Optimised Constraint Solving for Real-World Problems

by

Johannes Hendrik Taljaard

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science (Computer Science) in the Faculty of Science at Stellenbosch University

Supervisor: Prof. W.C. Visser
Co-supervisor: Prof. J. Geldenhuys

December 2019

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2019

Abstract

Optimised Constraint Solving for Real-World Problems

J.H. Taljaard

Department of Mathematical Sciences, Division of Computer Science
University of Stellenbosch
Private Bag X1, Matieland 7602 , South Africa.

Thesis: M.Sc (Computer Science)

December 2019

Although significant advances in constraint solving technologies have been made during the past decade, Satisfiability Modulo Theories (SMT) solvers are still a significant bottleneck in verifying program properties. To overcome the performance issue, different caching strategies have been developed for constraint solution reuse. One of the first general frameworks for doing such caching was implemented in a tool called Green. Green allows extensive customisation, but in its basic form it splits a constraint to be checked into its independent parts (called factorisation), performs a canonisation step (including renaming and reordering of variables) and looks up results in a cache. More recently an alternative approach was suggested: rather than looking up *sat* or *unsat* results in a cache, it stores *models* (in the satisfiable case) and *unsatisfiable cores* (in the unsatisfiable case), and reuses these objects to establish the result of new constraints. This model reuse approach is re-implemented in Green and investigated further with an extensive evaluation against various Green configurations as well as incremental sat solving. The core findings highlight that the factorisation step is the crux of the different caching strategies. The results shed new light on the true benefits and weaknesses of the respective approaches.

Uittreksel

Optimiseerde Beperking-Oplos vir Regte Wêreld Probleme

J.H. Taljaard

Departement van Wiskundige Wetenskappe, Divisie van Rekenaar Wetenskap
Universiteit van Stellenbosch
Privaatsak X1, Matieland 7602 , Suid Afrika.

Tesis: M.Sc (Rekenaar Wetetenskap)

Desember 2019

Alhoewel daar die afgelope dekade aansienlike vordering met beperking-oplos tegnologieë gemaak is, is Bevredigbare Modulo Teorieë (BMT) oplossers steeds 'n belangrike knelpunt in die verifiëring van programme se eienskappe. Deur die werkverrigting kwessie te oorkom, is verskillende stoorstrategieë ontwikkel vir die hergebruik van beperkinge se oplossings. Een van die eerste algemene raamwerke om sulke stoorwerk te doen, is geïmplementeer in 'n program genaamd Green. Green laat uitgebreide aanpassing toe, maar in sy basiese vorm verdeel dit 'n beperking in sy onafhanklike dele (genaamd faktorisering), voer 'n kanoniseringsstap uit (insluitend die hernoem en herrangskik van veranderlikes) en soek resultate in 'n kasgeheue. Meer onlangs is 'n alternatiewe benadering voorgestel: waar in plaas van *bevredigend* of *onbevredigend* waardes in 'n kasgeheue op te soek, dit *modelle* (in die bevredigende geval) en *onbevredigende kerns* (in die onbevredigende geval) stoor, word hierdie voorwerpe hergebruik as die resultaat van nuwe beperkinge. Hierdie nuwe modelhergebruik-benadering word geïmplementeer in Green en word verder ondersoek met 'n uitgebreide evaluering teen verskillende Green-konfigurasies sowel as inkrementele bevredigbare-oplossing. Die kernbevindinge beklemtoon dat die faktoriseringstap die kern van die verskillende stoorstrategieë is. Die resultate werp nuwe lig op die werklike voordele en swakhede van die onderskeie benaderings.

Acknowledgements

For in him all things were created: things in heaven and on earth, visible and invisible, whether thrones or powers or rulers or authorities; all things have been created through him and for him. He is before all things, and in him all things hold together.
– Colossians 1:16-17

Thanks be to the **Lord** for this opportunity to gain knowledge, hone skills and labour alongside some of the most astute professors and researchers the University of Stellenbosch has to offer. Thank you for Your patience, guidance, peace and faithfulness, supplying the means and funds to pursue this research, for having brought this work to completion and all the help in doing so.

I would like to express my sincere gratitude to my supervisors **Prof. Willem Visser** and **Prof. Jaco Geldenhuys** for their guidance, inspiration and close collaboration from my Honours and throughout my Masters. Your lessons will stick with me through the years to come. Prof. Visser, thank you for teaching me problem breakdown into simpler components yet keeping the big picture in mind. Prof. Geldenhuys, thank you for the time you took to sit with me and carefully explain concepts and to help me talk through the problem.

I am grateful for **Dr. Tarl Berry** for his suggestions and crucial advice during the time of research. Thank you to the **Computer Science division** for the space where I could peacefully work and complete this research and thesis. I would also like to thank **Andrew J. Collett**, for his insight and help with the machine for experiments and the various problems that popped up moving to a different system. Thank you for setting up the machine on which I worked to finish this thesis. **Shane Josias** for his suggestions and availability as a sound board during the time of research. **Francois du Toit** for his availability as a sound board as well and assistance in proof reading. I would like to express my gratitude to **my family**, especially my parents – their love and support were essential for the completion of the thesis work.

Finally, my appreciation to **NRF** for funding my research and the assistance of **Bernd Fischer** to obtain research funds – without him the research would not have been possible either.

Contents

List of Figures

List of Tables

Acronyms

CNF	Conjunctive Normal Form
CS	Conditional Statement
JPF	Java PathFinder
SAT	Satisfiable (or feasible)
SMT	Satisfiability Modulo Theories
SPF	Symbolic PathFinder
UNSAT	Unsatisfiable (or infeasible)

Nomenclature

Canonisation represents each individual constraint into normal form.

Conditional Statement is a decision point in a program, which make up part of the execution paths of a program.

Factorisation splits a constraint into its independent factors (or sub-constraints).

Green is an SMT solver caching solution developed by Prof. W. Visser and Prof. J. Geldenhuys.

Grulia is a (Julia type) service within the Green framework.

Julia is a general purpose caching framework for formulas from an SMT solver, developed by Dr. A. Aquino and Prof. M. Pezzè.

Propositional Formula (in propositional logic) is a type of syntactic formula which is well formed and has a truth value.

SAT solver determines the satisfiability of formulas generated during the analysis of a program.

Satisfiability Modulo Theories encompass a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. An example is linearisation.

Symbolic execution means to use symbolic values, instead of actual data, as input values to determine what inputs cause each part of a program to execute, as stated by King (1976).

Chapter 1

Introduction

1.1 Problem Statement

Many program verification techniques produce propositional logic formulas that include linear integer arithmetic. Questions like whether a given formula is satisfiable, what variable assignments (= *models*) satisfy it, and how many such models exist (defined by Morgado *et al.* (2006)), are typically generated. Many symbolic and concolic program analysis techniques use Satisfiability Modulo Theories (SMT) solvers to verify properties of programs. In recent years, the performance of SMT solvers have improved dramatically, but even more advances are needed to handle ever-increasing targets. Symbolic and concolic execution are two examples of popular SMT-based program analysis techniques that have gained popularity for generating high-coverage tests, checking feasible execution paths, and detecting subtle errors in programs. Although SMT solvers are powerful, very large inputs still require long running times.

One way of tackling scalability is *memoisation*. SMT solvers can provide solutions more quickly if they cache their results. The logic behind memoisation is simple: expensive solver invocations can potentially be avoided, as long as the overhead of storing and retrieving results to and from a cache is low enough.

To overcome the performance issue of SMT solvers, different caching strategies have been developed for constraint solution reuse. One of the first general frameworks for doing such caching was implemented in a tool called Green, envisioned and developed by Visser *et al.* (2012). Green allows extensive customisation, but in its basic form it splits a formula to be checked into its independent parts (called factorisation), performs a canonisation step (includ-

ing renaming and reordering of variables) and looks up results in a cache. More recently an alternative approach was suggested by Aquino *et al.* (2017) (and improved in Aquino *et al.* (2019)): rather than looking up *sat*/*unsat* results in a cache, it stores models (in the sat case) and unsatisfiable cores (in the unsat case), and reuses these objects to establish the result of new queries. This approach will be referred to as Julia (in reference to the latest version).

This thesis evaluates various approaches for caching during satisfiability checking. Firstly the exact analyses as published previously by running the Julia tool on the original benchmarks are repeated (the replication introduces a more recent version of Julia into the comparison). Lastly, Julia is re-implemented within the Green framework (calling it Grulia), and all three tools are compared against a current version of Z3 (an SMT solver) for doing satisfiability checking. The results shed new light on the true benefits and weaknesses of the two respective approaches for memoisation (reusing models and unsatisfiable cores versus reusing satisfiability results).

1.2 Thesis Goals

The thesis will explore the following research questions:

1. Which of the popular caching frameworks seem best suited for analysis of programs during symbolic/concolic execution?

2. What is the relevancy of caching frameworks like Green or Julia with the increase of solver performance?

3. What is the impact of pre-processing, or specifically factorisation (where constraints are split into independent parts), of constraints on solving and solution caching?

4. What difference emerges between caching for symbolic and concolic analyses?

1.3 Thesis Structure

Chapter 2 provides a detailed background on the main technologies and explains the different frameworks (Green and Julia) involved in this optimisation approach. Furthermore the chapter takes a look at other solution caching techniques.

Chapter 3 describes the implementation of the Grulia caching service in Green, along with that of the factorisation service.

Chapter 4 presents the evaluation and results achieved by the new services.

Chapter 5 concludes this paper and highlights a few observations from this work.

Chapter 2

Background

This chapter provides background information on constraint solution reuse, symbolic execution, the tools involved in this study and other key concepts. Section 2.1 gives further background information to understand Symbolic PathFinder (SPF), followed by Section 2.2 which provides minimal yet necessary information about concolic execution. Section 2.3 discusses the tools involved in this study, followed by a section with a view on the other comparable tools and strategies. The chapter concludes with Section 2.5 as a summary.

2.1 Symbolic Execution

King (1976) was one of the first to propose the use of symbolic execution for test generation. The basic approach involves executing a program with symbolic inputs rather than concrete inputs. Path conditions that describe the constraints on the inputs under which a specific path can be executed are collected from branching conditions during symbolic execution. In addition, whenever a constraint is added to the path condition, the resulting constraint is checked for feasibility. If it is not feasible, the path is terminated and not analysed further. The feasibility check is performed by external constraint solvers.

One can think of the analysis performed during symbolic execution as searching for feasible execution paths in a tree (sometimes referred to as an execution tree) where edges represent path conditions. At any point during this search the current path condition must be feasible, and a solution to the path condition will represent inputs that when used during execution will reach this location in the code. For example, if a location in the analysis is reached

```
1  public boolean foo(int i, int j) {
2      if (i > 5) {
3          if (j > 5)
4              i += 5;
5      } else {
6          if (j < 5)
7              if (j >= 5)
8                  i -= 5;
9      }
10     if (i == 0)
11         return true;
12     else
13         return false;
14 }
```

Figure 2.1: Simple Java program example.

where an assertion is violated the solution to the path condition will produce inputs that can be used to execute the program to show the violation.

The fundamental problem with symbolic execution is that the execution tree can become very large, in fact, infinitely large. Searching through this space is typically limited by using a depth limit that indicates how deep the analysis may go. Note of course that it is possible to miss errors, if the depth limit is too shallow to reach the error. It is definitely desirable to perform the analysis as fast as possible and it is well known that one of the main inefficiencies during symbolic execution is the time spent doing the feasibility check.

In practice, a symbolic execution involves replacing concrete inputs with corresponding symbolic values, tracking the flow of these symbolic inputs through the execution, and the extraction of conditional statements to build (feasible) path conditions. A program like the code fragment in Figure 2.1[1], operates on concrete input such as `i`$= 2$ and `j`$= 7$ or other valid integers. Symbolic execution transforms the inputs such that it can work with arbitrary constants, which represents fixed unknown values (call them symbolic variables). For example the symbolic variables I and J (not mentioned elsewhere in the program) are used instead of the concrete values of `i` and `j`. Typically the symbolic variables are bounded, but research such as that of Jaffar *et al.* (2012), have been done to handle unbounded variables[2]. To prevent the text from becoming too cluttered, the bounds are not explicitly written in the examples in the section, but are still mentioned for clarity.

A conditional statement (CS) whose variables have been changed to sym-

[1]Most of the braces are absent to shorten the code example.

[2]The constraints encountered and analysed in this thesis' experiments are all bounded.

bolic values is referred to as a constraint. The transformed constraint is in the form of first order logic, making it possible for a Satisfiability Modulo Theories (SMT) solver to evaluate it. The target constraint ϕ for the feasibility check is obtained from a transformation of some conditional statement CS_1 to a constraint ϕ_1, which forms as a clause in the larger constraint ϕ. The SMT solver will evaluate each constraint and assert if a constraint is satisfiable (feasible) or unsatisfiable (infeasible)[3]. A constraint is typically made up of all the previous constraints in the path leading up to the target constraint. Meaning that within a nested CS (such as present in Figure 2.1) the constraint is not made up of only the inner CS, but also captures the outer CS (and the preceding path). Therefore construction of a constraint is the transformation of some CS_2 to the constraint ϕ_2, and conjoined with the previous constraint(s) along the path, such that $(\phi : [\phi_1 \wedge \phi_2])$. For example the CS in line 2 and line 3 in Figure 2.1 becomes $I > 5$ and $J > 5$, respectively, and the two constraints make up the constraint $\phi : [(I > 5) \wedge (J > 5)]$ to reach line 4.

Two figures will suffice as an illustration to assist in a clearer understanding of how a symbolic execution analysis executes on a program. Figure 2.1 is the source code of a simple program, and Figure 2.2 represents the symbolic execution tree of the code. As the target program gets executed, the analysis (depth-first search in this case) takes place, recording the necessary data. Each CS in the program is represented as a node in the tree that indicates which line of code is encountered given the corresponding path condition. The edges follow the program flow during the analysis. The path represents the resulting constraint following the program flow during the analysis. The line under the stated constraint in the node represents the line that produces the given constraint. The shaded node at the end of the path represents the final outcome of that path. Given the input variables `i` and `j`, consider the corresponding symbolic values of I and J, both constrained to the range of $[-10, 10]$.

The program starts with the method call and moves on to the first branching point at line 2, with the analysis recording the CS and generating the equivalent constraint $\phi_1 : [I > 5]$. A solver call is made to evaluate the constraint. Upon proving the satisfiability of the constraint, the program continues to line 3. The constraint derived from it, is the CS itself, translated to $[J > 5]$, and the previous state $[I > 5]$ resulting in the final constraint that is $\phi_2 : [(I > 5) \wedge (J > 5)]$. Another solver call is made, asserting that the constraint is satisfiable and the program flow moves to line 4 and then to line 10 where another condition is encountered. The added condition checks if $[I = 0]$ which is added to the constraint, but with the execution of line 4 there is another condition placed on I as well, such that the constraint $\phi_3 : [(I > 5) \wedge (J > 5) \wedge (I + 5 = 0)]$ is obtained, and is asserted as unsatisfiable. The other branch gives the constraint $\phi_4 : [(I > 5) \wedge (J > 5) \wedge (I + 5 \neq 0)]$

[3]Another possibility is to calculate the number of satisfying values (or the model count) of the constraint.

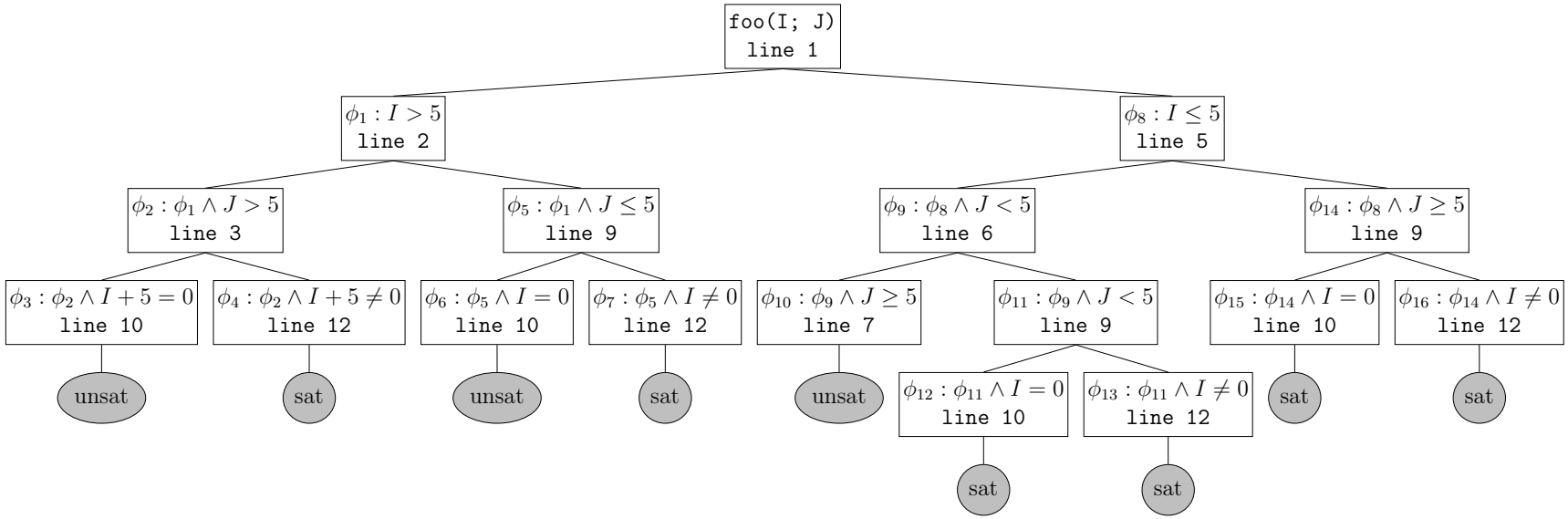


Figure 2.2: Symbolic execution tree of the sample program.

and is evaluated to be satisfiable. The program flow continues to line 13 and returns to the method call. The end of this path has been reached, ending the analysis thereof and backtracking to the previous state.

The analysis negates the last clause, resulting in the constraint $\neg[J > 5]$ which can be simplified to $[J \leq 5]$. The final constraint is achieved by adding this state to the previous state, which produces $\phi_5 : [(I > 5) \wedge (J \leq 5)]$. The constraint is evaluated with another solver call, determining the satisfiability. The constraint is satisfiable, which allows the program to move to line 10, which repeats the branching point of $[I = 0]$. The constraint $\phi_6 : [(I > 5) \wedge (J \leq 5) \wedge (I = 0)]$ is unsatisfiable, and the other branch with the constraint $\phi_7 : [(I > 5) \wedge (J \leq 5) \wedge (I \neq 0)]$ is asserted as satisfiable. The program continues to line 13 and returns to the method call, which results in the end of this path's analysis. This also concludes the analysis of the left side of the tree.

The analysis backtracks to a previous unsolved state, which is the else of the condition of line 2. Again the negation of the condition is taken, resulting in $\neg[I > 5]$ as the constraint, which is simplified to $\phi_8 : [I \leq 5]$. The constraint is evaluated by the solver, proving that it is satisfiable. The program flow continues to line 6, encountering a new CS and translating it and adding it to the previous state, which results in $\phi_9 : [(I \leq 5) \wedge (J < 5)]$. The satisfiability is proved and the program flow proceeds to line 7. The new CS results in the constraint $\phi_{10} : [(I \leq 5) \wedge (J < 5) \wedge (J \geq 5)]$. The constraint contains a contradiction and is proved as unsatisfiable and therefore the path is unsatisfiable. Thus line 8 will never be executed. The new constraint to be evaluated follows the same procedure as before, giving the constraint $\phi_{11} : [(I \leq 5) \wedge (J < 5) \wedge (J < 5)]$[4]. The constraint is asserted as satisfiable, and the program flow moves to line 10. Again asserting the constraint of $\phi_{12} : [(I \leq 5) \wedge (J < 5) \wedge (J < 5) \wedge (I = 5)]$ as satisfiable and the program continues to line 11 and returns to the method call. The other branch produces the constraint $\phi_{13} : [(I \leq 5) \wedge (J < 5) \wedge (J < 5) \wedge (I \neq 5)]$ which is evaluated as satisfiable. The program continues to line 13 and returns to the method call. Thus concluding the analysis of this path and branch.

The analysis backtracks to a previous unsolved state, which produces the constraint $\phi_{14} : [(I \leq 5) \wedge (J \geq 5)]$. The solver call proves its satisfiability, allowing the program flow to line 10 of the program. The left branch represented by the constraint $\phi_{15} : [(I \leq 5) \wedge (J \geq 5) \wedge (I = 0)]$ is satisfiable and results in the program reaching line 11 to return to the method call. The right branch produces the constraint $\phi_{16} : [(I \leq 5) \wedge (J \geq 5) \wedge (I \neq 0)]$ which is evaluated as satisfiable and allows the program to move to line 13 and returns to the method call. The analysis backtracks, finding there are no more unsolved

[4] Note that this constraint can be further simplified by removing the redundant clause, with further pre-processing of the constraint as an intermediate step, to produce the constraint $[(I \leq 5) \wedge (J < 5)]$, which is argued to make it easier for the solver to evaluate.

states and therefore concludes the analysis of the program.

The symbolic execution tree displays the program flow, for example if the input ranges from 6 to 10 (with the first constraint) the true case of the CS is satisfied. If the input is less than or equal to 5, it satisfies the false case of the CS. Note that for the execution tree a range is specified for the input values to determine possible solutions to satisfy the constraint. In practice during symbolic execution (for satisfiability checking) the solver will return only a single value (that exists in that range of possible solutions), i.e., $i = 6$ (true case) or $i = 5$ (false case), and not the range itself.

Programs can be analysed with symbolic input or could be done by tracking how concrete inputs are used to execute code and perform a symbolic analysis on the side. With symbolic input, more constraints are obtained since more states are generated, whereas with concrete input a single program flow is followed.

Some popular symbolic execution tools such as KLEE[5], SPF, Crest[6], JBSE[7] (developed by Braione *et al.* (2016)), jCute[8], CuteR[9] and Pex (designed by Tillmann and de Halleux (2008)) allow for a variety of uses such as automatic test generation and bug finding.

One of the added bonuses of symbolic execution is combating accidental correctness[10] in a program, since all the input parameters are tested. This allows for testing at the boundary cases, as path execution is done in a more general sense than a single case of actual data would. With a single concrete input only one path might be explored like in Figure 2.3, whereas symbolic execution will explore all of the possible paths (thus testing the boundary cases as well).

Symbolic PathFinder

Symbolic PathFinder (SPF)[11] is a symbolic execution tool for Java programs. SPF extends the Java PathFinder (JPF)[12] (developed by NASA[13]) analysis engine to allow symbolic execution. SPF combines the source code analysis

[5] `https://klee.github.io`

[6] `http://www.burn.im/crest`

[7] `https://github.com/pietrobraione/jbse`

[8] `http://osl.cs.illinois.edu/software/jcute`

[9] `https://github.com/cuter-testing/cuter`

[10] Accidental correctness refers to the case where it seems like the program is functioning in the correct manner by using flawed logic or introducing accidental errors. An example would be a simple function of adding two values written as (`a + b`) but the actual code is implemented as (`a * b`). Testing this program with input values `a` $= 2$ and `b` $= 2$ gives the correct answer of 4. If this program is not further tested, one would assume the program is correct.

[11] `https://github.com/SymbolicPathFinder/jpf-symbc`

[12] `https://github.com/javapathfinder/jpf-core`

[13] `https://ti.arc.nasa.gov/tech/rse/vandv/jpf`

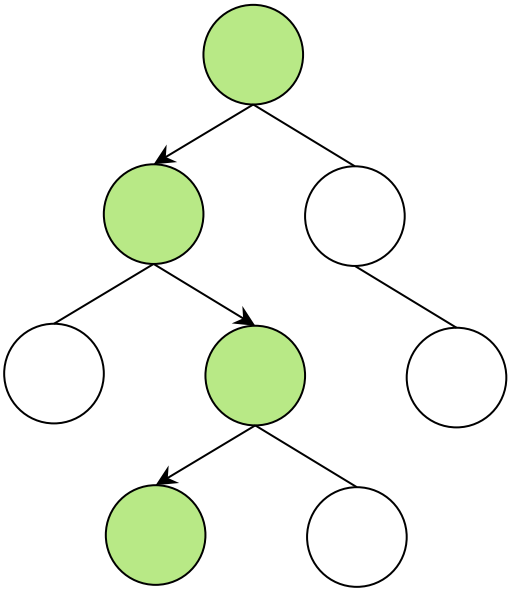


Figure 2.3: State space with single path execution vs. full coverage.

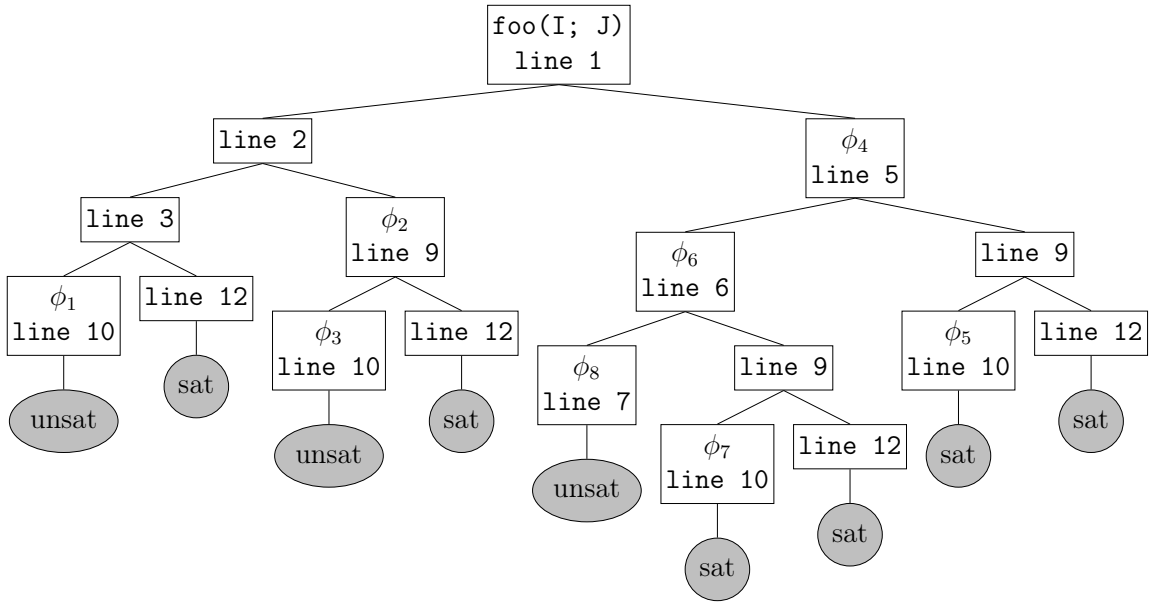
with constraint solving to generate test cases for programs. The tool can use various back-end solvers for constraint solving. Part of the experiments are performed by attaching the Green framework as the back-end solver, to test improvement of the analysis running time. The interested reader can find a detailed description of how SPF operates in the paper of Păsăreanu *et al.* (2013).

2.2 Concolic Execution

Concolic is a portmanteau of two words: *conc*rete and symb*olic*. Concolic execution is broadly similar to symbolic execution, except for a few key differences.

During concolic execution the program is executed with concrete inputs, but the analysis keeps track of the corresponding symbolic constraints or conditional statements along the concrete path that is executed. When the end of a path is reached (some paths are still unexplored as shown in Figure 2.3), the path condition for this executed path is then manipulated to generate new concrete inputs to explore a different path. This manipulation is typically to negate the last constraint obtained to mimic a depth-first traversal of the symbolic execution tree of the program. Concolic execution does not make a solver call for each encountered edge of the execution tree, although each edge traversed along a path is evaluated given the concrete values. Concolic execution typically starts with a single run of the program with the user specified (or predefined) values of the variables.

Two figures will suffice as an illustration to assist in a clearer understanding



$\phi_1 : [(I > 5) \wedge (J > 5) \wedge (I + 5 = 0)]$
$\phi_2 : [(I > 5) \wedge (J \leq 5)]$
$\phi_3 : [(I > 5) \wedge (J \leq 5) \wedge (I = 0)]$
$\phi_4 : [I \leq 5]$
$\phi_5 : [(I \leq 5) \wedge (J \geq 5) \wedge (I = 0)]$
$\phi_6 : [(I \leq 5) \wedge (J < 5)]$
$\phi_7 : [(I \leq 5) \wedge (J < 5) \wedge (J < 5) \wedge (I = 0)]$
$\phi_8 : [(I \leq 5) \wedge (J \geq 5) \wedge (J < 5)]$

Figure 2.4: Concolic execution tree of the sample program.

of how a concolic execution analysis executes on a program. Figure 2.1 is the source code of a simple program, and Figure 2.4 represents the execution tree of the code. As the target program gets executed, the analysis (in a depth-first fashion in this case) takes place, recording the necessary data. Each CS in the program is captured in the tree with a node that indicates which line of code is encountered given the corresponding path condition. The edges follow the program flow during the analysis. Given the input variables `i` and `j`, consider the corresponding symbolic variables, I and J, both constrained to the range of $[-10, 10]$. In the execution tree each ϕ indicates a solver call that has been invoked.

The program starts with the method entry point at line 1 and moves to line 2, given the input values $I = 6$ and $J = 6$, the analysis records the CS and the equivalent constraint obtained is $[I > 5]$. The program executes the CS with the input values and finds the condition `true`, moving the program flow onto line 3. The new CS and the constraint (adding the previous condition to the current) $[(I > 5) \wedge (J > 5)]$ are recorded. The program evaluates the CS as `true` and the flow continues to line 4 placing another condition on the constraint and the flow continues to line 12. The constraint $[(I > 5) \wedge (J > 5) \wedge (I + 5 \neq 0)]$ is evaluated as satisfiable and returns to the method call, concluding this path. The analysis goes back to the previous clause that is not negated, and negates it, resulting in a solver call to check the satisfiability of $\phi_1 : [(I > 5) \wedge (J > 5) \wedge (I + 5 = 0)]$, which is unsatisfiable. Note that this is the first time a solver call has been made. The run of this path is ended and the analysis picks the previous constraint not yet negated and negates the clause, which is the else of the CS at line 3, which results in the new constraint $\phi_2 : [(I > 5) \wedge (J \leq 5)]$. A solver call is made to test satisfiability of the constraint and to obtain satisfying values (say $I = 6$ and $J = -10$). A new program run is performed with the new input values, whereby the program flow moves from line 2 to the else condition of line 3 and then to line 12. The evaluation of the constraint finds it to be satisfiable and returns to the method call.

The analysis negates the last non-negated condition, calling the solver with the constraint $\phi_3 : [(I > 5) \wedge (J \leq 5) \wedge (I = 0)]$ which is unsatisfiable and concludes the analysis of the left side of the execution tree. The analysis picks the last condition not yet negated and negates that, which is the CS at line 3, resulting in the constraint $\phi_4 : [I \leq 5]$. A solver call is made to evaluate the satisfiability of this branch which leads to line 13 and the method returns. Taking the negation of the previous constraint, the result is $\phi_5 : [(I \leq 5) \wedge (J \geq 5) \wedge (I = 0)]$ with a solver call giving the answer as satisfiable, and generates the new input of $I = 0$ and $J = 5$. The program flow continues to line 11 and the method returns.

With the negation of the previous non-negated condition, the constraint $\phi_6 : [(I \leq 5) \wedge (J < 5)]$ is obtained, where the solver call gives the solution as satisfiable and the new inputs as $I = 0$ and $J = -10$. The program flows

proceeds to line 13 whereupon returning to the method call. The analysis again negates the last condition which gives the constraint $\phi_7 : [(I \leq 5) \wedge (J < 5) \wedge (J < 5) \wedge (I = 0)]$ which is asserted as satisfiable with a solver invocation. The program is executed with the previously stated input values, and the program flows through to line 11 finding no new paths and returns to the method call.

The last non-negated condition (line 7) is negated, resulting in the constraint $\phi_8 : [(I \leq 5) \wedge (J \geq 5) \wedge (J < 5)]$, which contains a contradiction. Therefore ϕ_8 is unsatisfiable. No unexplored or non-negated constraints are present and therefore the analysis terminates.

Coastal

Coastal[14] is a concolic execution tool for Java programs, which is chosen for this thesis since it operates on Java programs as well. Having both Coastal and SPF operating on Java programs a comparison can be performed on the effect of caching in both settings. Coastal instruments the byte code to analyse the source code of a program in question. The execution paths are traced and explored with a specified strategy, which can be one of the options provided by the user. For the comparison in the thesis, the depth-first strategy is employed. Similar to SPF, Coastal can attach various back-end solvers for constraint solving. Part of the experiments are performed where the Green framework is also attached to Coastal to test improvement in the analysis running time.

2.3 SMT solving

Many symbolic program analysis techniques use Satisfiability Modulo Theories (SMT) solvers to verify properties of programs. This section describes one SMT solver named Z3, as well as describing two existing frameworks (Green and Julia) that provide caching layers before invoking an SMT solver.

Z3

One of the best known (and NP-complete) problems in mathematics and computer science is THREE-SAT. The SAT problem is common in many applications. Much research have been devoted to efficiently translate various problems into SAT problems, which can then be evaluated by SAT solvers.

One of the earliest approaches to solving SAT problems (and theorem proving) was done by Davis and Putnam (1960) and Davis *et al.* (1962). The algorithm from their work is referred to as DPLL (the authors – Davis, Putnam,

[14] `https://github.com/DeepseaPlatform/coastal`

Logemann and Loveland). It is essentially a backtracking algorithm that explores all possible variable assignments. DPLL was further improved by Tinelli (2002) and Ganzinger *et al.* (2004) and still forms the basis of many successful modern solvers.

Further research spent on SAT solvers, for example such as done by Eén and Sörensson (2004) performed their study on simplifying the understanding and creation of SAT solvers. They have presented their work with their proof of concept SAT solver. The design and creation of a robust SAT or SMT solver is a difficult and time consuming endeavour. SMT solvers are not more powerful than SAT solvers, but encapsulate SAT solving, taking more knowledge into consideration while evaluating the given problem. As such, SMT solvers can tackle more complex theories including the theory of reals (among many other theories) and quantified[15] constraints.

One of the most popular SMT solvers is Microsoft's Z3[16] (simply referred to as Z3), and with its continued growth in popularity and robustness the solver is considered for this study's comparison. Z3 was designed and released by Microsoft in 2007, and they are at the time of writing still actively updating and improving the solver. It is a complex program, using some of the latest research to develop its solving strategies[17].

For solving constraints, there are different configurations in Z3. One of Z3's features is its incremental solving mode, which can operate in two fashions: stack-based and assumption-based. Stack based solving, as implied with the data structure, functions by means of push and pop commands. The idea is to start with a known state, adding a new assertion to it, and then re-evaluating the state. To demonstrate this with an example, say there is a constraint $\phi : [\phi_1 \wedge \phi_2 \wedge \phi_3]$. With incremental mode, the first clause ϕ_1 is asserted. Z3 stores the state internally. With ϕ_2 pushed onto the stack, the assertion is added to the previous one and the state is evaluated. The same is repeated for ϕ_3, with the final state returned containing the solution. Solving constraints in this manner is arguably faster.

Green

Green[18], designed and created by Visser *et al.* (2012), is a framework which among many features, allows the user to use the framework for constraint solving purposes. Green is an active open source project that gets improved upon by various different contributors.

Green is fundamentally a caching layer that aims to improve the performance for various kinds of constraint analyses and is typically used during

[15]Referring to Quantification Logic.

[16]`https://github.com/Z3Prover/z3`

[17]See `https://github.com/Z3Prover/z3/wiki/Publications` for their latest research contributions.

[18]`https://github.com/GreenSolver/green`

symbolic execution. Most of its features are specifically designed for constraints in Conjunctive Normal Form (CNF) and containing only linear integer arithmetic. In addition to its role as a caching layer, Green also serves as an interface to various back-end solvers, for example SMT solvers such as Z3, or model counters such as Barvinok[19]. Z3 is an external library accessed directly from Java through the command line, or through an interface with Java bindings. In this thesis the focus is placed on Green's use as a front-end to Z3 and the interest lies in the amount of reuse that it is possible to obtain from caching sat/unsat results, and whether or not this saves any time over calling Z3 directly. One of Green's most useful features is that it caches results across various external analyses. For example doing symbolic execution of one program could lead to constraint solving results that are reused in the analysis of another program.

Green uses a pipeline architecture where each service in a pipeline transforms the input and passes it to the next service; the last step is a service that invokes Z3. However, right before passing a constraint to Z3, this service checks a cache and passes the result (cached or computed) back up the pipeline to the caller. This architecture makes it easy to extend a service by introducing or altering the steps in its pipeline. For example, in the rest of this work the final step (which invokes Z3), will be replaced with a new step based on model-reuse (see Section 2.3 that expounds on this).

A typical pipeline for checking satisfiability consists of the following services (as shown in an abstract view in Figure 2.5[20]):

Factorise: This first step splits the input constraint into a number of independent factors (sub-constraints). Two clauses in a constraint are independent if none of the variables in one clause can affect the solution in the other clause. Since the input constraint is in CNF, each of the factors must be satisfiable for the input constraint to be satisfiable. For example $\phi : [(a > 5) \wedge (b < 7)]$ would become $\phi_1 : [a > 5]$ and $\phi_2 : [b < 7]$.

Canonise: After the input is split into independent factors, a constraint is converted to a canonical form (see Visser *et al.* (2012) for details). Part of this step is to rename the variables according to the lexicographic order they appear in the constraint[21]. Further transformation is done such that all variables and constants only appear on the left side of the equation. Furthermore the equation is multiplied by -1 to change the operator from $>$ to $<$ or from $\geq$ to $\leq$. Another step, only included if the operator is $<$, involves adding 1 on the left side of the equation to transform the operator to $\leq$. Finally all of the transformed clauses are

[19] `http://barvinok.gforge.inria.fr`

[20] The image is adapted from Figure 1 in Visser *et al.* (2012).

[21] Note that this renaming service is later separately used for pre-processing.

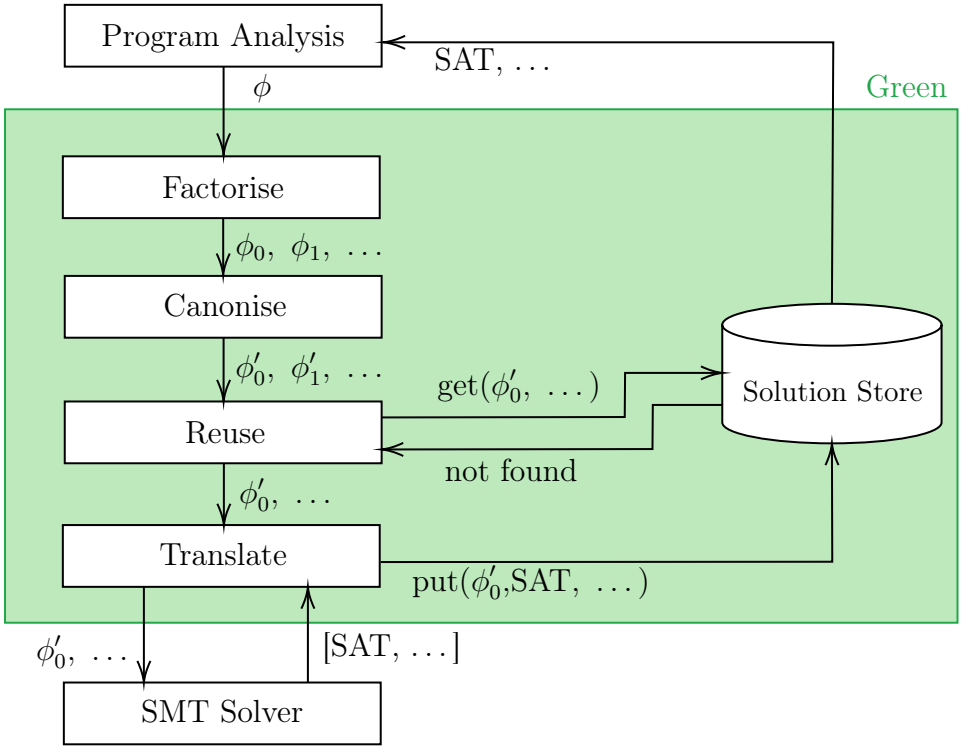


Figure 2.5: Program analysis with basic Green pipeline as caching layer.

aggregated again in CNF. For example $\phi : [(a > 5) \wedge (b < 7)]$ would become $\phi_1 : [(-v_0 + 6 \leq 0) \wedge (v_1 - 6 \leq 0)]$.

Z3Service: The last service in the pipeline (SMT solver) uses Z3 to check for satisfiability, if the result is not already cached. A key-value store (the `Solution Store` in Figure 2.5) called Redis[22] is used. To cache these results the following is done: the key is taken as the constraint and the value as a boolean value representing the satisfiability result returned by Z3.

Julia

An intricate, though novel, approach to optimise SMT solution caching was initially proposed by Aquino *et al.* (2017). Their approach reuses models (which are variable assignments for satisfiable constraints) and unsatisfiable cores (explained later in the section) of already-solved constraints to find solutions for incoming constraints. The first prototype is implemented in a C++ tool called Utopia, but since the first publication they have also added an improved Java version, called Julia presented by Aquino *et al.* (2019). Both Utopia[23] and Ju-

[22] `http://redis.io`

[23] `https://bitbucket.org/andryak/utopia_qflia/src/master`, although this repository is no longer available at the time of writing.

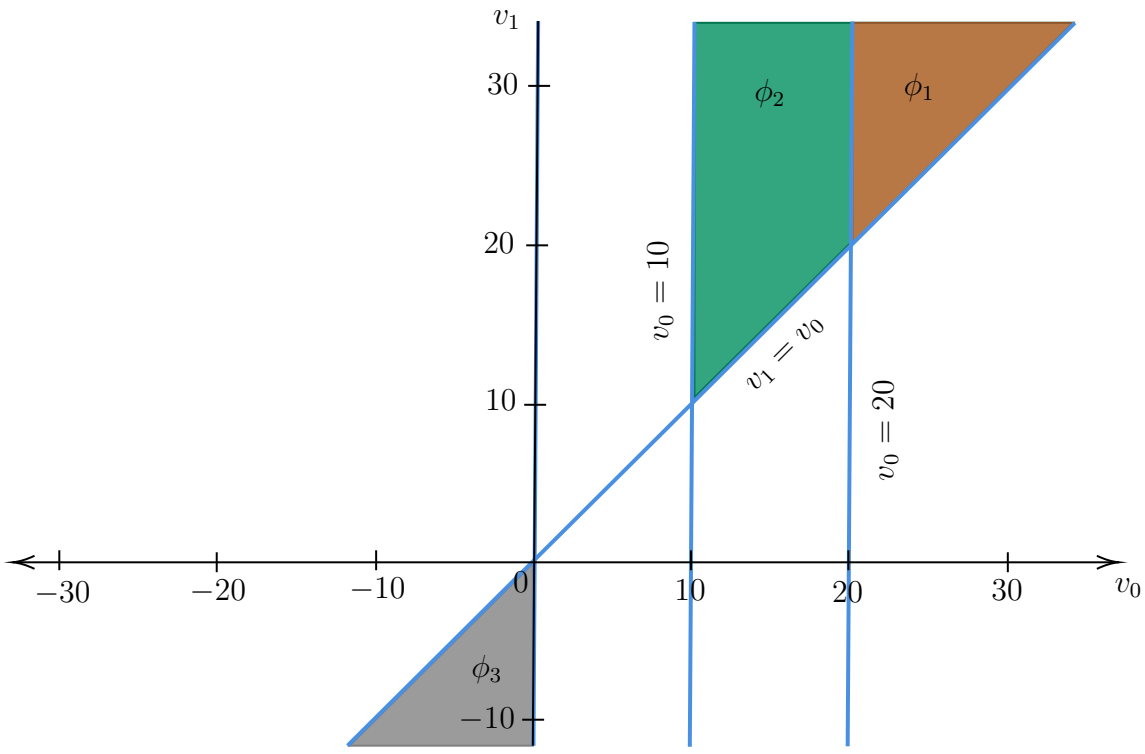


Figure 2.6: Intuitive 2D solution space analogy.

lia[24] have open-source repositories on Bitbucket which were used to replicate their benchmarks and study the implementations. Specifically the benchmarks presented in the paper of Aquino *et al.* (2017) are replicated since those results were more detailed for comparison. In this thesis the focus is mostly on the Julia implementation and the thesis will refer to this tool throughout the document.

The fundamental idea is to not reuse sat/unsat results, but rather to reuse previous solutions (models and *unsat-cores*) instead. It therefore exploits the behavioural similarity of constraints with regard to solutions. In other words, the same solution may satisfy two different constraints. For example in c_1 : $[(v > 10) \wedge (v \leq 20)]$ and c_2 : $[(v > 10) \wedge (v < 30)]$, the model $v = 20$ is satisfiable for both c_1 and c_2. This might not seem immediately obvious as a good idea: how could one expect that a model for one constraint to also be a model for another? The trick that makes this work is to have a fast hash function that links the constraints that have a high likelihood of having the same solution space. In Green terminology one can think of this as replacing the canonisation step with a fast approximation. In the Julia approach this fast approximation is called the *sat-delta* calculation (explained in the next section).

[24] `https://bitbucket.org/andryak/julia/src/master`

What Julia attempts with the *sat-delta* calculation, is a way to quickly determine a relation between the solution spaces of two constraints or, in other words, to match the solution spaces of constraints instead of their structural similarity. For an intuitive example, take a look at Figure 2.6, where the solution space of a given constraint $\phi_1 : [(v_0 > 20) \wedge (v_1 > v_0)]$ is represented by the brown coloured area. Given another constraint $\phi_2 : [(v_0 > 10) \wedge (v_1 > v_0)]$, its solution space is contained in the teal coloured area which is merged with the solution space of ϕ_1. A third constraint is presented as $\phi_3 : [(v_0 < 0) \wedge (v_1 < v_0)]$ with the solution space captured in the gray area. The idea is that ϕ_1 and ϕ_2 would match closer to one another (having scores with a small difference), because their solution spaces are closely situated. The fast *sat-delta* calculation would calculate a score for ϕ_3 that is greater in difference compared to that of ϕ_1 or ϕ_2, since its solution space is quite far from them. The satisfiability of ϕ_2 can be tested with the satisfiable model of ϕ_1, instead of the model of ϕ_3.

SAT-Delta

The *sat-delta* calculation provides a score for a constraint, with respect to a solution space. This value is used for the look-up in the cache and the latter is kept sorted with respect to these values. The *sat-delta* calculation computes the "distance" of a constraint with respect to one or more reference models[25] in the solution space. If that distance is zero, it means that one of the reference models satisfy the constraint, otherwise it is a positive number in relation to the distance of the solution space of that constraint. It is not important whether or not the reference models satisfy the constraint; the distance metric is more nuanced. The argument of Julia is that identifying the constraints based on a common set of reference models increases the chance of assigning similar scores to constraints that share some models.

An example illustrated in Figure 2.7 with a rule plot to visualise the score in relation to the reference model. For some input constraint ψ_1, and given reference model M_{ref}, the score (*sat-delta*) is computed and indicated with the symbol $\xrightarrow{s\delta}$. The evaluation of ψ_1 results in a score of 20. A model that satisfies this constraint is M_{ψ_1}. The same procedure is repeated for ψ_2 and ψ_3, with scores of 50 and 100, respectively. Then there is some ψ_4 evaluated with a score not equal to zero, and close to the *sat-delta* of ψ_1 and the *sat-delta* of ψ_2. Therefore the constraint is evaluated with M_{ψ_1} and M_{ψ_2}, and either or neither can satisfy the constraint. But the argument is that this test is faster and has greater gain, than simply calling the solver. There can also be some ψ_5 that obtains a score of 0, which means that a reference model satisfies this constraint.

Two possible problems arise when too many *sat-delta* values are mapped closely together. Many models could be evaluated before either a satisfiable one

[25]Reference model is a predefined model which captures the variable value assignments.

$\psi_1(M_{\text{ref}}) \xrightarrow{s\delta} 20$
$\psi_2(M_{\text{ref}}) \xrightarrow{s\delta} 50$
$\psi_3(M_{\text{ref}}) \xrightarrow{s\delta} 100$
$\psi_4(M_{\text{ref}}) \xrightarrow{s\delta} 30 \rightarrow M_{\psi_1}|M_{\psi_2}$
$\psi_5(M_{\text{ref}}) \xrightarrow{s\delta} 0$

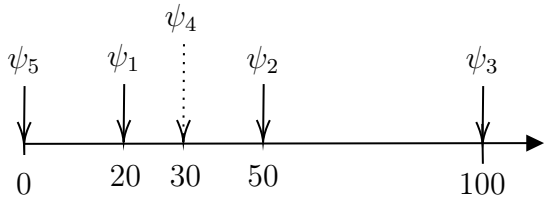


Figure 2.7: Distance approximation with *sat-delta*.

is found or, worse still, when it is determined that there is no such model and that the solver must be invoked to find the solution. The second possibility is that the correct solution might be missed if one selects too few models. Therefore it is crucial to have a good mapping of the distance values to models and by implication avoiding mismatches, which is what *sat-delta* attempts to accomplish.

The *sat-delta* calculation, summarised in equation (2.2), computes a score for each of the clauses in a constraint. Given that the constraint is in Conjunctive Normal Form, the clause scores are summed to produce the constraints *sat-delta* value. The intuition is that constraints with similar solution spaces have similar scores when calculating their distance with some specified reference models. The *sat-delta* for a constraint is computed as

$$\texttt{sat-delta}(\phi, S_m) = \text{average}(\sum_{\substack{C \in \phi \\ M \in S_m}} \texttt{sat-delta}'(C, M)) \tag{2.1}$$

where S_m is the set of reference models, M is a model contained in the set and C is a clause in the given constraint ϕ.

Recall that for the canonisation step, 1 is added to the left side of the equation if the operator is strictly less than, changing it to $\leq$. Earlier it was mentioned that *sat-delta* mimics the canonisation effect. Looking at equation (2.2) (which is adapted from the paper of Aquino *et al.* (2017)), one can see a similarity in calculation. The score for a clause $C = L \odot R$ is computed as

$$\texttt{sat-delta}'(L \odot R, M) = \begin{cases} 0 & \text{if } M_L \odot M_R \\ |M_L - M_R| & \text{if } \odot \in \{\leq, =, \geq\} \\ |M_L - M_R| + 1 & \text{if } \odot \in \{<, \neq, >\} \end{cases} \tag{2.2}$$

where M_X is the value of expression X under the value assignment of model M, and $\odot$ is a placeholder for the possible operations $\{\leq, =, \geq, <, \neq, >\}$. As an illustration, consider the constraint:

$$\phi\colon [(x > 5) \wedge (x = y - 1) \wedge (y \leq 7)]$$

and some arbitrary reference model

$$M : (x = 0, y = 0).$$

For the first clause $[x > 5]$, the resulting calculation is found that

$$\begin{aligned} \texttt{sat-delta}'(x > 5, M) &= |M_x - 5| + 1 \\ &= |0 - 5| + 1 \\ &= 6. \end{aligned}$$

Similarly, $\texttt{sat-delta}'(x = y - 1, M) = 1$ and $\texttt{sat-delta}'(x \leq 7, M) = 0$. Finally, the values are added to produce $\texttt{sat-delta}(\phi, S_m) = 7$. The sum gives an estimate of the distance of the reference model from the constraint's solution space.

When using more than one reference model, the average *sat-delta* value with all the reference models are taken as indicated in equation (2.1). The resulting value provides an approximation of distance with respect to all the reference models, therefore closer approximating the solution space of the constraint. The resulting value is used as index in the cache to find or update the stored sat/unsat answer. The cost of calculating the *sat-delta* value is directly related to the number of given reference models.

The section has discussed the *sat-delta* calculation over the theory of linear integer arithmetic. What makes this technique more useful, is that it can be applied to different theories, such as booleans, strings and others. The other theories are beyond the scope of this thesis, and therefore are left for future work.

UNSAT-Cores

Obtaining the unsatisfiable subset of a constraint to prove unsatisfiability has been around at least circa 1987 (see Reiter (1987)) and improved upon by many. Some of the popular work on proving unsatisfiability and employing unsatisfiable subsets have been done by Gleeson and Ryan (1990), de la Banda *et al.* (2003), Bailey and Stuckey (2005) and Liffiton and Malik (2013). The idea is not novel, but few constraint solution caching frameworks have implemented this technique.

Julia is one of the few caching frameworks that tries to exploit this technique to gain more solution reuse from input constraints. Julia requires an input constraint in CNF, and produces either a satisfying model, or a minimal unsatisfiable subset (or *unsat-core*) that proves unsatisfiablity. For example given the unsatisfiable constraint

$$[(x = y) \wedge (x \neq y) \wedge (x > y)], \tag{2.3}$$

possible *unsat-cores* are $[(x = y) \wedge (x \neq y)]$, $[(x = y) \wedge (x > y)]$, $[(x = y) \wedge (x \neq y) \wedge (x > y)]$. The first two subsets are minimal (in other words, contain

the fewest clauses). The minimal *unsat-core* is required to reduce caching overhead and execution time for unsatisfiable testing of a target constraint. The *unsat-core* provides an advantage over the typical unsat solution[26] that is stored. One such advantage is that less memory is consumed since a smaller solution (less string characters) is stored. Another advantage is the higher probability that an *unsat-core* like the constraint $[(x = y) \wedge (x \neq y)]$ will be present in more constraints, than compared to finding the complete constraint $[(x = y) \wedge (x \neq y) \wedge (x > y)]$ present in other constraints. Within the basic Green pipeline, the constraint (like equation (2.3)) is stored as the key and the value as false, will produce only a cache hit if a constraint with the exact same syntax is queried.

It is easy to obtain the *unsat-cores* with a solver like Z3. One has to enable the correct settings and construct the assertions properly in a certain manner and the solver does the rest behind the scenes. The correct program settings to configure is to enable `produce-unsat-cores` (allowing the solver to track the asserts) and disable `auto-config` (to obtain the minimal *unsat-core*). The next step for the translation to Z3, is to construct each clause as a named assert. Z3 can then identify each clause and return the combination of identifiers which cause the constraint to be unsatisfiable. The caching framework does a reverse mapping based on the identifiers to construct an understandable *unsat-core* whereby the information is ready to be stored for future constraint matching.

The Algorithm

The explanation of Julia's algorithm is done with the assistance of Figure 2.8.

`sat-delta:` The algorithm starts by calculating the *sat-delta* value `sd` of the input constraint with respect to a fixed set of reference models `M` (lines 6–8). The value gives the average distance from satisfiability of the input constraint from the models in `M`.

`SATcache.extract:` Next, a fixed number of `K` models are retrieved from the sat cache (line 10). The value of `K`, just as `M`, is predetermined by the user, and stays constant throughout the computation. The models are selected for their proximity to `sd`.

`satisfies:` If any of the models satisfy the constraint, the algorithm returns true immediately (lines 11–12).

`UNSATcache.extract:` The same procedure is followed for the *unsat-cores* from the unsat cache (in line 14).

`sharesUnsatCore:` If any *unsat-core* is found in `constraint`, the algorithm returns false immediately (lines 15–16).

[26]Typically the unsat solution is stored as a simple `false` boolean value along with the constraint as identifier.

```
1  // M = a set of reference models
2  // K = bound on number of models/cores to extract
3
4  boolean solve(constraint):
5      total = 0
6      for m in M:
7          total += sat-delta(constraint, m)
8      sd = total / |M|
9
10     models = SATcache.extract(sd, K)
11     for m in models:
12         if satisfies(constraint, m): return true
13
14     cores = UNSATcache.extract(sd, K)
15     for c in cores:
16         if sharesUnsatCore(constraint, c): return false
17
18     sat = SMTsolver(constraint)
19     if sat: SATcache.store(sd, constraint.getModel())
20     else: UNSATcache.store(sd, constraint.getCore())
21     return sat
```

Figure 2.8: Summary of the Julia algorithm.

`SMTsolver:` Once the algorithm reaches line 18, the answer has not been found in the caches. An SMT solver is invoked to compute the result, and the answer is cached and returned (lines 19–21).

Julia contains two additional optimisations, the one discussed in the next chapter under Section 3.1 where there is a check that, if the *sat-delta* in line 7 is 0, the method call can return that the constraint is satisfiable (a reference model satisfies the constraint). The other optimisation is a third cache that is consulted before line 9 in case a single cache model satisfies the constraint. All such code have been switched off for this thesis. This is a very good optimisation, since it can further cut out a lot of unnecessary computation, as the exact constraint and solution may be in the cache. It is turned off in the initial study to effectively test the Julia algorithm. Similarly, this kind of cache is disabled for Grulia, for comparison reasons in the replication study and also to effectively test Grulia.

2.4 Other Related Work

Yang *et al.* (2012) have performed initial work on memoised symbolic execution using Tries. Recal is a caching tool constructed by Aquino *et al.* (2015) where a

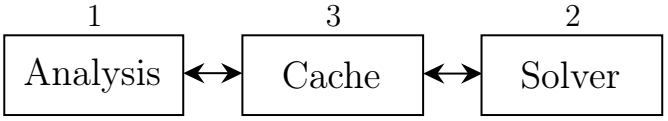


Figure 2.9: Program analysis with constraint solving, enhanced with caching.

target constraint is simplified based on a set of rules and is transformed into a matrix where the information can be converted into a canonical form for better matching to previous solutions. The tool is further improved with the version Recal+ where the tool looks at the structural composition of the constraint for implied logical satisfiability with solution reuse. GreenTrie, developed by Jia *et al.* (2015) which is similar to Recal+, is an extension to Green. Optimising constraint solving by introducing an assertion stack has been tried by Zou *et al.* (2015). The aim here is to maintain a stack of formulas and declarations, which is provided by the symbolic executor. Zou *et al.* (2015) cache each query result of the stack for further reuse and avoiding redundant queries. Brennan *et al.* (2017) developed Cashew[27] which is built on top of Green, and is designed to process and cache constraint solutions in the theory of linear integers and strings.

In the work of Aquino *et al.* (2017) and Aquino *et al.* (2019) a comparative study is done, where Green, GreenTrie, Recal, Recal+ and Julia are compared, and in which it is shown that Julia outperforms the other caching tools. Based on this recent study, the thesis only compares Julia with Green and ignores the other caching tools.

2.5 Summary

In summary many different tools and concepts were explained. To capture the information in an abstract view, see Figure 2.9. The arrows indicate the flow of information. There are three parts:

1. Constraints are generated during some form of program analysis. The assumption made in this work is that this analysis is a symbolic execution of the program.

2. The generated constraints must be checked for satisfiability by an SMT solver. For this work the assumption is that this step is accomplished by Z3.

3. In order to speed up the satisfiability check, we insert a caching approach between the analysis and the solver. The focus here is to evaluate different approaches to caching implemented in the Green framework.

[27] `https://cashew.vlab.cs.ucsb.edu`

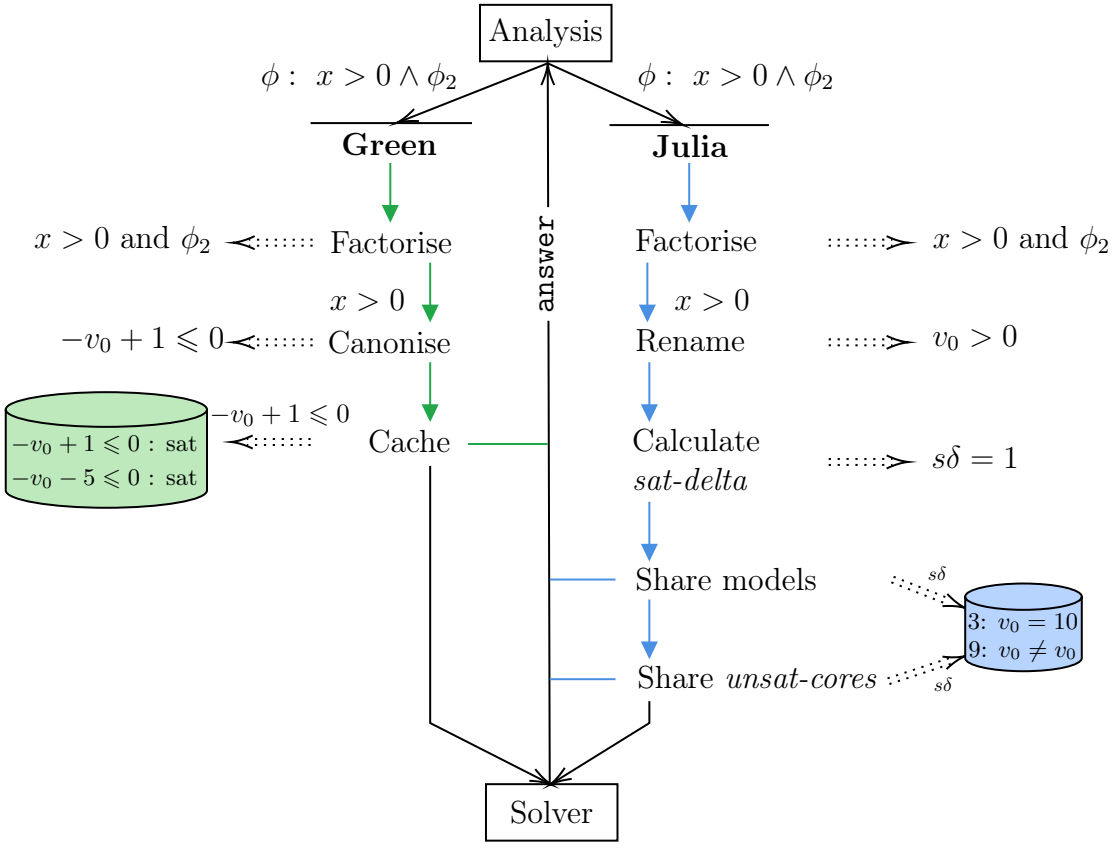


Figure 2.10: Green vs. Julia caching.

Figure 2.10 represents in summary the two different caching tools that perform pre-processing of constraints and provides a speed up to present solutions for the analysis. For Green the pre-processing is factorisation and canonisation of the constraints. Whereas Julia executes factorisation and a simple renaming of the variables in the constraints. For simplicity the second factor (ϕ_2) is ignored in the Figure 2.10.

Green's caching layer checks for exact matches, whereupon sat/unsat solutions are stored. The solutions are stored in a key-value store, with the constraint as key and solution as value. Julia's caching layer conducts an approximate matching with *sat-delta*, where it gets the closest matches to the target's *sat-delta*. Then those matches are picked one at a time, and tested to see if a model satisfies the constraints (in the sat case) or implicitly proves that the constraint is unsat with an *unsat-core* (in the unsat case). Julia's solving layer produces a model or *unsat-core* for the target constraint. The solutions are stored in two separate stores, with an entry having the *sat-delta* value as identifier and another parameter referencing the solution. In Green's solving layer, the sat/unsat is computed. Z3 is an SMT solver, used in the solver layer by most solution caching frameworks, to compute solutions for constraints.

Chapter 3

Design and Implementation

The main focus of this chapter is to illustrate how the Grulia service (in Section 3.1) is added to Green to allow a comparison between Green (without Grulia) and Green with Grulia. In addition a discussion is presented on improving the factorisation step of Green with an algorithm based on Union-Find (in Section 3.2).

3.1 Grulia

Julia is implemented as a service in Green, and this new service is called Grulia (as in Green+Julia). To be clear, Grulia is an implementation within Green and functions as a service which replicates the functionality of the Julia algorithm. See Figure 3.1 for an abstraction of the Grulia pipeline flow (accentuated with the blue box) within the Green framework. All the components will be discussed, since either a component had to be newly created or improved.

The Grulia service is signified by the Julia algorithm component in the figure. Having Grulia as a service in Green, makes it helpful and more suitable to compare the classic Green pipeline for satisfiability, with one that shares some of the exact same components but also includes the Julia approach. Specifically, the pipelines are:

Green: `(Factorise (Canonise (Z3)))` (see Figure 2.5)

Grulia: `(Factorise (Rename (Grulia (Z3))))` (see Figure 3.1)

Factorise Both pipelines use the same Factoriser service, which is improved with a new algorithm and is further discussed in Chapter 3.2.

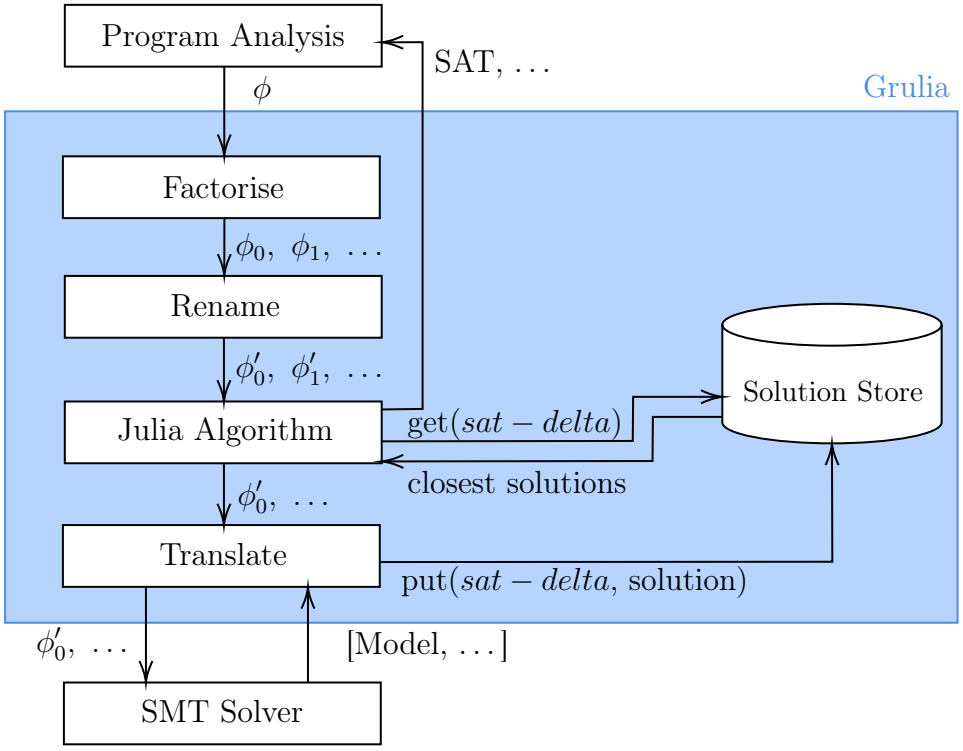


Figure 3.1: Program analysis with Grulia pipeline in Green framework.

Rename The Renamer service is a stripped down version of the Canoniser service, with only the renaming feature. It is a light-weight service to accomplish the renaming of variables in lexicographic order for constraints. The renaming functionality is still needed for the model assignments (value substitution) for the Grulia service. Note that the Renamer and the *sat-delta* calculations in Grulia serve as an approximation for the canonisation step in Green, and one of the important aspects of an evaluation of Grulia is to see how well this works.

Renaming is done by using a visitor pattern to step through the expression tree, making a copy of variables' details except giving them a new name with a prefix "v" and a number. The number typically depends on the number of variables, for consistency, counting from 0. The new variable is then pushed onto the stack. Upon completion of the visitor pattern on the expression, the stack is empty and all variables are renamed and the result is sent to the rest of the pipeline. For example the input would be $\phi_1 : [(a > 5) \wedge (b < 7)]$, and $\phi_2 : [(c > 5) \wedge (d < 7)]$ then the variables of ϕ_1 and ϕ_2 would be renamed to $[(v_0 > 5) \wedge (v_1 < 7)]$ if they have the same bounds.

Cache Layer Omission Each solver service in Green either extends a `SATService` or a `ModelService`. The former is for returning a sat/unsat answer. The latter is for returning a model as solution. Both services have two

solving methods, one involving a caching layer in which the cache is queried to find the solution if the target constraint has already been evaluated and the second method not. To remind the reader, Green's caching (by means of `MemStore` or `RedisStore`) works like a key-value store, containing the constraint and its solution. If the solution is not found at the caching layer, the constraint is then passed on to the solving layer of the service.

During the replication phase it is noted that for the experiments in Aquino *et al.* (2017) the third cache feature is disabled, as mentioned at the end of Section 2.3. This cache functions similar to the caching layer of the `SATService`. Therefore to stay true to the replication, the cache-less solving method of the `SATService` is used to omit Green's hash caching layer for Grulia.

SAT-Delta Calculation The first step to the Julia algorithm is computing the *sat-delta* of a constraint, see Figure 3.2 as summary of the *sat-delta* calculation procedure. This happens after the constraint is passed from the Renamer to Grulia. In terms of Green, a visitor pattern is used to step through the expression (line 10). For each variable the given reference solution (set with line 8) is pushed onto the stack (as a substitution step). After substitution the *sat-delta* equation (see equation (2.2) in Section 2.3 for reference) gets executed. One can have any number of reference solutions. The *sat-delta* of a clause is calculated with a given reference solution, aggregated together with those of the other clauses, and then that value is passed back up as the evaluated *sat-delta* value for that constraint with the specified reference solution (line 13). The details of the calculation are described in Section 2.3 (under Julia).

The lines 8–23 are repeated for any number of reference solutions. The final *sat-delta* of the constraint is the sum of all the recorded *sat-delta* values of the different solutions and then taking the average (line 26). One effective optimisation which have been included, is to check for *sat-delta* values of 0: in such cases, the corresponding reference model satisfies the constraint and the solution is returned immediately (lines 15–19). The check is included since it is implemented in Julia.

A difference to note is that a `Double` is used to represent the average used for the *sat-delta* value, whereas Julia uses a custom data structure called `BigRational` that just represents values as fractions and can store larger values.

Share Models After the *sat-delta* is computed, it is used to extract the K closest models from the store. These are then checked to see if any of them satisfy the constraint. The sat store is queried to verify that it is not empty, otherwise a call is made to the solver for evaluation. Upon checking

```
1  private Double calculateSATDelta(Expression expr) {
2      Double result = 0.0;
3      GruliaVisitor gVisitor = new GruliaVisitor();
4      try {
5          // Repeat for given solutions.
6          for (int i = 0; i < REF_SOL_SIZE; i++) {
7              // Set given reference solution.
8              gVisitor.setRefSol(REFERENCE_SOLUTIONS[i]);
9              // Step through the expression.
10             expr.accept(gVisitor);
11             // Obtain the expression's satDelta.
12             // Clause values already aggregated.
13             satDelta = gVisitor.getResult();
14
15             if (Math.round(satDelta) == 0) {
16                 // The computation produced a hit,
17                 // satisfying the expression.
18                 expr.satDelta = 0.0;
19                 return 0.0;
20             } else {
21                 // Record calculated satDelta.
22                 result += satDelta;
23             }
24         }
25         // Calculate average satDelta.
26         result = result/REF_SOL_SIZE;
27         // Store the value in the expression.
28         expr.satDelta = result;
29     } catch (VisitorException x) {
30         result = null;
31         log.fatal("encountered an exception", x);
32     }
33     return result; // Final satDelta value of expression.
34 }
```

Figure 3.2: Java code excerpt of top layer *sat-delta* calculation implementation.

old solutions, a sorted set[1] is extracted which consists of models less than or equal to the specified number of matches to obtain (the value K in the Julia algorithm). A match in this case is the closest model or models to the target constraint, based on the *sat-delta* value. The extraction process is handled by the store and is explained later in this section.

After extraction, the constraint is evaluated with each model (picking from the smallest *sat-delta* difference to the largest). If the constraint is not satisfied with a chosen model, test the next one, and so on until either a satisfying model is found, or the set is exhausted. A model is tested by substituting the given model's variable assignments to the corresponding variables in the target constraint, evaluating the constraint and verifying the satisfiability. The substituting and evaluation process is done with a visitor stepping through the constraint. If one of the chosen models satisfies the constraint, return true immediately. If the set is exhausted – meaning none of the chosen models satisfy the target constraint – return false, causing the next step of checking if any *unsat-cores* are shared.

Share *unsat-cores* If none of the proximity models satisfy the constraint, it is tested for unsatisfiability by checking the shared *unsat-cores*, which is done in a similar fashion to the shared models. If the unsat store is not empty, a sorted set[2] is extracted which contains *unsat-cores* less than or equal to the specified number of matches to obtain (the value K in the Julia algorithm). Again a match is defined by the closest constraint or constraints to the target constraint, based on the *sat-delta* value. The retrieval from the unsat store is done in a similar fashion to the sat store.

From the set, pick an *unsat-core* (working from the smallest *sat-delta* difference to the largest) and evaluate if the constraint contains the *unsat-core*. If a picked *unsat-core* is not present in the target constraint, pick a next one, and continue in this manner. An *unsat-core* is evaluated by checking if each of the clauses in the *unsat-core* are present in the target constraint. If all of the clauses are present it means that *unsat-core* is shared by the target constraint, where upon proving the constraint's unsatisfiability. If an *unsat-core* is shared, the function returns true immediately, signifying the constraint is unsat. If all the matches are evaluated and no shares are found, a false is returned, resorting to the next step in the program – invoking the solver to compute the solution.

Binary Search Store The computed solutions from the solver are amassed in the store. The initial implementation of the replication study included

[1]Sorted based primarily on the *sat-delta* value, and secondarily on the solution size or otherwise the string representation length. Here the solution size refer to the number of variables contained in the model.

[2]Same sorting criteria as specified for the models, except the size of the solution refers to the number of clauses contained in the *unsat-core*.

faithful implementations of the same data structures as Julia since the main interest was replicating previous results. For example, only the *sat-delta* values of the constraints and the corresponding cache solutions are stored. Like Julia, a priority queue was used to extract the K closest *sat-delta* values from the cache. Initially Green only had the `RedisStore` with a limited interface and could not easily implement the retrieval of multiple entries based on a specified calculated criteria. Therefore a new data structure was introduced to Green to serve as Grulia's store. After overcoming some of Green's limitations and a faithful replication was achieved, some of the storage structures were improved. The initial replication prototype started with something similar to Julia's sorted list implementation. The major drawback of this kind of implementation (working with a list) is that it has close to linear time execution.

The Grulia store was augmented by using a sorted `TreeSet` implementation, which provided the imperative requirements such as keeping the nodes sorted, containing only unique nodes and present quick access and retrieval of the contained nodes. A node in the tree contains the vital information, such as the *sat-delta* value and the solution to the constraint, which is either a model (sat case) or an *unsat-core* (unsat case). Moving from the linear list to a sorted `TreeSet` structure reduced the execution time of searching and retrieval to logarithmic time. The search occurs by means of a filter, including results only in close proximity to the *sat-delta* of the constraint in question. Using binary search to find the target *sat-delta* and extract K solutions around (above and below) the target. If the value of K is greater than the store's size, return all the entries. Otherwise check for K entries around the target *sat-delta*. Instead of linearly searching through a big list, binary search is used to find the target *sat-delta*. Using two pointers, one to look at the entries smaller than the target, and one to the entries greater than the target (in terms of *sat-delta* value). Alternating between the head set and the tail set, the entry with the closest (or smaller distance) *sat-delta* is chosen and added to a new list. This is done until K closest entries are picked.

A supplementary filter is applied in Julia for the extraction of the models. Before accepting a model solution in the list of K entries, the model size is assessed to see if it is greater than or equal to the target constraint. This is overcome in the older version of Julia, with a simple implementation of substituting in the value of zero if a chosen model has too few variable assignments. In the thesis' experiments, the default zero substitution is turned off, and Grulia rather use the model size filter, which is also applied to *unsat-cores* regarding the number of clauses.

Note that a similar store and search is done for both the sat and unsat case. A more refined implementation for the unsat case would be to follow a similar approach to Julia – using a `BloomSet`, which applies a Bloom filter on the information stored in the structure. A Bloom filter, conceived by Bloom (1970), provides a probabilistic data structure to quickly determine whether or not an element is in a set. This enables a faster check for all available

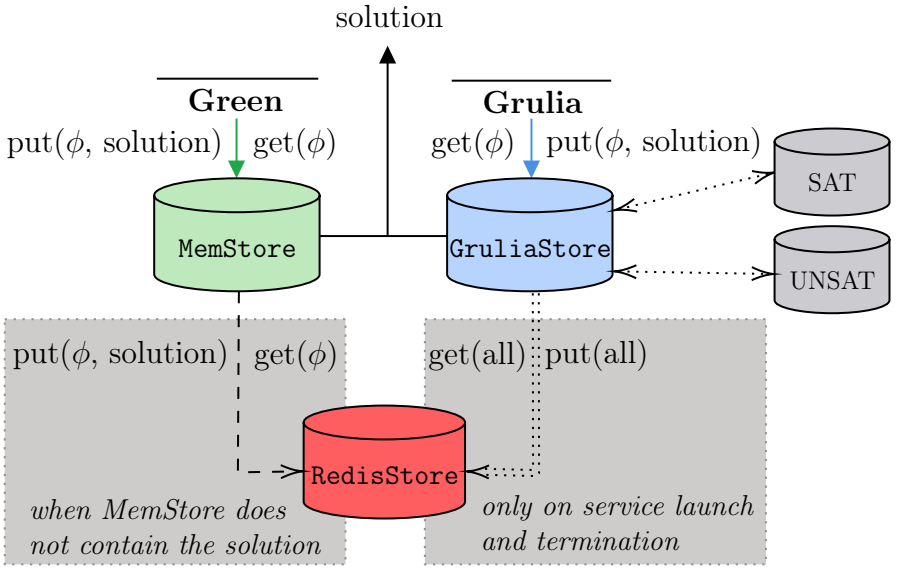


Figure 3.3: Green vs. Grulia hybrid persistent caching.

unsat-cores. The Bloom filter is not implemented because the main goal was to test the model reuse and only use a simple implementation for unsat cores. The simple unsat store already provided major reuse increase, and therefore the Bloom filter implementation is left for future work.

Hybrid Persistence Redis (the back-end storage unit that Green typically uses) is an in-memory store with a persist memory feature where its content is written to the hard disk. The initial experimental results showed that Green compared significantly worse to Grulia. Profiling Green revealed the main bottleneck at that time was Redis, whereupon it became noticeable that Redis is not truly an in-memory store. The cause for reduction in execution time is due to latency of the Redis store, which uses network communication for storing data even though storage takes places on the local host. These findings led to implementing a true in-memory storage unit for Green, called `MemStore`, such that Green can be compared with Grulia more accurately because Grulia has an in-memory store. The `MemStore` implements a `HashMap` where the key-value pairs are stored. This in-memory store also resulted in a significant speed-up for Green.

The main benefit of Redis, and by extension Green, is to persist data across runs. To retain this functionality, `MemStore` is extended (see Figure 3.3) to work with a secondary store (if enabled) which is preferably a persistent store like Redis. The functionality is done in such a manner that each entry that is added to `MemStore` is also added to the secondary store. Upon querying `MemStore` for the solution of a target constraint, its data is searched through to identify the target constraint. If it is not found in `MemStore`, a query is then made to the secondary store for the solution. If the solution is present, it

is added to `MemStore` and the solution is also returned. Otherwise the solver is invoked to compute the solution, adding the solution to `MemStore` and also to the secondary store.

Hybrid persistence refers to storage that uses a fast in-memory store combined with a slower persistent store on disk. Adding such functionality to Grulia, allows Grulia results also to be persisted across runs, similar to Green. The `TreeSet` data storage is in-memory which results in one losing the information of the cached results once the program terminates. A simplistic way to make the store persistent, is to flush all entries from Grulia's store to a secondary store (that is persistent) if the secondary store is enabled (see Figure 3.3). To write these entries to the persistent store, the entries of Grulia's store need to be serialisable objects. Adding an entry to Redis involves the entry's hash-code as key, and the entry object as the value. Redis gives some difficulty for the Grulia storage mechanism, since Redis returns a single entry and the Grulia approach requires a set of entries. A simplistic way to obtain the persistent information for Grulia's store is to load all entries from the persistent store once the Grulia service starts. Initially a check is done to see if a secondary store is enabled and contains items, followed by fetching all of the items. From a given entry object in the Redis store, one can test its instance to determine if it should be placed in the sat or unsat cache when loading in the solutions. One by one each item is filtered to the corresponding sat or unsat cache. After all the entries are loaded the Grulia service continues with its usual procedures. Upon completion of the program all the content of the Grulia store is flushed to Redis for persistent storage.

***Unsat-cores* in Green** A significant difference between Green and Julia was that Green did not support the calculation of *unsat-cores*. Therefore the compatibility of *unsat-cores* had to be added to Green. Green only mimicked *unsat-cores* by finding the smallest factored constraint that proves unsat and stores this factor instead of the *unsat-core*. For the *unsat-cores* first adjustments had to be made to the translation of Green expressions to what Z3 can understand. Initially this was done in the SMT-LIB (string) translation of Green, and then later when shifting to Z3 Java, done in the Context translation. The major problem was that Green concatenated all the constraints in one big assert before sending it to Z3. This made the different clauses indistinguishable for Z3. The change made that each clause of the constraint is added as singular asserts with a given name as an identifier.

Lastly the correct Z3 settings has to be set, such as enable production of *unsat-cores* and to disable `auto-config` to get the minimal *unsat-core*. The new model and *unsat-core* translation work was done with a new `ModelCoreService` class in Green. The new service makes use of a new data structure called `ModelCore`, which stores either a model or *unsat-core* as solution to the target constraint.

3.2 Factoriser

The process of splitting a constraint into its independent parts, which as the results in the evaluation chapter (Chapter 4) will show, is a crucial step in the caching and solving process. Factorisation is usually the first transformation in the process of constraint solving, which places an emphasis on the importance of having a proper implementation that is fast[3], as not to be a bottleneck in the whole analysis and solving process. To recall what factorisation is: the input constraint is split into a number of independent factors (sub-constraints). Two clauses in a constraint are independent if none of the variables in the one clause can affect the solution in the other clause. Since the input constraint is in CNF each of the factors must be satisfiable for the input constraint to be satisfiable. For example $\phi : [(a > 5) \wedge (b < 7)]$ would become $\phi_1 : [a > 5]$ and $\phi_2 : [b < 7]$. Logically if both clauses are satisfiable it implies that ϕ is satisfiable as well.

A part of the thesis work includes incorporating a new and improved technique for factorisation in Green. A technique using the Union-Find algorithm is suggested as improvement and has been evaluated to be a significant improvement to time execution of a Green analysis, compared to the original Factoriser service.

Union-Find

The Union-Find algorithm (also known as a disjoint-set data structure or set union problem) starts with a number of singletons (that are disjoint elements). The algorithm works by performing a number of find and union operations. According to Galil and Italiano (1991), by definition there are two invariants that always stay true: (i) the sets should be disjoint (non-overlapping), only joined upon a criteria of equivalence, and (ii) the representative of each set (also referred to as the root) is one of the elements contained in the set. The user can specify the criteria for equivalence for the union operation. The union operation connects two objects (object that is an element or set of elements), and the find query checks if there is a path connecting one object to another. Each union operation reduces the number of components (be it singletons or sets) by 1.

Three main operations:

`make-set(`e`)`**:** make a singleton set containing the element e, and its representative that is the element e (a unique id). The operation has $O(1)$ time complexity, so initialising n elements has $O(n)$ time complexity.

`union(`A`,`B`)`**:** combine the two objects A and B into a new set named A, where A,B can be an element or set, but is required to be disjoint. The union

[3]It goes without saying that it is critical to produce accurate results as well.

```
function makeSet(element)
    if element not in tree:
        add element to the tree as a singleton
        element.parent = element
        element.rank = 0
```

Figure 3.4: Pseudo-code of redefined make-set.

operation uses the find operation to determine the roots of the sets A and B belong to. If the roots are distinct, the sets are combined by attaching the root of the one set to the root of the other. If this is simply done with making A a child of B, the height of the tree can grow as $O(n)$. One can prevent this by using union by rank or by size (later discussed under optimisations).

`find`(e)**:** return the root of the unique set containing the element e.

Optimisations

Union by Rank One optimisation is to introduce a rank or size to each set. Typically upon make-set the element's rank is set to 0 (see Figure 3.4), increasing the rank with one when a set is merged with this set. The rank is taken into account when doing the union operation. The set with the lower rank, is merged with the set with the higher rank, taking on the root of the set with the higher rank as indicated in Figure 3.5.

Path Compression Typically `find`(e) follows the chain of parent pointers from e up the tree until it reaches a root element, whose parent is the element itself. This root element is the representative member of the set to which e belongs, and may be e itself. Path compression is used to combat tall trees and to flatten the structure of the tree. One way is to use the path splitting that Tarjan and van Leeuwen (1984) proposed. The procedure requires to follow the parent pointers from e until repeating an element; then returning the repeated element, as indicated in Figure 3.6. Upon following the parent pointers, every element's pointer is changed to the root of the set. This is valid, since each element visited on the way to a root is part of the same set. The resulting flatter tree speeds up future operations not only on these elements, but also on those referencing them. This one-pass algorithm for find is more efficient while retaining the same worst-case complexity.

Design

The set union problem has many variants and can be applied as a solution to many instances. In the context of Green, it is applied to quickly establish

```
function union(element1, element2)
    root1 = find(element1)
    root2 = find(element2)
    if (root1 == root2):
        return root1

    rank1 = root1.rank
    rank2 = root2.rank
    root = null
    if (rank1 < rank2):
        root1.parent = root2
        root = root2
    else if (rank1 > rank2):
        root2.parent = root1
        root = root1
    else:
        root2.parent = root1
        root1.rank = rank1 + 1
        root = root1
    return root
```

Figure 3.5: Pseudo-code of redefined union.

```
function find(element)
    parent = element.parent
    while (parent != element):
        run through the parents of each element, and
        assign the same root to each element
    return element
```

Figure 3.6: Pseudo-code of redefined find.

dependency among clauses of a given constraint. The relation pertains to the clauses of a constraint, by looking at the variables contained in each clause, and the sets containing the same variables are merged. Afterwards one is left with a number of disjoint sets, that are the different independent components (or factors) of a given constraint.

The initial implementation done in Green with this algorithm, was with a graph data structure as a network connectivity solution, which performed immensely slow. The overhead included storing extra information for the edges, to signify connected components and then mainly to check the graph for cycles.

Switching over to a better abstraction of the algorithm, making use of a tree structure, and a quick-find and quick-union implementation reduced the execution time of the algorithm. The find operation then functions by looking

if element a and element b have the same root. The union operation performed faster, because all that has to happen is changing the root of the set element b belongs to, to the root of the set element a belongs to. This results in $O(n)$ with find and union in the worst-case.

The factoriser service employing the new algorithm, shows a phenomenal improvement in execution time. The optimised service is rather used (to minimise pre-processing overhead of the constraints) for the experiments and evaluation. The algorithm involving union by rank and path compression, achieves an amortised cost of $O(\log(n))$ per operation.

Application

Working through a concrete example, and for simplicity using a constraint with four clauses. Given a constraint:

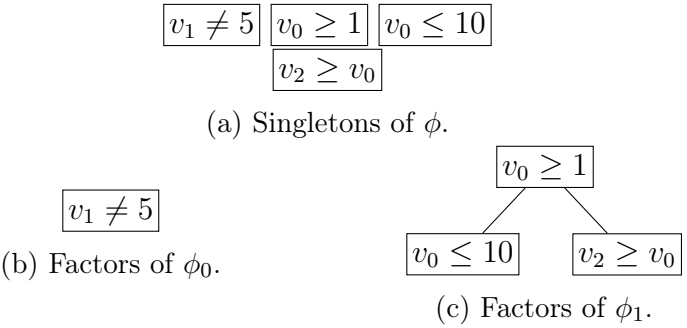
$$\phi : [(v_0 \geq 1) \wedge (v_1 \neq 5) \wedge (v_0 \leq 10) \wedge (v_2 \geq v_0)]$$

the possible resulting factors are as shown in Figure 3.7. The four clauses of ϕ are $p_0 : [v_0 \geq 1]$, $p_1 : [v_1 \neq 5]$, $p_2 : [v_0 \leq 10]$ and $p_3 : [v_2 \geq v_0]$.

Each clause is individually observed to test for independence using the containing variables as criteria. At hand with the example, say p_0 is observed, then `make-set(`p_0`)` is called, placing the clause in a singleton set, and assigning a rank of 0 to the clause. The clause contains the variable v_0, which has no factor associated with it, using `find(`p_0`)` to check if there is a different root associated to this set, and assigning that root's set as factors to the variable. Next the program calls make-set on p_1, following the same procedure as before, resulting in p_1 that is in its own set and the only factor associated with the variable v_1.

With the next clause the variable v_0 is observed, which already has a factor associated with it. Therefore the factor's root is determined with `find(`p_2`)` to get the `root` of the set, and then followed with `union(root,` p_2`)` to add the clause to the factors associated with v_0. Both clauses' ranks are equal, therefore `root` stays the root of the set and its rank is increased by one.

The last clause, p_3, contains two variables one of which has factors associated with it and the other not. This is overcome by using the root of the factors associated with the known variable v_0 and assigning those factors to v_2 as well. The `root` of the set is determined and the call `union(root,` p_3`)` is made, adding p_3 to the set of `root` (which is p_0). The new root of the set is determined by looking at the rank of the two elements. The clause, p_0, has a greater rank than p_3 and therefore the former is left as the root for the set. Note that due to union by rank the root of p_3 is p_0 and not p_2. Thus it would occur that the resulting tree like structure as in Figure 3.7 is achieved, signifying the different factors.



(a) Singletons of ϕ.

(b) Factors of ϕ_0.

(c) Factors of ϕ_1.

Figure 3.7: Factors of ϕ as disjoint-sets.

3.3 Summary

In summary two tools/pipelines are presented that does pre-processing of constraints, has a caching layer, and a solving layer. For Green the pre-processing is factorisation and canonisation of the constraints. Whereas Grulia does factorisation and a simple renaming of the variables in the constraints. Green's caching layer checks for exact matches, whereupon sat/unsat solutions are stored. The solutions are stored in a key-value store of the constraint as key and solution as value. Grulia's caching layer does a vague matching with *sat-delta*, where it gets the closest matches to the target's *sat-delta*. Then those matches gets picked one at a time, and tested if it satisfies the constraints (in the sat case) or shows the constraint unsat (in the unsat case). The matches are models in the sat case and *unsat-cores* in the unsat case. The solutions are stored in two separate `TreeSet`s, with a node having the *sat-delta* value as identifier and the solution. In Green's solving layer, the sat/unsat is computed whereas with Grulia a model or *unsat-core* is produced (with extra flags set for the solver).

Chapter 4

Evaluation

This chapter includes the discussion of numerous experimental results for the analysis of the effectiveness and the efficiency of the tools with different solution caching strategies. The different tools are evaluated across four categories of input constraints: i) the parsed data sets from Klee and JBSE (replication experiments), ii) constraints obtained from program analysis with SPF (industrial experiments), iii) constraints obtained from program analysis with Coastal (concolic experiments), and iv) artificially constructed constraints (generated experiments). Lastly the efficiency of the new factoriser service will be tested. The chapter concludes with some observations drawn from the various experiments.

4.1 Experimental Setting

All experiments ran on a machine with 4 Intel Xeon(R) E5-2640v2 CPUs with 8 cores and 16 threads each, running at 2.00GHz. The machine has 283GB DDR3 memory at 1866MHz. This machine is chosen for a stable environment and since there is no resource contention, as one might find on a desktop computer. To simulate the performance of a desktop computer[1], the experiments run inside a Docker[2] container with a clean version of the Ubuntu 18.04 LTS operating system. Each Docker container is configured with 15GB of memory. All the experiments are run sequentially to further minimise resource contention. Each experiment is run 10 times to eliminate noise from inaccurate time measurement.

[1] In terms of resource allocation, and as proof that the tools can run viably on any computer.

[2] `https://www.docker.com`

Below is the breakdown of the different tools in the sat-checking experiments, showing the services in the pipeline and which solutions (sat answers or models) retrieved from the solvers and also which internal storage is used. Z3Java is the Java bindings that run through the Green framework.

Green: `Factoriser, Canoniser, Z3Java (sat) +MemStore`

Grulia: `Factoriser, Renamer, Grulia, Z3Java (model) +GruliaStore`

Julia: `Factoriser (implicit renamer), Z3 (model) +Repository`

Z3Fact: `Factoriser, Z3Java (sat) +MemStore`

Z3Cache: `Z3Java (sat) +MemStore`

In replication experiments:

Z3: `Z3Java (sat)`

In industrial experiments (these two are run interfaced through SPF, also using the Java bindings):

Z3: `Z3 (basic mode - sat)`

Z3Inc: `Z3 (incremental mode - sat)`

In concolic experiments (command-line interfaced through Coastal):

Z3: `Z3 (basic mode - model)`

The execution environment is set up with Microsoft's Z3 version 4.8.4 (simply referred to as Z3), the latest at the time of writing, Jedis 2.9.0, Redis 5.3.0 and Java version 8. The latest version of Julia at the time of writing is used[3]. The Green framework is continuously updated and improvements made, and this happened during writing as well. The same version of Green is used for the Green, Grulia and Z3Fact experiments[4].

The same reference solutions specified in the replication experiments for Grulia, are used in the other experiments. All of the data sets to evaluate each tool on are in the quantifier-free linear integer arithmetic logic.

[3]Using commit *772dbde*.

[4]For reproducibility and research, the tool and all experimental data are available at: `https://bitbucket.org/Developer_Jan/green/src/master`. The work for the experiments are contained in a separate instance of Green, storing the necessary files on the BitBucket repository with the core files of commit *cc03417* on BitBucket (for the thesis experiments and setup) agreeing with that of commit *a1261b0* on the GitHub fork (`https://github.com/JHTaljaard/green`) of Green (for integration with the active framework).

Replication Experiments

Green, Grulia and Julia are compared across the same data sets (containing different constraint examples) and the other tools for evaluation are Z3, Z3Fact and Z3Cache. The data sets are the same used in the experiments of Aquino *et al.* (2017), and contain approximately 800 000 constraints (before factorisation); they are available on Bitbucket[5].

Before the execution of each data set, the respective cache of each tool is cleared to get the reuse of a single experimental run. To determine the satisfiability of each constraint, Grulia and Julia each calculate the constraint's *sat-delta* value with respect to three reference models:

- the model that sets all variables to −10 000,
- the model that sets all variables to 0, and
- the model that sets all variables to 100,

and a K value of 10 (following Aquino *et al.* (2017) for the replication of the more detailed results). To clarify small discrepancies, it is of note to mention that in the paper of Aquino *et al.* (2019), Julia uses −1 000, 0, 100 as reference models. This experiment uses the same reference models as per the former paper, for both Grulia and Julia, which is stated in the list above.

The hash cache (the third cache mentioned at the end of Chapter 2) is disabled in Julia for the experimentation of this replication study, because the results in the paper of Aquino *et al.* (2017) do not use it. A comparison is drawn between the three implementations: first investigating the effectiveness (reuse rates) of the caches, and then turning attention to the efficiency (running times) of the tools.

Industrial Experiments

As the title of the thesis suggests, the different tools are evaluated on real-world applications. For this experimental setting a sample of real-world Java programs are used for the SPF analysis consisting of 23 programs. Some of the programs are typical symbolic execution examples that are available with the SPF package and the rest are publicly available on GitHub. These programs together make up around 3.8 million constraints. The settings for SPF in order to execute programs to foster controlled experiments on identical sets of path conditions are, **enabled** `multiple errors`, and **disabled** `optimised choices`.

The baseline for the comparison is set using Z3Inc through SPF, the reason for incremental mode being that it seems the fastest setting for obtaining a solution from the solver. As a sanity check with all the different pre-processing

[5] `https://bitbucket.org/andryak/julia/overview`

Z3 is run through Green with no pre-processing and only a cache (`MemStore`) enabled represented by Z3Cache.

The settings for Z3 are the same as before, for Grulia the models are enabled, and also producing unsat cores. Additionally `auto-config` is **disabled** (to get smaller cores), which allows for more effective reuse, and using less storage. From previous analyses, the `auto-config` option showed interesting behaviour, with it being enabled, less reuse is obtained, for instance with the *BinTree* example (one of the programs used for analysis) the unsat reuse was 23%. With `auto-config` disabled the results show 99% reuse in the unsat case of *BinTree*.

A secondary store (Redis) which is persistent is enabled. This means having the in-memory storage capabilities, with the store flushing to the persistent store to have the persistent storage for solutions across runs. Across runs here means running the same program analysis twice, followed by clearing the storage units and then moving on to the next program analysis.

Concolic Experiments

The different tools are evaluated on real-world applications with a concolic analysis. For this experimental setting a sample of real-world Java programs are used for the analysis with Coastal, consisting of 13 programs. These are a subset of the same programs used in the SPF analysis, because Coastal does not cater for all the instructions encountered in the complete sample. These programs account for about 37 516 constraints. The settings for Coastal in order to execute programs to foster controlled experiments on identical sets of path conditions are, running it with the `quiet` mode to omit writing of any textual output except for the result reporting, **enabled** `constant elimination`, like SPF also using a depth-first search and using a single thread for analysis. For further consistency and to enable comparison between a symbolic and concolic analysis any search depth limits have been removed from the program analyses and therefore also reduced the search space by decreasing the input parameters for the programs. The corresponding reduced SPF results are in Appendix B for comparison.

The baseline for the Coastal analysis is set using Z3 interfaced through Coastal. Again as a sanity check with all the different pre-processing, Z3 is run through Green with no pre-processing and only a cache (`MemStore`) enabled represented by Z3Cache.

Generated Experiments

What are the worst-case scenarios for Grulia? Two possible cases can be 1) where the cache is flooded with irrelevant solutions having close *sat-delta* values causing the search to miss the correct solution, resulting in a solver call. The other is 2) where the constraints are formed in such a manner that no previous

model can satisfy it. With this in mind a few automatic synthetic constraints are constructed, to shine light on Grulia's worst-case scenarios and to further justify the trends appearing in the other experiments.

The different versions of bounded constraint generation are displayed in Figure 4.1. Each version generates one constraint and each loop creates two or three clauses for a variable depending on the condition. One clause serves as a lower bound and the other as an upper bound for the variable, the optional third clause place a dependency on another variable by linking two variables. There are four main constants: `max`, `numVars` and `addDependence` or `randomDependence`. All are set and fixed at the start of the run, but these values can be adjusted to get various effects on the constraints generated. The constant `max` influences the range of the bounds for the constraint. The constant `numVars` is the total number of unique variables for the constraint. l and u are any random value in the specified range. The flag `addDependence` adds the third clause if enabled, which makes the current clause dependent on the next one, by containing the same variable and introducing a new variable for the upper bound. The final ϕ is the complete constraint built and returned for evaluation. In an abstracted view the total number of clauses generated in a version are either $2 \times$ `numVars` or $3 \times$ `numVars` $+ 1$.

Version 1 generates a set of constraints with a setting to make the constraints dependent or not. For example the first constraint looking only at 3 variables with dependency will be:

$$[(150 < v_0 < 1000) \wedge (v_0 \leq v_1) \wedge (250 < v_1 < 1400)$$

$$\wedge (v_1 \leq v_2) \wedge (200 < v_2 < 2200) \wedge (v_2 \leq v_3) \wedge (v_3 < 3200)],$$

and without dependency:

$$[(150 < v_0 < 1000) \wedge (250 < v_1 < 1400) \wedge (200 < v_2 < 2200)].$$

Version 2 is similar to Version 1, except the dependent constraints are randomly added, which is based on a 60% chance to add dependent clause for the experiment.

Version 3 chains the clauses, making the lower bound of the clause the upper bound of the previous clause. For example with dependency true:

$$[(150 < v_0 < 1000) \wedge (v_0 \leq v_1) \wedge (1000 < v_1 < 1400)$$

$$\wedge (v_1 \leq v_2) \wedge (1400 < v_2 < 2200) \wedge (v_2 \leq v_3) \wedge (v_3 < 3200)],$$

and without dependency:

$$[(150 < v_0 < 1000) \wedge (1000 < v_1 < 1400) \wedge (1400 < v2 < 2200)].$$

Version 4 decreases the range of the bounds based on δ, for an acceptable model. For example using $\delta = 50$, with dependency:

$$[(150 < v_0 < 200) \wedge (v_0 \leq v_1) \wedge (250 < v_1 < 300)$$

$$\wedge (v_1 \leq v_2) \wedge (200 < v_2 < 250) \wedge (v_2 \leq v_3) \wedge (v_3 < 1250)],$$

and without dependency:

$$[(150 < v_0 < 200) \wedge (250 < v_1 < 300) \wedge (200 < v_2 < 250)].$$

Each version is run on one of the tools building 100 constraints with each containing 500 unique variables. Each version is run with the dependency condition on and off, which gives seven different runs that will be analysed in the rest of the chapter. For the version with the fixed bounds a value of 50 is used for δ, as indicated in Figure 4.1.

4.2 Effectiveness

Once the Grulia service computes the average *sat-delta* value of a target constraint, it extracts the ten closest models from the sat cache. The service checks whether any of the extracted models is also a solution to the target constraint. If a model satisfies the target constraint, it is counted as a *sat cache hit*. If no such model is found, a *sat cache miss* is recorded. The same operations are repeated for the unsat cache with the corresponding *unsat cache hits* and - *misses* recorded in a similar fashion. If both cases, sat and unsat, result in a cache miss, Z3 is invoked to produce a solution for the target constraint. In the sat case the Grulia service then stores the constraint's *sat-delta* value paired with the model in the cache. Otherwise (in the unsat case), the Grulia service stores the constraint's *sat-delta* value paired with the *unsat-core*. The sat and unsat cases are handled similar in Julia. With Green and Z3Fact, one solution store is queried for a sat value (in both sat and unsat cases) if the solution is not present count it as a cache miss (which can be split respectively). In the case of a cache miss, Z3 is invoked to produce the sat value.

Replication Experiments

The experimental results given in Table 4.1, where the constraints and the cache hit rate for each program are split into the respective sat and unsat cases. The first column gives the names of the benchmark programs. The second column indicates the percentage of input constraints (before any factorisation) that were found to be sat. Note however that Julia uses a different implementation of factorisation than Green and Grulia and sometimes produces slightly different number of independent factors. The difference is the

$\max = 500$
for i in numVars:
    $l = \texttt{random}(0, \max/2)$
    $u = \texttt{random}(l, \max \times 2)$
    $c : l < v_i < u$
    $\texttt{if}(\text{addDependence})$
        $c : c \wedge v_i \leq v_{i+1}$
$\phi : c \wedge v_{\text{numVars}} < \max \times 2 + u$

(a) V1: Fixed dependence

$\max = 500$
for i in numVars:
    $l = \texttt{random}(0, \max/2)$
    $u = \texttt{random}(l, \max \times 2)$
    $c : l < v_i < u$
    $\texttt{if}(\text{randomDependence})$
        $c : c \wedge v_i \leq v_{i+1}$
$\phi : c \wedge v_{\text{numVars}} < \max \times 2 + u$

(b) V2: Random dependence

$\max = 500$
$prev = \texttt{random}(0, \max/2)$
for i in numVars:
    $l = prev$
    $u = \texttt{random}(l, \max \times 2)$
    $prev = u$
    $c : l < v_i < u$
    $\texttt{if}(\text{addDependence})$
        $c : c \wedge v_i \leq v_{i+1}$
$\phi : c \wedge v_{\text{numVars}} < \max \times 2 + u$

(c) V3: Chained clauses

$\max = 500$
$\delta = 50$
for i in numVars:
    $l = \texttt{random}(0, \max)$
    $u = l + \delta$
    $c : l < v_i < u$
    $\texttt{if}(\text{addDependence})$
        $c : c \wedge v_i \leq v_{i+1}$
$\phi : c \wedge v_{\text{numVars}} < \max \times 2 + u$

(d) V4: Fixed bounds

Figure 4.1: Formula versions for artificial generated constraints.

		Green		Grulia		Julia		Z3Fact		Z3Cache	
Program	sat%	sat	unsat	sat	unsat	sat	unsat	sat	unsat	sat	unsat
afs	100.0	99	0	99	0	99	0	98	0	76	0
floppy	100.0	99	0	99	0	99	0	99	0	0	0
diskperf	96.6	99	99	99	99	99	99	99	99	0	0
kbfiltr	93.6	98	66	99	66	99	66	96	66	0	0
cdaudio	85.5	99	98	99	98	99	99	99	97	0	0
wbs	59.8	99	98	99	98	99	98	97	97	0	50
treemap	51.2	99	99	98	99	99	99	99	99	96	95
dijkstra	50.6	96	97	99	97	96	97	87	28	0	0
collision	22.4	99	99	99	99	99	99	98	99	71	75
tcas	2.2	99	99	99	99	99	99	98	98	0	28
reverseword	0.2	99	99	99	99	99	99	99	99	0	99
grep	99.9	99	32	99	73	99	73	99	0	0	0
division	0.1	0	0	0	99	0	99	0	0	0	0
knapsack	0.0	81	57	100	99	100	99	81	57	0	57
multiplication	0.0	0	0	0	99	0	99	0	0	0	0
swapwords	0.0	0	0	0	98	0	96	0	0	0	0
ball	100.0	89	0	96	0	95	0	88	0	3	0
list	78.3	87	86	97	94	96	94	87	86	87	86
old-tax	67.4	31	50	79	50	79	50	31	50	31	50
new-tax	67.3	29	50	78	50	78	50	29	50	29	50
block	100.0	37	0	56	0	36	0	35	0	33	0
avl	93.4	95	72	97	88	73	92	93	70	74	56
average	57.7	74	54	79	72	79	73	73	49	22	29

Table 4.1: Reuse rate (%) of solutions in replication data set.

result of factor ordering done differently in each tool. Both have the optimisation of finishing evaluating the factors of a constraint once an unsat condition is encountered (because it is in CNF one unsat condition makes the whole constraint unsat, and the analysis can continue to the next constraint). Say a constraint is factored into five factors, where one of the factors causes the constraint to be unsat, then the case might be that Green finds the contradicting condition as the second factor, whereas Julia might find it as the fourth factor (evaluating two more sat factors).

The rest of the table shows the percentage of cache hits over total constraints for the sat and unsat cases across the five tools. Note that in the original paper of Aquino *et al.* (2017) the reuse rates were not broken out by sat versus unsat results. The following discussion is dedicated to show that this adds additional insight into the performance of the various tools.

The four groupings capture the subdivision of the benchmark results according to their outcomes:

Similar: The first 11 examples show minimal differences in reuse rates across the four tools.

Unsat: The next five examples are shaded and represent the cases where there are a large percentage of unsat constraints.

Models: The next grouping of four (unshaded) are examples where the reuse of models works particularly well.

Misc: The last two examples (shaded) show more variable performance.

The **Similar** grouping shows that any form of reuse, be that based on models or the syntactic reuse of Green works well on these examples. They are thus not very discriminating, and hence somewhat uninteresting to us. The *afs* and *dijkstra* example shows with Z3Fact the benefit of canonisation, since it got a lower reuse rate in both cases compared to Green. With Z3Cache there are a few interesting examples like *afs*, *wbs*, *collision* and *tcas* where some reuse are obtained, and *treemap* and *reverseword* where high reuse are obtained. These high reuse examples are part of the scenario where one sees pre-processing might sometimes be unnecessary.

The **Unsat** grouping shows one thing very clearly and that is the cases where the use of *unsat-cores*, as used by Grulia and Julia, makes a substantial difference in the reuse these tools get. Comparing the unsat column of Grulia with Green's, it is noticeable that Green's strategy of using just the independent factor that was found to be unsat, is not nearly as efficient as the *unsat-core* returned by Z3. For the *knapsack* example, it shows 0% sat constraints, and yet has sat reuse, that is because when the input constraints are factorised, 11 sat constraints are produced of a total of 7 662 constraints. Both Grulia and

Julia achieve 100% sat reuse, because one of the reference models satisfies the sat constraints, and therefore no cache miss is calculated. On closer inspection on the other examples, the high amount of unsat reuse obtained by Grulia and Julia is due to a small subset of *unsat-cores* being reused in syntactically different constraints. The *knapsack* example also shows that factorisation and canonisation sometimes fail on syntactically different constraints (looking at Green, Z3Fact and Z3Cache vs Grulia). Although Z3Cache obtains unsat reuse in this example, the absence of sat reuse is of no surprise, since there is no factorisation to achieve sat constraints like with the other tools.

The *swapwords* example indicates a difference in reuse among Grulia and Julia. This is because Julia makes six solver calls, and Grulia three. A phenomenon appears where the number of solver calls (in the case of Grulia) are either one, two or three, depending on the *unsat-core* returned by the solver. In this example there are many possible *unsat-cores* and any is valid, although not all are shared among the different constraints. For example, the *unsat-core* $[(v_0 = 1) \wedge (v_0 \neq 1)]$ is shared among more constraints than $[(v_0 = 60) \wedge (v_0 \neq 60)]$. This inconsistency from the solver is still undetermined and beyond the scope of the thesis.

Note that *grep* shows the same behaviour as with *afs* and *dijkstra*, having the same explanation for the difference in reuse between Green and Z3Fact.

The **Models** grouping has the examples which best show the advantage of reusing previous solutions in both the sat and unsat case. The sat case will carry the focus here, since the previous discussion explains the unsat case. Essentially what is happening in the sat case is that constraints have small syntactic changes, but the solutions stay the same. Note that Grulia and Julia perform the same kind of operation on the sat cases in this grouping. Even though they execute a similar operation, they obtain different results in the reuse. Upon inspections it shows that Grulia and Julia has greater reuse since the reference models satisfy some constraints An example constraint, from the one data sample, that explains how the phenomenon of sharing models performs better than identical factors, works as follows. Consider the following where the first constraint encountered is

$$\phi_1 : [(v_0 \leq 7\,999\,999) \wedge (v_0 \leq 3\,499\,999) \wedge (v_0 \leq 5\,499\,999)],$$

followed later by

$$\phi_2 : [(v_0 \leq 7\,999\,998) \wedge (v_0 \leq 3\,499\,998) \wedge (v_0 \leq 5\,499\,998)].$$

Green will think both these are different and will not be able to get any reuse, whereas the other tools will reuse a solution, for example $v_0 = 0$. This phenomenon indicates cases, such as sorting where numerous comparisons (in the form of $v \leq k$, where v is a variable and k is a constant) are performed, where the model caching strategy can be better suited for the analysis of a

program. The effect is that for example the model $v = 0$ continues to satisfy the constraints as k increases, whereas Green will not find any matches. Conversely Green will excel in constraints of the form $v = k$, since it does less computation compared to the model caching strategy.

Looking at Green's and Z3Fact's sat and unsat columns (which have identical results), these examples show that sometimes constraints are so syntactically different that not even canonisation can help to increase reuse. Z3Cache shows (sat and unsat columns of *list*, *old-tax* and *new-tax* being similar to Green and Z3Fact) that sometimes pre-processing does not make a difference to improve reuse. Although *ball* is the exception to this observation in this grouping, showing very little sat reuse.

The **Misc** grouping shows much of what has been discussed above, but has some interesting anomalies as well. For example Grulia and Julia differ on the sat case for *block* and *avl* because the models that Z3 return are different. Uncertainty remains after further investigation into what the cause could be (but it could be as simple as a small difference in the encoding of the constraint when sent to Z3), but it clearly indicates the results of reusing models is not very stable. Lastly the *block* example shows that neither technique works well, since many syntactically different constraints (not good for Green or Z3Fact) occur and they don't share solution spaces (not good for model reuse).

In conclusion reusing *unsat-cores* shows a clear edge when working with unsat constraints. However one might wonder how often will unsat constraints be encountered, especially ones where the *unsat-cores* are similar but these are not syntactically the same as one of the independent factors. This is explored in the next section.

Industrial Experiments

An expansive experiment is done by attaching Grulia, Green, Z3Fact and Z3Cache to SPF, to see how it performs on constraints that are generated during symbolic execution. Note that SPF checks the feasibility of a constraint at every branching point in the Java program being analysed, if it finds an infeasibility it doesn't consider exploring that execution path any further. The results are shown in Table 4.2. Each group is sorted according to the percentage of sat constraints present. The value is calculated by recording the number of input constraints (while still unprocessed) given to each tool. Note that some of the examples have the same names as in the Replication Experiment data set, but they are not the same. Here they refer to Java programs that implement a *TreeMap*, a car's breaking system (*WBS*), traffic collision avoidance system (*TCAS*) and an actual implementation of the Dijkstra algorithm (*Dijkstra*), and the constraints produced when doing a symbolic execution of the code. In

		Green		Grulia		Z3Fact		Z3Cache	
Program	sat%	sat	unsat	sat	unsat	sat	unsat	sat	unsat
WBS	100.0	99	0	99	0	99	0	0	0
Stack	100.0	99	0	99	0	99	0	0	0
FlapController	97.1	99	99	99	99	99	99	99	98
Strings	88.1	99	99	99	99	99	99	0	0
ObjectRec	59.0	99	99	99	99	99	99	0	0
CLI	53.6	99	99	99	93	99	98	-	-
Jadx	25.9	99	99	99	98	99	99	-	-
MagicIndex	100.0	89	0	97	0	89	0	0	0
Sorting	100.0	0	0	2	0	0	0	0	0
Remainder	85.1	13	0	37	33	1	0	0	0
Dijkstra	70.7	14	0	62	56	0	0	0	0
BubbleSort	66.8	26	0	0	1	0	0	0	0
Median	66.8	26	0	0	1	0	0	0	0
Operations	64.1	91	73	94	71	87	71	0	0
TreeMap	100.0	97	0	89	0	96	0	0	0
SortedListInt	84.1	99	97	89	94	99	97	0	0
BinTree	75.2	97	71	90	8	92	5	0	0
BinomialHeap	63.8	98	81	98	38	93	26	0	0
NanoXML	61.5	99	99	99	27	99	99	-	-
Triangle	50.9	95	89	97	51	87	87	0	0
TCAS	48.6	99	96	99	67	99	95	0	0
Flink	44.7	99	95	99	7	91	0	-	-
CoinChange	2.4	98	55	99	6	96	7	0	0

Table 4.2: Reuse rate (%) of solutions with SPF analysis.

the other data set it is just lists of constraints that in all likelihood came from doing a similar analysis, but not the same analysis.

The dashes (-) in Z3Cache are examples where the Green framework used an overburdening amount of memory causing the framework to crash and the analysis could not be completed. The subdivision of the benchmark results are done in three groupings:

Similar: The first seven examples show minimal differences in, yet great, reuse rates across the four tools.

ModelCores: The next grouping of seven (shaded) are examples where Grulia obtains better reuse than Green.

Semantics: The next nine (unshaded) are examples representing the cases where there are much better unsat reuse in Green than Grulia.

In the **Similar** grouping, Green and Grulia gets similar reuse in almost all cases. The Z3Fact gets similar results to the previous two in most cases. The grouping shows is nothing of interest, except a few outliers that will be discussed.

Z3Cache gets no reuse, except in *FlapController* where it achieves reuse remarkably close to that of Green. With reuse obtain within Z3Cache, that means the constraints generated must be identical, which is a strange phenomenon considering how symbolic execution generates constraints. Upon further inspection it shows that *FlapController* is a multi-threaded program with interleaving, meaning similar constraints will be encountered among the different threads. *SortedListInt* shows less reuse with Grulia, because compared to Green, due to constraints with similar structures the possible model is missed among the 127 289 entries in the sat cache showing that this strategy is not robust in all situations, although many constraints are still satisfied with one of the reference solutions.

One might also notice the 100% sat precedence in the *WBS* example in Table 4.2, where *wbs* in Table 4.1 has 59.8%. Even though the latter is based on an analysis of the real program (represented with the former), with the analysis of *WBS* the unsat constraints could not be produced. It is only worthy of note that it is two complete separate examples, therefore the difference despite the same name. The same is also true for *TreeMap*, *Dijktra* and *TCAS*.

The **ModelCores** grouping represents cases where the models and the *unsat-cores* seem like a useful strategy. Grulia shows greater reuse in the sat case of *MagicIndex*, *Sorting*, *Remainder* and *Dijkstra* where the constraints shared more models. Grulia further shows better unsat reuse in *Remainder* and *Dijkstra* where the constraints had common *unsat-cores*. These four programs are examples where models and *unsat-cores* definitely work well. Grulia also obtains better sat reuse in *Operations* than Green, but marginally worse unsat reuse. Although the example contains a higher percentage of sat constraints which attributes a greater value on Grulia from the sat reuse. The *Median* and *BubbleSort* examples show better sat reuse with Green and a little unsat reuse with Grulia.

From Table A.2 in Appendix A, one can see that no factorisation took place on the constraints[6] and that the constraints consist of many clauses, which indicates that there is difficulty to find common models in the two examples. Both examples have surprisingly similar results. Upon program inspection after the analysis, it is revealed that the *Median* program implements a bubble sort algorithm to determine the median.

The observation made during the previous experiment of when the model caching strategy might be better, fail on the sorting examples such as *Bubble-*

[6]Indicated by the ratio of 1.0 in the column with the number of factors over the number of constraints.

Sort and *Sorting*. The influencing factor here though, is the lack of factorisation[7] making it more difficult to share models.

The **Semantics** grouping shows less reuse in the unsat case with Grulia, because with the possible solution missed among the extracted *unsat-cores*. Grulia obtaining less unsat reuse goes against the assumption that *unsat-cores* will give better reuse since it is more probable that an *unsat-core* would be present in a constraint than another exact unsat factor. What counts for Green's benefit is the canonisation, transforming the unsat factors to look similar if they have the same structure. This indicates that a better approach for Grulia will be to rather look at all the possible *unsat-cores* for comparison when checking for shares.

Running the canoniser with Green the unsat reuse is boosted, for example with *BinTree* it is boosted from 8% to 44%. Z3Fact further shows the usefulness of the canoniser in the Green pipeline with the lower reuse compared to Green, specifically again with *BinTree*. It is also worthy to note, that although *Flink* shows about 44.7% sat queries, with the factorisation step they are split up numerously such that the example changes from unsat majority to sat majority with 90% sat constraints processed. *Triangle*, *TCAS* and *CoinChange* show marginally better sat reuse with Grulia, but again similar to *BinTree* shows weak unsat reuse.

Looking at the `sat%` constraints, surprisingly a large number of unsat constraints are present in some real-world programs. The takeaway from these examples though is that there are many more sat constraints than unsat. Which is inevitable during symbolic execution but it might not be true for other use-cases of constraint analysis.

Concolic Experiments

A comparative experiment is done by attaching Grulia, Green, Z3Fact and Z3Cache to Coastal, to see how it performs on constraints that are generated during concolic execution. Note that Coastal only makes a solver call for feasibility of a constraint at every leaf in the execution tree of the Java program being analysed, if it finds an infeasibility it doesn't consider exploring that execution path any further.

The result is shown in Table 4.3. The benchmark results are subdivided into three groupings according to their results:

Similar: The first four are examples presents nothing of interest where three of the four tools performed the same.

ModelCores: The next five (shaded) examples show better reuse rates with Grulia.

[7]Since no factorisation could take place the constraints remain long and complex.

		Green		Grulia		Z3Fact		Z3Cache	
Program	sat%	sat	unsat	sat	unsat	sat	unsat	sat	unsat
WBS	100.0	99	0	99	0	99	0	0	0
Stack	100.0	99	0	99	0	99	0	0	0
Remainder	74.7	0	0	0	0	0	0	0	0
ObjectRec	52.1	99	99	99	99	99	98	0	0
MagicIndex	100.0	77	0	91	0	77	0	0	0
Sorting	100.0	0	0	27	0	0	0	0	0
SortedListInt	79.8	93	80	95	91	92	80	0	0
BinTree	50.4	97	71	94	99	95	5	0	0
CoinChange	15.8	96	83	99	83	80	0	33	0
BinomialHeap	39.6	99	87	99	64	98	40	0	0
Operations	28.3	90	73	87	38	86	71	0	0
BubbleSort	6.2	0	41	30	4	0	41	0	0
Triangle	1.7	0	89	42	20	0	87	0	0

Table 4.3: Reuse rate (%) of solutions with Coastal analysis.

Semantics: The next grouping of four (unshaded) are examples where there are better unsat reuse in Green than the model caching tool.

Each group is sorted according to the percentage of sat constraints present. The value is calculated by recording the number of input constraints (while still unprocessed) given to each tool.

In the **Similar** grouping the first observation is that Green, Grulia and Z3Fact performed similar and secondly that Z3Cache obtained no reuse. One outlier is Z3Fact that shows marginal benefit of the canonisation step since Green obtained better unsat reuse in the *ObjectRec* example. In this grouping there is nothing of interest to further discuss.

In the **ModelCores** grouping Grulia shows better reuse in both the sat and unsat case. *MagicIndex* and *Sorting* are two examples where reusing models show an advantage over Green's sat/unsat answers. *SortedIntList* and *BinTree* are two examples where Grulia obtains better unsat reuse compared to the other tools. These four examples correspond with the same trend as in the previous experiments, regarding models and *unsat-cores*. Another outlier is *CoinChange* that shows 33% reuse in the sat case of Z3Cache, signalling that some constraints were exact matches without pre-processing.

In the **Semantics** grouping Green shows greater unsat reuse compared to the *unsat-core* reuse. *BubbleSort* and *Triangle* show greater sat reuse with Grulia than Green, but a significant smaller amount of unsat reuse. Another observation is that Z3Fact performs close to Green in this grouping as well,

Program	Green	Grulia	Z3Fact
version1T	0	0	0
version1F	6	99	0
version2	0	80	0
version3T	0	0	0
version3F	0	85	0
version4T	0	0	0
version4F	99	99	9

Table 4.4: Reuse rate (%) of solutions of the generated constraints.

casting doubt on the effectiveness of the canoniser, except for the unsat case of *BinomialHeap*.

Generated Experiments

The program labels in Table 4.4 indicate whether the dependency is enabled (T) or disabled (F) for each version. Version 2 has the random condition which means there is only one case. Note in Table 4.4 Grulia obtains high reuse in all instances where there is no dependency placed on clauses, which shows the strength of reusing models. Version 2 with the high reuse is odd because there is a high probability of dependent clauses. Running the same version with a higher amount of variables and greater value of `max` shows a more realistic low average of reuse. Looking at the column of Z3Fact surprisingly the hypothesis of only using factorisation on constraints fails on these generated constraints. In version4F, Green shows great reuse and comparing with only the factoriser shows the benefit of the canonisation step.

4.3 Efficiency

The previous section shows that model reuse is a good alternative option for reusing satisfiability results, but equally important is that it must be faster than just redoing the work. In other words it must be faster than for example just rerunning the constraint solver. Furthermore all the tools actually show really good reuse, but are they faster than the solver? Therefore the running time is consider in this section, which measure only the solving time of a tool (that is, the time the tool took to process all the constraints) which means the time overhead of the entire analysis is excluded. As an attempt to obtain reasonably sound timing results, each tool is run ten times on all the data sets. The two outliers of the runs (the run with the maximum running time and the run with the minimum running time) are removed before taking the average of the results.

Program	#cstrs	$\frac{\#\text{factors}}{\#\text{cstrs}}$	Z3Java (ms)	Green	Grulia	Julia	Z3Fact	Z3Cache
treemap	332 950	6.3	340199	0.122	0.394	3.387	0.070	0.071
diskperf	103 505	27.4	252772	0.112	0.103	0.246	0.057	0.992
grep	100 126	46.9	1191256	0.106	0.116	0.255	0.057	0.990
floppy	100 006	16.5	149513	0.118	0.111	0.206	0.062	1.537
cdaudio	55 329	12.4	78348	0.129	0.153	0.322	0.068	1.006
reverseword†	38 104	8.0	27393	0.072	0.087	0.115	0.056	0.051
multiplication†	25 217	1.0	19097	1.718	0.312	0.225	1.113	1.008
tcas	13 476	9.7	18205	0.079	0.119	0.180	0.062	0.692
avl	11 161	2.2	17971	0.216	0.423	2.187	0.156	0.344
knapsack†	7 651	1.0	215829	0.930	0.551	0.180	0.601	0.479
collision	6 812	4.2	5770	0.104	0.153	0.203	0.084	0.273
division†	1 257	1.0	13834	1.157	0.385	0.198	1.065	0.957
list	876	1.0	268	0.284	0.500	0.403	0.205	0.153
block	505	1.0	426	0.798	2.873	1.714	0.737	0.660
wbs	239	5.7	110	0.155	1.118	0.300	0.118	0.891
ball	210	2.0	256	0.406	0.789	0.246	0.289	0.984
afs	203	16.2	390	0.164	0.197	0.290	0.121	0.264
kbfiltr	188	4.4	257	0.891	0.755	0.132	0.728	1.016
swapwords†	173	1.0	2203	0.928	0.371	0.144	0.756	0.756
dijkstra	85	22.7	1052	0.693	0.608	0.638	0.619	1.067
new-tax	55	1.0	13	1.462	5.231	1.231	0.769	0.692
old-tax	43	1.0	11	1.455	3.364	1.364	3.364	0.727

† Majority unsat constraints.

Table 4.5: Running times (normalised) of replication data set.

Replication Experiments

Table 4.5 shows the number of input constraints per example under the column `#cstrs`. The number of constraints are indicated to give some insight on the running times. This can be looked at in conjunction with the `sat%` column in Table 4.1. The following column indicates the average number of factors per constraint. The Z3Java column shows the running time of the solver. The experiment of `TABLE IV` in the paper of Aquino *et al.* (2017) (since it is more detailed than the recent paper) is redone with the latest Green, Grulia and Julia, which are displayed in the next three columns, where each entry shows the ratio over the running time of Z3Java. The column Z3Fact shows the ratio of running factoriser and Z3Java, over Z3Java alone. The last column Z3Cache shows the ratio of running Z3Java with storage. Note the best timings (i.e. lower ratios) of Green, Grulia and Julia are highlighted in a lighter shade, and separately Z3Fact and Z3Cache in a darker shade when one of them are the fastest. Any case in which the best timing has a ratio greater than 1 indicates that Z3Java by itself is the fastest. In Z3Fact, Z3Cache, Green and Grulia the tools use the Green framework's Z3 with Java bindings, whereas Julia does not make use of any Java bindings to interface with Z3.

Looking at the results, the first observation is that Grulia runs longer than Green in 14 out of the 22 cases, with most of the times where Grulia is faster is those that obtained high unsat reuse. Julia performs better than Green in 9 cases (out of 22). As before with the reuse results it is not surprising to see that Julia performs better in the unsat cases. Grulia runs faster than Julia in half of the examples.

However, what is much more striking about these results are the performance of Z3Fact and Z3Cache. Z3Fact is the fastest in 10 out of the 22 cases, with Z3Cache accounting for another 5 cases. That means that out of the 22 examples the scenario is that either doing nothing or just splitting the constraint up into independent factors being faster in 15 out of the 22 cases. The only exceptions are some of the unsat cases, *kbfiltr* and *ball*.

Note *old-tax* and *new-tax* look like significant difference with Z3Fact vs Z3Cache, but the differences are only mere milliseconds, both completed in less than 1 second. Similarly for *reverseword* there is a 200 ms difference between Z3Fact and Z3Cache. In *swapwords* both have the same running time, because of the nature of the constraints they do not produce any factors.

Between Z3Fact and Green are only 2 examples slower with Z3Fact (*division* and *old-tax*). Between Z3Fact and Julia are only in 7 examples slower with Z3Fact. Between Z3Fact and Z3Cache are only 7 examples slower with Z3Fact out of 22. This shows Z3Fact is a good compromise between caching and pre-processing of constraints.

The *avl* example shows great difference in running time among Grulia and Julia. The case being that the example runs less then 120 ms, which makes it more difficult to distinctly compare the tools. The *block* example showed

poor reuse among the tools and it can be seen across the model caching tools Grulia and Julia – the case being varied ratio but slow running time due to long waiting time for model solution from the solver. The *multiplication* example shows slow running time with Green due to bad unsat reuse and better performance among the model caching tools. The *wbs* example displays varied ratio with Grulia being the slowest and Julia the fastest and Green second. With Grulia close to 99% of the time is spent waiting for the solution from the solver, the same case happens for *avl, ball, block, dijkstra, kbfiltr, list, old-tax, new-tax* and *treemap*, where Julia spent much less time waiting for the solver solution. The other significant outlier is *treemap* where Julia spent most of the time computing the *sat-delta* check where Grulia computed this much quicker.

Further analysis reveals that the average number of factors per constraint can be a predictor of how well the factoriser, and by implication Z3Fact, can perform. In most cases where the average is larger than 4.0, like with the first six examples, Z3Fact outperforms the other tools. A few outliers to this trend are present, for example *dijkstra* having about 22.7 factors per constraint where Z3Fact is not the winner in the example, but still it performs close to the winner. In the other examples where the average is 1.0 Z3Fact performs noticeably slower.

The replication results do not correspond with that of Aquino *et al.* (2017). The authors of this thesis conjecture that this might be due to running an older version of Z3 and/or an older version of Green. Their results can therefore not be reconciled with the results obtained in this study.

Industrial Experiments

Table 4.6 shows the number of input constraints per example under the column `#cstrs`. The number of constraints are indicated to give some insight on the running times. This can be looked at in conjunction with the `sat%` column in Table 4.2. The rest of the columns shows the ratio of time of that tool taken over Z3Inc on its own (Z3 with incremental mode). The Z3 column shows running time of the solver in basic mode, for comparison with the incremental mode's speed. The next six columns shows the running time of the three main tools, Green, Grulia and Z3Fact, all three with a persistent cache and therefore a second run. The last two columns are a sanity check, with Z3Cache and its persistent storage and second run. The second run refers to the prepared cache from the first run, to display any improvement if at all for reuse across runs.

Note that the best timings (i.e. lower ratios) of Green, Grulia, Z3Fact and Z3Cache are highlighted in a darker shade. Any case in which the timing has a ratio greater than 1 indicates Z3 incremental mode by itself is the fastest. As a baseline for this experiment, Z3 incremental mode is run on command-line (outside of Green, but inside SPF). Grulia and Green has the same settings as with the replication run, except that they additionally have a persistent

Program	#cstrs	$\frac{\#\text{factors}}{\#\text{cstrs}}$	Z3Inc (ms)	Z3	Green	Run 2	Grulia	Run 2	Z3Fact	Run 2	Z3Cache	Run 2
NanoXML	871 580	5.9	185 186	4.920	0.893	0.898	1.251	1.236	0.422	0.416	-	-
Jadx	658 475	3.0	496 531	1.519	0.413	0.412	0.490	0.487	0.229	0.224	-	-
Strings	557 838	14.4	117 000	4.262	0.548	0.547	0.567	0.573	0.275	0.266	6.866	0.564
SortedListInt	340 114	3.9	76 804	3.883	0.588	0.518	7.458	10.605	0.312	0.251	6.190	0.473
ObjectRec	282 088	10.5	31 247	5.359	1.080	1.078	0.992	1.011	0.617	0.601	9.493	1.417
Stack	131 070	11.3	30 718	3.234	0.444	0.452	0.414	0.403	0.231	0.222	5.349	0.483
WBS	27 646	8.2	6 231	2.971	0.547	0.544	0.528	0.503	0.324	0.316	5.049	0.608
FlapController	14 860	2.0	3 722	1.593	0.271	0.266	0.307	0.259	0.173	0.162	0.110	0.114
TreeMap	151 944	6.9	42 804	3.328	1.253	0.582	11.367	11.988	1.021	0.338	5.171	0.413
Median	103 950	1.0	19 968	3.788	6.723	1.161	30.242	31.066	5.820	0.784	5.609	0.595
BubbleSort	103 950	1.0	20 049	3.702	6.596	1.147	30.274	30.024	5.880	0.767	5.606	0.592
Sorting	80 638	1.0	20 867	3.288	6.483	1.005	21.877	21.552	4.945	0.597	4.785	0.445
BinomialHeap	47 460	7.7	7 864	4.043	1.721	1.001	8.813	8.688	3.530	0.934	7.161	0.804
BinTree	15 226	5.9	3 250	3.290	1.634	0.932	14.884	15.643	2.496	0.820	4.859	0.741
Dijkstra	12 512	1.0	3 820	3.173	5.231	1.293	8.695	4.994	4.485	0.854	4.422	0.605
MagicIndex	7 200	9.9	2 157	3.378	2.235	1.235	2.975	0.610	1.736	0.834	4.012	0.688
TCAS	4 390	9.6	796	4.505	1.494	1.456	4.165	2.795	1.030	0.954	6.574	1.366
CLI	364 492	4.9	47 967	6.668	2.440	2.441	3.509	3.403	1.568	1.461	-	-
CoinChange	23 682	2.0	1 482	3.374	4.656	2.250	41.203	27.920	7.938	2.315	8.127	2.005
Operations	15 618	2.0	3 689	10.605	6.822	6.150	12.154	9.857	6.074	5.166	16.421	2.150
Triangle	2 206	1.0	434	4.627	1.776	1.475	6.657	8.145	1.399	1.184	4.604	1.530
Flink	1 020	7.1	779	16.116	7.356	7.433	8.552	7.651	5.633	4.755	-	-
Remainder	956	1.0	718	6.019	55.901	52.567	67.848	58.479	58.242	51.156	10.794	3.708

Table 4.6: Running times (normalised) of SPF analysis on programs.

storage enabled. Similar to the previous experiment, Z3Fact is run with only a factoriser service, and additionally with a persistent storage and a Z3 service in the pipeline. In Z3Fact, Z3Cache, Green and Grulia the programs use the Green framework's Z3 with Java bindings.

The process of this experiment started with Green that performed significantly slow. The in-memory storage enhancement to Green improved the running time of the tool. The question then arose, are any of these caching tools faster than one of the fastest solvers? Therefore the experiment moved away from Z3 (basic mode) as baseline to rather use Z3Inc for comparison. The picture changed significantly, showing that the tools are outperformed by Z3Inc in the examples from *TreeMap* to *Remainder* (that is 15 examples). Therefore the persistent storage was activated for the different tools to measure the running time reuse across runs (indicated by the columns *Run 2*) to evaluate the relevancy of the caching tools.

Table 4.6 is sorted into three categories:

Intra-run: The first eight examples where the first run of one of the tools beats Z3Inc.

Across-runs: The next grouping of nine (shaded) are examples where only the second run beats Z3Inc.

Misc: The next six (unshaded) are examples representing the cases where almost none of the tools beat Z3Inc.

Each category is sorted from most to least number of constraints obtained from the analysis. Keep in mind that the rest of the table show normalised values, even though a large number of constraints are evaluated and the running time is quite long, the normalised value can be small, for example in the case of *Jadx*. The *Median* result will be tied in the discussion with *BubbleSort* only referring to the latter, since *Median* uses the same bubble sort algorithm and obtain similar results.

The **Intra-run** grouping focuses on the first runs from the caching tools that were the fastest. The second run being faster in this grouping should be a given although a few outliers are present which will be discussed later. Z3Fact is the fastest in all examples of this run, except for *FlapController* which achieved similar reuse to Z3Cache yet is slower due to the overhead of the factoriser. Do note the interesting scenario where *Jadx* gives many constraints for evaluation whereupon Green and Grulia is faster than Z3Inc, but a smaller example like *BubbleSort* they are significantly slower than Z3Inc due to no factorisation. Further inspection for *Jadx* shows that 26% of the Grulia service running time is spent waiting for the solver solutions, 20% (30 seconds) is the *sat-delta* computation, 35% is checking shared models and 12% of the time is checking for shared *unsat-cores*. The breakdown is noted because

in other examples the other components take negligible time consumption and most of the service running time is spent waiting for the solver solutions. Other outliers are *SortedListInt* and *ObjectRec* where there is slower running time in the second run. The case here shows one of the weaknesses of the Julia algorithm where the cache is populated with many solutions (with *sat-delta* values in close proximity) but the viable solutions are not found, therefore resorting to solver calls. With *Jadx* and *ObjectRec* Green performs quite close to the second run, which is ascribed to the few solver calls that are made in the first run and the rest of the running time is spent on the cache especially the communication with the persistent store in the second run. Recall that Green follows the greedy approach with the persistent store, meaning in the worst-case a call is made to Redis for each constraint to obtain the solution. This grouping shows that reuse helps in 8 of 23 examples.

In the **Across-runs** grouping encapsulate programs that produce constraints that are structurally similar, resolving to high reuse and fast analysis upon a second run. Furthermore cases like *Dijkstra* and *BubbleSort* obtained no or little reuse, yet the analysis ran faster with Green compared to Grulia. The outliers of Grulia and large ratios such as *TreeMap* and *BubbleSort* will be discussed later, because it is a greater overarching phenomenon. The grouping shows that reuse across runs helps in 17 of 23 examples.

In the **Misc** grouping Z3Fact is still the fastest in the second run among the caching tools, although significantly slower than Z3Inc. *CLI* is an example where Grulia spends 50% of the service running time waiting for solver solutions, and the other significant time consuming components are checking for shared models and *unsat-cores*. With *CoinChange* both Z3Fact and Z3Cache runs slower than Green in the first run but have a close running time in the second run. Green runs faster than Z3Fact and Z3Cache in the first run of *CoinChange*, which makes sense since it got better unsat reuse on an example that has majority unsat constraints. *Remainder* is an example where pre-processing of constraints are a hindrance and it can be better simply ignoring it and rather use a cache only. The group portrays, with 6 of 23, examples that sometimes caching and reuse do not help and can simply run the solver alone.

Overarching observations: Z3Cache shows arguably that pre-processing might be unnecessary if one works with constraint reuse across runs. Z3Cache is always slower than the other tools in the first run over all examples except for *Remainder*. In the first two groupings Z3Cache comes close to Z3Fact in the second run, and in the last grouping it is faster than Green in the second run. Z3 is in most cases slower than Z3Inc, except for small examples, yet

in some cases basic Z3 is still faster than some of the tools in a few cases for example *BubbleSort*, *CoinChange* and *Remainder*.

Again the column with average number of factors server as a predictor, hinting that with an average greater than 3.0 Z3Fact will perform well.

Although big examples (not only in number of constraints, but more so in the number of clauses[8] as well) do pose a long running time and out of memory issue (as indicated by the dashes (-) in the table). In most cases (for example *TreeMap*, *BubbleSort*, *BinTree*, *Dijkstra*, *CoinChange* and *Remainder*) 93% of Grulia service running time is spent waiting for model solutions from Z3 (except for the named exceptions above). The time spent waiting for model solutions (the solver calls to produce a model) is greater than the solver call time to produce the simple sat/unsat solution. Other exceptions include *Strings*, *ObjectRec* and *FlapController* where less than 10% of the time consumption is taken by the solver and a greater amount of time is taken up by the *sat-delta* computation and Grulia store extraction and checking shared solutions. One definite improvement can be made to Grulia to have a hybrid system with Green, having a hash storage to check the solution before spending time to compute the *sat-delta*, and also will resolve missing solutions in the Grulia store.

Concolic Experiments

The Industrial Experiments show the sweet spot for the solver, where the constraints are in the quantifier free integer domain and only produce a single value solution (sat/unsat). The experiment showed that the caching tools have difficulty to keep up with performance. What if the solution type changed to something more difficult? How will the picture change of the caching tools' benefit? Therefore the running time of a concolic analysis are taken into consideration where the solutions to compute consist of models (a more expensive computation).

Table 4.7 is sorted according to the cases where Grulia has the quickest running time, then Green, followed by Z3Fact and lastly Z3Cache, with each grouping sorted according to the number of constraints produced. The normalised values are obtained by taking the running time of the tool divided by the running time of Z3.

CoinChange, *Stack*, *Sorting*, *SortedListInt* and *MagicIndex* are examples where Grulia obtained better reuse in Table 4.3 and a corresponding better performance in Table 4.7. This observation contradicts the SPF results where Grulia obtained great reuse and yet performed sub-par in running time. Threat to validity involves concern that the programs of Table 4.7 might be set up in such a manner generating too little constraints that the overhead of Grulia is not fully exposed.

[8]See Table A.2 for the average number of clauses per constraint.

Program	#cstrs	$\frac{\#\text{factors}}{\#\text{cstrs}}$	Z3 (ms)	Green	Grulia	Z3Fact	Z3Cache
CoinChange	95 436	5.6	4025	0.084	0.068	0.217	0.104
Stack	2 046	6.8	7170	0.098	0.068	0.083	1.134
Sorting	1 438	1.0	35751	0.177	0.105	0.157	0.160
SortedListInt	694	2.6	13437	0.059	0.037	0.058	0.209
MagicIndex	400	5.8	9320	0.234	0.010	0.173	0.150
BinTree	15 226	5.9	383259	0.043	0.087	0.096	0.104
BinomialHeap	7 182	6.5	239652	0.018	0.036	0.041	0.092
Triangle	2 206	1.0	74757	0.017	0.088	0.017	0.082
ObjectRec	1 654	5.6	5046	0.102	0.232	0.217	1.487
BubbleSort	23 262	1.0	493217	0.080	0.129	0.058	0.087
Operations	15 618	2.0	340378	0.072	0.164	0.066	0.188
WBS	1 150	5.2	45785	0.011	0.020	0.010	0.102
Remainder	1 146	1.0	946068	0.010	0.027	0.011	0.009

Table 4.7: Running times (normalised) of Coastal analysis on programs.

The model and *unsat-core* reuse shows an improvement when doing concolic analysis. Recall that from the SPF analysis it is noticed that the model calls are more expensive than simple sat/unsat calls, causing Grulia to be slower than the other tools. In this experiment all the other tools also make solver calls requesting models which further contributes to Grulia not under-performing in comparison of execution time with the concolic analysis.

Another observation is that once again Z3Fact competes with the other tools in performance, displaying the strength of factorisation. Lastly the *Remainder* example showed no reuse (from Table 4.3) along with no factorisation which is reflected with Z3Cache. In this example Z3Cache shows that preprocessing was unnecessary (and actually costs extra) whereby it was faster simply passing the constraint to the solver as is and then storing the solution.

Comparing the Coastal analysis with that of SPF, the caching tools show much improvement in execution time, mainly because the solver takes a long time to compute the model solutions. The model solver calls are more expensive than the sat/unsat calls, which is further substantiated by comparing the Z3 running time in Table 4.7 with Z3Inc running time in Table B.2.

Generated Experiments

From Table 4.8 Grulia's running time is incongruous, that is it runs significantly slower than the other two configurations in most cases. Grulia obtains more reuse than Green in version2, but still runs a few milliseconds slower than Green. Another outlier is version3T with the dependency enabled where Grulia crashes due to an out of memory issue. It will be recalled that version3T is where all the constraints are chained and therefore no factorisation can take place. A strange phenomenon appears where even though both Green

Program	Green	Grulia	Z3Fact
version1T	27 931	242 684	27 521
version1F	7 804	1 769	7 379
version2	3 006	3 246	2 707
version3T	30 016	-	93 093
version3F	8 799	340 170	7 550
version4T	32 857	283 556	32 692
version4F	791	1 281	10 684

Table 4.8: Tool performance (in ms) on generated constraints.

and Z3Fact get no reuse, Green with the canoniser runs faster than just with the factoriser. The general trend shows that the dependent constraints are solved slower than the constraints with independent clauses, showing that dependent clauses are more difficult to solve. Another jarring outlier is version3F where Grulia still runs significantly slower compared to the other two tools, this further shows how difficult it is to obtain a model for chained clauses.

Upon closer inspection with all the slow cases of Grulia it shows that it is not the caching strategy of Grulia that displays poor performance but rather most (between 78% and 98%)[9] of the running time is spent waiting for a solution from Z3. Comparing this to Green that gets no reuse in most cases and still runs faster than Grulia, further shows that it is cheaper to calculate simple sat/unsat solutions compared to models/*unsat-cores*. The long duration of model computation is clearly visible in these cases due to the constraints consisting of a large amount of clauses. Additionally it is noticeable that in all cases where the dependency is enabled the program runs slower.

These large generated constraints differ from the real-world examples displayed in the previous experiment, since most often the constraints contain a few clauses and in specific instances where the constraints do contain many clauses it is still less than those generated in this experiment.

Factorisation

The factorisation experiment is composed with the same environment as the Industrial Experiments (Section 4.1), comparing the original factoriser service in Green with the new service using the new Union-Find algorithm.

The factorisation effects between the two algorithms are the same in terms of producing equivalent number of factors from the analysis. Only the running time will be inspected to see if the enhanced service offers improved running

[9]Looking from version1T to version4T, with only version4F being an exception showing between 20% and 70% time spent in the solver. It is a significant difference of 50% because working with an example that takes only a few milliseconds to execute (from about 400 ms to about 1 200 ms).

Program	#cstrs	$\frac{\#\text{factors}}{\#\text{cstrs}}$	$\frac{\#\text{clauses}}{\#\text{cstrs}}$	OLD	NEW
NanoXML	871 580	5.9	40.7	126 004	65 974
Jadx	658 475	3.0	66.8	180 597	91 551
Strings	557 838	14.4	17.7	24 247	13 254
CLI	364 492	4.9	52.2	85 450	57 811
SortedListInt	340 114	3.9	18.9	24 575	14 143
ObjectRec	282 088	10.5	20.2	18 225	13 443
TreeMap	151 944	6.9	19.1	12 197	199
Stack	131 070	11.3	15.0	4 833	151
BubbleSort	103 950	1.0	15.4	9 832	5 434
Median	103 950	1.0	15.4	9 666	5 339
Sorting	80 638	1.0	18.3	9 260	159
BinomialHeap	47 460	7.7	19.9	4 560	452
WBS	27 646	8.2	15.0	1 560	947
CoinChange	23 682	2.0	9.0	40 313	1 161
Operations	15 618	2.0	40.9	17 395	17 953
BinTree	15 226	5.9	16.6	1 523	872
FlapController	14 860	2.0	3.1	302	306
Dijkstra	12 512	1.0	16.4	1 766	1 351
MagicIndex	7 200	9.9	19.2	731	430
TCAS	4 390	9.6	13.5	320	291
Triangle	2 206	1.0	5.9	157	247
Flink	1 020	7.1	222.3	4 676	3 023
Remainder	956	1.0	47.1	16 931	42 781
TOTAL	3 818 915	-	-	595 120	337 272

Table 4.9: Factoriser performance (in ms) on real-world examples with SPF.

time. Looking at Table 4.9 (sorted according to the number of constraints) one would assume that the run-time columns should decrease as the number of constraints decrease, which is not the case. Possibly the size of the constraints might play a role, which is indicated by the column with the number of clauses per constraint, but there is no definite correlation between this column and the run-time columns as well (comparing for example *Flink*, *BubbleSort* and *TreeMap*).

Taking a closer look at Table 4.9 one can see in most cases the new technique shows great improvement. The difference in execution time is especially highlighted by the *TOTAL* row indicating about 56.6% increase in performance. Further observation shows that *TreeMap*, *Stack*, *Sorting*, *BinomialHeap* and *CoinChange* are 5 examples where the new algorithm performs phenomenal. Contrary to this are *Operations*, *FlapController*, *Triangle* and *Remainder* which are 4 examples where the new algorithm performs slower.

The other 14 out of 23 examples stand as markers for better performance with the new algorithm.

On some small examples such as *TCAS* and *Triangle* the difference is less clear, since the overhead of the old algorithm is not present. Interestingly comparing *ObjectRec* (where there are many constraints) with *Operations* (where there are fewer constraints) it is noticeable that the factorisers ran longer on the latter example. Also with *Operations* it is of note that the new algorithm runs slower than the old one, but compare it with *CoinChange* (having the same ratio of factors) there is a major difference in performance between the two factorisers. Similarly with *FlapController*, having the same ratio of factors and a smaller ratio of clauses, it shows faster execution than the previous two examples.

The two columns indicating the ratio of factors and the ratio of clauses are added to the table to assist in determining if these characteristics indicate some trend in execution time differences, but on contrary no clear trend is visible. Therefore the efficiency of the algorithms are probably influenced by something else hidden in the nature of the constraints, for instance *Remainder* is an example where the new algorithm performs about 40% slower than the original. Additionally with the *Remainder* example the old algorithm also ran slower on this small example compared to other examples that show similar characteristics based on the abstract composition. The clear understanding of this phenomenon is outside the scope of this thesis and is left for future work.

4.4 Summary

The evaluation has looked at a number of different experiments and scenarios, which reveal insight into a few caching strategies.

As a replication study, the implementation of Grulia is deemed successful since similar reuse results were obtained. Looking at the running times, further optimisation of Grulia is needed to improve the execution time, although much of the time is actually spent waiting for the model solutions from the solver. This is further backed by the concolic analysis where Grulia was faster for almost half of the examples. There are examples where reusing models do show an advantage, depending on the nature of the constraints. Although model computation is more expensive than the sat/unsat call, Grulia performs better for the concolic cases. Additionally reusing *unsat-cores* shows it definitely adds benefit to a caching strategy for greater reuse.

Large constraints with many dependent clauses are not good for the caching tools, mainly because factorisation cannot take place. Factorisation is the key difference in all the pre-processing to make reuse more effective. The example of *Remainder* displays a different nature of constraints where the caching tools struggle, and the solvers find the solutions much easier. Therefore further

research can be beneficial to draw on certain techniques from solvers for the caching tools to make the processing of constraints easier.

Caching tools show distinct relevance with concolic analysis, whereas with symbolic execution solver performance is keeping up and a caching tool is more useful with a prepared cache.

Chapter 5

Conclusion

This thesis set out to explore certain research questions, which will be addressed below.

Which of the popular caching frameworks seem best suited for analysis of programs during symbolic/concolic execution?

The thesis replicated the results of Aquino *et al.* (2017), but the work has produced somewhat different insights. One part of the explanation for the run time discrepancies is that the replication might not be faithful enough. In particular, the simple implementation of *unsat-cores* in Grulia produces differences in its run time, compared to Julia. Splitting the reuse results into the sat and unsat cases reveals more insight into the different strategies. Although the reuse in the sat cases (between the two main caching strategies) are quite similar, some exceptions (such as constraints with only changing constants) exist where the model reuse prevails. In the unsat case the results show the dramatic impact of reusing *unsat-cores* over unsat factors. Therefore showing that this enhancement should be added to the basic Green pipeline.

What is the relevancy of caching frameworks like Green or Julia with the increase of solver performance?

Z3's incremental mode displays greater performance over Z3 in basic mode, but factorisation with storage still outperforms the former. Therefore maybe incremental solving with a cache might be a good compromise. One of the great benefits of Green is to have persistent storage for reuse across runs. Therefore one can look at adding such a persistent store with the incremental solver,

similar to the Z3Cache in the experiments. The first run of an analysis will be slow, since all the solutions need to be calculated and stored. The second run of the analysis will almost only contain the overhead of the storage, since there is no other over-head such as pre-processing of the constraints. Although the second run will show greater benefit if the same program is re-analysed (as shown by Z3Cache in the experiments) compared to reuse across runs with different programs, since the constraints are stored at the highest level. It is not surprising that, as SMT solvers continue to evolve, the improvement that systems such as Green/Grulia and Julia add, decreases. Nevertheless, they bring their own advantages (such as caching across runs and tools, and support for services other than satisfiability).

What is the impact of pre-processing, or specifically factorisation (where constraints are split into independent parts), of constraints on solving and solution caching?

It is interesting to note how well Z3 and Z3Fact perform. In fact, apart from a small (noisy) exception, the combination of factorisation and straightforward invocation of Z3 appears to be the optimal approach as long as the number of unsatisfiable constraints is low. In cases where most constraints are expected to be satisfiable, this would be a good approach to take. Z3Fact performed much better when the constraints were of such a nature to produce numerous factors. Z3Fact further showed only a few cases where canonisation might be good. The more interesting result with Z3Fact is that factorisation is shown to be the cornerstone for the caching strategies to perform well. The results show that the role of operations such as canonisation, and other "advanced" techniques, might need to be reconsidered.

What difference emerges between caching for symbolic and concolic analyses?

Symbolic analysis displayed the sweet spot for the solver, where the constraints are in the quantifier free integer domain and only produce a single value solution (sat/unsat). The experiments showed that the caching tools have difficulty to keep up with performance when the incremental solver is involved. The difficulty of concolic execution is that it is not that obvious how to implement an incremental solver for the analysis. The solver in its basic mode performed immensely slow, therefore the caching tools display a greater relevancy. The model-core reuse strategy shows advantage over the sat/unsat alternative. For concolic analysis a model is required for the constraint, which is more expensive to compute, therefore the caching strategies (and more so the model-core approach) improved the analysis run time.

Conclusions to highlight from the experiments are:

- *sat-delta* beats canonisation for constraints of the form: $v \leq k$,
- factorisation is the main constraint pre-processing step that makes any of the caching strategies effective,
- reusing *unsat-cores* definitely adds benefit to a caching strategy for greater reuse,
- caching tools show distinct relevance with concolic analysis, whereas with symbolic execution solver performance is keeping up and a caching tool is more useful with a prepared cache.

Future work includes analysing improved and much larger (in terms of analysis size and variety of programs) benchmarks for the concolic execution.

Regardless, future research could continue to explore alterations to Grulia where there is more of a hybrid Green storage mechanism involved. For Grulia it can be to activate the Green caching layers that were disabled for the replication and experiments.

In addition, a third scenario in terms of caching and reuse might prove a useful area for future research, where the tools run with a semi-filled cache. The third scenario would create a more realistic environment in the sense of the user analysing different programs in one execution, and thus reuse and caching would be of great benefit. The actual impact of this scenario is more difficult to approach and determine, since for example one influencing factor among many is the order in which one run the programs for analysis and populate the cache.

Further work is certainly required to disentangle these complexities in the examples where the caching tools and specifically the factorisation performs poorly, such as the example of *Remainder*.

In Green's current state the framework still cannot solve non-linear constraints, and therefore it would be interesting to see how the different caching strategies compare on such constraints.

Take home message, if you plan to do a single analysis (with symbolic execution), it can be faster to just run Z3 incremental mode. If you want to analyse a program or different programs multiple times, it will definitely still be useful to run a persistent storage. Regarding concolic execution, solution caching still offers great benefit and it is up to the user to decide whether classical caching is the approach or model-cores will be better suited for the analysis. Despite all that said, at the very least, the recommendation will be to have a factorisation step to reduce the constraint size – this gives the best trade-off between effective reuse and not too much extra computation and time consumption.

Appendices

Appendix A

Constraint Details

A.1 Replication Experiments

Program	#cstrs	sat%	$\frac{\#\text{factors}}{\#\text{cstrs}}$	$\frac{\#\text{clauses}}{\#\text{cstrs}}$
grep	100 126	99.9	46.9	212.7
diskperf	103 505	96.6	27.4	41.3
dijkstra	85	50.6	22.7	85.5
floppy	100 006	100.0	16.5	27.1
afs	203	100.0	16.2	39.4
cdaudio	55 329	85.5	12.4	30.1
tcas	13 476	2.2	9.7	18.8
reverseword	38 104	0.2	8.0	18.0
treemap	332 950	51.2	6.3	20.3
wbs	239	59.8	5.7	11.2
kbfiltr	188	93.6	4.4	9.5
collision	6 812	22.4	4.2	11.7
avl	11 161	93.4	2.2	23.6
ball	210	100.0	2.0	6.0
knapsack	7 651	0.0	1.0	521.1
division	1 257	0.1	1.0	183.7
swapwords	173	0.0	1.0	175.5
multiplication	25 217	0.0	1.0	34.6
block	505	100.0	1.0	7.7
list	876	78.3	1.0	6.1
new-tax	55	67.3	1.0	5.6
old-tax	43	67.4	1.0	4.6

Table A.1: Constraints obtained from the replication data set.

A.2 Industrial Experiments

Program	#cstrs	sat%	$\frac{\#\text{factors}}{\#\text{cstrs}}$	$\frac{\#\text{clauses}}{\#\text{cstrs}}$
Strings	557 838	88.1	14.4	17.7
Stack	131 070	100.0	11.3	15.0
ObjectRec	282 088	59.0	10.5	20.2
Magic	7 200	100.0	9.9	19.2
TCAS	4 390	48.6	9.6	13.5
WBS	27 646	100.0	8.2	15.0
BinomialHeap	47 460	63.8	7.7	19.9
Flink	1 020	44.7	7.1	222.3
TreeMap	151 944	100.0	6.9	19.1
NanoXML	871 580	61.5	5.9	40.7
BinTree	15 226	75.2	5.9	16.6
CLI	364 492	53.6	4.9	52.2
SortedListInt	340 114	84.1	3.9	18.9
Jadx	658 475	25.9	3.0	66.8
Operations	15 618	64.1	2.0	40.9
CoinChange	23 682	2.4	2.0	9.0
FlapController	14 860	97.1	2.0	3.1
Remainder	956	85.1	1.0	47.1
Sorting	80 638	100.0	1.0	18.3
Dijkstra	12 512	70.7	1.0	16.4
BubbleSort	103 950	66.8	1.0	15.4
Median	103 950	66.8	1.0	15.4
Triangle	2 206	50.9	1.0	5.9

Table A.2: Constraints obtained from the SPF analysis.

A.3 Concolic Experiments

Program	#cstrs	sat%	$\frac{\#\text{factors}}{\#\text{cstrs}}$	$\frac{\#\text{clauses}}{\#\text{cstrs}}$
Stack	1 023	100.0	6.8	9.0
BinomialHeap	4 982	39.6	6.5	14.8
BinTree	7 613	50.4	5.9	16.7
MagicIndex	200	100.0	5.8	11.1
CoinChange	108	15.8	5.6	43.7
ObjectRec	827	52.1	5.6	9.3
WBS	575	100.0	5.2	9.7
SortedListInt	347	79.8	2.6	7.9
Operations	7 809	28.3	2.0	41.1
Remainder	573	74.7	1.0	127.5
BubbleSort	11 631	6.2	1.0	13.7
Sorting	719	100.0	1.0	10.1
Triangle	1 103	1.7	1.0	6.4

Table A.3: Constraints obtained from the Coastal analysis.

A.4 Reduced SPF Experiments

Program	#cstrs	sat%	$\frac{\#\text{factors}}{\#\text{cstrs}}$	$\frac{\#\text{clauses}}{\#\text{cstrs}}$
Stack	2 046	100.0	6.8	9.0
BinomialHeap	7 182	93.8	6.5	15.3
BinTree	15 226	95.1	5.9	16.6
MagicIndex	400	100.0	5.8	11.1
ObjectRec	1 654	95.2	5.6	9.0
WBS	1 150	100.0	5.2	9.7
CoinChange	95 436	72.7	4.8	127.2
SortedListInt	694	96.0	2.6	7.9
Operations	15 618	80.4	2.0	40.9
Remainder	1 146	87.3	1.0	127.5
BubbleSort	23 262	53.1	1.0	13.3
Sorting	1 438	100.0	1.0	10.1
Triangle	2 206	50.9	1.0	5.9

Table A.4: Constraints obtained from the reduced SPF analysis.

A.5 Generated Experiments

Program	#cstrs	sat%	$\frac{\#\text{factors}}{\#\text{cstrs}}$	$\frac{\#\text{clauses}}{\#\text{cstrs}}$
version4F	100	100.0	500.0	1000.0
version3F	100	39.0	500.0	1000.0
version1F	100	31.0	500.0	1000.0
version2	100	0.0	201.5	1299.1
version3T	100	38.0	1.0	1501.0
version1T	100	0.0	1.0	1501.0
version4T	100	0.0	1.0	1501.0

Table A.5: Constraints obtained from the artificial generation.

Appendix B

Reduced Results of SPF

For consistency with the concolic and symbolic analysis, any search depth limits were removed from the analyses. Along with the removal of the limits, the search space had to be reduced such that the analysis could 1) complete and 2) complete within reasonable time for experimental results. The reduction is realised by either changing the input size or decreasing the iterations of certain program functions. After the settings were determined for the Coastal setup, the same settings were repeated for the SPF setup. Therefore the reduced SPF examples are obtained which are set up according to the concolic analysis of Table 4.7 for comparison.

B.1 Reuse Performance

		Green		Grulia		Z3Fact		Z3Cache	
Program	sat%	sat	unsat	sat	unsat	sat	unsat	sat	unsat
WBS	100.0	99	0	99	0	99	0	0	0
Stack	100.0	99	0	99	0	99	0	0	0
SortedListInt	89.9	94	80	96	91	93	80	0	0
ObjectRec	76.1	99	99	99	88	99	98	0	0
Operations	64.1	91	73	94	71	87	71	0	0
MagicIndex	100.0	82	0	95	0	82	0	0	0
Sorting	100.0	0	0	28	0	0	0	0	0
Remainder	87.3	14	0	24	0	0	0	0	0
BubbleSort	53.1	79	41	20	11	41	41	0	0
BinTree	75.2	97	71	90	8	92	5	0	0
BinomialHeap	65.6	98	84	99	54	92	27	0	0
Triangle	50.9	95	89	97	51	87	87	0	0
CoinChange	11.7	99	99	99	96	99	93	0	0

Table B.1: Reuse rate (%) of SPF on reduced examples.

The results in Table B.1 are similar to that of Table 4.2 and therefore only a few differences will be discussed. When looking at the results, the constraints are influenced with the analysis input change. The sat% percentage differ with Table 4.2 that of the original experiment. *ObjectRec* shows less unsat reuse compared to Green in Table 4.2. *BubbleSort* shows much better reuse with Green and also has greater unsat reuse than compared to Table 4.2. *CoinChange* shows better unsat reuse than Green in Table 4.2.

B.2 Running Times

Program	#cstrs	Z3Inc (ms)	Green	Grulia	Z3Fact	Z3Cache
Triangle	2 206	448	1.629	6.112	1.254	4.080
Stack	2 046	661	0.859	0.626	0.531	2.782
ObjectRec	1 654	418	1.129	0.983	0.732	3.000
WBS	1 150	376	1.152	0.867	0.782	2.851
SortedListInt	694	267	1.625	2.255	1.187	2.543
BubbleSort	23 262	3 707	2.961	23.26	2.551	3.742
BinTree	15 226	3 140	1.564	14.472	2.080	4.146
BinomialHeap	7 182	1 442	1.494	5.623	2.345	4.499
Sorting	1 438	517	3.954	11.894	2.983	2.621
MagicIndex	400	169	2.840	2.953	2.195	2.604
CoinChange	95 436	5 957	15.217	19.531	12.836	85.319
Operations	15 618	3 784	6.123	11.475	5.385	14.584
Remainder	1 146	339	23.652	101.761	25.976	23.855

Table B.2: Running times (normalised) of SPF on reduced examples.

Table B.2 is sorted similar to Table 4.6 for easier reference. Once again differences occur with the change of input for each program of the analysis. With the smaller search space, most program analyses completed under a second for both the solver and the caching tool. On the these smaller examples the overhead of the caching tools are too great to show improvement over Z3Inc. Keep in mind these examples are setup according to the concolic analysis of Table 4.7 for comparison.

List of References

Aquino, A., Bianchi, F.A., Chen, M., Denaro, G. and Pezzè, M. (2015). Reusing constraint proofs in program analysis. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pp. 305–315. ACM, New York, NY, USA. ISBN 978-1-4503-3620-8.
Available at: `http://doi.acm.org/10.1145/2771783.2771802`

Aquino, A., Denaro, G. and Pezzè, M. (2017). Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In: *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pp. 427–437. IEEE Press, Piscataway, NJ, USA. ISBN 978-1-5386-3868-2.
Available at: `https://doi.org/10.1109/ICSE.2017.46`

Aquino, A., Denaro, G. and Pezzè, M. (2019). Reusing solutions modulo theories. *IEEE Transactions on Software Engineering*, pp. 1–21.

Bailey, J. and Stuckey, P.J. (2005). Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V. and Cabeza, D. (eds.), *Practical Aspects of Declarative Languages*, pp. 174–186. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-30557-6.

Bloom, B.H. (1970 July). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, vol. 13, no. 7, pp. 422–426. ISSN 0001-0782.
Available at: `http://doi.acm.org/10.1145/362686.362692`

Braione, P., Denaro, G. and Pezzè, M. (2016). JBSE: A symbolic executor for java programs with complex heap inputs. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 1018–1022. ACM, New York, NY, USA. ISBN 978-1-4503-4218-6.
Available at: `http://doi.acm.org/10.1145/2950290.2983940`

Brennan, T., Tsiskaridze, N., Rosner, N., Aydin, A. and Bultan, T. (2017). Constraint normalization and parameterized caching for quantitative program analysis. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 535–546. ACM, New York, NY, USA. ISBN

978-1-4503-5105-8.
Available at: http://doi.acm.org/10.1145/3106237.3106303

Davis, M., Logemann, G. and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, vol. 5, no. 7, pp. 394–397.

Davis, M. and Putnam, H. (1960 July). A computing procedure for quantification theory. *J. ACM*, vol. 7, no. 3, pp. 201–215. ISSN 0004-5411.
Available at: http://doi.acm.org/10.1145/321033.321034

de la Banda, M.G., Stuckey, P.J. and Wazny, J. (2003). Finding all minimal unsatisfiable subsets. In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pp. 32–43. ACM, New York, NY, USA. ISBN 1-58113-705-2.
Available at: http://doi.acm.org/10.1145/888251.888256

Eén, N. and Sörensson, N. (2004). An extensible sat-solver. In: Giunchiglia, E. and Tacchella, A. (eds.), *Theory and Applications of Satisfiability Testing*, pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-24605-3.

Galil, Z. and Italiano, G.F. (1991). Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, vol. 23, no. 3, pp. 61–63.

Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A. and Tinelli, C. (2004). Dpll(t): Fast decision procedures. In: Alur, R. and Peled, D.A. (eds.), *Computer Aided Verification*, pp. 175–188. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-27813-9.

Gleeson, J. and Ryan, J. (1990). Identifying minimally infeasible subsystems of inequalities. *INFORMS Journal on Computing*, vol. 2, pp. 61–63.

Jaffar, J., Navas, J.A. and Santosa, A.E. (2012). Unbounded symbolic execution for program verification. In: *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pp. 396–411. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-29859-2.
Available at: http://dx.doi.org/10.1007/978-3-642-29860-8_32

Jia, X., Ghezzi, C. and Ying, S. (2015). Enhancing reuse of constraint solutions to improve symbolic execution. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pp. 177–187. ACM, New York, NY, USA. ISBN 978-1-4503-3620-8.
Available at: http://doi.acm.org/10.1145/2771783.2771806

King, J.C. (1976 July). Symbolic execution and program testing. *Commun. ACM*, vol. 19, no. 7, pp. 385–394. ISSN 0001-0782.
Available at: http://doi.acm.org/10.1145/360248.360252

Liffiton, M.H. and Malik, A. (2013). Enumerating infeasibility: Finding multiple muses quickly. In: Gomes, C. and Sellmann, M. (eds.), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*,

pp. 160–175. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-38171-3.

Morgado, A., Matos, P., Manquinho, V. and Marques-Silva, J. (2006). Counting models in integer domains. In: Biere, A. and Gomes, C.P. (eds.), *Theory and Applications of Satisfiability Testing - SAT 2006*, pp. 410–423. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-37207-3.

Păsăreanu, C.S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P. and Rungta, N. (2013 Sep). Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. In: *Automated Software Engineering*, vol. 20, pp. 391–425. ISSN 1573-7535.
Available at: `https://doi.org/10.1007/s10515-013-0122-2`

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial intelligence*, vol. 32, no. 1, pp. 57–95.

Tarjan, R.E. and van Leeuwen, J. (1984 March). Worst-case analysis of set union algorithms. *J. ACM*, vol. 31, no. 2, pp. 245–281. ISSN 0004-5411.
Available at: `http://doi.acm.org/10.1145/62.2160`

Tillmann, N. and de Halleux, P. (2008 April). Pex - white box test generation for .net. In: *Proc. of Tests and Proofs (TAP'08)*, vol. 4966 of *LNCS*, pp. 134–153. Springer Verlag.
Available at: `https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/`

Tinelli, C. (2002). A dpll-based calculus for ground satisfiability modulo theories. In: *European Workshop on Logics in Artificial Intelligence*, pp. 308–319. Springer.

Visser, W., Geldenhuys, J. and Dwyer, M.B. (2012). Green: Reducing, reusing and recycling constraints in program analysis. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pp. 58:1–58:11. ACM, New York, NY, USA. ISBN 978-1-4503-1614-9.
Available at: `http://doi.acm.org/10.1145/2393596.2393665`

Yang, G., Păsăreanu, C.S. and Khurshid, S. (2012). Memoized symbolic execution. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pp. 144–154. ACM, New York, NY, USA. ISBN 978-1-4503-1454-1.
Available at: `http://doi.acm.org/10.1145/2338965.2336771`

Zou, Q., An, J., Huang, W. and Fan, W. (2015 Dec). Integrating assertion stack and caching to optimize constraint solving. In: *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, pp. 397–401.