and j which is the case here, it will just add the string $i > j$ to STORE.

   (d) It will also add $i \leq j$ to STACK for later exploration.

3. From the **assignment statement** $r = i - j$, the verifier will try to see if $i - j$ can produce any numeric value and then it will add $r : val(i - j)$ to VARSTORE. Since i and j have no value assigned, the parser will add the pair $r : i - j$ to VARSTORE.

4. At line 4, the *else* statement is ignored for now, since it will be handled later when exploring STACK after the branch following from the *if* is explored completely.

5. Line 7 is reached: **return** r. The **return** statement indicates that a leaf has been reached. Therefore the verifier performs a final consistency check like as follows.

   - At this step, STORE = $[i > j]$ and VARSTORE = $[r : i - j]$. From these, our verifier can derive that $r > 0$ and updates STORE to $[r > 0, i > j]$.

- The algorithm compares the constraints in the STORE with the specification (which is $r > 0$) and it sees that the STORE is complying with it. So the program is consistent at this point.
- Then the algorithm queries STACK, retrieves its top elements, and restores back STORE to what it was at this alternative point of execution. STORE becomes $i \leq j$. The *else* branch is now about to be explored.

6. The algorithm then repeats the same steps as it followed for the *if* part for each line following the *else* statement starting at line 4.

7. Again, when reaching the **return** r statement, it performs a consistency check.

   - STORE:= $[i \leq j]$, VARSTORE:= $[r : j - i]$.
   - Parser derives $r \geq 0$ from STORE and VARSTORE.
   - Then the algorithm compares the STORE with specifications $r \geq 0$. Since they are not inconsistent, this means that the code satisfies the user defined specification and passes the verification step so far.

8. Since there is no more execution point in STACK to be restored and explored, our verifier knows that the complete execution path from the input program is explored and then it declares that the **program has been verified**.

9. As side information valuable for the analysis of our verifier, the process will also provide us with the depth of tree traversal and the nodes generated during this tree traversal.

Figure 5.2 shows the tree generated from the traversal of the absolute-difference program.

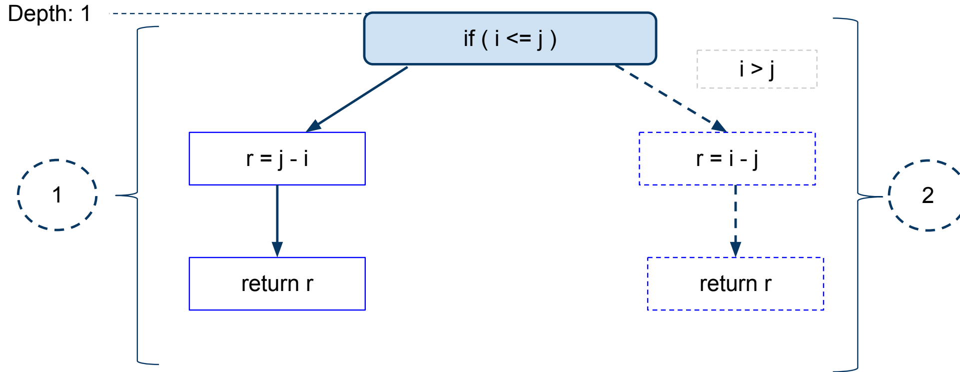

Figure 5.2: Tree for Absolute Difference Algorithm

5.4.2 Insertion Sort Algorithm

The previous example of a numerical problem shows how the symbolic verification steps work with this type of problems. Now, we analyze another example of a sorting algorithm, **insertion sort** (Algo. 10). The verifier works for an arbitrary array length, but due to limited space, we only consider the insertion sort working on an array of length 2. After sorting the two elements of the array, the end result must satisfy this constraint $a[0] \leq a[1]$, which is the specification here.

Steps in Brief

In the following description, we have used notation $\langle T_i \rangle$ and $\langle F_i \rangle$, in which i denotes the line number in Algo. 10 and T, F denote the true or false alternative of a conditional statement respectively. E.g., line 5 of Algo. 10 is $while(i < l)$. The true alternative, which is $i < l$ will be denoted by $\langle T_5 \rangle$ and false alternative, which is $i \geq l$ will be denoted by $\langle F_5 \rangle$.

- From the **first iteration**, for $i = 0$, the true alternative of the first conditional statement $\langle T_5 \rangle$ and false alternative of second conditional statement $\langle F_8 \rangle$ will be selected. After the first iteration: STORE:= $\emptyset$ and VARSTORE:= $[v : a[0]]$.
- From the **second iteration**, for $i = 1$, the verifier will select the execution path with

Algorithm 10 Insertion Sort

Ensure: /** $\forall i$, $a[i] \leq a[i+1]$ **/

```
1:  function INSERTION(int *a, int l)
2:      int i = 0
3:      int j = 0
4:      int v = 0
5:      while (i < l) do
6:          v = a[i]
7:          j = i − 1
8:          while (j ≥ 0 && a[j] > v) do
9:              a[j + 1] = a[j]
10:             j = j − 1
11:         end while
12:         a[j + 1] = v
13:         i = i + 1
14:     end while
15:     return a
16: end function
```

$\langle T_5, T_8, F_8 \rangle$alternatives. After the second iteration: STORE:= $[0 \geq 0, a[0] > a[1]]$ and VARSTORE:= $[v : a[1], a[1] : a[0], a[0] : a[1]]$.

- Next the verifier proceeds for the **third iteration**, where $\langle F_5 \rangle$ will be selected since $i = 2$, $l = 2$, and $i \not< l$. So, it goes to line 14 which is the **return** statement. Thus the verifier has explored one complete execution path and it is at the leaf level of that branch.

- At this stage, it is time for the verifier to check the **consistency**. At this point, STORE:= $[0 \geq 0, a[0] > a[1]]$ and VARSTORE:=$[v : a[1], a[1] : a[0], a[0] : a[1]]$. From the comparison and assignment constraints the parser derives the rule $a[1] > a[0]$, which is then compared against the specification $a[0] \leq a[1]$. They are consistent here.

- After one branch is completely explored, it will look into the **STACK** to check if there is still some branche to be explored. It will resume execution from the node which is at the STACK top. In this case, we will have a valid alternative from line 8, which is $a[j] \leq v$ (also at the top of the STACK). Then the verifier starts exploring this branch in the same way as described above.

In case of **sorting algorithms**, we check the consistency only after the completion of all the iterations of the loops so that it will build the complete set of constraints. In any intermediate stage, the constraint store would not be sufficiently developed to check against the specification since a sorting algorithm will always sort some portion of the array in a specific loop depending on the**loop invariant** for that algorithm. Only after all the loop is completed it can guarantee a completely sorted order of the input array.

The tree generated from the tree traversal of the insertion sort program (see Algo. 10) is illustrated in Figure5.3. From this algorithm, applied to an array of length 2, maximum depth of tree traversal is 7, current depth is 6, and number of nodes is 18. The order of the tree traversal (in DFS way) is $1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 14 \rightarrow 16 \rightarrow 11 \rightarrow 11 \rightarrow 18$.

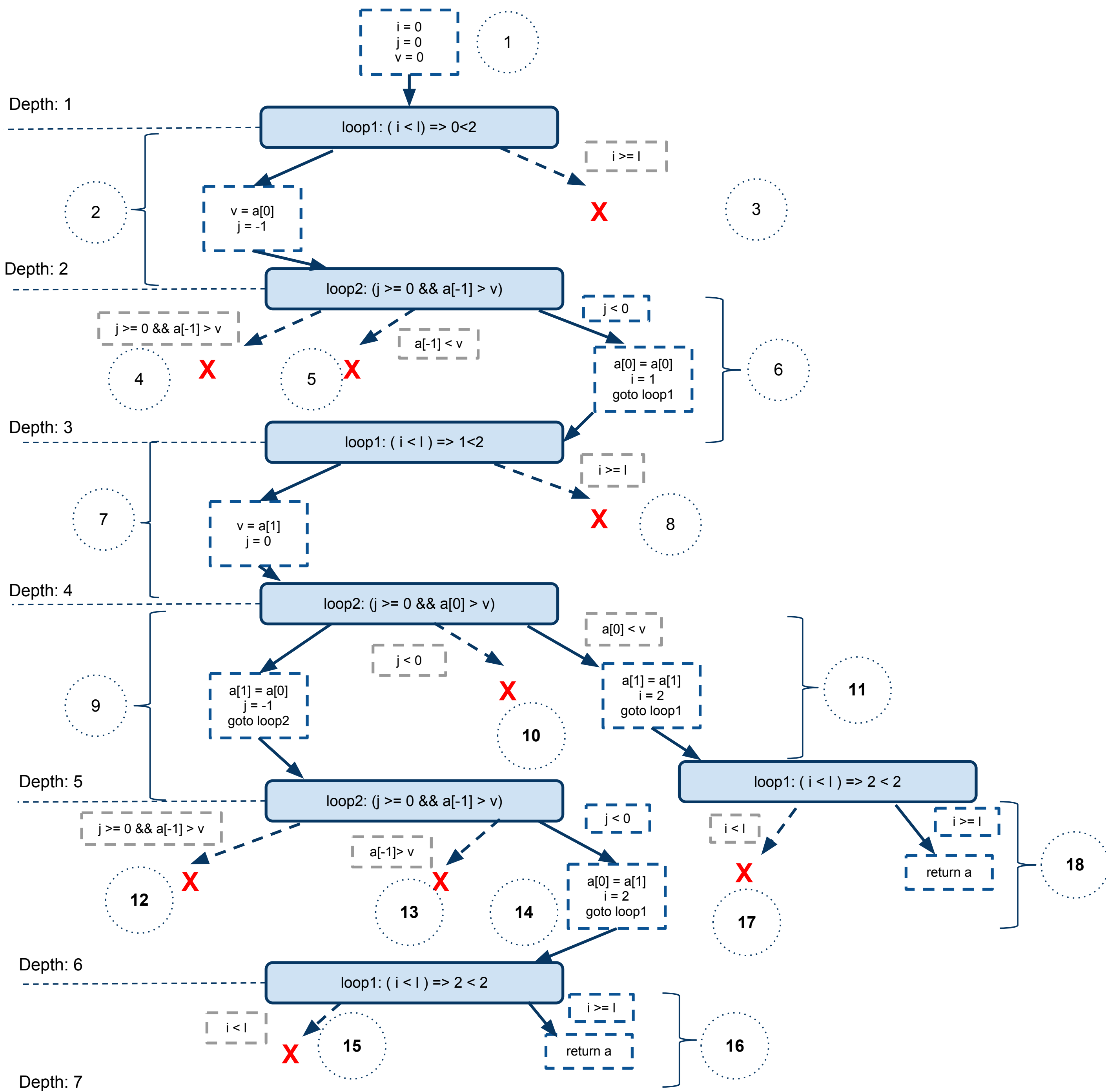

Figure 5.3: Tree for Insertion Sort Algorithm

5.4.3 Insertion Sort with Error

Here we illustrate the same insertion sort algorithm with an error. Let us change line 8 of Algo. 10 to **while(**$j \geq 0$ **&&** $a[j] < v$**)** . Lets see how the verifier will detect the error even without calling the solver.

Steps in Brief

- The **first iteration** will generate STORE:= $\emptyset$ and VARSTORE:= $[v : a[0]]$.
- From the **second iteration**, for $i = 1$, the execution path will select $< T_5, T_8, F_8 >$ alternatives. So it will create STORE:= $[a[0] < a[1]]$ and VARSTORE:= $[v : a[1], a[1] : a[0], a[0] : v]$.
- After the **third iteration**, where false alternative of the first **if** condition will be selected since i = 2 and l = 2 it goes to line 19 which is the **return** statement. Since this is the leaf level of that branch, the verifier checks the consistency.
- **Consistency check:** at this point, STORE:= $[a[0] < a[1]]$ and VARSTORE:= $[v : a[1], a[1] : a[0], a[0] : v]$. From the comparison and assignment constraints the parser derives the rule $a[1] < a[0]$, which is then compared against the specification $a[0] \leq a[1]$, and it is inconsistent with the specification here.
- The verifier marks this erroneous branch and starts to explore other alternative branches which are in STACK.
- When all the branches in STACK are explored, then the verifier checks if it encountered any inconsistency earlier and since it did in this case, it will declare that the **Program is not verified**.

The tree traversal will be similar as in Figure 5.3 except some changes in constraints. And the error that we discussed here will be caught after the first branch is completely explored which is at depth 7.

Chapter 6

Experiments and Results

In the previous section, we illustrated our approach by examples. We currently have the verifier working for numerical problems and sorting algorithms. Following we set our experimental goals. We then describe the experimental methodology we followed, report the results and analyze them.

Goal for the Experiments

Our goal was to make verification process more reliable and practical. In order to assess our work, we need to fully understand the working of our technique. We also need to conduct experiments to provide us a clear comparison with the state of the art technique used in constraint-based software verification CPBPV. This can be expressed in terms of the following goals:

- **Goal 1:** We are interested in understanding how our verifier behaves, how it exploits the tree structure in terms of the number of nodes generated, maximum depth reached, and the average depth of the tree. On a comparative scale how does it perform with different classes of the programs: both correct and incorrect?

- **Goal 2:** Though CPBPV constitutes a reasonable framework for constraint-based software verification, we showed in Chapter 4 and 5 that there is a need for improvement: for instance relieving the verification process from computationally intensive solving. That is why we proposed a different approach using **symbolic analysis** of the constraints, while they follow a **numerical approach**. Here we are interested in

analyzing through experiments what exactly the benefits of our approach over them are. In particular, we are interested in comparing the two approaches in terms of:

  - speed;
  - error detection capability;
  - space management capability; and
  - scalability.

- **Goal 3:** Numerical solving processes introduce lots of noise in the solution as false positive or rounding errors due to floating-point numbers with potential false negative[1] or/and false positives, while symbolic approaches do not. Both numerical and symbolic approaches can be tedious for complex problems and that is the reason why we kept the symbolic analysis at its simplest level. Here our question is: does our technique ever generate false positive? False negative?

6.1 Experiment Setup

Our verifier is written in Python and our experiments were conducted on a system with 2.13 GHz Intel Core(TM)2 CPU and 2 GB of RAM. We describe the experiments conducted that allowed us to address our goals in what follow:

- **For Goal 1,** the verifier is run on bubble sort, insertion sort and selection sort algorithms to test the working of our verifier with different classes of programs. The experiments are conducted on both correct and incorrect sorting algorithms for a comparative measurement of our verifier's performance. Also we have analyzed the impact of number of constraints on the verifier running time, meaning how the number of constraints grows with the array size and how it affects the overall time taken by

[1] A false negative is a case in which we conclude that the code is correct while it is not. False negatives are much more dangerous than false positive as they cause to overlook problems in code.

the verifier for tree traversal. For the behavioral analysis we measure the following variables generated from the tree traversal of the input program:

  - **Number of nodes** generated while traversing the tree;
  - **Current depth** of the tree at which the verifier stopped;
  - **Maximum depth** of the tree traversal; and
  - **Time taken** by the verifier to verify the program.

- **For Goal 2,** we compared our results with the results from CPBPV when we ran the experiments on correct and incorrect sorting algorithms to compare our approach with them.

- **For Goal 3,** we have used a numerical solver to solve the constraints generated from traversing a branch generated from the correct insertion sort algorithm. And also we have run our verifier on the same set of constraints. This way, we were able to show how the risk of false positive or/and false negative was avoided in the case of our approach.

6.2 Results

In this section, we present the experimental results corresponding to the experiments we just described.

6.2.1 Experimental Results for Goals 1 and 2

For Goals 1 and 2, we carried out experiments with various sorting algorithms (both correct and incorrect) to understand the behavior of our verifier. We also compared the correct and incorrect case, to show if our verifier behaves differently for these two cases. We then compared our results with CPBPV to analyze what exactly the benefits of our approach over them are.

Correct Input Code

The following Tables 6.1, 6.2, and 6.3 show the experimental results when our verifier runs on the respective sorting algorithms with a correct input code. The graphs from these results are shown in Figure 6.1. The graphs show time(ms)/no of nodes/max depth/current depth (in y-axis) for various array lengths (in x-axis). The y-axis of the graph is drawn in **logarithmic** scale.

Table 6.1: Results for **correct insertion** sort

Length of the Array	Time taken (sec)	No of nodes	Max Depth	Current Depth
2	0.031	18	7	6
3	0.016	48	11	8
4	0.063	168	16	10
5	0.187	768	22	12
6	0.812	4368	29	14
7	5.562	29568	37	16
8	46.578	231168	46	18

Table 6.2: Results for **correct selection** sort

Length of the Array	Time taken (sec)	No of nodes	Max Depth	Current Depth
2	0.032	23	8	8
3	0.047	111	14	14
4	0.125	927	22	22
5	1.796	14911	32	32
6	58.718	477311	44	44

Observation: From the graph we see that we do better for insertion sort. Our verifier runs for array length of 5 and 6 for bubble and selection sort respectively while it runs for array length of 8 for insertion sort. The tree from both bubble and selection sorting

Table 6.3: Results for **correct bubble sort**

Length of the Array	Time taken (sec)	No of nodes	Max Depth	Current Depth
2	0.016	15	6	6
3	0.046	79	12	12
4	0.125	671	20	20
5	1.500	10815	30	30
6	48.650	346239	42	42

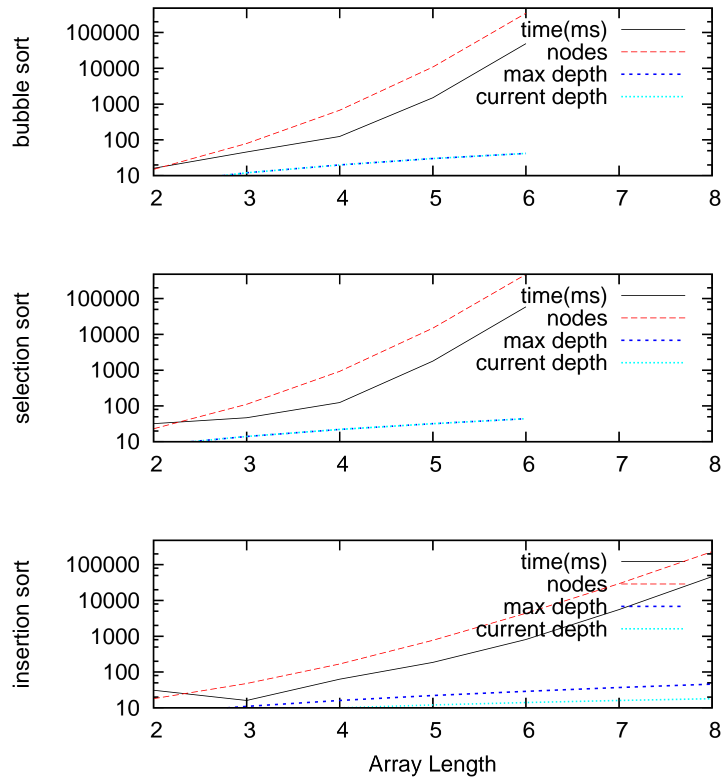

Figure 6.1: **correct** sorting algorithms

algorithms are much more complex and bigger in terms of nodes and maximum depth reached. That is why the verifier acts differently for different sorting algorithms.

Incorrect Input Code

The following Tables 6.4, 6.5, and 6.6 show the experimental results when our verifier runs on the respective sorting algorithms with an incorrect input code. The graphs from these results are shown in Figure 6.2. The first, second and third graphs show time(sec), no of nodes, max depth respectively for the three sorting algorithms in y-axis and array lengths in x-axis. These graphs are plotted in **regular** scale.

To test incorrect insertion sort, first we changed line 8 of the insertion sort algorithm (Algo. 10) to **while (j $\geq$ 0 && a[j] $<$ v)**. Similarly we have changed some valid lines with invalid logic for selection and bubble sort to make them incorrect.

Table 6.4: Results for **incorrect insertion** sort

Length of the Array	Time Taken (sec)	Number of Nodes	Max Depth	Current Depth
2	0.015	16	7	7
4	0.015	41	16	16
8	0.063	127	46	46
16	0.218	443	154	154
32	2.249	1651	562	562
40	5.448	2543	862	862
50	13.162	3928	1327	1327
64	33.330	6371	2146	2146
100	200.546	15353	5152	5152

Observation: From the graphs, we see that our verifier works with much bigger arrays (length 100) than what it was able to do for correct case (length 8). Though here we have the results till array length of 100, our guess is, it can work for even bigger array size. In terms of time, we do better for selection sort and in terms of other parameters we do better

Table 6.5: Results for **incorrect selection** sort

Length of the Array	Time Taken (sec)	Number of Nodes	Max Depth	Current Depth
2	0.031	15	8	8
4	0.047	43	22	22
8	0.062	147	74	74
16	0.156	547	274	274
32	0.702	2115	1058	1058
40	1.249	3283	1642	1642
50	2.326	5103	2552	2552
64	4.855	8323	4162	4162
100	16.784	20203	10102	10102

Table 6.6: Results for **incorrect bubble** sort

Length of the Array	Time Taken (sec)	Number of Nodes	Max Depth	Current Depth
2	0.016	11	6	6
4	0.031	39	20	20
8	0.047	143	72	72
16	0.389	543	272	272
32	4.381	2111	1056	1056
40	11.381	3279	1640	1640
50	25.765	5099	2550	2550
64	67.053	8319	4160	4160
100	398.45	20199	10100	10100

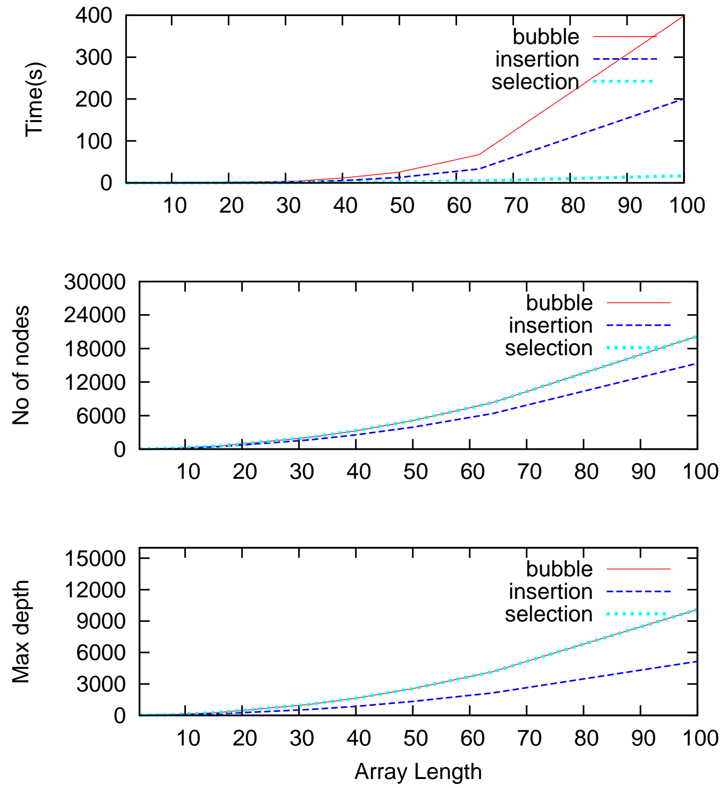

Figure 6.2: **incorrect** sorting algorithms

for insertion sort. By using symbolic methods, the verifier was able to detect the error very fast for all of these algorithms. Number of nodes and maximum depth are larger for bubble and selection algorithm because of the complex nature of the program and more nested if structure than the insertion sort algorithm. But in terms of time taken, selection sort is much faster than the other two sorting algorithms. The number of constraints generated is much smaller for selection sort than for the other two algorithms because of the different way this algorithm works. Hence, the amount of time for symbolic analysis is smaller for selection sort.

Impact of the Number of Constraints on our Verifier's Running Time

Tables 6.7, 6.8, and 6.9 show the impact of the number of constraints on the verifier's running time. As expected, the number of constraints grows with the array length, and hence the running time of our verifier. In particular, we observe that for insertion sort and bubble sort, the number of constraints grows quadratically with the size of the input, while it only grows linearly for selection sort. This difference can be explained by the difference in the behavior of the sorting algorithms (see Analysis of Results w.r.t. Goal 1 for more details). The time grows as fast as the number of constraints.

Table 6.7: From insertion sort

Length of the Array	Time Taken (sec)	Number of Constraints
2	0.015	1
4	0.015	6
8	0.063	28
16	0.218	120
32	2.249	496
64	33.33	2016

Table 6.8: From selection sort

Length of the Array	Time Taken (sec)	Number of Constraints
2	0.031	1
4	0.047	3
8	0.062	7
16	0.156	15
32	0.702	31
64	4.855	63

Table 6.9: From bubble sort

Length of the Array	Time Taken (sec)	Number of Constraints
2	0.016	1
4	0.031	6
8	0.047	28
16	0.389	120
32	4.381	496
64	67.053	2016

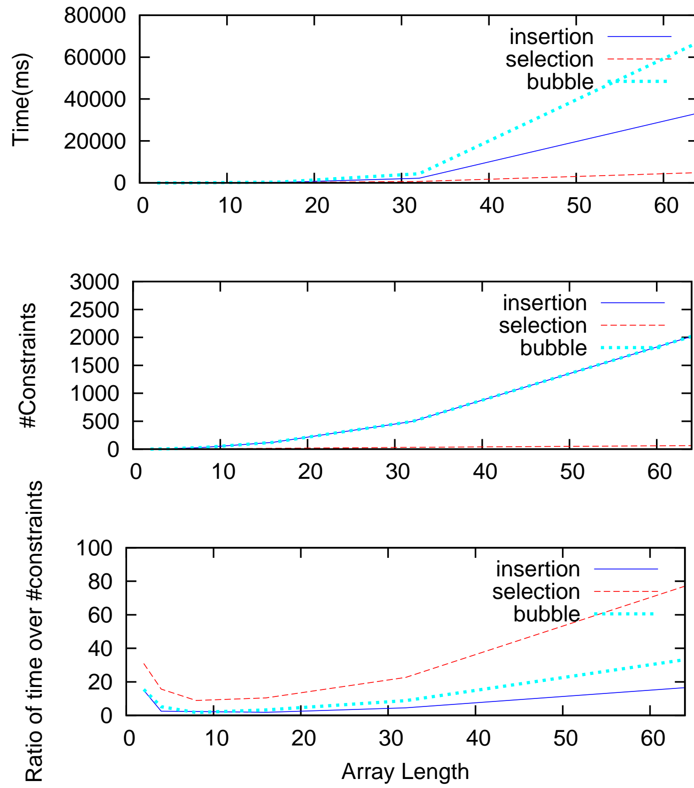

Figure 6.3: Relation between Time Taken and Number of Constraints

Both Correct and Incorrect Input Code

In Figure 6.4, we show the comparison graphs between the correct and incorrect sorting algorithms. Here we have only shown the difference between running time and no of nodes generated for the correct and incorrect sorting algorithms since the other parameters are same for both cases. The graphs show time(ms)/no of nodes (in y-axis) for various array lengths (in x-axis) for both correct and incorrect case. To make the graph readable, we have limited the range to 32 in x-axis even though for incorrect case, the verifier works for array length of 100 which we have shown earlier. The y-axis of the graph is plotted in **logarithmic** scale. Observation from these graphs are discussed in "Analysis of Results w.r.t. Goal 1".

Observation: The graphs show that the verifier is faster when the code is incorrect. For the incorrect case, the verifier runs for array length of 100 while for correct case it runs up to size 8. Also the number of nodes is drastically reduced since we are able to detect the errors very early in the branch. Since for the correct case, we produce a much bigger tree with large number of nodes and branches than the incorrect case, the verifier performs a lot better in the incorrect case than the correct one. These results show that our verifier works with array **length of 100** for incorrect case, which to the best of our knowledge, CPBPV does not. This is an improvement we achieved over the existing benchmark techniques.

Analysis of Results w.r.t. Goal 1

We have carried out our experiments with various sorting algorithms, both correct and incorrect. Here we discuss how the experiments helped us to establish our **Goal 1** which is to understand the workings of our verifier.

- In general, the experiments show that our approach is feasible. Both for correct and incorrect case, our algorithm is able to determine the compliance between the code and its specification with simple symbolic steps without using any numerical solver.
- Our algorithm works faster when the input code is incorrect than in the correct case.

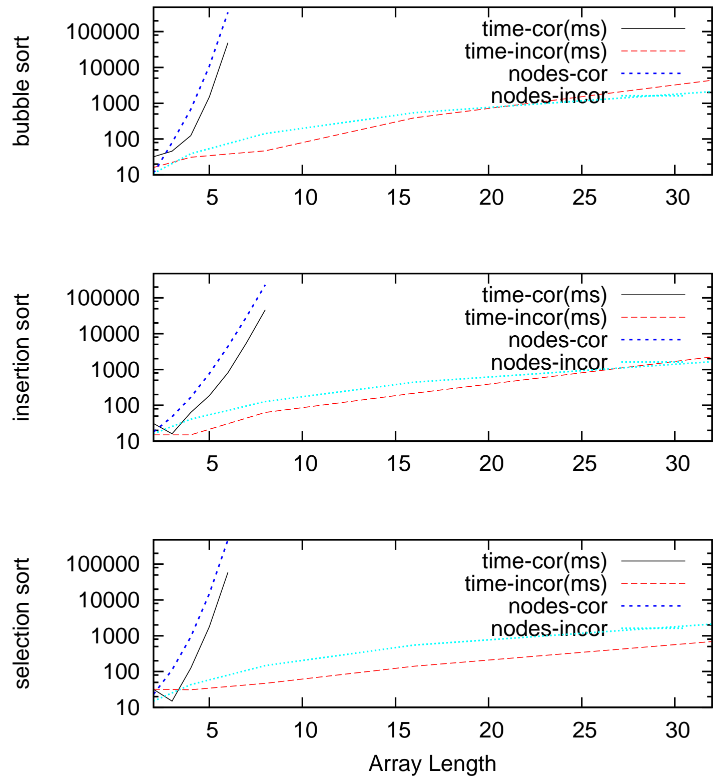

Figure 6.4: **incorrect and correct** sorting algorithms together

For the incorrect case, we have tested our approach with array length of 100 while for correct case it works up to size 8. Since we detect the error earlier in the branch and that branch is pruned from the tree for incorrect case, the amount of tree traversal is much smaller than the correct case.

- In the correct case, the verifier works better for insertion sort than for the bubble and the selection sort. The reason for this is that the behavior of the verifier varies based on the size of the tree developed from the program, which in turn is based on the nested loop structure of the code. For bubble and selection sort, the nesting structure of the loop is different than for the insertion sort. The larger the nesting level, the larger the generated tree. Hence, the amount of time for tree traversal gets longer.

- Analysis of the relation between the verifier's running time and the number of constraints:

  - From the results in Tables 6.7 and 6.9, which correspond to insertion and bubble sort respectively, it is clear that the number of constraints grows approximately with a factor of n^2, where n is the length of the array. As a result the time to treat each node grows and the total running time as well. It grows by the same factor as the number of constraints. Similarly, Table 6.8, which corresponds to selection sort, shows that the number of constraints grows with a factor of n, where n is the array length. Because of the structure of the algorithm, for selection sort, the comparison constraints are always formed with each array element paired with the immediate next array element; e.g., for array of length 3, there are three array elements: $a[0]$, $a[1]$, and $a[2]$. Therefore the comparison constraints for selection sort will be: $a[0] \leq a[1]$ and $a[1] \leq a[2]$, while, in case of insertion and bubble sort, all of the array elements will be paired with each other in the form of comparison constraints. As a result, the comparison constraints for insertion and bubble sort will be: $a[0] \leq a[1]$, $a[0] \leq a[2]$, and

$a[1] \leq a[2]$. Thus the number of constraints for selection sort is less than the other two sorting algorithms. Hence the time taken by our verifier for selection sort is also less than the verifier running time for insertion and bubble sort.

- Another observation from the results (see Figure 6.3) is that the ratio of time over number of constraints starts with a high peak for small arrays and then gradually decreases with array length until a certain point is reached and then starts increasing again. The reason is that the impact of the number of constraints over the verifier running time is not so significant until a certain amount of constraints (which is around 30 for insertion and bubble sort applied to array of size 8) are generated. So the verifier running time remains almost constant for small arrays even if number of constraints grows. Hence the ratio of time over number of constraints decreases almost until array of size 8 is reached. For array of size bigger than 8, the verifier running time in treating the number of constraints at each executable node grows significantly and hence the ratio of verifier running time over the number of constraints also grows.

Analysis of Results w.r.t. Goal 2

We have compared our experimental results for sorting algorithms: both correct and incorrect with CPBPV. Here we discuss how the experiments helped us to establish our **Goal 2**.

- In terms of **speed**, our observation is the following:
  - The results show that, we are still slow in the correct case compared to CPBPV. It seems we are slower per node when the number of the constraints from the specification and the code gets larger in count. Since each of these constraints are checked against each of the constraints in the specification, it tends to take more time than the numerical constraint solving at this stage. Also to be consistent with RealPaver format, we did not handle disjunction of constraints. In our

case, each constraint in disjunction is treated separately. Hence, each of the constraints in the disjunction generates a separate sub-tree when together; they should have produced just one sub-tree. That is why we produce tree which is same depth but much wider than CPBPV.

- Though we are slower than CPBPV when the input code is correct, our algorithm is much better when the code is incorrect. In this case, our verifier works with array length of 100, which to the best of our knowledge, CPBPV does not. In case the code is incorrect, we caught the errors earlier than CPBPV because the error is detected as soon as that erroneous line is encountered. For numerical process, they still develop a whole lot of the tree before they actually solve the system at some decision point (conditional statement) to detect that error, which takes more time. So CPBPV develops a tree which is much deeper than us.

- In terms of **error detection** capability, our verifier performed much better than CPBPV. Their approach does not work for array length of 32 for bubble sort while our algorithm works with array length of 100 if the input code is incorrect. Also we could catch some other types of errors symbolically like *variable declaration missing* or *array index out of range*which to the best of our knowledge, CPBPV does not.

- In terms of **space management** capability, CPBPV generates lot of variables while developing the tree, which fills up their repository of constraints and they faced the challenge of controlling it. In our case, we were able to restrict the STORE size by updating the variables with their recent values.

- In terms of **scalability**, we are more scalable than CPBPV in incorrect case. Since our verifier handles array of size 100 while CPBPV stops at 16.

6.2.2 Experimental Results for Goal 3

Our third goal was to understand whether our verifier would help avoid false positives/negatives. In order to assess this, we coupled our verifier with a numerical solver: the interval constraint solver RealPaver [35, 36].

The anticipated problem with RealPaver is that it would generate solutions even if there are not. As a result, we explicitly used it to check the consistency of the program with the specifications, in the case of the correct insertion sort program (see Algorithm 10). When doing this for inputs of length 3 only, we already observed the anticipated problem. Let us take a look at this case:

The specification for this case is $a[0] \leq a[1] \wedge a[1] \leq a[2]$. Thus the negation of the specification is $a[0] > a[1] \vee a[1] > a[2]$. Since RealPaver doesn't handle disjunctions, each of these constraints in negation of the specification will be combined with the constraints from the code into two separate CSPs. In the figures below, we showed the results provided by RealPaver for only one of the CSPs: the one considering the following constraint from the negation of the specification $a[0] > a[1]$ (the results for the other CSPs are similar).

Hence, RealPaver was used to solve the following CSP: $C \wedge \neg S$, where C is the set of constraints generated from the first branch of the code and $\neg S$ is $a[0] > a[1]$. At the same time, we performed a purely symbolic comparison between C and $\neg S$ using our verifier. For a correct sorting algorithm, this CSP should not have any solution: because the code conforms to its specifications.

The following experiment (see Figure 6.6) shows that by using RealPaver, we obtain some solutions while our symbolic verifier did not produce any. Figure 6.5 shows the CSP written in RealPaver format and the output from it. Figure 6.6 shows how our verifier works at this stage with the same set of constraints.

Note: The example below was run on a simple array of integers and yet RealPaver generated solutions. In a more conventional setting for checking consistency, larger arrays of intervals would be considered. In such settings, RealPaver would produce an even larger number of false positives. However, since small and simple inputs already allowed to show

```
/*
* CSP written in RealPaver format

* CSP generated from correct Insertion
sort program and the negation of the
specification
*/

Constants
i=3,
j=-1,
l=3;

Variables
int a[0..2] in [1,2];

Constraints
a[2]>=a[1],
a[2]>=a[0],
a[1]>=a[0],
a[0]>=a[1];
```

```
/* REALPAVER solution */

INITIAL BOX
  a[0] in [1 , 2]
  a[1] in [1 , 2]
  a[2] in [1 , 2]

INNER BOX 1
  a[0] = 2
  a[1] = 2
  a[2] = 2

  precision: 0, elapsed time: 0 ms

INNER BOX 2
  a[0] = 1
  a[1] = 1
  a[2] = 2

  precision: 0, elapsed time: 0 ms

INNER BOX 3
  a[0] = 1
  a[1] = 1
  a[2] = 1

  precision: 0, elapsed time: 0 ms

END OF SOLVING
  Property:     reliable process (no solution is lost)
  Elapsed time: 0 ms
```

Figure 6.5: **RealPaver solution** to the CSP generated from correct insertion sort algorithm

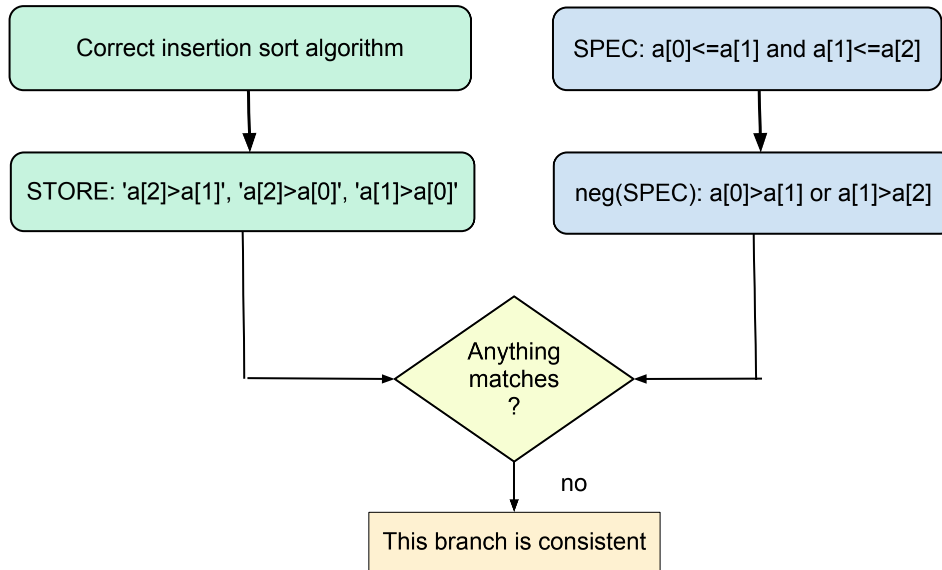

Figure 6.6: **Verifier solution** from correct insertion sort algorithm

the shortcomings of the numerical approach, and for the sake of simplicity of the above figures, we did not report more experiments.

Analysis of Results w.r.t. Goal 3

The union of all the boxes returned by RealPaver represents the solution to the CSP $(C \wedge \neg S)$, where C is the set of constraints derived from the first branch of the insertion sort program and S is the set of constraints derived from the specification. Since this CSP has solutions in RealPaver, our verifier decides that the input code is inconsistent with its specification while that is not the case for the correct insertion sort algorithm. Therefore this is an example of a situation in which using a numerical solver caused our verifier to generate a false positive.

The reason why RealPaver generates solutions is that it does not allow to model strict inequalities such as '$>$' or '$<$'. Strict inequalities always have to be replaced by '$\geq$' or '$\leq$'. Because of the extra equality ('$=$') in the relation, solutions can be found although they do not correspond to any reality in the verification process: we obtain false positives.

Using our purely symbolic approach, this problem is avoided and our verifier is able to decide that the constraints in STORE have nothing in common with the constraints from the negation of the specification, hence the branch is correct.

We tested our approach with a simple insertion sort algorithm, which is non-numerical in nature. But with a more conventional numerical problem, which deals with floating-point numbers, the verifier might run into the problem of false-positives. Due to the challenges associated with floating-point numbers (described in Section 3.2.2), our verifier might: (1) return solutions to a constraint system that is inconsistent; (2) incorrectly follow an execution path and can lead to incorrect combination of constraints for which a correct program will prove to be faulty. In future work, we discuss ways to incorporate techniques in our verifier to better deal with floating-point numbers, such as the symbolic execution with floating-point numbers described in Section 3.2.3.

General Conclusion about Our Experiments

Overall, the experimental results reported above clearly show that our proposed approach works well to detect the consistency/inconsistency without using a numerical solver. We believe, from analyzing our experimental results, that our symbolic solver, if integrated to the existing framework of CPBPV, has the potential to simplify the verification process by reducing the overhead associated with the constraint solving techniques and by allowing catching errors in code faster and more reliably.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Our aim is, given a specification and an implementation of it, to verify that they comply. Although there exist many tools for conducting software verification, none of them has proved to be really simple and cost-effective. Recently, constraint programming techniques for software verification have emerged. CPBPV is currently the leading constraint-based framework for software verification. It achieves significant gain in performance and provides new functionalities compared to other tools for software verification. However, there exist areas for improvement as pointed out in Chapters 3 and 4. In particular, the main overhead associated with CPBPV is its numerical constraint solving process and its overflowing constraint store. The numerical solving process may produce noise and rounding errors due to floating-point number computations while determining solutions.

In this work, we proposed a different way of detecting the consistency between the program and its specification. We followed a structure similar as CPBPV for exploring the execution paths of the program to be verified, but we used simple **symbolic steps** instead of a numerical solving process for checking the compliance between the code and its specification. The symbolic steps involve simple forms of string matching techniques to check if any constraint generated so far negates the specification. Besides, our logic reassigns values to existing variables without the need for creating new variables, which helps control the size of the constraint store.

We implemented our approahc for checking the consistency between the code and its specification symbolically instead of numerical constraint solving to reduce the overhead

associated with the numerical constraint solving process for software verification. We conducted experiments that show that our algorithm works well in deciding the consistency/inconsistency of the program with its specification without the aid of a numerical solver. We achieved a better performance than CPBPV when the input code was incorrect: with our symbolic verification steps, we could catch errors faster than CPBPV. Our approach works on sorting algorithms for **array length of 100** in the incorrect case while CPBPV stops at array length 16: this is clearly an achievement over the existing constraint-based software verification frameworks and it shows the promise of our approach.

7.2 Future Work

Based on our experimental results, we observed that our algorithm (1) takes exponential time with correct sorting algorithms for increasing array lengths. For correct sorting algorithms the verifier works till array length of 8. Besides, it has structural limitations: (2) It does not handle very complex numerical expressions except simple arithmetic operations. (3) It does not handle recursion and procedure calls and returns. (4) It works with input program written in C and with some specific syntax as described in Section 5.3. (5) It does not handle floating-point number constraints.

Although it performed much better than CPBPV with incorrect input codes, the above observations indicate that there is room for further improvements. In what follows, we draw directions for future work:

1. Currently the verifier takes exponential time for verifying correct sorting algorithms and therefore cannot process input sizes more than 8. In order to address this issue, we plan to:

    (a) Modify the exploration process by both integrating a bi-directional search and using the weakest precondition strategy. In bi-directional search, we start traversing from the start node and goal node simultaneously. As a result, the number

of nodes to be explored is significantly reduced and so is the time to solution. We plan to study the feasibility of this approach in our future work;

(b) Modify the structure of our tree by limiting its width: in the case of correct pieces of code, CPBPV and our verifier should generate the same number of nodes. However, at this stage of design of our verifier, it is not the case because our branching factor is much larger. We plan to embrace disjunctions in our symbolic solver to be able to prune a lot of the unnecessary width.

2. The verifier currently works with simple numerical expressions. It contains a built-in function that processes arithmetic expressions, and it can be very easily upgraded in order to process complex arithmetic expressions.

3. To handle recursion and function calls, we will have to integrate modular verification techniques in our approach.

4. The parser in our verifier is written to identify specific types of program statements. This logic can be easily modified to identify more general program structures.

5. Our verifier currently does not handle floating-point number constraints. In order to incorporate this feature, we need to use techniques such as, *normalization*, *projection functions* etc. as described under the title "Symbolic Execution of Floating-Point Computation" in section 3.2.3.

We also plan to enhance our verifier so as to catch as many types of errors as possible, such as infinite loops or type checking. We would also like to explore the possibilities of integrating our work in the existing frameworks for software verification to analyze the performance and efficiency of the integrated model.

References

[1] C. Acosta. A Constraint-based Approach to Verification of Programs with Floating-Point numbers. Master's thesis, The University of Texas at El Paso, December 2007.

[2] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *Computing Surveys*, 14(2):159–192, 1982.

[3] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137–147.

[4] A. W. Appel. *Modern Compiler Implementation in {C,Java,ML}*. Cambridge University Press, 1998.

[5] K. R. Apt and M. G. Wallace. Constraint Logic Programming using ECL^iPS^e. Cambridge University Press,1 edition, 2007.

[6] R. C. Backhouse. *Program Construction and Verification.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[7] O. Balci. Validation, verification, and testing techniques throughout the life cycle of a simulation study. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 215–220, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[8] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984. http://www.cs.ru.nl/∼henk/Personal Webpage.

[9] H. P. Barendregt. *Lambda Calculi with types*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[10] B. Beizer. *Software Testing techniques (2nd ed.).* Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[11] F. Benhamou, D. McAllester, and P. Van Hentenryck. Clp (intervals) revisited. Technical report, Providence, RI, USA, 1994.

[12] F. Benhamou and J. William. Applying interval arithmetic to real, integer and boolean constraints. *JOURNAL OF LOGIC PROGRAMMING*, 32(1):1–24, 1997.

[13] B. Benjamin Blanc, F. Bouquet, A. Gotlieb, B. Jeannet, T. Jeron, B. Legeard, B. Marre, and C. Michel. The V3F project. In *CSTVA'06*, 2006.

[14] S. Berezin. Model Checking and Theorem Proving: a Unified Framework. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, January 2002.

[15] A. Berztiss and M. A. Ardis. Formal Verification of Programs. Curriculum Module SEI-CM-20-1.0, December 1988.

[16] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations: Research articles. *Softw. Test. Verif. Reliab.*, 16:97–121, June 2006.

[17] B. Botella, A. Gotlieb, and C. Michel. Symbolic Execution of Floating-point Computations: Research Articles. *Softw. Test. Verif. Reliab.*, 16:97–121, June 2006.

[18] M. Ceberio, C. Acosta, and C. Servin. A Constraint-based approach to Verification of Programs with Floating-point Numbers. In *2008 International Conference of Software Engineering Research and Practice*, pages 225–230, 2008.

[19] D. Champeaux. Bidirectional Heuristic Search Again. *J. ACM*, 30:22–32, January 1983.

[20] A. K. Chandra and V. S. Iyengar. Constraint Solving for Test Case generation. In *ICCD 1992: Proceedings of the 1991 IEEE International Conference on Computer*

Design on VLSI in Computer & Processors, pages 245–248. IEEE Computer Society, 1992.

[21] Y. Cheon, A. Cortes, M. Ceberio, and L. T. Gary. Integrating Random Testing with Constraints for Improved Efficiency and Diversity. In *Proceedings of SEKE 2008, The 20-th International Conference on Software Engineering and Knowledge Engineering*, pages 861–866, 2008.

[22] J. G. Cleary. Logical Arithmetic. *Future Computing Systems*, 2:125–149, 1987.

[23] B. Cohen, W. T. Harwood, and M. I. Jackson. *The Specification of Complex Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[24] H. Collavizza and M. Rueher. Exploration of the Capabilities of Constraint Programming for Software Verification. In *TACAS*, pages 182–196, 2006.

[25] H. Collavizza, M. Rueher, and P. Hentenryck. CPBPV: a Constraint-Programming Framework for Bounded Program Verification. *Constraints*, 15:238–264, April 2010.

[26] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, first edition, 1990.

[27] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[28] R. A. DeMillo and J. A. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[29] M. S. Deutsch. Software Verification and Validation: Realistic project approaches. Prentice-Hall, Englewood Cliffs, 1982.

[30] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18:453–457, August 1975.

[31] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[32] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. *SIGPLAN Not.*, 37:191–202, January 2002.

[33] J. Gleick. A Bug and a Crash. New York Times Magazine, December 1996. http://www.around.com/ariane.html.

[34] A Gotlieb, B. Botella, and M. Rueher. Automatic Test data Generation using Constraint Solving Techniques. In *ISSTA 1998: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–62. ACM Press, 1998.

[35] L. Granvilliers. RealPaver User's Manual, August 2004. "http://pagesperso.lina.univ-nantes.fr/∼granvilliers-l/realpaver/src/realpaver-0.4.pdf".

[36] L. Granvilliers and F. Benhamou. Algorithm 852: RealPaver: an Interval Solver using Constraint Satisfaction Techniques. *ACM Trans. Math. Softw.*, 32:138–156, March 2006.

[37] D. Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[38] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12:576–580, October 1969.

[39] C. P. Hollocker. The Standardization of Software Reviews and Audits. *in G.G. Schulmeyer & J.I. McManus (Eds.), Handbook of Software Quality Assurance*, pages 211–266.

[40] IEEE. IEEE standards for Software Reviews and Audits. IEEE Computer Society, June 1989. www.cs.ucf.edu/∼workman/cen5016/IEEE1028.pdf.

[41] A. Jesdanun. GE Energy acknowledges Blackout bug. The Associated Press, February 2004. http://www.securityfocus.com/news/8032.

[42] S. Khanna. Logic Programming for Software Verification and Testing. *The Computer Journal*, 34(4):350–357, 1991.

[43] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Verification using Weakest Precondition strategy. In *In Computer Aided Verification of Information Systems A Practical Industry Oriented Approach: Proceedings of the Workshop CAVIS03*, February 2003.

[44] B. Marre and C. Michel. Improving the Floating point Addition and Subtraction Constraints. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 360–367, Berlin, Heidelberg, 2010. Springer-Verlag.

[45] H. C. Michael. Why Engineers Should Consider Formal Methods. In *In 1997 AIAA/IEEE 16th Digital Avionics Systems Conference*, pages 11–16, 1997.

[46] C. Michel. Exact Projection Functions for Floating point number Constraints. In *Proceedings of 7th AIMA Symposium*, 2002.

[47] H. D. Mills, V. R. Basili, J. D. Gannon, and R. G. Hamlet. *Principles of Computer Programming, A Mathematical Approach.* Newton, Mass.:Allyn and Bacon, 1986.

[48] R. Moore. *Introduction to Interval Analysis.* Prentice Hall, first edition, 1966.

[49] G. J. Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. *Communications of the ACM*, 21(9):760–768.

[50] G. J. Myers and C. Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[51] R. S. Pressman. *Software Engineering: A practitioners approach.* McGraw-Hill, fourth edition, 1996.

[52] B. Roman. Modelling Soft Constraints: A Survey. *Neural Network World*, 12:421–431, 2002.

[53] F. Rossi, P. V. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., New York, NY, USA, 2006.

[54] S. Schach. *Object-Oriented Software Engineering.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[55] A. M. Stavely. *Toward Zero-Defect Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1998.

[56] G. Tan. A Collection of Well-Known Software Failures, August 2009. http://www.cse.lehigh.edu/ gtan/bug/softwarebug.html.

[57] H. Theresa. Cause-Effect Graphing By Theresa Hunt. The WestFallTeam, 2007. http://www.westfallteam.com/Papers/Cause_and_Effect_Graphing.pdf.

[58] R. C. Waters. System Validation via Constraint Modeling. In *SIGPLAN Notices*, volume 26, pages 27–36, 1991.

[59] B. Weiß. Predicate Abstraction in a Program Logic Calculus. *Science Computer Program*, 76:861–876, October 2011.

[60] R. B. Whitner and O. Balci. Guidelines for selecting and using Simulation model Verification techniques. In *Proceedings of the 1989 Winter Simulation Conference*, pages 559–568. IEEE, 1989.

[61] E. Yourdon. *Structured Walkthroughs.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 1990.

[62] E. Yucesan and S. H. Jacobson. Building Correct Simulation Models is difficult. In *Proceedings of the 1992 Winter Simulation Conference*, pages 783–790. IEEE, 1992.

[63] E. Yucesan and S. H. Jacobson. Intractable Structural Issues in Discrete Event Simulation: Special cases and Heuristic approaches. *ACM Transactions on Modeling and Computer Simulation*, 6(1):53–75, 1996.

Resources

Few useful tools used in this work are given below with information on where to get them.

Python An interpreted high level programming language which is versatile, easy to use and cross platform. Our verifier is written in Python.
Available at http://www.python.org/

ScribTex Online latex editor and compiler, useful because documents can be edited and compiled from *any* internet connected browser, hence any machine that's connected to the internet.
Available at http://www.scribtex.com

Cygwin A collection of tools and utilities under windows, proving features and usefulness of Linux. Few tools used in this project are Latex and GnuPlot.
Available at http://www.cygwin.com/

RealPaver An interval software for modeling and solving nonlinear systems. It is an useful modeling language for numerical constraint solving. We have used this tool while experimenting with numerical solvers.
Available at http://sourceforge.net/projects/realpaver/

Curriculum Vitae

Shubhra Datta, daughter of Swapan Kumar Datta and Jaya Datta was born on 4th February 1985 in Howrah, India. She completed her secondary and higher secondary education in July 2003 under West Bengal Board of Higher Secondary Education. She entered West Bengal University of Technology in the Fall of 2003 and she graduated with a bachelor's degree in Computer Science and Engineering in the Spring of 2007. In Fall 2007 she joined a software firm called Oracle Financial Services Software Ltd. as an Associate Consultant and continued to work there till Fall 2008. In the Spring of 2009 Shubhra resumed her studies at The University of Texas at El Paso for her master's degree. She did her Thesis with Dr. Martine Ceberio. Besides while pursuing her master's degree in Computer Science she worked as a Research Assistant under Dr. Nigel Ward for the Spring 2009. After that she got an offer from the Center for research, evaluation and planning department (CIERP) of UTEP to work as Research Assistant. Since 2010 she became a full-time employee of the Center and still working there as Data Analyst.

Email:
shubhra.datta@gmail.com
sdatta2@miners.utep.edu

Permanent Address:
Khanyer Bagan Mondal Para, Kona Road
Howrah, West-Bengal, India. Zip – 711104