Working Towards the Verified Software Process

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Bruce M. Adcock, B.S., M.S., M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University
2010

Dissertation Committee:

Bruce W. Weide, Advisor
Atanas Rountev
Paolo Sivilotti
ABSTRACT

Numerous pieces of the software verification puzzle need to fit together in order to achieve that vision. First, there must be a programming language that gives some hope of specifying and implementing verifiable code (‘assertive code’). Second, there needs to be a correct procedure for generating verification conditions (VCs) from this assertive code. Third, there must be one or more automated theorem provers to process the VCs and determine their validity or invalidity. Finally, it is essential that there is an interface for programmers to write their code and verify or debug it without forcing them to understand how particular automated theorem provers work. This dissertation covers the latter two of these topics using the Resolve language and verification framework.

A common failing of most automated verification techniques is the inability to be truly automatic in all circumstances. Experience with proof assistants shows that they need to be ‘nudged’ in the correct direction by an intervening human, in all but the simplest cases. Otherwise, the option is to use tools not geared to handle the rich mathematics used in the Resolve programming language. By using specialized decision procedures that take into account the known structure of the generated VCs, we strive to accomplish one of two possibilities for each VC: first, we hope to prove it outright; otherwise, we hope to simplify the VC to such an extent that another prover is able to handle it. We detail work on SplitDecision, a tool that simplifies VCs for
Resolve. We further explain work that has helped to ensure SplitDecision is among the fastest and least memory intensive automated provers available, by introducing “lazy copying” to Resolve/C++ (in which SplitDecision is written), with specific work to maintain the value semantics integral to Resolve’s design.

Sometimes (perhaps this is even the modal case) at least one VC is not proved because it is not valid. It is clear that the ways we present errors and debug code must be rethought in a verified software paradigm. This area of research has received little attention to date, so we lay out criteria for how to approach debugging, along with potential methods of doing so.
To my parents of over 31 years,

my wife, Mousam, of more than 8,

and son, Rayhan, of almost 1.
ACKNOWLEDGMENTS

It is safe to say that this dissertation would not be what it is without the opportunities available to me at The Ohio State University. I have enjoyed continual encouragement during my time here, both through the Department of Mathematics, and later the Department of Computer Science and Engineering.

I would like to thank my advisor, Bruce W. Weide, for both providing excellent guidance and support while I did my research. Even when our interests diverged, he was willing to accommodate my work.

Paul Sivilotti is one of reasons I started work on a Ph.D. in computer science, with his intriguing 560 course and later on, his graduate courses. Nasko Rountev went out of his way to make sure students had a course on compilers, and I cannot thank him enough for that. Paolo Bucci, Wayne Heym, Bill Ogden, Murali Sitaraman, and Harvey Friedman all helped to make our weekly RSRG meetings enjoyable and thought provoking.

I would also like to thank my parents, Charles and Beth Adcock. They always made sure I never ran out of things to learn at home and taught me my work should be something I enjoy. Given all the trouble I used to get into, I am thankful they decided to stick with it. The encouragement of my in-laws, Matsyendra and Chandralekha, has meant a great deal to me. The time that my brother-in-law, Manish, lived with us are three years I will never forget (in a good way). My dependable beagle, Wall-E,
was essential, forcing me to take regular breaks that cleared my mind and caused me to see problems from new perspectives.

Friends in school helped me make it this far. From Craig Lennon, who put up with my stupid questions in our abstract algebra courses, to erica Whitaker, who handed me the form to switch to computer science after seeing how frustrated I was that one quarter, I still treasure my long-standing friendships from the Mathematics Department. The camaraderie among Jason Kirschenbaum, Derek Bronish, and myself in particular has made it possible to get this far while still enjoying the whole experience.

I do not believe I would have ended up going on to graduate school if not for the wonderful experience I had at Lafayette College. Everyone there provided strong encouragement, but it was Robert Roots’ mentorship that really allowed me to see what academia was about.

Finally, I would like to thank my beautiful wife, Mousam, for enduring nine long years of being with someone in graduate school. I cannot begin to express how much I love her. My son Rayhan, now only nine months old, has provided a new impetus to finish my degree I did not know could exist. When I look at him, he I encourages to think of what kind of person I want his dad to be, and work all the harder. I did not realize how much more love I had in me until he came along.

Bruce Adcock
Columbus, Ohio
November 30, 2010
VITA

February 22, 1979  ......................... Born: Albany, NY, USA

May 2001  ................................. B.S. Mathematics,
                        Lafayette College,
                        Easton, PA, USA

March 2004  ............................... M.S. Mathematics,
                        The Ohio State University,
                        Columbus, OH, USA

December 2008  ............................. M.S. Computer Science & Engineering,
                        The Ohio State University,
                        Columbus, OH, USA

Autumn 2002—Present  ........................ Graduate Research Associate,
                        Graduate Teaching Associate
                        The Ohio State University,
                        Columbus, OH, USA

PUBLICATIONS

Research Publications

Timothy Miller, James Dinan, S. Kasiviswanathan, Renji Thomas, Bruce M. Adcock,
Radu Teodorescu. Parichute: Generalized Turbocode-Based Error Correction for
Near-Threshold Caches. In International Symposium on Microarchitecture (MICRO),

Scott Pike, Wayne Heym, Bruce M. Adcock, Derek Bronish, Jason Kirschenbaum,
Bruce W. Weide. Traditional Assignment Considered Harmful. In OOPSLA 09:
Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented


FIELDS OF STUDY
Major Field: Computer Science and Engineering

Studies in Software Verification: Prof. Bruce W. Weide
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvi</td>
</tr>
<tr>
<td><strong>Chapters:</strong></td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1 Resolve</td>
<td>5</td>
</tr>
<tr>
<td>1.1.2 Verification conditions</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Going forward</td>
<td>17</td>
</tr>
<tr>
<td>2. SplitDecision, an Automated Prover for Verification Conditions</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>19</td>
</tr>
<tr>
<td>2.2 An example</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Development</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Results</td>
<td>28</td>
</tr>
<tr>
<td>2.5 Future work</td>
<td>34</td>
</tr>
<tr>
<td>2.6 Related work</td>
<td>35</td>
</tr>
</tbody>
</table>
3. Lazy Copying ........................................ 38
  3.1 Resolve and value semantics .................. 38
  3.2 Adding efficient copying to Resolve ........ 40
    3.2.1 Data representation in Resolve .......... 40
    3.2.2 Implementation of lazy copying ........ 44
    3.2.3 Additional details .................... 46
  3.3 Results ....................................... 47
  3.4 Future work .................................. 52
  3.5 Related work ................................ 54

4. Debugging in a World of Verification ........... 55
  4.1 Tools at our disposal ....................... 55
  4.2 An example .................................. 59
    4.2.1 Some correct code ..................... 59
    4.2.2 Some incorrect code ................... 66
    4.2.3 Correcting some incorrect code ....... 74
    4.2.4 Verification and the user ............. 78
  4.3 Tracing a verification condition to its source 80
    4.3.1 Origin of a VC ......................... 82
  4.4 Tracing counterexamples back ................. 88
  4.5 Failing gracefully (for the programmer) .... 93
    4.5.1 An example of failure .................. 93
    4.5.2 A framework for failure ............... 99
  4.6 A new interface for debugging .............. 100
    4.6.1 An example workflow ................... 100
    4.6.2 Final details ......................... 123
  4.7 Future work ................................ 125
    4.7.1 Front end ................................ 125
    4.7.2 Back end ................................ 127
  4.8 Related work ................................ 128

Bibliography ........................................ 130
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Procedure for breaking a fact of $\tau \neq \omega$, where $\tau$ and $\omega$ are strings. Here, $m$ is a new variable of type integer, and $\tau$, $\omega$ are strings of the type of $v$ and $z$.</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of SplitDecision and Isabelle in terms of categorizing VCs. The last column examines our best attempts at identifying VCs, using Z3 to find 2 more are valid, and a tool Contra to find 52 more invalid. There are 1,494 VCs that both SplitDecision and Isabelle find to be true.</td>
</tr>
<tr>
<td>2.3</td>
<td>Comparison of SplitDecision and Isabelle times. Separate comparisons done between VCs both verify, VCs both are unable to categorize, those that only Isabelle can categorize (i.e., prove), and those that only SplitDecision can categorize (i.e., prove or find invalid).</td>
</tr>
<tr>
<td>2.4</td>
<td>Number of VCs found valid, unsolved, or invalid by SplitDecision within 100 ms each.</td>
</tr>
<tr>
<td>3.1</td>
<td>Examination of 10 VCs, comparing the number of Copy_To calls made without lazy copying, the number of Copy_To calls made with lazy copying, the number of actual calls to Duplicate_To with lazy copying, the number of actual copies prevented, and the number of Copy_To’s called on an uninitialized object with lazy copying. All percentages are in comparison to the number of Copy_Tos without lazy copying. Percentages at the bottom are the average of percentages.</td>
</tr>
<tr>
<td>3.2</td>
<td>Examination of 10 VCs, comparing the result of adding a memory pool to Resolve/C++ with a fixed size of four. Shown is the number of requests for new memory against the actual uses of New, with lazy copying in place.</td>
</tr>
</tbody>
</table>
3.3 Comparison of total times to process 2,035 VCs using SplitDecision in seconds, with and without lazy copying, and with and without a memory pool. Also shown is the speedup factor compared to no lazy copying and no memory pool. ........................................ 50

4.1 Sample portion of a symbolic tracing table ........................................ 58

4.2 Symbolic tracing of Iterative code of UnboundedInteger ................. 61

4.3 Symbolic tracing of Iterative code for Multiply using IntegerFacility 71

4.4 Rules for breaking a VC so that givens and the statement to prove do not contain the logical operators ∨, ∧, ⇒, or ⇔. We are considering VCs written in the form \((\bigwedge_{i \in \{1,...,n\}} G_i) \Rightarrow C\). In the case of looking at the givens, we shall refer to the given to be singled out as \(G_j\) for some \(j \in \{1,...,n\}\). ....................................................... 81

4.5 Comparison of finding counterexamples using either both SplitDecision and Contra, or just Contra. Times are in seconds. ......................... 89
## LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Contract for SetTemplate</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Contract for Remove of SetTemplate</td>
<td>9</td>
</tr>
<tr>
<td>1.3 An attempt at an iterative implementation of Subtract for SetTemplate</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Specification for Copy_To in Resolve/C++</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Example iteration through a Queue</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Contract for UnboundedInteger’s Multiply</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Iterative implementation for UnboundedInteger’s Multiply</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Contract for UnboundedInteger’s IsPositive</td>
<td>60</td>
</tr>
<tr>
<td>4.4 Contract for UnboundedInteger’s Add</td>
<td>60</td>
</tr>
<tr>
<td>4.5 Contract for UnboundedInteger’s Negate</td>
<td>60</td>
</tr>
<tr>
<td>4.6 Contract for the IntegerFacility component</td>
<td>66</td>
</tr>
<tr>
<td>4.7 Contract for Integer’s Multiply</td>
<td>67</td>
</tr>
<tr>
<td>4.8 An attempt at an Iterative implementation for Integer’s Multiply</td>
<td>67</td>
</tr>
<tr>
<td>4.9 Contract for Integer’s Add</td>
<td>68</td>
</tr>
<tr>
<td>4.10 Contract for Integer’s Negate</td>
<td>68</td>
</tr>
<tr>
<td>4.11 A non-obvious Iterative implementation for Integer’s Multiply</td>
<td>75</td>
</tr>
</tbody>
</table>
4.12 Contract for Integer's Subtract. 75

4.13 An implementation of Iterative for Integer's Multiply that could be verified. 77

4.14 Contract for ArrayAsString2Template 84

4.15 Contract for enhancement Contains of ArrayAsString2Template 85

4.16 Implementation of enhancement Contains of ArrayAsString2Template 85

4.17 A useful enhancement to SetTemplate. 93

4.18 An attempted implementation of UniteAndIntersect 94

4.19 A corrected implementation of UniteAndIntersect 99

4.20 Contract for Sqrt, an enhancement to IntegerFacility 101

4.21 Contract for Average enhancement to IntegerFacility 115
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 System overview of a Resolve development and verification system. The thinly dashed line denotes the focus of this dissertation, with the greatest emphasis given to the thickly dashed line.</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Graph of facts and obligations for Iterative implementation of Subtract for SetTemplate.</td>
<td>13</td>
</tr>
<tr>
<td>1.3 Unsimplified VC generated to prove ( t_9 = t_0 ) at location indicated by “1” in the graph in Figure 1.2.</td>
<td>14</td>
</tr>
<tr>
<td>1.4 VC generated to prove ( t_9 = t_0 ) at location indicated by “1” in the graph in Figure 1.2.</td>
<td>15</td>
</tr>
<tr>
<td>1.5 One of two VCs generated to prove ( s_7 \cap \text{tmp}_7 = \emptyset ) using the graph in Figure 1.2 at location “2”, on passing through the if branch.</td>
<td>16</td>
</tr>
<tr>
<td>1.6 Second of two VCs generated to prove ( s_7 \cap \text{tmp}_7 = \emptyset ) using the graph in Figure 1.2 at location “2”, on going around the if branch.</td>
<td>16</td>
</tr>
<tr>
<td>2.1 System overview of a Resolve development and verification system. The dotted line highlights the main focus of this chapter.</td>
<td>19</td>
</tr>
<tr>
<td>2.2 The initial example VC</td>
<td>22</td>
</tr>
<tr>
<td>2.3 VC from Figure 2.2, negated and put in disjunctive normal form.</td>
<td>23</td>
</tr>
<tr>
<td>2.4 The VC after length of concatenations and pruning of inequalities are applied, after 7 applied steps.</td>
<td>24</td>
</tr>
<tr>
<td>2.5 Processing of the VC after a total of 14 applied steps.</td>
<td>24</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>2.6</td>
<td>Processing of the VC after a total of 15 applied steps.</td>
</tr>
<tr>
<td>2.7</td>
<td>Processing of the VC after a total of 21 applied steps.</td>
</tr>
<tr>
<td>2.8</td>
<td>Histogram of the number of VCs that can individually be processed, over time to process (2035 VCs total), with steps of 100ms.</td>
</tr>
<tr>
<td>2.9</td>
<td>Graph of the percent of VCs processed, over time to process, partitioned into “valid” (1748 VCs in total) and “unsolved” (285 VCs in total). The VCs found false by <code>SplitDecision</code> are not shown.</td>
</tr>
<tr>
<td>3.1</td>
<td>How swap actually is implemented for non-primitive types.</td>
</tr>
<tr>
<td>3.2</td>
<td>Dependencies of code written for <code>SplitDecision</code>. Boxes with rounded corners are “abstract”. Most importantly, all arrows move from left to right; if a component does utilize aliasing internally, it cannot escape.</td>
</tr>
<tr>
<td>3.3</td>
<td>Before and after calling <code>a.Copy_To(b)</code> for a non-primitive type. In this case, “rc” is the reference count.</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison of times without lazy copying or a memory pool, with the speedup factor from using lazy copying.</td>
</tr>
<tr>
<td>3.5</td>
<td>The data structure of a logical formula in <code>SplitDecision</code> using lazy copying.</td>
</tr>
<tr>
<td>3.6</td>
<td>Zoomed in view of the data structure shown in Figure 3.5.</td>
</tr>
<tr>
<td>4.1</td>
<td>System overview of a Resolve development and verification system. The dotted line highlights the main focus of this chapter.</td>
</tr>
<tr>
<td>4.2</td>
<td>A VC generated for the <code>requires</code> clause of the first <code>Add</code> used in the implementation in Listing 4.8.</td>
</tr>
<tr>
<td>4.3</td>
<td>A VC generated for the <code>requires</code> clause of the second <code>Add</code> used in the implementation in Listing 4.8.</td>
</tr>
<tr>
<td>4.4</td>
<td>A VC generated for the <code>requires</code> clause of <code>Subtract</code> used in the implementation in Listing 4.11.</td>
</tr>
</tbody>
</table>
4.5  A VC generated for the `ensures` clause of `LinearIterativeEndConditionBug`, an enhancement `Contains` of `ArrayAsString2Template` shown in Listing 4.16 ................................. 86

4.6  Origin of VC shown in Figure 4.5 before splitting ......................... 87

4.7  The result after processing the VC from Figure 4.5 with `SplitDecision`. 90

4.8  Example VC where facts irrelevant to the proof are less irrelevant to the counterexample. ......................................................... 91

4.9  VC from Figure 4.8 after potential simplification. ........................ 92

4.10 One of two unproven VCs generated from code in Listing 4.18. ...... 95

4.11 Second of two unproven VCs generated from code in Listing 4.18. .. 95

4.12 VC from Figure 4.10 after processing by `SplitDecision`. ............. 96

4.13 Attempted binary search implementation of `Sqrt` in Listing 4.20. ... 102

4.14 Showing some contracts to operations used in Listing 4.13. .......... 104

4.15 Explaining the different paths and requirements. ...................... 105

4.16 Selecting code shows where the facts are used, to help the programmer determine whether it is correct or necessary. ..................... 107

4.17 Process to see if a fact is unnecessary in an unsolved VC. ........... 108

4.18 Examining one of the counterexamples showing that the code is defective.110

4.19 Examining the second counterexample showing that the code is defective.111

4.20 Examining what in the code was necessary to prove one of the require-
ments. ................................................................. 112

4.21 Second attempt at binary search implementation of `Sqrt` in Listing 4.20.114

4.22 Third attempt at binary search implementation of `Sqrt` in Listing 4.20.116
4.23 Cleaning up the cleaned up third attempt binary search implementation of \texttt{Sqrt} in Listing 4.20. ................................................................. 118

4.24 Fourth attempt at binary search implementation of \texttt{Sqrt} in Listing 4.20, now with \texttt{confirm} statements. ................................. 119

4.25 Final attempt at binary search implementation of \texttt{Sqrt} in Listing 4.20. 121

4.26 Showing the user which prover was involved at each point in the code for verifying our implementation of \texttt{Sqrt}. ................................. 122

4.27 Giving a symbolic trace while programming, along with a found counterexample. ................................................................. 125
CHAPTER 1

Introduction

For all the advances in software engineering it seems that, in many ways, the current state forces programmers to live in the Dark Ages. We provide them tools to help trace back a bug once it has happened, both delighting them by letting them act the sleuth, and frustrating them by forcing them to act the sleuth. For certain bugs, those most egregious and obviously wrong ones such as dereferencing null pointers [18, 23, 19] or buffer under- and overflows [33, 13, 60], we provide static tools to help track them down. As helpful as these tools are, they come with major caveats. Do they guarantee completeness? No. Do they even guarantee soundness? No. They can make guarantees of few false positives, and many of the bugs found. Static checking [38, 9] leaves us with a slight distaste in our mouths, comforting us when there is no more flagging of real errors, but we leave ever wondering.

So in the end check their code forces programmers to use various means of dynamically testing their code [43]. But as we know from Dijkstra [15],

Program testing can be used to show the presence of bugs, but never to show their absence.

Again, the state of the art in dynamic checking must always leave us uneasy, unsure of when the unknown will strike.
There is, of course, a dream to do better by truly verifying the code; by showing that not only does a specific property hold, but also that we can describe a behavior and demonstrate that the code realizes it correctly. By definition, dynamic checking will take resources at runtime. It certainly might allow a developer to handle problems as they happen, but cannot guarantee the problem will not occur. If there were a way to check that statically, there would be no need for the dynamic checking. Clearly, verification can and should happen statically if we are to have complete confidence.

There are many different variations of software verification. For this dissertation, we shall be discussing verification that is full functional, total, and with automated formal reasoning. Having these properties would essentially meet Hoare’s grand challenge of a verifying compiler [27]. We will discuss these in turn.

• By full functional, we mean the functionality of an operation is fully described in a formal language. Indubitably, there must be some way of describing the functionality, as otherwise there is nothing to check. Using rigorous, formal mathematical statements to describe the functionality removes unintended ambiguity and makes automated checking potentially possible.

• Total means the execution will terminate. Another way to state this is that we must demonstrate progress, at all instances of recursion and loops (we disallow infinite recursion and infinite loops).

• We desire to verify with automated formal reasoning, for there to be any hope of its use. The actual proving must occur with tools with no requirement of interaction, and formally, so there is no question of their veracity.¹

¹One might wonder how we know the proof is correct if no one would ever look at it. An automated theorem prover can (potentially) generate a proof certificate, a version of the proof written for a
Many groups are working to achieve this [31, 36, 4], with The Ohio State University’s (OSU) Resolve/Reusable Software Research Group (RSRG) among them. Every attempt at verification has a different methodology, up to and including the language used in the specifications and the tools that do the actual proving, but the focus in this dissertation using work at OSU is to provide a solid example. Many of the attempts involve trying to verify programs using in-production languages such as C or Java, languages that were never designed for verification, and indeed contain features that run counter to being able to reason about them. Resolve, the language used by RSRG, is a research language designed with a focus on being verifiable (while also being highly expressive and maintaining a reasonable amount of efficiency) [45]. If there is any hope at software verification, Resolve would demonstrate it.

Figure 1.1 shows the system overview of developing and verifying with Resolve. The focus of this dissertation is within the outer dashed line, with a greater focus within the inner dashed line, but the overall intended architecture is important to understand. On the software side, the desired functionality appears in the specifications; one or more implementations are written in assertive code, that is, code with annotations that are necessary to justify its correctness. These specifications and assertions are written using Resolve’s mathematical language developed for this purpose. This language may rely on libraries of general math definitions and proven theorems found useful for a wide range of applications, as well as more specialized definitions and theorems.

Using the specifications and the code, we may determine what actually needs to be verified, and what possible accumulation of facts might help demonstrate that. Limited proof system. Proof certificates are so low-level that no human would ever wish to look at them either, but a tool that could verify each step is correct would be simple enough to show its correctness with a traditional proof by social processes. Ideally, we will eventually verify even the automated prover itself, so that not even the proof certificate step would be necessary.
These are more specifically known as VCs, or *verification conditions*, and are created automatically by a VC generator. The VCs are fed to one or more automated provers, tools that attempt to show without human intervention that each VC is valid, and thus that aspect of the code is correct; or invalid, meaning there exist reachable program states where the code is incorrect. While any VCs continue to be found invalid or unsolved, the programmer must modify the code to correct the errors and justify its correctness. Once it is found to be correct, we may then compile the code.

This dissertation focuses on the feedback loop between an invalid or unproven VC and the code. In Chapter 2, we examine a tool, *SplitDecision*, designed from the ground up to automatically handle generated VCs. The feedback loop, for practical reasons, needs to happen quickly, and Chapter 3 discusses some of the work done with lazy copying for this while hinting at the implications it can have on a modular
language such as Resolve. Finally, Chapter 4 examines the requirements to make the feedback loop useful for someone writing code, and possible interfaces to aid in this.

1.1 Background

Throughout this dissertation, we will use the Resolve language. Below is background information on the Resolve language and the VCs generated.

1.1.1 Resolve

Resolve is a research language designed with the understanding that specifications and programming languages themselves must be carefully designed to support modular (component-wise) verification. Resolve/RSRG created it to identify language features and programming methods that allow the programmer to maintain the expressiveness and efficiency provided by a typical mainstream language for sequential object-based programming, while supporting easy software component reusability and, most importantly, the foundations necessary to verify code in a modular way. The Resolve discipline is a framework for software development in which one writes contracts consisting of model-based formal specifications, and implements these contracts in various realization modules [8, 16].

An important tenet of Resolve is to prevent unintentional aliasing and minimize (but allow) the need to intentionally alias. Resolve handles the latter by taking data structures that typically use aliasing, such as a singly-linked list or a tree, and encapsulating them in modular components, such as ListTemplate or TreeTemplate, respectively, that properly abstract the desired features while not exposing any references to the client. We handle the former through many specific language features, the most important being the use of swapping for data movement [47]. All
types inherently get the first-class operator ‘:=:’, which, on \( x :=: y \), ensures \( x \) has \( y \)'s old value, and vice versa. Resolve prevents further unintentional aliasing by syntactically forbidding repeated arguments in procedures [34], and disallowing shallow copying [24],\(^2\) among other places.

A major aspect of Resolve is its strong reliance on mathematical modeling, enforced not just in how we write the specifications, but also in how the programmer must think about the code they rely upon. This enforcement is pushed by the point that a client can only think of a component in terms of its contract. Take, for example, the contract for \texttt{SetTemplate} in Listing 1.1.

```plaintext
contract SetTemplate (type Item)
    uses UnboundedIntegerFacility

    math subtype SET_MODEL is finite set of Item

    type Set is modeled by SET_MODEL
    exemplar s
    initialization ensures
        s = empty_set

    procedure Add (updates s: Set, clears x: Item)
        requires
            x is not in s
        ensures
            s = #s \union \{\#x\}

    procedure Remove (updates s: Set, restores x: Item, replaces xCopy: Item)
        requires
            x is in s
        ensures
            s = #s \setminus \{x\} and
            xCopy = x

    procedure RemoveAny (updates s: Set, replaces x: Item)
        requires
            s \neq empty_set
        ensures
            x is in #s and
            s = #s \setminus \{x\}

    function Contains (restores s: Set, restores x: Item): control
        ensures
            Contains = (x is in s)

    function IsEmpty (restores s: Set): control
        ensures
            IsEmpty = (s = empty_set)
```

\(^2\)As swapping is the only first-class data movement operator, one must use a \texttt{Replica} function that forces deep copying instead, and is not implicitly guaranteed for every type.
function Size (restores s: Set): Integer
    ensures
    Size = |s|
end SetTemplate

Listing 1.1: Contract for SetTemplate

Clients of `SetTemplate` must think of a `Set` object in terms of its mathematical model, stated here to be a finite set of “Item”, where the client provides `Item` based on what they need. This is important: it tells the client that the value of a variable of type `Set` will not allow duplicate items (items are duplicate if their mathematical model values are the same); the client also knows they are not allowed to distinguish between the `Sets` normally described as `{a,b}` and `{b,a}`, as in set theory these are one and the same; finally, the client knows that on the declaration of a variable of type `Set`, it will have a mathematical value of \(\emptyset\).

Every operation a programmer either uses as a client or implements as a developer states the requirements of what is necessary to call the operation, its precondition written in a `requires` clause, and what it guarantees afterwards, its postcondition written in an `ensures` clause. For example, the operation `Add` has a `requires` clause of `x is not in s`. If \(x = a\) and \(s = \{a,b\}\), then we are not allowed to call `Add`. This allows any developer of a realization of `SetTemplate` to fail to handle such a case; the client is already making that guarantee. `Add` also has an `ensures` clause of `s = s \cup \{x\}`. Here, `s` refers to the `incoming` value of `s`, whereas `s` means the outgoing value. A client can rely on this guarantee, while a developer must make sure it is the case. Finally, additional syntactic sugar allows us to state common contractual occurrences without writing everything out. The parameter mode
clears x is equivalent to adding is_initial (x) to the ensures clause. Similarly, restores x is equivalent to having x = #x as part of the ensures clause.

A client of SetTemplate can very easily run into a situation where their knowledge of the states of variables is relational. Consider if we know that the mathematical values of an object of type Set, s, and one of type Item, x, are \{a\} and b, respectively (with a \(\neq\) b). On calling Add (s, x) we now know \(s = \{a, b\}\), and x has an initial value for its type, Item. What should happen if we then call RemoveAny (s, x)? Is it that \(s = \{a\}\) and \(x = b\), or is it that \(s = \{b\}\) and \(x = a\)? The truth is, we have no way of knowing, as the contract for RemoveAny allows both behaviors. It is important to remember that we designed this behavior into SetTemplate: if the client did not want this behavior, they should have used a different component.

Resolve may have additional annotations, depending on the situation. Every loop, for example, requires an invariant given by a maintains clause. Further, as we require our verification to be total, both loops and recursive functions require progress metrics, given by decreases clauses. Such statements are integer expressions that must decrease after each iteration of the loop (and must always be non-negative). A further place for annotations is with confirm statements, where a programmer can document formally what they believe to be true, and also have this checked by a theorem prover. This has an additional benefit; we can use such documentation as a fact by the theorem prover for later things to prove, and thus serve as a “bridge” in a difficult VC proof.
1.1.2 Verification conditions

For code written in Resolve, the task of verification is a matter of proving that a proposed realization satisfies its specification. We can generate the VCs in a natural and fairly straightforward way [51]. As an example, we will consider the enhancement Subtract of SetTemplate.

```resolve
contract Subtract enhances SetTemplate

procedure Subtract (updates s: Set, restores t: Set)
    ensures
    s = #s \ t
end Subtract
```

Listing 1.2: Contract for Remove of SetTemplate

Simply, the Subtract operation takes two Sets \( s \) and \( t \), and updates \( s \) to be all elements originally in \( s \) except those in \( t \). An attempted implementation of this is in Listing 1.3.

```resolve
realization Iterative implements Subtract for SetTemplate

procedure Subtract (updates s: Set, restores t: Set)
    variable tmp: Set
    loop
        maintains s union tmp union t = #s union #tmp union t and
        s intersection tmp = empty_set and
        t = #t
        decreases |s|
        while not IsEmpty (s) do
            variable x: Item
            RemoveAny (s, x)
            if not Contains (t, x) then
                Add (tmp, x)
            end if
        end loop
    s :=: tmp
end Subtract
```

Listing 1.3: An attempt at an iterative implementation of Subtract for SetTemplate

When we ask “how do we make a VC”, we are really asking, “what do we need to prove, and what information is available to do it”. What we need to prove is relatively straightforward, as we have already stated most of the obligations in Resolve:
• **Recursion progress metrics.** A recursive operation must have a progress metric, where (potentially) we need to check two things. At the start of the operation, we must verify the proposal of a metric which is an integer greater than or equal to 0.\(^3\) Second, at any place where there is a recursive call, we must check that the metric measured at this point in the code is now less than the metric when checked at the start.\(^4\)

• **Loop progress metrics.** Loops have very much the same requirements on their metrics as recursive operations do, but with some slight changes. When coming upon a loop, we must show that the progress metric is greater than 0 if the loop’s condition is true. A VC is also generated to show that, on any given iteration of the loop, the progress metric has decreased through an iteration of the loop. Finally, we check that after completing any iteration of the loop, the progress metric is still non-negative.

• **Loop invariants.** When coming upon a loop, it is necessary to demonstrate that each loop invariant is true. Similarly, each loop invariant must be true after any possible iteration of the loop, assuming it is true at the start of that iteration.

• **Preconditions.** Both the client and the VC generator understands each operation call by its specification, so a VC must check any operation with a requires clause at the site of the call.

\(^3\)Other methods of dealing with progress may demand that the progress metric expression is either a natural or an ordinal, and therefore have no need to make this check here. This version of Resolve does not include ordinal types and considers natural numbers to be a subset of the integers.

\(^4\)It is *not* necessary to check that the metric is non-negative at this point, as the initial check confirms that any values that meet the requires clause of the operation will satisfy this already.
• **Postconditions.** Conversely, we must check an operation’s *ensures* clause with a VC at the end of the body implementing it.

• **Parameter modes.** In OSU’s Resolve, each parameter has a mode, providing common obligations on parameters for the *ensures* clause shown as part of the operation signature. As stated earlier, common ones (that likewise generate a VC) are *restores* and *clears*.²

• **Assertions.** A programmer may choose to annotate code with a *confirm* statement. While certainly not mandatory in Resolve, this gives the programmer a way to document their reasoning formally. A VC results to prove the statement needing confirmation, and VCs afterward can use it as a fact to help with their proofs.

• **Abstraction functions.** Any realization for a component that defines a new type needs to have a connection between the mathematical model for the new type and the mathematical model for its representation, known as an abstract relation.⁶ It is necessary to describe this in the realization but results in a VC only when there is a constraint on the new type to a subspace of its mathematical model space.

• **Representation invariants.** To aid in the development of a realization, programmers write the useful conventions that always hold both at the beginning and end of the kernel operations for that type. The invariants given are those

---

²There are other parameter modes: *updates*, *replaces*, and *alters*. These do not, however, cause the generation of VCs (a syntactic check handles their requirements instead).

⁶In OSU’s Resolve, we limit this relation to a function, for simplicity. However, it is known how to generalize to relations [53].
that are often useful in the proof of other VCs, thus the need for the convention to be formally written down and proven at the end of each operation, or ones that aid in the programmer’s understanding of the type.

Consider how what it takes to prove that the realization in Listing 1.3 meets the obligation of the restores parameter mode \( t \) of Subtract, \( i.e., t = \#t \). Do we need to consider every iteration of the loop? No, as this is the point of the loop invariants, to describe what is invariant through any number of iterations of the loop. What about the loop invariant clause at the end of an iteration of the loop, that \( s \ intersection \ tmp = empty\_set \)? How does one handle the possibility of going through the if branch, along with the possibility of not? We do this by (in effect, at least) generating a directed, acyclic graph of the accumulated knowledge and obligations demonstrated in Figure 1.2.

Walking through the code, we start by writing down a symbolic value of each variable and any properties about them immediately before and after each statement. These are exactly the points at which one could imagine pausing the program to look at variable values without stepping into called operations. We use a subscript with each variable to let us know which state we are examining, and work to relate the current state of the variable with the next oldest state. We show these facts in the boxes with a dotted border. Some states may have obligations we need to prove; these appear in boxes with solid borders. We connect each box with an arrow if we may progress from one state to the next, adding in the fact of the path condition where applicable. We do not need to draw a back edge from the end of an iteration of the while loop back to the top, as the loop invariant is there to eliminate this case. The directed acyclic graph drawn fully demonstrates our formal knowledge of values.
Figure 1.2: Graph of facts and obligations for Iterative implementation of Subtract for SetTemplate.
of each variable. Snippets of code are also shown, to help relate the graph with a realization. These do not otherwise provide any insight into the correctness of the code at each state of execution.

What, then, does it mean to meet the obligation labeled by a “1” in Figure 1.2? To show that $t_g = t_0$ is valid, we may make use of all givens accumulated along any path that can reach that point. We collect them by starting at the top of the graph, and include known facts, obligations that must be previously proven, and path conditions, to get the VC shown in Figure 1.3.

<table>
<thead>
<tr>
<th>Prove</th>
<th>$t_g = t_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given</td>
<td>$tmp_9 = s_8$</td>
</tr>
<tr>
<td></td>
<td>$\land tmp_8 = s_9$</td>
</tr>
<tr>
<td></td>
<td>$\land tmp_1 = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\land t_g = t_8$</td>
</tr>
<tr>
<td></td>
<td>$\land t_8 = t_1$</td>
</tr>
<tr>
<td></td>
<td>$\land t_1 = t_0$</td>
</tr>
<tr>
<td></td>
<td>$\land s_8 = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\land s_1 = s_0$</td>
</tr>
<tr>
<td></td>
<td>$\land tmp_8 \cap t_8 = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\land tmp_8 \cap s_8 = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\land t_8 \cup (tmp_8 \cup s_8) = t_8 \cup (tmp_1 \cup s_1)$</td>
</tr>
</tbody>
</table>

Figure 1.3: Un simplified VC generated to prove $t_g = t_0$ at location indicated by “1” in the graph in Figure 1.2.

This is a rather unwieldy VC even for the most obvious cases, which we in fact are dealing with. Therefore, in VC generation we also do any available variable substitutions when possible, allowing us to minimize the total number variables and
Prove $t_0 = t_0$

Given
- $\emptyset \cup \text{tmp}_8 \cup t_0 = s_0 \cup \emptyset \cup t_0$
- $\emptyset \cap \text{tmp}_8 = \emptyset$
- $\text{tmp}_8 \cap t_0 = \emptyset$

Figure 1.4: VC generated to prove $t_9 = t_0$ at location indicated by “1” in the graph in Figure 1.2.

to simplify the VC (without using any knowledge from the underlying theory, in this case about sets). The result appears in the VC in Figure 1.4. The VC demonstrates our dependence on loop invariants properly describing the loop; despite covering the code from start to finish, it is in fact a VC with very few accumulated givens. It should be unsurprising, however, that the VC is easily proven; looking through the code in Listing 1.3, it is clear that the value of $t$ is never altered. This mimics the general pattern we see with the VCs and the code; properties easily seen to be true in the code should generate VCs that are easily seen to be valid. Even though very few facts will turn out to be necessary for any particular instance, the VC generator does not know which ones are important, so we provide all facts to an automated theorem prover.

To handle the case of multiple paths, we create a VC for each path. Generation of the VCs for the obligation labeled “2” gives those shown (after substitutions) in Figures 1.5 and 1.6, demonstrating either going through the if branch or falling through, respectively. In both instances, the single given necessary to prove the VC is $s_2 \cap \text{tmp}_2 = \emptyset$, highlighted as lines 1.1 and 1.2. Clearly, the VCs are not the same as they have different givens and obligations.
Prove \((s_2 \setminus \{x_4\}) \cap (tmp_2 \cup \{x_4\}) = \emptyset\)

Given
\[
\begin{align*}
& \quad s_2 \neq \emptyset \\
& \quad s_2 \cup tmp_2 \cup t_0 = s_0 \cup \emptyset \cup t_0 \\
& \quad s_2 \cap tmp_2 = \emptyset \\
& \quad tmp_2 \cap t_0 = \emptyset \\
& \quad 0 \leq |s_2| \\
& \quad is\_initial(x_3) \\
& \quad x_4 \in s_2 \\
& \quad is\_initial(x_6) \\
& \quad x_4 \notin t_0
\end{align*}
\]

(1.1)

Figure 1.5: One of two VCs generated to prove \(s_7 \cap tmp_7 = \emptyset\) using the graph in Figure 1.2 at location “2”, on passing through the **if** branch.

Prove \((s_2 \setminus \{x_4\}) \cap tmp_2 = \emptyset\)

Given
\[
\begin{align*}
& \quad s_2 \neq \emptyset \\
& \quad s_2 \cup tmp_2 \cup t_0 = s_0 \cup \emptyset \cup t_0 \\
& \quad s_2 \cap tmp_2 = \emptyset \\
& \quad tmp_2 \cap t_0 = \emptyset \\
& \quad 0 \leq |s_2| \\
& \quad is\_initial(x_3) \\
& \quad x_4 \in s_2 \\
& \quad x_4 \in t_0
\end{align*}
\]

(1.2)

Figure 1.6: Second of two VCs generated to prove \(s_7 \cap tmp_7 = \emptyset\) using the graph in Figure 1.2 at location “2”, on going around the **if** branch.
The real difficulty, of course, lies in simplifying and establishing the validity of these VCs, particularly because they may deal with sophisticated mathematical entities. Important points about this are at the heart of this dissertation.

1.2 Going forward

There are three parts to the thesis of this work:

1. It is possible to write an automated prover that deals effectively with the rich mathematics necessary for proving VCs generated for Resolve.

2. An automated prover such as SplitDecision can be efficient, both in time and memory, while making few if any concessions as to completeness.

3. Failures of verifying code, either through finding them defective or being unable to say otherwise, are presentable to the programmer such that

   (a) we do not need to show actual VCs, and

   (b) we can find helpful information for discovering the issue and fixing it, all while never actually executing the code.

Chapter 2 introduces SplitDecision, an automated prover made for the very purpose of handling Resolve VCs. We further examine SplitDecision’s efficiency. Chapter 3 introduces “lazy copying”, an enhancement made to Resolve/C++’s foundation to specifically help SplitDecision maintain a low memory profile, but is beneficial to anyone code written in Resolve/C++. Chapter 4 discusses the issue of debugging realizations in a world of software verification.
CHAPTER 2

SplitDecision, an Automated Prover for Verification Conditions

At the heart of the current work for creating a verifying compiler is an automated prover that soundly determines either the validity or invalidity of the vast majority of VCs that are likely given to it. This integral piece appears in the place of the dashed box in Figure 2.1. Numerous such tools exist: Z3 and Simplify are commonly used in the software verification community, as is Isabelle. With relatively mature tools already established, it would seem to be unnecessary, and perhaps somewhat less than sensible, to start work on a new one.

Work on a verifying compiler quickly revealed the current tools to be somewhat unsuited to Resolve VCs. Most tools in use focus on a few mathematical theories that are convenient for languages heavily concerned with primitive program types (especially integers and arrays) and the management of the heap. Others have the intended use as interactive proof assistants. In either case, they were never intended to handle the rich mathematical theory developed over the past 30 years for Resolve. We additionally suffer from having to treat these tools as essentially black boxes. What built-in rules do they apply? When supplied additional lemmas for a theory, which ones work? Even if a tool provides a trace through the proof, it is invariably not
Figure 2.1: System overview of a Resolve development and verification system. The dotted line highlights the main focus of this chapter.

intended to be human readable. \textit{SplitDecision} was born out of an attempt to handle these issues. It is an endeavor to, first and foremost, deal with the mathematical theories used in Resolve. It further allows us to test new proof rules for efficacy and efficiency, and provides human-intelligible feedback about the proof process which we can then adjust and explore.

2.1 Overview

The initial algorithms on finite strings, a collection of decision procedures [20], inform the design of \textit{SplitDecision}. Resolve’s (finite) string theory consists of the following:

- the empty string, $\Lambda$, that contains no entries;
- singleton strings of objects such as $\langle x \rangle$ (also known as “stringletons”);
• the concatenation operator, \( \circ \), which concatenates two strings together;

• the length operator, so that \(|\alpha|\) defines how many entries there are in the string;

• The Val operator, where Val \((\alpha, i, x)\) means that the \(i\)th position of the string \(\alpha\) (with index starting at 1) has value \(x\); and

• The Winc operator, with Winc\((\alpha)\) meaning that, for any \(x, y, i, j\) such that Val \((\alpha, i, x)\), Val \((\alpha, j, y)\), and \(x \leq y\), then \(i \leq j\) (note that we therefore allow an arbitrary ordering on objects).

Beyond this, we have added more to the theory. In particular, we have substring, where substring \((\alpha, i, j)\) is the substring of \(\alpha\) from \(i\) to \(j - 1\) (starting at index 0) if \(0 \leq i < j \leq |\alpha|\), and \(\Lambda\) otherwise. Symbols (\(i.e.,\) variables) are only allowed on one side of an equality or inequality, and only the logical operators \(<, \leq, =, \neq, \wedge, \vee,\) and \(\neg\) can be used. Formulas must be put into disjunctive normal form, or DNF, meaning it is in the form of a disjunction of conjunctions of literals. Each literal is called a conjunct, while each conjunction of literals is called a disjunct. The final restriction is that \(\neg\)s must be pushed as far “down” the formula as possible, and minimized through recognizing double negation or, for example, converting \(\neg(a \leq b)\) to \(b < a\).

**SplitDecision** has expanded, keeping the design decisions needed for the above rules in mind. The rules address a sizable subset of Resolve’s theory of strings and either find VCs concerning strings valid\(^7\) or push them down to existential sentences in Presburger arithmetic, which is itself decidable [48].

In general, under these procedures, **SplitDecision** takes a VC, negates it, and converts it to DNF with the proper logical operator restrictions, \(i.e.,\) the problem turns

\(^7\)The rules can potentially find invalidity, but the VC in question would necessarily be degenerately simple in this case.
into one of deciding whether the negation is satisfiable. For this reason \texttt{SplitDecision} is specifically tuned to \textit{ruling out} the possibility of counterexamples by finding contradictions in each disjunct, or “split”. If \texttt{SplitDecision} finds a contradiction in a split, we remove it because it is therefore not satisfiable. The standard contradictions, such as $x \neq x$, $A \land \neg A$, and $0 = 1$, are of course found using general-purpose strategies. The decision procedures for strings add two new additional contradictions to look for, namely $\langle x \rangle = \Lambda$ and $\text{Val}(\Lambda, i, x)$. Every other algorithm in the string theory decision procedure works to arrive at one such contradiction, without needing to backtrack.

In order to handle these rules, \texttt{SplitDecision} was designed to do two things very efficiently: first, to substitute one subformula for another, in part handled by maintaining the VC in a canonical form so as to aid in quickly finding things to be matched; second, to take a disjunct and split it into multiple parts, a common occurrence as the rules essentially induce case analysis by causing splits. As an example, consider the situation where one fact in a disjunct is $\langle x \rangle = \alpha \circ \beta$, where $\alpha$ and $\beta$ are strings of $x$’s type. The algorithm indicates that we split the disjunct into two, where this fact is replaced in one with $\langle x \rangle = \beta \land \alpha = \Lambda$, and the other with $\langle x \rangle = \alpha \land \beta = \Lambda$. Inside the former split, we may then substitute all instances of $\beta$ with $\langle x \rangle$ and $\alpha$ with $\Lambda$, and similarly for the latter.

\texttt{SplitDecision} has quickly grown beyond its origins of handling strings, for one simple reason: with a focus on string theory, \texttt{SplitDecision} was able to prove more VCs, faster, than our main automated prover at the time, \texttt{Isabelle}, and thus we were encouraged to broaden its scope. It has since added the ability to handle finite sets, tuples (\textit{i.e.}, Cartesian products), and indeed some integer theory in an attempt to see where a similar exploration of decision procedures for these theories would lead. The
original focus on efficient splitting and substitutions continues to guide exactly what rules make sense to be added to \texttt{SplitDecision}. As the entirety of the original string theory rules, as well as a description of the fragment of string theory for which it is complete, are found in [20], we shall instead motivate by an example.

2.2 An example

What follows are the actual steps taken by \texttt{SplitDecision} to process a generated VC,\footnote{The VC in question is for the \texttt{ensures} clause of \texttt{RemoveLeft} for the \texttt{Recursive} implementation of an enhancement \texttt{FullReverse} of \texttt{ListTemplate}.} shown in Figure 2.2. For the sake of clarity, facts irrelevant to the proof have been removed. Additionally, the names of variables have been changed to more easily discern their type: greek letters are used for strings of objects; the middle portion of the alphabet for integers; and the end of the alphabet for variables of type object. The unadulterated version of this VC takes under 250 milliseconds on an 2.8 GHz Intel Xeon with 1 GB of RAM at http://resolve.cse.ohio-state.edu:8080/ResolveVCWeb as of this writing, and no other tools that we use (\texttt{Z3}, \texttt{Isabelle}) are able to prove this VC. \texttt{SplitDecision}'s proof applies 108 small steps, so we will only show steps that help to demonstrate its process.

\begin{align*}
\text{Prove} & \quad \alpha = \beta \\
\text{Given} & \quad \beta \circ \gamma = \Lambda \circ \alpha \circ \delta \\
& \quad \land \ i + |\alpha \circ \delta| = |\gamma| + |\alpha| \\
& \quad \land \ i \neq 0 \\
& \quad \land \ i \geq 0
\end{align*}

Figure 2.2: The initial example VC
To better see why the VC is true and what the pitfalls might be, it is worth trying it by hand first. The last two givens, 2.3 and 2.4, imply that $i = 0$. We may then take the given 2.2 to tell us $|\delta| = |\gamma|$, as $|\alpha \circ \delta|$ is the same as $|\alpha| + |\delta|$ and $|\alpha|$ can be subtracted from both sides. Since $\gamma$ and $\delta$ are suffixes in the first given 2.1, and they have the same length, it must be that $\gamma = \delta$, and so they can be removed from the concatenation. We then essentially have what we wish to prove.

The proof above requires a substantial amount of hand waving. We wish SplitDecision to both be able to prove it, and to do so formally. However, we would like to also be able to understand this proof. SplitDecision’s algorithms start by taking the VC, negating it, and putting it into disjunctive normal form, shown in Figure 2.3 (when only one disjunct exists, we defer to showing it as just the conjunction itself).

$$\beta \circ \gamma = \Lambda \circ \alpha \circ \delta$$
$$\land \ i + |\alpha \circ \delta| = |\gamma| + |\alpha|$$
$$\land \ i \leq 0$$
$$\land \ 0 \leq i$$
$$\land \ \alpha \neq \beta$$

Figure 2.3: VC from Figure 2.2, negated and put in disjunctive normal form.

The first major step that SplitDecision takes is replacing the length of the concatenation of strings as the sum of the lengths of each individual string, specifically replacing $|\alpha \circ \delta|$ with $|\alpha| + |\delta|$. A second step is “pruning” inequalities, which in this case means knowing that the only way $i \leq 0$ and $0 \leq i$ could be true is if $i = 0$. Our current state of affairs is shown in Figure 2.4.
\[ i = 0 \]
\[ \land \beta \circ \gamma = \Lambda \circ \alpha \circ \delta \]
\[ \land |\gamma| + |\alpha| = i + |\delta| + |\alpha| \]
\[ \land \alpha \neq \beta \]

Figure 2.4: The VC after length of concatenations and pruning of inequalities are applied, after 7 applied steps.

At any available situation, \texttt{SplitDecision} works to apply substitutions, and in this case it replaces all instances of \( i \) with 0, and then removes the fact “\( i = 0 \)”. As it knows 0 is the identity for addition, it removes any additions of 0 (which in this case arose from \( i \)). \texttt{SplitDecision} also removes summands it finds on both sides of an equality or inequality, so \( |\gamma| + |\alpha| = i + |\delta| + |\alpha| \) is now \( |\gamma| = |\delta| \). Finally, there is a string decision procedure step which removes all instances of \( \Lambda \) from a concatenation. \texttt{SplitDecision} has now processed the VC to what is shown in Figure 2.5.

\[ |\gamma| = |\delta| \]
\[ \land \alpha \circ \delta = \beta \circ \gamma \]
\[ \land \alpha \neq \beta \]

Figure 2.5: Processing of the VC after a total of 14 applied steps.

We see from looking back at our human attempt that \texttt{SplitDecision} has brought us to a very similar state. All that is missing is the final deduction, which required the greatest amount of hand waving on our part. At this point the steps diverge. An
important rule for \texttt{SplitDecision} when handling strings is to recognize that a situation of \(\alpha \circ \delta = \beta \circ \gamma\) means either \(\alpha \circ \delta = \beta \circ (\zeta \circ \delta)\) for some string \(\zeta\), or \(\alpha \circ \delta = (\alpha \circ \xi) \circ \gamma\), for some string \(\gamma\). From this, it is able to derive Figure 2.6.\(^9\)

\[\begin{align*}
\gamma &= \zeta \circ \delta \\
\land \quad \alpha &= \beta \circ \zeta \\
\land \quad |\gamma| &= |\delta| \\
\land \quad \alpha &\neq \beta \\
\lor \quad \delta &= \xi \circ \gamma \\
\land \quad \beta &= \alpha \circ \xi \\
\land \quad |\gamma| &= |\delta| \\
\land \quad \alpha &\neq \beta
\end{align*}\]

Figure 2.6: Processing of the VC after a total of 15 applied steps.

That is, by splitting into two and introducing a new variable in each split, \texttt{SplitDecision} is next able to substitute out two other variables. The design of \texttt{SplitDecision} facilitates case exploration by continual splitting and substitutions based on the rule applied; in every instance no backtracking is necessary, because a case that cannot happen will simply be removed upon finding the contradiction. There is always a step towards progress, even if it does not directly match human reasoning.

Take, for example, the VC after substitutions have been done and the lengths of concatenations properly broken, shown in Figure 2.7. It is at this point straightforward\(^9\)

\(^9\)To see how this rule makes sense, realize that either \(|\alpha| \leq |\beta|\), or \(|\alpha| \geq |\beta|\), so intuitively either \(\beta = \alpha \circ \xi\) for some \(\xi\), or \(\alpha = \beta \circ \zeta\) for some \(\zeta\).

\(^{10}\)Note that the splits appear in a different order, caused by \texttt{SplitDecision}'s enforcement of a canonical ordering.
to see what can be done; remove common summands in each split, \( i.e., \) deduce \(|\gamma| = 0\) for the first and \(|\zeta| = 0\) for the second. Then, SplitDecision can deduce \(\gamma = \Lambda\) and \(\zeta = \Lambda\), respectively, leaving the problem to be one of substitutions, and of finding the contradictions laid apparent at that point. SplitDecision is continually striving to make the best choice, but a major tenet for it is that no matter the order of rule application, the time it takes to process a VC will be the only difference in the end result. In this situation, SplitDecision applies the rule that deals with string inequality, shown in Table 2.1, which is a less direct path in the proof. From here, the processed VC has quadrupled the number of splits, from two splits to eight! Within roughly 80 small steps, SplitDecision is able to narrow the possibility of a contradiction down to two specific possibilities, each of which is dispatched right away.

\[
|\gamma| = |\gamma| + |\xi| \\
\land \alpha \neq \alpha \circ \xi \\
\lor |\delta| = |\delta| + |\zeta| \\
\land \beta \neq \beta \circ \zeta
\]

Figure 2.7: Processing of the VC after a total of 21 applied steps.

It is possible, although preferably unlikely, that SplitDecision is unable to verify or refute a particular VC. In this case, SplitDecision provides the altered formula, negated again for the user or next prover to work on.
Table 2.1: Procedure for breaking a fact of $\tau \neq \omega$, where $\tau$ and $\omega$ are strings. Here, $m$ is a new variable of type integer, and $\tau$, $\omega$ are strings of the type of $v$ and $z$.

<table>
<thead>
<tr>
<th>Reasoning</th>
<th>Split</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$ is empty so $\omega$ is not</td>
<td>$\tau = \Lambda$ \wedge 0 &lt; $</td>
</tr>
<tr>
<td>$\omega$ is empty so $\tau$ is not</td>
<td>$\omega = \Lambda$ \wedge 0 &lt; $</td>
</tr>
<tr>
<td>Both are nonempty but lengths are different</td>
<td>$0 &lt;</td>
</tr>
<tr>
<td>Both are nonempty and lengths are the same, so an entry is different</td>
<td>$0 &lt;</td>
</tr>
</tbody>
</table>

2.3 Development

SplitDecision is written in Resolve/C++ [28], a disciplined way of designing C++ code that mimics how one develops code in the Resolve programming language the tool itself is developed to verify. The primary reason motivating this choice is simply that Resolve/C++ code is simultaneously powerful, efficient, and easy to write. An ancillary technical benefit is that the code itself is theoretically easily translatable into ordinary Resolve code that will be verifiable using SplitDecision.

As Hoare himself points out [27], it is unnecessary to verify the verifier. Instead the verifier may produce a certificate, that is, a proof at a “low enough level of sophistication” that it can be verified by a human, or by a “proof checker” whose correctness is easier to establish manually. Needless to say, it is still undesirable to
design an unsound verifier. By designing \texttt{SplitDecision} in the same language that we strive to verify, we leave open the possibility to verify the correctness of the verification itself, with certificates required only to break the seemingly inescapable circularity.

Pointers are often considered integral to writing anything nontrivial in C++, something that might be detrimental to verifying \texttt{SplitDecision}. The Resolve discipline rejects this conventional wisdom; while certainly aliasing exists at the foundational level (as we will see in Chapter 3), any other necessary use of pointers is encapsulated properly in the design of components that offer pointer-free abstractions for clients. This can be best seen by noting that, as of November 18, 2010, \texttt{SplitDecision} is made up of 57,805 lines of code spread out over 159 files. These numbers do not include any existing Resolve/C++ components that worked “as is” for \texttt{SplitDecision}, nor any Resolve/C++ foundation files, \textit{e.g.}, C++ code for built-in Resolve/C++ types. Of these 159 files, a total of 2 (1.4%) have any mention of pointers, specifically an implementation of \texttt{Array} and \texttt{List} (encapsulations of dynamic arrays and singly-linked lists for Resolve/C++), both of which required necessary changes due to the details discussed in Chapter 3. That is, of 57,805 lines of code, only 359, or 0.62%, have \textit{any} mention of pointers.\footnote{And this, of course, is an upper bound, as not all lines of code in these files are for managing pointers.} It should be clear that pointers cause little difficulty in reasoning about \texttt{SplitDecision}’s code, nor does a nearly complete lack of pointers hamstring the power of Resolve/C++ code.

\section*{2.4 Results}

To test \texttt{SplitDecision}, we applied it to every VC generated for every component in the Resolve software component catalog as it existed on October 15, 2010. This
repository contains realizations or enhancements (or both) for components such as Queue, Stack, Array, List, Set, Integer, and UnboundedInteger as well as a number of known defective components of this nature. The mathematical types involved for the repository are strings, finite sets, tuples, and integers, along with an uninterpreted type “object” in the case of templates. Some VCs include local mathematical definitions used in specifications, which SplitDecision currently does not deal with, and existential quantifiers, which SplitDecision has a limited ability to handle.

While Resolve also includes Z3 in its list of automated provers, the current usage of Z3 puts it under numerous disadvantages; with some exceptions it is only useful for VCs involving integers.¹² For this reason, we shall compare SplitDecision against Isabelle only. The computer used is a 2.8 GHz Intel Xeon with 1 GB of RAM. SplitDecision was allowed to run with no timeout until it finished applying rules, while Isabelle had a timeout of 3 seconds.

Table 2.2: Comparison of SplitDecision and Isabelle in terms of categorizing VCs. The last column examines our best attempts at identifying VCs, using Z3 to find 2 more are valid, and a tool Contra to find 52 more invalid. There are 1,494 VCs that both SplitDecision and Isabelle find to be true.

<table>
<thead>
<tr>
<th></th>
<th>SplitDecision</th>
<th>Isabelle</th>
<th>Combined</th>
<th>Additional tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>1,748 (85.90%)</td>
<td>1,570 (77.15%)</td>
<td>1,824 (89.63%)</td>
<td>1,826 (89.73%)</td>
</tr>
<tr>
<td>Unsolved</td>
<td>285 (14.00%)</td>
<td>465 (22.85%)</td>
<td>209 (10.27%)</td>
<td>155 (7.62%)</td>
</tr>
<tr>
<td>Invalid</td>
<td>2 (0.10%)</td>
<td>0 (0.00%)</td>
<td>2 (0.10%)</td>
<td>54 (2.65%)</td>
</tr>
</tbody>
</table>

¹²Specifically, we have given Z3 limited information about the Resolve mathematical theories due to efforts being applied elsewhere. Another student is currently working on making Z3 more useful in proving Resolve VCs.
SplitDecision finds 1,748 of the VCs valid (85.90%) and 2 to be invalid (0.10%). It is unable to categorize 285 VCs (14.00%). The comparison of SplitDecision and Isabelle is shown in Table 2.2. Notably, SplitDecision is able to verify a 178 more VCs than Isabelle. This is not the whole story, however. SplitDecision and Isabelle overlap in verifying 1,494 VCs, meaning that they complement each other to some extent, together verifying 1,824 VCs (89.63%). As an interesting aside, if we make use of additional tools at our disposal for Resolve VCs (Contra and Z3), we find there are only 155 VCs we are unable to categorize (7.62%).

Table 2.3: Comparison of SplitDecision and Isabelle times. Separate comparisons done between VCs both verify, VCs both are unable to categorize, those that only Isabelle can categorize (i.e., prove), and those that only SplitDecision can categorize (i.e., prove or find invalid).

<table>
<thead>
<tr>
<th></th>
<th>Isabelle</th>
<th>SplitDecision</th>
<th>SplitDecision/Isabelle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both Categorized as Valid</td>
<td>12m 8.378s</td>
<td>1m 5.739s</td>
<td>11.080×</td>
</tr>
<tr>
<td>Both Unsolved</td>
<td>21m 24.330s</td>
<td>2m 11.810s</td>
<td>9.744×</td>
</tr>
<tr>
<td>Only SplitDecision</td>
<td>19m 54.803s</td>
<td>37.576s</td>
<td>31.797×</td>
</tr>
<tr>
<td>Only Isabelle</td>
<td>1m 0.113s</td>
<td>58.260s</td>
<td>1.032×</td>
</tr>
<tr>
<td>Total</td>
<td>54m 27.624s</td>
<td>4m 53.385s</td>
<td>11.138×</td>
</tr>
</tbody>
</table>

A comparison of execution times between Isabelle as SplitDecision is shown in Table 2.3. To process all 2,035 VCs, Isabelle took roughly $54\frac{1}{2}$ minutes. For the same VCs, SplitDecision took less than 5 minutes, meaning SplitDecision is over 11 times faster than Isabelle on average. As seen in the table, SplitDecision truly shines for the 1,750 VCs it is able to categorize, shown in the first and third rows. The only place Isabelle is able to come close is for the 76 VCs it can prove that SplitDecision
Figure 2.8: Histogram of the number of VCs that can individually be processed, over time to process (2035 VCs total), with steps of 100ms.

cannot. While even then Isabelle takes 2 seconds longer, that is a worthwhile tradeoff if it means solving the VCs.

The impressive numbers for SplitDecision can be seen in Figure 2.8, a histogram of the number of VCs processed by SplitDecision (using increments of 100 ms). 1,928 VCs are processed in at most 200 ms each. The remaining 107 VCs take longer, most of which are shown. There are 4 VCs not shown; the longest takes 29.880 seconds.

Quickly processing the files is not very useful if SplitDecision leaves most of them unsolved. Figure 2.9 breaks up Figure 2.8’s in to the VCs SplitDecision proves and the ones left uncategorized.\textsuperscript{13} Notably, we see that 84\% of the VCs are proved in at most 200 ms. Most of the VCs that take longer are in fact VCs SplitDecision cannot

\textsuperscript{13}The two VCs SplitDecision finds invalid are not shown. They took 20 ms and 70 ms.
Figure 2.9: Graph of the percent of VCs processed, over time to process, partitioned into “valid” (1748 VCs in total) and “unsolved” (285 VCs in total). The VCs found false by SplitDecision are not shown.
categorize. A better examination of \texttt{SplitDecision}'s work in the first 100 ms is shown in Table 2.4. In 20 ms each \texttt{SplitDecision} processed over a quarter of the VCs, in 30 ms each over half, and in 50 ms each over three-quarters. Even if we consider only the VCs \texttt{SplitDecision} can verify or find invalid, it takes a mere 20 ms each to categorize over a quarter of the VCs, 30ms each to handle over half, and 60 ms each to deal with more than three-quarters.

Table 2.4: Number of VCs found valid, unsolved, or invalid by \texttt{SplitDecision} within 100 ms each.

<table>
<thead>
<tr>
<th>Time Range</th>
<th>Valid</th>
<th>Unsolved</th>
<th>Invalid</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 10ms</td>
<td>119</td>
<td>0</td>
<td>0</td>
<td>119 (5.85%)</td>
</tr>
<tr>
<td>10ms–20ms</td>
<td>506</td>
<td>8</td>
<td>0</td>
<td>514 (25.26%)</td>
</tr>
<tr>
<td>20ms–30ms</td>
<td>442</td>
<td>27</td>
<td>1</td>
<td>470 (23.10%)</td>
</tr>
<tr>
<td>30ms–40ms</td>
<td>229</td>
<td>15</td>
<td>0</td>
<td>244 (11.99%)</td>
</tr>
<tr>
<td>40ms–50ms</td>
<td>116</td>
<td>18</td>
<td>0</td>
<td>134 (6.58%)</td>
</tr>
<tr>
<td>50ms–60ms</td>
<td>85</td>
<td>33</td>
<td>0</td>
<td>118 (5.80%)</td>
</tr>
<tr>
<td>60ms–70ms</td>
<td>68</td>
<td>12</td>
<td>0</td>
<td>80 (3.93%)</td>
</tr>
<tr>
<td>70ms–80ms</td>
<td>52</td>
<td>12</td>
<td>1</td>
<td>65 (3.19%)</td>
</tr>
<tr>
<td>80ms–90ms</td>
<td>22</td>
<td>15</td>
<td>0</td>
<td>37 (1.82%)</td>
</tr>
<tr>
<td>90ms–100ms</td>
<td>30</td>
<td>24</td>
<td>0</td>
<td>54 (2.65%)</td>
</tr>
</tbody>
</table>

\texttt{SplitDecision} clearly demonstrates itself to be a strong automated prover for these Resolve VCs. It validates both the design decisions made in Resolve to allow for code to be both verifiable and efficient, first in that the code being verified is written in Resolve, and that \texttt{SplitDecision} is written in Resolve/C++. Finally, \texttt{SplitDecision} demonstrates that the decision procedures in [20] are practical, as are the new procedures that handle other theories designed in the same spirit.
2.5 Future work

SplitDecision is an ongoing project. It was originally built with the consideration that anything left unsolved could be passed on to a tool such as Isabelle. It has, instead, grown to be able to verify outright a large fraction of the VCs we have examined thus far. However, decision procedures for all of the core Resolve theories are not yet developed or implemented, only those that are currently in use with Resolve at OSU. Indeed, most of them are undecidable, but decidable fragments may be investigated. These features will be added as they are incorporated into the Resolve compiler. The best ways to handle tree theory, partial functions, and multisets, for example, are still ongoing research questions.

A large repository of generated VCs, 2,035 at this time and growing, come from implemented realizations and we can use them to determine where SplitDecision’s weaknesses are for the current theories. Currently, the biggest area of consideration is in handling local mathematical definitions, i.e., specialized definitions useful for a particular component, enhancement, or realization. These are often used either to make the contracts more human readable, or to encapsulate quantifiers, which are problematic for theorem provers in general. In the future, SplitDecision needs to decide when it is acceptable to expand the definition and when not to.

Along these same lines, a proper way to provide SplitDecision with rules for new mathematical functions and definitions is a major next step. SplitDecision was originally designed with specific decision procedures for a fixed set of functions in mind, but as we add new ones it is apparent that a language for providing new rules needs to exist, for better prototyping abilities. It is unlikely that this would become the de
facto method of adding finalized rules, for speed reasons, but being able to test new procedures quickly is essential as SplitDecision matures.

A better comparison of Isabelle and SplitDecision is in order. Specifically, it would be helpful to know which VCs Isabelle does well with, and which it does not. Identifying empirically which theories are the strengths of each prover, and likewise which are the weaknesses, would allow us to learn from each other about beneficial rules.

At this time, SplitDecision has a fixed order for the rules that it applies, not due to the rules being brittle, but because of an early design decision. Studying an effective method for contextually ordering of rules can potentially offer speed benefits, by either removing applications of rules in cases they are unnecessary, or reordering their application for greater efficacy.

Finally, we wrote SplitDecision modularly in Resolve/C++ with the Resolve discipline, meaning the change of components is very easy. An exploration of better realizations for the components may provide some benefit. One possibility is to wrap C++’s Standard Template Library components. Perhaps some of the STL implementations are better suited for SplitDecision. If not, that is a useful result in itself; if so, then SplitDecision improves.

2.6 Related work

The idea of combining disparate decision procedures originates from Nelson and Oppen [42]. The simplifier they describe, however, allows direct reasoning about functional programming primitives such as car and cdr, and so is not directly related to SplitDecision. Additionally, SplitDecision makes no attempt to separate the decision
procedures between mathematical theories, and instead considers them to be a unified whole. Such situations are rare, but definitely not disallowed.

**Simplify** [14] is a verification condition simplifier used in static checkers such as ESC/Java. Its purpose restricts it to the discovery of a predefined subset of potential run-time errors such as out-of-bounds array accesses. We hope that, short of theoretical limits such as the halting problem, SplitDecision will be capable of identifying arbitrary errors as we intend Resolve for full functional verification.

**Calysto** [2] is an extended static checker that uses the automated theorem prover/SAT solver SPEAR. It generates VCs based on an intermediate object code representation of the whole program. The SPEAR simplification is fully automatic and supports Boolean logic, bit vector operations, and bit-accurate arithmetic. The primary distinction between this work and ours is the use and design of SPEAR to check verifications resulting from the compiled object code, whereas we use SplitDecision to check verification conditions that operate at a higher level of abstraction, giving it a chance to “scale up” to arbitrary code via judicious use of abstraction and, of course, modularity.

**Isabelle** [59, 44] is the main outside tool used with Resolve. It offers a strong ability as an interactive proof assistant, but for the same reason suffers somewhat when used automatically. Working with Isabelle for Resolve is an ongoing effort.

Verifying Resolve can also be currently done with **Z3** [39, 35, 40], which another verification project, Boogie [6], uses as a backend. We are currently exploring how Boogie fits in with our tools.

**Yices** [54, 7, 17, 30] is an SMT solver made to handle linear arithmetic, bit-vectors, lambda expressions, quantifiers, and recursive datatypes, for example. We might consider it as an additional tool for our tool chain, but as an SMT solver its theories
are likely needed to be more separable than what SplitDecision handles. Isabelle is able to utilize it as an internal solver, so this is a more likely proposition. PVS [46] is a theorem prover that can use Yices. It, like Isabelle, gears itself to interactive use, limiting its usefulness for us. ACL2 [32] likewise suffers this problem and is more suited to doing inductive proofs.
CHAPTER 3

Lazy Copying

Having a verifiable language, such as Resolve, is of limited practical value if the executables are inherently too slow or require too much memory. This concern was raised in particular with SplitDecision written in Resolve/C++. Here we discuss a modification to the foundation of Resolve/C++ that allows for making copies of objects efficiently while still maintaining value semantics.

3.1 Resolve and value semantics

Resolve/C++ employs many tactics to allow for value semantics while maintaining efficiency. For example, the swap operation is available for all types (except pointers) as the data movement operator [24, 47]. A major upshot is that it prevents unintentional aliasing, a mistake quite easy to make in C++. Simple things, such as repeated arguments, can also cause unintentional aliasing and are avoided via the discipline.

C++ provides a default assignment operator and copy constructor that make shallow copies, again causing (probably) unintentional aliasing. This is prevented in Resolve/C++ by making these operations private.

To avoid shallow copying means there is no “built-in” method of copying an object in Resolve/C++. If copying is necessary, the Copy_To extension may be implemented.
with the kernel operations of the type in question, based on the specification found in Listing 3.1. As any actual aliasing must be encapsulated within a component under the Resolve discipline, this leaves us without cross-component aliasing, but at a cost of time and space for deep copies.

```c++
procedure Copy_To (  
    produces T& x  
) is_abstract;  
/*!  
 preserves self  
 ensures  
   x = self  
 */
```

Listing 3.1: Specification for Copy_To in Resolve/C++.

Most of the time, this is perfectly fine. However, there are instances where the ability to make copies is necessary. Take, for example, the following logical formula

\[(a \lor b) \land c \land d \land e\]

converted to disjunctive normal form as

\[(a \land c \land d \land e) \lor (b \land c \land d \land e)\]

using one of De Morgan’s Laws. If we were to represent a logical formula using a tree data structure (and we do in SplitDecision), then unless some fancy special-purpose footwork is done it is necessary to copy the part of the logical formula that represents \(c \land d \land e\). Further, one can imagine this to get much worse with the potential of converting to disjunctive normal form having an exponential increase in formula size. There is further salt on the wound if it is later found out that, say, \(b \land c\) is a logical impossibility, and thus the whole copy was an unnecessary expenditure.

To solve this, we could implement some form of lazy expansion of the logical formula as needed, making copies only when necessary. For someone developing the
logical formula implementation, it is likely this technique would complicate the design, and worse yet, the benefits would only be felt for that specific realization. If we wished to reuse the lazy expansion for something else, it would have to be re-implemented for the new component.

The goal of this chapter is to show how to make lazy copying available for all components with (except rarely) no additional work for the component implementation.

3.2 Adding efficient copying to Resolve

Before understanding how lazy copying can be added to all Resolve components at once, with little or no component-specific code, it is necessary to grasp how data representations are handled internally in Resolve/C++.

3.2.1 Data representation in Resolve

A key aspect of Resolve/C++ is ensuring that swap, provided for all Resolve/C++ types with the operator &=, is a constant-time operation. As every non-primitive type uses a Representation object to store its data members, supplying constant-time swapping turns out to be straightforward conceptually. Instead of an object internally being a collection of its fields, it is a pointer to a collection of the fields. Figure 3.1 demonstrates how this facilitates swapping of any two objects in constant time: we simply swap the pointers to the two collections of data members.

All of this is hidden from anyone encapsulating the Representation object (i.e., using it to store the data members of the type), as the access to the representation is through specially designed accessors. Every change to the foundation of Resolve/C++

\footnote{Thus, Resolve/C++ makes use of pointers even within its foundation, but in a controlled, disciplined manner.}
Figure 3.1: How swap actually is implemented for non-primitive types.

has to keep this transparency in mind. For example, Resolve/C++ makes use of lazy initialization, which itself is completely hidden from both the developer of the component and the client using it. As constructors and destructors have no side effects in Resolve/C++, the pointer to the actual fields in `Representation` does not allocate memory for the fields until it is necessary. Instead it points to `NULL`, to represent an object in its initially-declared state. Thus, if the fields are never needed, we save on having to allocate (and later deallocate) objects unnecessarily. In this case, we need not make sure the memory is properly initialized, or, in the case of pointer use, cleaned up with a finalize operation. If the fields are ever accessed, Resolve/C++ guarantees that the type’s `Initialize` operation is called before the access. Since components are designed generics, i.e., with C++ templates, all any code can do by default to an object of an unrestricted parametric type is initialize, finalize, swap, or clear. With lazy initialization, none of these access a field. For example, take the code shown in Listing 3.2.
Inside the `while` loop, the object `x` is a parameterized type `Item`, and `Dequeue` and `Enqueue` can only do the four operations listed above. Thus, a `NULL` pointer is created on the declaration of `x`, `Dequeue` internally uses swap and destroys the `NULL` pointer, `Enqueue` internally creates an object of type `Item` and uses swap, and finally, `x` goes out of scope—destroying a `NULL` pointer in the end. At no point did `x`’s fields get accessed, so that at no point nothing of type `Item` causes `Initialize` to be called.

```c
while (q.Length () > 0)  
  /*!
     alters self, q
     maintains
     self * q = #self * #q
     decreases  
     |q|
  */
{
  object catalyst Item x;
  q.Dequeue (x);
  self.Enqueue (x);
}
```

Listing 3.2: Example iteration through a `Queue`

The Resolve discipline does not allow any use of aliasing to escape from a component, meaning any use of pointers must be hidden within the implementation. See, for example, the chain of component dependencies for `SplitDecision` and its related projects in Figure 3.2 (including some components not currently in use). While this certainly looks complicated, we know without a doubt that there is no aliasing from one component to another, as each component was designed specifically to avoid this (in fact, a vast majority of the components do not use pointers at all in their implementations). The graph would actually be a mere three layers deep (from left to right), if not for partial instantiations of components (where some, but not all, of the template parameters are filled in), common component instantiations between executables made for convenience, and checking components (where the calls to a component’s kernel operations are layered with checking code to catch illegitimate runtime use of
Figure 3.2: Dependencies of code written for SplitDecision. Boxes with rounded corners are “abstract”. Most importantly, all arrows move from left to right; if a component does utilize aliasing internally, it cannot escape.
component). Without these, the layers would consist of the executables, followed by component realizations, component enhancement realizations, and useful utility class realizations, followed by the specifications for each realization.

Most importantly, there are no recursive data structures in either Resolve or Resolve/C++. That is, a Tree implementation might internally be recursive in its own representation, but a Queue implementation cannot use a Stack implementation which then uses the original Queue implementation.

3.2.2 Implementation of lazy copying

We now come to how lazy copying is implemented. The argument that it works is inductive, which applies because there are no recursive data structures in the sense mentioned earlier, and hence have no cycles in the component-to-component dependency graph. The easiest copy is for primitive types such as Integer or Boolean (but not pointers): simply use the built-in assignment operator. It is almost as simple in the case of non-primitive types. To aid us, the structure that is pointed to inside Representation has a new field: an Integer named ref_count. This field initializes to one, and represents the number of objects sharing the data representation. Any time a.Copy_To(b) is called, for some objects a and b, we decrease the reference count for what b points to by one, point b to the same data as a, and increment the reference count, as in Figure 3.3. If any reference count goes to 0, we know that nothing is referring to the object in question and call the Finalize operation for it.

The question now is what to do in the case of an object trying to access its fields. We take the conservative approach if the reference count is greater than 1 and force this object to do the “real” copy operation, called Duplicate_To.

44
All types in Resolve have default Initialize and Finalize operations, namely ensuring that each field has its Initialize and Finalize operations called, respectively. There are times, of course, when this is not enough. For example, if a component uses pointers, a developer must ensure memory is properly allocated on initialization, and cleaned up on finalization. In these instances, the developer may write their own Initialize and/or Finalize operations. Similarly, Resolve/C++ provides a default Duplicate_To operation: on a.Duplicate_To(b), the fields of a are copied to the fields of b, naturally using Copy_To. That is, when copying is “actually done” the copy is pushed down one layer, where it is also done lazily.

There are certain types in Resolve/C++ that do not have a Copy_To operation, most importantly pointers. In this case, a developer must write their own implementation of Duplicate_To, to handle the special data structure brought on by aliasing. This might also be necessary if the data representation uses Records, Resolve/C++’s version of structs (similar to Representation). If any of the fields in a Record

Figure 3.3: Before and after calling a.Copy_To(b) for a non-primitive type. In this case, “rc” is the reference count.
does not have \texttt{Copy\_To}, the \texttt{Record} itself likewise does not, so some special-purpose \texttt{Duplicate\_To} code is needed for the \texttt{Representation} fields that are \texttt{Records}.

### 3.2.3 Additional details

The discussion in Section 3.2.2 makes no mention of handling the combination of lazy initialization and lazy copying. It turns out to be almost as easy to handle as with primitive types: on a call of \texttt{a.Copy\_To(b)}, where \texttt{a} has not had \texttt{Initialize} called yet, and therefore has an internal pointer of \texttt{NULL}, simply set \texttt{b} to have an internal pointer of \texttt{NULL} as well. This demonstrates the importance of building lazy copying into the foundation; had it been designed to be specific to one type, we would not have been able to handle this situation as efficiently. The very act of the implementor checking the state of \texttt{a} would necessarily force the \texttt{Initialize} operation to be called.

To avoid creating a copy when data needs to only be read from a field (not written), a special kind of accessor naming convention was added for use when implementing kernel operations. If a field is accessed by \texttt{self[length]}, for example, changing this to \texttt{self[length\_ro]} lets the system know it is unnecessary to make an actual copy. This is an inconvenient hack, and a better work-around is something to be explored.

While modifying the internals of Resolve/C++’s foundation, a basic memory pool mechanism was added. Each type holds a buffer of a fixed size (defaulting to four fields for \texttt{Representation}). When memory is requested for a \texttt{Representation}’s fields, the pool is checked first before allocating the memory. When memory is no longer needed, it is added to the pool if there is room.
3.3 Results

To see the effectiveness of adding lazy copying, we ran *SplitDecision* using it on 10 VCs: the longest and shortest running VCs for *SplitDecision*, and 8 randomly selected VCs. The results for lazy copying are shown in Table 3.1. We monitored the total number of *Copy_To* calls without lazy copying, the total number of *Copy_To* calls with lazy copying, the actual *Duplicate_To* calls with lazy copying, and *Copy_Tos* that simply resulted in the representation pointer being set to `NULL` from lazy initialization, as discussed in Section 3.2.3. Additionally, the total number of requests for new memory for *Representation* fields was monitored with lazy copying in use, along with the actual number of *New* calls made when the memory pool was empty.

*SplitDecision* is an effective benchmark for lazy copying for several reasons. First, it is one of the largest projects ever built using Resolve/C++, using 36 different instantiations of Resolve components. Additionally, while *SplitDecision* requires the use of *Copy_To* for many of its operations, it was designed to avoid making copies unless absolutely necessary, and therefore is not calling *Copy_To* when it is known not to be useful. Finally, the theories used between one VC and the next can range wildly, meaning different parts of *SplitDecision*’s code are exercised depending on what is required for a particular VC.

Looking at Table 3.1, we see that on average, only 46.70% of the *Copy_To* calls occur when using lazy copying, and only 3.19% on average actually result in a real copy (ranging from as low as 0.33% to as much as 7.07%). Of the average of 96.81% of prevented real copies, an average of 14.56% are prevented by utilizing lazy initialization.

Table 3.2 shows the usefulness of adding memory pools, even with lazy copying on. Memory pools of size four for each instance also turns out to be highly effective. In
Table 3.1: Examination of 10 VCs, comparing the number of `Copy_To` calls made without lazy copying, the number of `Copy_To` calls made with lazy copying, the number of actual calls to `Duplicate_To` with lazy copying, the number of actual copies prevented, and the number of `Copy_To`'s called on an uninitialized object with lazy copying. All percentages are in comparison to the number of `Copy_Tos` without lazy copying. Percentages at the bottom are the average of percentages.

<table>
<thead>
<tr>
<th>Copy_Tos w/o Lazy Copying</th>
<th>Copy_Tos w/ Lazy Copying</th>
<th>Actual Copies</th>
<th>Prevented Copies</th>
<th>Lazy Inits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>184</td>
<td>96</td>
<td>13</td>
<td>171</td>
<td>32</td>
</tr>
<tr>
<td>603</td>
<td>300</td>
<td>2</td>
<td>601</td>
<td>131</td>
</tr>
<tr>
<td>1,274</td>
<td>582</td>
<td>47</td>
<td>1,227</td>
<td>158</td>
</tr>
<tr>
<td>1,439</td>
<td>600</td>
<td>70</td>
<td>1,369</td>
<td>170</td>
</tr>
<tr>
<td>3,782</td>
<td>1,654</td>
<td>139</td>
<td>3,643</td>
<td>328</td>
</tr>
<tr>
<td>3,496</td>
<td>1,676</td>
<td>63</td>
<td>3,433</td>
<td>610</td>
</tr>
<tr>
<td>7,051</td>
<td>3,408</td>
<td>176</td>
<td>6,875</td>
<td>793</td>
</tr>
<tr>
<td>8,515</td>
<td>4,237</td>
<td>187</td>
<td>8,328</td>
<td>1,762</td>
</tr>
<tr>
<td>143,799</td>
<td>66,700</td>
<td>4,647</td>
<td>139,152</td>
<td>16,345</td>
</tr>
<tr>
<td>1,437,354</td>
<td>597,236</td>
<td>37,100</td>
<td>1,400,254</td>
<td>185,131</td>
</tr>
<tr>
<td>1,607,497</td>
<td>676,489</td>
<td>42,444</td>
<td>1,565,053</td>
<td>205,460</td>
</tr>
</tbody>
</table>
Table 3.2: Examination of 10 VCs, comparing the result of adding a memory pool to Resolve/C++ with a fixed size of four. Shown is the number of requests for new memory against the actual uses of `New`, with lazy copying in place.

<table>
<thead>
<tr>
<th>New Requests</th>
<th>Actual New</th>
<th>Actual %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,028</td>
<td>401</td>
<td>39.01%</td>
</tr>
<tr>
<td>2,515</td>
<td>608</td>
<td>24.17%</td>
</tr>
<tr>
<td>4,627</td>
<td>715</td>
<td>15.45%</td>
</tr>
<tr>
<td>5,660</td>
<td>1,410</td>
<td>24.91%</td>
</tr>
<tr>
<td>8,786</td>
<td>1,185</td>
<td>13.49%</td>
</tr>
<tr>
<td>14,496</td>
<td>1,254</td>
<td>8.65%</td>
</tr>
<tr>
<td>12,863</td>
<td>1,596</td>
<td>12.41%</td>
</tr>
<tr>
<td>45,559</td>
<td>2,392</td>
<td>5.25%</td>
</tr>
<tr>
<td>384,779</td>
<td>30,665</td>
<td>7.97%</td>
</tr>
<tr>
<td>5,627,710</td>
<td>217,843</td>
<td>3.87%</td>
</tr>
<tr>
<td>6,108,023</td>
<td>258,089</td>
<td>4.23%</td>
</tr>
</tbody>
</table>

total, only 4.23% of the requests for new memory actually required a call to `New`. Over the 10 VCs this ranged from an astoundingly low 3.87% to a more modest 39.01%.

We can also examine how `SplitDecision` improves in terms of speed. Figure 3.4 shows the result of running `SplitDecision` on 2,035 VCs without lazy copying or a memory pool, against the speedup factor for that VC when run with lazy copying enabled. These timings were done on a 2.2 GHz Intel Core 2 Duo with 4 GB of RAM. While there is a lot of variance in the difference, we see a general improvement from enabling lazy copying.

Table 3.3 compares the total time of running `SplitDecision` on all 2,035 VCs with or without lazy copying, and with or without the use of memory pools (of size four). Both lazy copying and the use of memory pools demonstrate individual improvements, but interestingly just memory pools on their own is the largest speedup overall. It
Figure 3.4: Comparison of times without lazy copying or a memory pool, with the speedup factor from using lazy copying.

Table 3.3: Comparison of total times to process 2,035 VCs using **SplitDecision** in seconds, with and without lazy copying, and with and without a memory pool. Also shown is the speedup factor compared to no lazy copying and no memory pool.

<table>
<thead>
<tr>
<th>SplitDecision</th>
<th>No Memory Pool</th>
<th>Memory Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without Lazy Copying</td>
<td>260.679</td>
<td>224.278</td>
</tr>
<tr>
<td>With Lazy Copying</td>
<td>251.941</td>
<td>246.321</td>
</tr>
</tbody>
</table>
is likely that lazy copying requires a different size for the memory pools used. Lazy
copying does allocate less to the heap—a randomly selected VC uses 3.15 MB without
lazy copying, but 2.74 MB with—so it will be interesting going forward to see what
interactions are causing this difference. It is worth noting that the difference in the
average time to process a VC is quite small. With no lazy copying or memory pool,
the average time is 128.1 ms. With only a memory pool, this is 110.2 ms. With lazy
copying and a memory pool, it is 121.0 ms.

![Figure 3.5: The data structure of a logical formula in SplitDecision using lazy copying.](image)

Another worthwhile question is the effect of lazy copying on both the code and
the data structure for SplitDecision. A total of two components used by SplitDecision
make use of pointers, and therefore need an implementation of Duplicate_To. No
other changes are needed. What is gained is that every Resolve/C++ type has a
Copy_To operation available if necessary. As mentioned earlier, there are a total of
36 different instantiations of realizations in SplitDecision. Of these, four made use of
the custom Duplicate_To. That is, of the two components requiring Duplicate_To implementations, SplitDecision instantiates them with template parameters to a total of four different possibilities. None of this, of course, need be known by the programmer.

The actual data structure in SplitDecision is of course a bit complicated by these optimizations. A high-level view of a VC represented in SplitDecision is shown in Figure 3.5. A zoomed-in view of some of the lower portion is shown in Figure 3.6. Notably, there are many horizontal arrows, demonstrating where lazy copying is active at the time. At the bottom of the figure, we see a massive number of arrows pointing to NULL, a demonstration of the combination of lazy initialization and lazy copying. Again, the design of components in Resolve/C++ prevents the possibility of any arrow going upwards; this structure, no matter how complicated it is, is a directed acyclic graph. While it should be clear that aliasing is integral to the efficiency of code in Resolve/C++, Figures 3.5 and 3.6 clearly demonstrate that it is Resolve’s commitment to value semantics via strict abstraction boundaries that makes this possible to understand and reason about, one level at a time.

3.4 Future work

One area of future improvement is finding a way to determine whether an access to Representation is a “get” or a “set”, instead of adding the “_ro” to a Representation accessor. C++ does not provide a way to automatically convert a field access into a get or set, meaning it needs to be explored further.

Further work in seeing what size memory pool is most effective is in order. Using a pool of size four has been very convenient, but it is possible that a larger or smaller size could provide a greater benefit (and Table 3.3 strongly suggests that it needs to
Figure 3.6: Zoomed in view of the data structure shown in Figure 3.5.
be changed for lazy copying). Another possibility is the use of free lists instead; this would have the drawback of never giving back memory until the end of the program, but would offer the potential to greatly minimize the total number of `new` called.

Both lazy copying and use of a memory pool were developed under the assumption that the program is sequential. While threaded programs are not currently accommodated in Resolve/C++, understanding the difficulties these changes present will be important in going forward.

Work is ongoing on the actual development of full Resolve (without C++). At the moment, Resolve does not compile to executable code. When it does, finding a way to bring the gains explored here would be highly beneficial to demonstrating Resolve to be more than a “toy” language.

### 3.5 Related work

The use of lazy copying, also called copy-on-write, is not new. It is common, for example, in functional languages such as Lisp [57, 50, 3]. Many scripting languages either have copy-on-write built in, or on a type-by-type basis [56]. The implementation presented here represents the first procedural implementation that uses value semantics. Further, all other known implementations, functional or procedural, use garbage collection and are therefore unable to fully make use of the reference counting added for the purposes of lazy copying. Resolve/C++, on the other hand, disallows arbitrary recursive data structures and therefore can rely *solely* on reference counting.
CHAPTER 4

Debugging in a World of Verification

In the current state of the world, finding out there is an error in one’s code (or a potential error) leads to debugging. In a world with software verification where there is a problem in verifying one’s code, finding out there is an error in one’s code (or a potential error) leads to debugging. Are the debugging processes different? Yes.

This chapter deals with the specific issue of handling the situation shown in the dotted line in Figure 4.1, when the automated prover or provers either come back saying the code is invalid, or there is not enough justification of correctness. In either case, the code is not compiled for execution. Further, as we shall see, automatically discovering that there is an error is different than being able to correct it. This chapter discusses techniques to help make the connection, by making sure the programmer is well-informed of the situation without being inundated with unnecessary details.

4.1 Tools at our disposal

Various tools are mentioned throughout this chapter. One tool in particular, SplitDecision, will be relied upon heavily for this discussion, and more can be read about it in Chapter 2. Beyond SplitDecision, two other automated theorem provers
Figure 4.1: System overview of a Resolve development and verification system. The dotted line highlights the main focus of this chapter.

are in use with Resolve at The Ohio State University, and as such will be mentioned throughout the chapter. They are Isabelle and Z3, and are used to varying degrees.

Isabelle is meant to be a proof assistant, but has been somewhat repurposed in the OSU Resolve tools to be an automated theorem prover, requiring no help from the user. Unlike SplitDecision, its proof techniques are not specifically designed to handle the mundaneness of actual Resolve VCs, and so it can be considered anything but efficient when asked to prove one. Using Isabelle does provide a strong advantage, even if not directly. We may also employ it to verify the actual lemmas used in our Resolve mathematical theories, an important part of satisfying the need to be sound.

Z3 from Microsoft Research, on the other hand, is constructed for proving VCs. While it comes with a strong background in integer theory, it is designed to be given rules for handling other theories. Z3 only really shines currently for VCs resulting from programs written in Resolve (Resolve VCs) that compute primarily with integers.
This is partly because we have not fully exploited Z3’s capabilities, and partly because Z3 is inherently limited in some ways.

Tools such as SplitDecision, Isabelle, and Z3 specialize in proving, not disproving, theorems.\textsuperscript{15} This generally leaves us with the ability to classify VCs as “valid” or “unsolved”. It is in our best interest to be able to also categorize VCs as invalid when possible. First, it provides an ability to say “invalid” instead of merely “unsolved”, along with a counterexample that demonstrates why it is invalid. Second, it allows us to check VCs already shown valid for errors in the provers, a useful instrument for the development of validation tools built without validation tools already in existence.\textsuperscript{16}

Contra is the main tool used in the OSU Resolve tools to find counterexamples. Designed at OSU, it looks for counterexamples among a particular range of values using the “small scope hypothesis” [29, 52]. This is the thesis that if a counterexample exists, there is likely to be a small one; near 0 for integers, of a small length for strings of items, etc. Many of the difficulties Contra faces are relevant to any such tool. We do not depend on Contra definitely finding a counterexample if there is one; rather, since this is impossible, we take advantage of the situation when Contra does find one.

A final common tool we will use is that of a symbolic tracing table. Whereas a regular trace through code will give variable values at each state index, a symbolic trace tracks the mathematical values at each state index, making use of the contracts of operations used and annotations. Likewise, it shows any path conditions at a particular state and obligations needed to progress from that state, all in terms of

\textsuperscript{15} Depending on the interface used, Z3 offers a “counterexample” once all of its applicable rules have run out. Unfortunately for our purposes, no guarantee is made that it is an actual counterexample.

\textsuperscript{16} Proof certificates, a low-level version of the proof that can be hand-checked or, more likely, checked by a dedicated prover (limited to a small set of rules and thus hand-verifiable by a human) removes the need for such a check. Many tools, such as SplitDecision, do not currently provide them.
the mathematical model values of the variables in question. This can, of course, be easily modified by plugging in specific mathematical values to see statically a trace of what would happen on a particular input. Note that this is still not equivalent to a standard “trace”. The symbolic traces used here rely on the contracts and annotations in the code; as these are likely to be relational, our symbolic tracing may only be able to describe the mathematical value of a variable using a relation with other variables.

Table 4.1: Sample portion of a symbolic tracing table

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Conditions</th>
<th>Facts</th>
<th>Obligations</th>
</tr>
</thead>
</table>
| 8     | $t_8 \neq \emptyset$ | $s_8 \cup t_8 = s_0 \cup t_0$
      |                  | $(s_8 \cap t_8) \cup tmp_8 = s_0 \cap t_0$
      |                  | $t_8 \cap tmp_8 = \emptyset$
      |                  | $0 \leq |t_8|$ |
|       | variable $x$: Item | $s_9 = s_8$
      |                  | $t_9 = t_8$
      |                  | $ts_9 = ts_8$
      |                  | $tmp_9 = tmp_8$
      |                  | $ss_9 = ss_8$
      |                  | $is\_initial(x_9)$ |
| 9     | $t_8 \neq \emptyset$ | $t_9 \neq \emptyset$ |
|       | RemoveAny $(t, x)$ | $x_{10} \in t_8$
      |                  | $t_{10} = t_8 \setminus \{x_{10}\}$
      |                  | $s_{10} = s_8$
      |                  | $ts_{10} = ts_8$
      |                  | $tmp_{10} = tmp_8$
      |                  | $ss_{10} = ss_8$ |
| 10    | $t_8 \neq \emptyset$ | |

A fragment of a sample symbolic tracing table is shown in Table 4.1, exploring code we will look at further in Section 4.5. At each state index, the variables are subscripted with a numeric value, to distinguish changes while progressing through the code. Thus we may read $s_9 = s_8$ and $s_{10} = s_8$ to mean the mathematical value of
s does not change in state indices 9 and 10 (equaling the value of s at state 8). It is also functional, so that knowing \( s_8 \) gives us \( s_{10} \). On the other hand, \( x_{10} \) can only be described relationally in terms of \( t_8 \), so we give it in that manner. For example, if \( t_8 = \{a, b\} \) for some values of \( a, b \), then \( x_{10} \) could either be \( a \) or it could be \( b \); which value \( x \) will obtain in state 10 is unknowable from the contract for \textit{RemoveAny}.

### 4.2 An example

It is instructive to see an example to understand the issues of debugging when dealing with full-functional verification. We will use a seemingly straightforward one: multiplication of two integers by repeated addition.

#### 4.2.1 Some correct code

We begin by looking at the contract of the operation \textit{Multiply} with integers that are \textit{unbounded}, given in Listing 4.1, and with a purported realization in Listing 4.2. The contracts of operations used in the realization are in Listings 4.3, 4.4, and 4.5.

```plaintext
contract Multiply enhances UnboundedIntegerFacility

  procedure Multiply (updates i: Integer, restores j: Integer)
  ensures
      i = #i * j
end Multiply

Listing 4.1: Contract for \textit{UnboundedInteger’s Multiply}.

realization Iterative implements Multiply for UnboundedIntegerFacility

uses IsPositive for UnboundedIntegerFacility
uses Add for UnboundedIntegerFacility
uses Negate for UnboundedIntegerFacility

procedure Multiply (updates i: Integer, restores j: Integer)
  variable p, nj, z: Integer
  if IsGreater (j, z) then
    loop
      maintains p + i * j = #p + #i * #j and
      nj + j = #nj + #j and
  end loop
```

\(^{17}\)For those familiar with the more conventional tracing tables used in Resolve, this differs by doing reasonable substitutions where possible (and hence not stating \( s_{10} = s_9 \)).
i = #i and z = #z and j ≥ 0

decreases j
while IsPositive (j) do
    Add (p, i)
    Increment (nj)
    Decrement (j)
end loop
else
    loop
        maintains p - i * j = #p - #i * #j and
        nj + j = #nj + #j and
        i = #i and z = #z and j ≤ 0
        decreases -j
    while not AreEqual (j, z) do
        Add (p, i)
        Decrement (nj)
        Increment (j)
    end loop
    Negate (p)
end if
i := p
j := nj
end Multiply

Listing 4.2: Iterative implementation for UnboundedInteger's Multiply.

contract IsPositive enhances UnboundedIntegerFacility

    function IsPositive (restores i: Integer): control
        ensures
            IsPositive = (i > 0)
    end IsPositive

Listing 4.3: Contract for UnboundedInteger's IsPositive.

contract Add enhances UnboundedIntegerFacility

    procedure Add (updates i: Integer, restores j: Integer)
        ensures
            i = #i + j
    end Add

Listing 4.4: Contract for UnboundedInteger's Add.

contract Negate enhances UnboundedIntegerFacility

    procedure Negate (updates i: Integer)
        ensures
            i + #i = 0
    end Negate

Listing 4.5: Contract for UnboundedInteger's Negate.

The contract for Multiply states that the operation takes two Integers, and alters the value of the first integer to be the product of the incoming values. To fully
verify the realization given, a total of 30 VCs must be checked. Of these, 13 can be
dispatched immediately once obvious substitutions have been applied, either due to
a conclusion being of the form \( a = a \), i.e., not requiring any math whatsoever, or
a conclusion also appearing as a given fact. It would be of little utility to go over
the remaining 17, but it is instructive to look at a symbolic tracing table with the
obligations included, shown in Table 4.2.

Table 4.2: Symbolic tracing of Iterative code of UnboundedInteger

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Conditions</th>
<th>Facts</th>
<th>Obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>procedure Multiply</strong> (updates i: Integer, restores j: Integer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><strong>variable</strong> p, nj, z: Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i_1 = i_0 j_1 = j_0 n_j_1 = 0 p_1 = 0 z_1 = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if IsGreater (j, z) then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 &lt; j_0</td>
<td>i_2 = i_0 j_2 = j_0 n_j_2 = 0 p_2 = 0 z_2 = 0</td>
<td>p_2 + i_2 \times j_2 = p_2 + i_2 \times j_2 \quad n_j_2 + j_2 = n_j_2 + j_2 \quad i_2 = i_2 \quad z_2 = z_2 \quad 0 &lt; j_2 \Rightarrow 0 &lt; j_2</td>
</tr>
</tbody>
</table>

*Table 4.2 continued on next page...*
Table 4.2 continued from previous page

<table>
<thead>
<tr>
<th>Loop</th>
<th>Maintains ( p + i \times j = #p + #i \times #j ) and ( \text{nj} + j = #\text{nj} + #j ) and ( i = #i ) and ( z = #z ) decreases ( j ) while ( \text{IsPositive (j)} ) do</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>( 0 &lt; \dot{j}_0, \dot{j}_3 ) ( p_3 + i_0 \times j_3 = i_0 \times j_0 ) ( \text{nj}_3 + j_3 = j_0 ) ( i_3 = i_0 ) ( z_3 = 0 ) ( 0 \leq j_3 )</td>
</tr>
<tr>
<td>4</td>
<td>( 0 &lt; \dot{j}_0, \dot{j}_3 ) ( i_4 = i_0 ) ( j_4 = j_3 ) ( \text{nj}_4 = \text{nj}_3 ) ( p_4 = p_3 + i_0 ) ( z_4 = 0 )</td>
</tr>
<tr>
<td>5</td>
<td>( 0 &lt; \dot{j}_0, \dot{j}_3 ) ( i_5 = i_0 ) ( j_5 = j_3 ) ( \text{nj}_5 = \text{nj}_3 + 1 ) ( p_5 = p_3 + i_0 ) ( z_5 = 0 )</td>
</tr>
</tbody>
</table>

Table 4.2 continued on next page...
<table>
<thead>
<tr>
<th>Decrement (j)</th>
<th>6 $&lt; j_0, j_3$</th>
<th>7 $&lt; j_0$</th>
<th>else</th>
<th>8 $\leq j_0$</th>
<th>loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_6 = i_0$</td>
<td>$i_7 = i_0$</td>
<td>$i_8 = i_0$</td>
<td>$p_8 = 0$</td>
<td>maintains $p - i * j = #p - #i * #j$ and $n j + j = #n j + #j$ and $i = #i$ and $z = #z$ and $j \leq 0$ decreases $-j$</td>
<td></td>
</tr>
<tr>
<td>$j_6 = j_3 - 1$</td>
<td>$j_7 \leq 0$</td>
<td>$j_8 = j_0$</td>
<td>$z_8 = 0$</td>
<td>$p_6 = p_3 + i_0$</td>
<td>$n j_6 + j_6 = n j_2 + j_2$</td>
</tr>
<tr>
<td>$n j_6 = n j_3 + 1$</td>
<td>$p_7 + i_0 \times j_7 = i_0 \times j_0$</td>
<td>$n j_8 = 0$</td>
<td>$z_8 = z_8$</td>
<td>$z_6 = z_2$</td>
<td></td>
</tr>
<tr>
<td>$p_6 = p_3 + i_0$</td>
<td>$n j_7 + j_7 = j_0$</td>
<td>$p_8 = 0$</td>
<td>$j_8 \leq 0$</td>
<td>$0 \leq j_6$</td>
<td></td>
</tr>
<tr>
<td>$z_6 = 0$</td>
<td>$z_7 = 0$</td>
<td>$z_8 = 0$</td>
<td>$j_8 \neq z_8 \Rightarrow j_8 &lt; 0$</td>
<td>$j_6 &lt; j_3$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 continued on next page...
... Table 4.2 continued from previous page

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>$j_0 \leq 0, j_9 \neq 0$</td>
<td>$i_9 = i_0$</td>
<td>$j_9 \leq 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$nj_9 + j_9 = j_0$</td>
<td>$p_9 - i_0 \times j_9 = -i_0 \times j_0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z_9 = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Add</strong> $(p, i)$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$j_0 \leq 0, j_9 \neq 0$</td>
<td>$i_{10} = i_0$</td>
<td>$j_{10} = j_9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$nj_{10} = nj_9$</td>
<td>$p_{10} = p_9 + i_0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z_{10} = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Decrement</strong> $(nj)$</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$j_0 \leq 0, j_9 \neq 0$</td>
<td>$i_{11} = i_0$</td>
<td>$j_{11} = j_9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$nj_{11} = nj_9 - 1$</td>
<td>$p_{11} = p_9 + i_0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z_{11} = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Increment</strong> $(j)$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>$j_0 \leq 0, j_9 \neq 0$</td>
<td>$i_{12} = i_0$</td>
<td>$j_{12} = j_9 + 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$nj_{12} = nj_9 - 1$</td>
<td>$p_{12} = p_9 + i_0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z_{12} = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p_{12} - i_{12} \times j_{12} = p_8 - i_8 \times j_8$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$nj_{12} + j_{12} = nj_8 + j_8$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i_{12} = i_8$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z_{12} = z_8$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$j_{12} \leq 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$j_9 &lt; j_{12}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>end loop</strong></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>$j_0 \leq 0$</td>
<td>$i_{13} = i_0$</td>
<td>$j_{13} = 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$nj_{13} = j_0$</td>
<td>$p_{13} = -i_0 \times j_0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z_{13} = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Negate</strong> $(p)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 continued on next page...
The code works iteratively, breaking the work into the two cases of \( j > 0 \) and \( j \leq 0 \). Notably, if \( j > 0 \) we may repeatedly add the value of \( i \) to \( p \), \( j \) many times, whereas otherwise we may repeatedly add \( i \) to \( p \), \(-j\) many times. In the second case, we now have the negative of what we want, so we negate the value of \( p \) once done.

The obligations are, as mentioned, rather easily seen to be true. Many are mere checks of tautologies, such as \( nj_2 + j_2 = nj_2 + j_2 \). Others, such as \( nj_{12} + j_{12} = nj_8 + j_8 \) can be done by direct substitutions and using a couple related facts.\(^{18}\)

\(^{18}\)Specifically, we know by the accumulated facts at states 12 and 8 that \( nj_{12} = nj_9 - 1 \), \( j_{12} = j_9 + 1 \), \( nj_8 = 0 \), and \( j_2 = j_0 \), so that this is equivalent to proving \( nj_9 + j_9 = j_0 \), a fact known from state 9.
4.2.2 Some incorrect code

The previous example demonstrates one of many possible implementations for Multiply when dealing with integers without bounds. While bounded integers have been, and continue to be, the standard programming integers, problems while using them continue, such as overflow bugs in binary search and mergesort [5].

The bounded integer component in Resolve uses two integer constants, MAX and MIN, whose only restrictions are $\text{MIN} \leq 0$ and $0 < \text{MAX}$, where the values of a bounded integer $a$ must satisfy $\text{MIN} \leq a \leq \text{MAX}$. Using nearly arbitrary constants such as MIN and MAX allows us to divorce ourselves from bounding issues tied to any particular computer architecture, and, as we shall see, allows for us to find counterexamples to incorrect code with smaller, easier to comprehend values.

A reasonable attempt to implement Multiply is in Listing 4.8. The contracts for Multiply, Add, and Negate are given in Listings 4.7, 4.9, and 4.10, respectively. The contract for the IntegerFacility component is in Listing 4.6, which is as one would expect it to be, with the main complication of requirements due to the bounds. The contract for IsPositive is the same as given earlier (excepting that the enhancement is now for IntegerFacility). For the contracts that have changed, they add the requirements that the client must ensure any updated values are within bounds. Finally, and most importantly, we have left the implementation of Multiply unchanged.

```plaintext
contract IntegerFacility
{
  definition MIN: integer
  satisfies restriction
  MIN <= 0

  definition MAX: integer
  satisfies restriction
  0 < MAX

  math subtype INTEGERMODEL is integer
  exemplar i

  constraint
```
\[ \text{MIN} \leq i \text{ and } i \leq \text{MAX} \]

type Integer is modeled by \text{INTEGERMODEL}

exemplar i

initialization ensures
\[ i = 0 \]

\text{procedure} Increment (\text{updates} i: \text{Integer})

\text{requires}
\[ i < \text{MAX} \]

\text{ensures}
\[ i = \#i + 1 \]

\text{procedure} Decrement (\text{updates} i: \text{Integer})

\text{requires}
\[ \text{MIN} < i \]

\text{ensures}
\[ i = \#i - 1 \]

\text{function} AreEqual (\text{restores} i: \text{Integer}, \text{restores} j: \text{Integer}): \text{control}

\text{ensures}
\[ \text{AreEqual} = (i = j) \]

\text{function} IsGreater (\text{restores} i: \text{Integer}, \text{restores} j: \text{Integer}): \text{control}

\text{ensures}
\[ \text{IsGreater} = (i > j) \]

\text{function} Replica (\text{restores} i: \text{Integer}): \text{Integer}

\text{ensures}
\[ \text{Replica} = i \]

\text{function} Min (): \text{Integer}

\text{ensures}
\[ \text{Min} = \text{MIN} \]

\text{function} Max (): \text{Integer}

\text{ensures}
\[ \text{Max} = \text{MAX} \]

end IntegerFacility

Listing 4.6: Contract for the IntegerFacility component.

\text{contract} Multiply \textbf{en enhances} IntegerFacility

\text{procedure} Multiply (\text{updates} i: \text{Integer}, \text{restores} j: \text{Integer})

\text{requires}
\[ \text{MIN} \leq i \times j \text{ and } i \times j \leq \text{MAX} \]

\text{ensures}
\[ i = \#i \times j \]

end Multiply

Listing 4.7: Contract for Integer's Multiply.

\text{realization} IterativeBoundingBug \textbf{implements} Multiply \textbf{for} IntegerFacility

\text{uses} IsPositive \textbf{for} IntegerFacility

\text{uses} Add \textbf{for} IntegerFacility

\text{uses} Negate \textbf{for} IntegerFacility

\text{procedure} Multiply (\text{updates} i: \text{Integer}, \text{restores} j: \text{Integer})

\text{variable} p, nj, z: \text{Integer}

\text{if} \text{IsGreater} (j, z) \text{then}
Listing 4.8: An attempt at an Iterative implementation for Integer’s `Multiply`.

For this implementation of `Multiply`, there are now 40 VCs. As the developer we might say that the code is the same, so what changed to make 10 more VCs to be checked? In fact, the code is not semantically the same; the contracts for components that we, as a client, are using have changed. Additionally, of these 40 VCs, only 27 are left after immediate removal of identities (again, we have 10 more VCs than in the unbounded implementation).

Listing 4.9: Contract for Integer’s `Add`.

Listing 4.10: Contract for Integer’s `Negate`.

68
\[ \text{MIN} \leq -i \quad \text{and} \quad -i \leq \text{MAX} \]

\[ \text{ensures} \]

\[ i + \#i = 0 \]

\end Negate

Listing 4.10: Contract for Integer’s Negate.

Clearly, these additional VCs\textsuperscript{19} must relate to the new requirements brought on by the modified requires clauses. Take, for example, a VC generated for the requires clause of the first Add, seen in Figure 4.2. The givens highlighted as 4.1, 4.2, 4.3, and 4.4 are all the facts needed to prove the claim.

A sketch of the proof works as follows. We break up the proof into two cases. First, consider \( i_0 \geq 0 \). In this case, Fact 4.3 is all we need, as then

\[ \text{MIN} \leq p_3 \leq p_3 + i_0. \]

For the alternative, when \( i_0 < 0 \), we may make use of Fact 4.4 to substitute \( p_3 \) out. If we do this, we are now trying to prove \( \text{MIN} \leq i_0 \times j_0 - i_0 \times j_3 + i_0 \), which can be written as

\[ \text{MIN} \leq i_0 \times j_0 + (j_3 - 1) \times (-i_0). \]

The given Fact 4.2 tells us \( j_3 - 1 \geq 0 \). Likewise, we know that \( i_0 < 0 \), meaning \( -i_0 > 0 \). So we are left with, when using Fact 4.1,

\[ \text{MIN} \leq i_0 \times j_0 \leq i_0 \times j_0 + (j_3 - 1) \times (-i_0) = p_3 + i_0, \]

which is what we want.\textsuperscript{20}

\textsuperscript{19}In actuality, all the VCs have been changed. For example, each one has as a given that the incoming values of \( i \) and \( j \) are within the range of MIN and MAX. A VC containing these additional facts as a given is at least as valid as one that does not; the original VCs that do not contain such givens were valid, meaning these new facts are not required to prove the conclusion.

\textsuperscript{20}As an interesting side note, neither Z3 nor Isabelle uses transitivity which we used here. This property is used by SplitDecision in a limited way, explaining why SplitDecision can prove this VC in about 200 ms, while both Z3 and Isabelle cannot.
Prove

MIN ≤ p₃ + i₀

Given

MIN ≤ j₀
∧ j₀ ≤ MAX
∧ MIN ≤ i₀
∧ i₀ ≤ MAX
∧ 0 < MAX
∧ MIN ≤ 0
∧ MIN ≤ i₀ × j₀
∧ i₀ × j₀ ≤ MAX
∧ 0 < j₀
∧ 0 < j₃
∧ MIN ≤ p₃
∧ p₃ ≤ MAX
∧ MIN ≤ j₃
∧ j₃ ≤ MAX
∧ MIN ≤ nj₃
∧ nj₃ ≤ MAX
∧ p₃ + i₀ × j₃ = i₀ × j₀
∧ nj₃ + j₃ = j₀

Figure 4.2: A VC generated for the requires clause of the first Add used in the implementation in Listing 4.8
There are, however, VCs that are more difficult to prove. The VC shown in Figure 4.3 is for the requires clause of the second Add statement. The two VCs are remarkably similar; instead of reference to variables with a subscript 3, they now have a subscript 9. We are in the else branch of the if-statement, however, so we now have a fact of \( j_0 \leq 0 \). Slightly different loop invariants have provided that \( j_9 < 0 \) and \( p_9 - i_0 \times j_9 = -i_0 \times j_0 \). These prove to be more than just cosmetic changes. If we assign the following values to the variables (produced by Contra), we properly meet the givens while not fulfilling what we want to prove, and thus have a counterexample: MIN = -2, MAX = 4, \( i_0 = -2 \), \( j_0 = -2 \), \( j_9 = -1 \), \( n_9 = -1 \), and \( p_9 = -2 \).

Not only does this mean that the VC is invalid; it also means the code is incorrect (unlike in the original example). We will explore how to fix this code; but it is also worth noting that the code is broken for the other portion of Add’s requires clause, that the final sum must not be greater than MAX. Likewise, going through the second path provides a counterexample to the restores parameter mode for \( j \). So what exactly is going on? To better understand, we can look at the symbolic tracing table where these values have been plugged in, seen in Table 4.3. The restrictions on the variables are described in the “Facts” column.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Conditions</th>
<th>Facts</th>
<th>Obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure Multiply (updates i: Integer, restores j: Integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Symbolic tracing of Iterative code for Multiply using IntegerFacility

Table 4.3 continued on next page...
Prove \[ \text{MIN} \leq p_9 + i_0 \]

Given \[ \text{MIN} \leq j_0 \]
\[ \land j_0 \leq \text{MAX} \]
\[ \land \text{MIN} \leq i_0 \]
\[ \land i_0 \leq \text{MAX} \]
\[ \land \text{MIN} \leq 0 \]
\[ \land 0 < \text{MAX} \]
\[ \land \text{MIN} \leq i_0 \times j_0 \]
\[ \land i_0 \times j_0 \leq \text{MAX} \]
\[ \land j_0 \leq 0 \]
\[ \land j_9 \leq 0 \]
\[ \land j_9 \neq 0 \]
\[ \land \text{MIN} \leq p_9 \]
\[ \land p_9 \leq \text{MAX} \]
\[ \land \text{MIN} \leq j_9 \]
\[ \land j_9 \leq \text{MAX} \]
\[ \land \text{MIN} \leq n_j_9 \]
\[ \land n_j_9 \leq \text{MAX} \]
\[ \land p_9 - i_0 \times j_9 = -i_0 \times j_0 \]
\[ \land n_j_9 + j_9 = j_0 \]

Figure 4.3: A VC generated for the \texttt{requires} clause of the second Add used in the implementation in Listing 4.8
Table 4.3 continued from previous page

<table>
<thead>
<tr>
<th>0</th>
<th>$i_0 = -2$  [j_0 = -2] [MIN = -2] [MAX = 4]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>variable</strong> $p, nj, z: \text{Integer}$</td>
<td><strong>variable</strong> $p, nj, z: \text{Integer}$</td>
</tr>
<tr>
<td>1</td>
<td>$i_1 = -2$  [j_1 = -2] [nj_1 = 0] [p_1 = 0] [z_1 = 0] [MIN = -2] [MAX = 4]</td>
</tr>
<tr>
<td>if IsGreater ($j, z$) then</td>
<td>if IsGreater ($j, z$) then</td>
</tr>
<tr>
<td>loop maintains $p - i \times j = #p - #i \times #j$ and</td>
<td>loop maintains $p - i \times j = #p - #i \times #j$ and</td>
</tr>
<tr>
<td>$nj + j = #nj + #j$ and</td>
<td>$nj + j = #nj + #j$ and</td>
</tr>
<tr>
<td>$i = #i$ and $z = #z$ and</td>
<td>$i = #i$ and $z = #z$ and</td>
</tr>
<tr>
<td>$j_0 \leq 0$</td>
<td>$p_8 - i_8 \times j_8 = p_8 - i_8 \times j_8$ [nj_8 + j_8 = nj_8 + j_8] [i_8 = i_8] [z_8 = z_8] [j_8 \leq 0] [j_8 \neq z_8 \Rightarrow j_8 &lt; 0]</td>
</tr>
</tbody>
</table>

Table 4.3 continued on next page...
Before calling \texttt{Add (p, i)} after state 9, we must meet the obligation that $\text{MIN} \leq p + i$. Now that we see there is a problem, we can try and solve it.\footnote{It is possible to find a counterexample for the other part of the \texttt{requires} clause for the same \texttt{Add} operation, that $p + i \leq \text{MAX}$; values of $\text{MIN} = -2$, $\text{MAX} = 1$, $i_0 = 1$, $j_0 = -2$, $r_0 = -1$, $n r_9 = -1$, and $p_9 = 1$ demonstrates such a situation.}

4.2.3 Correcting some incorrect code

The original logic of the second \texttt{while} loop in Listing 4.8 performs repeated addition, and then negates the answer (as we have really calculated $-i_0 \times j_0$ up to this point). In the bounded case, there is flawed logic; while we know, by the \texttt{requires} clause, that $\text{MIN} \leq i \times j \leq \text{MAX}$, we do not know that $\text{MIN} \leq -i \times j \leq \text{MAX}$. For this to be true for \textit{all} possible incoming values of $i$ and $j$, we in fact would need to require both $\text{MAX} \leq -\text{MIN}$ and $-\text{MAX} \leq \text{MIN}$. That puts us in the position of requiring $\text{MAX} = -\text{MIN}$, which is not the case for most implementations of bounded integers, and certainly not a consequence of the stated restrictions on \text{MIN} and \text{MAX}.

Is there any “easy” fix to this code? If we consider how the second \texttt{while} loop works, we do see an answer. With $j \leq 0$, we add $i$ to $p$ over the course of $-j$ many times, and then negate. In the \texttt{while} loop, the difference between $i \times j$ and $p$ increases.
Why not subtract \( i \) from \( p \) instead, \(-j\) many times, and thereby decrease the difference between \( i \times j \) and \( p \). As an added bonus, we have no need to call \texttt{Negate} afterwards.

We do need to change part of the loop invariant, to reflect that we are now not doing the “reverse” of what we had done before. This version is seen in Listing 4.11, with the contract for \texttt{Subtract} given in Figure 4.12.

---

**realization** IterativeNonObvious implements Multiply for IntegerFacility

uses IsPositive for IntegerFacility
uses Add for IntegerFacility
uses Subtract for IntegerFacility

procedure Multiply (updates i: Integer, restores j: Integer)
  variable p, nj, z: Integer
  if IsGreater (j, z) then
    loop
      maintains \( p + i \times j = \#p + \#i \times \#j \) and \( nj + j = \#nj + \#j \) and \( i = \#i \) and \( z = \#z \) and \( j \geq 0 \)
      decreases \( j \)
      while IsPositive (\( j \)) do
        Add (\( p, i \))
        Increment (\( nj \))
      Decrement (\( j \))
    end loop
  else
    loop
      maintains \( p + i \times j = \#p + \#i \times \#j \) and \( nj + j = \#nj + \#j \) and \( i = \#i \) and \( z = \#z \) and \( j \leq 0 \)
      decreases \(-j \)
      while not AreEqual (\( j, z \)) do
        Subtract (\( p, i \))
        Decrement (\( nj \))
      Increment (\( j \))
    end loop
  end if
  i :=: p
  j :=: nj
end Multiply

end IterativeNonObvious

Listing 4.11: A non-obvious Iterative implementation for Integer’s Multiply.

---

**contract** Subtract enhances IntegerFacility

procedure Subtract (updates i: Integer, restores j: Integer)
  requires MIN \( \leq i - j \) and \( i - j \leq \texttt{MAX} \)
  ensures \( i = \#i - j \)
end Subtract

Listing 4.12: Contract for Integer’s Subtract.
Here, we have 38 VCs (a savings from removing the \texttt{Negate} operation), with 25 requiring something more than completely trivial checking. It is worth looking at a VC for the \texttt{requires} clause of \texttt{Subtract}, for multiple reasons as we shall see. The VC to prove that the difference between the inputs to \texttt{Subtract} does not go below \texttt{MIN} is shown in Figure 4.4.

\begin{align*}
\text{Prove} & \quad \text{MIN} \leq p_9 - i_0 \\
\text{Given} & \quad \text{MIN} \leq j_0 \\
& \quad j_0 \leq \text{MAX} \\
& \quad \text{MIN} \leq i_0 \\
& \quad i_0 \leq \text{MAX} \\
& \quad \text{MIN} \leq 0 \\
& \quad 0 < \text{MAX} \\
& \quad \text{MIN} \leq i_0 \times j_0 \quad (4.5) \\
& \quad i_0 \times j_0 \leq \text{MAX} \\
& \quad j_0 \leq 0 \\
& \quad j_0 \leq 0 \quad (4.6) \\
& \quad j_0 \neq 0 \quad (4.7) \\
& \quad \text{MIN} \leq p_9 \quad (4.8) \\
& \quad p_9 \leq \text{MAX} \\
& \quad \text{MIN} \leq j_9 \\
& \quad j_9 \leq \text{MAX} \\
& \quad \text{MIN} \leq n_j_9 \\
& \quad n_j_9 \leq \text{MAX} \\
& \quad p_9 + i_0 \times j_9 = i_0 \times j_9 \quad (4.9) \\
& \quad n_j_9 + j_0 = j_0 \\
\end{align*}

Figure 4.4: A VC generated for the \texttt{requires} clause of \texttt{Subtract} used in the implementation in Listing 4.11
Without knowing which facts are relevant to prove the VC, Figure 4.4 is rather non-obvious. The lines highlighted are all that is required by SplitDecision to prove the VC. That these facts are relevant makes more sense when we see where they come from: 4.5 comes from the requires clause for Multiply; 4.6 from the branch condition of the if statement; 4.7 from the condition to be in the while loop; 4.8 as a constraint to being a bounded integer; and 4.9 from the very first loop invariant.

Not all provers have enough theory to show this VC to be true. Versions of SplitDecision from September 24, 2010 or earlier were not adequate to do so, for example (nor are Isabelle or Z3 as of versions available on the same date). While the code is correct, if it cannot be verified it might as well not be correct, for one must treat such code as suspect. For completeness, we present a modified version of the code that could be verified with SplitDecision before September 24 in Listing 4.13. These new loop invariants on the second loop are meant, beyond just getting the implementation to be proven, as a way to codify and document the programmer’s thinking as to why the code is correct. Either $p - i$ is bounded from below by $i \times j$ (written using $nj + j$, a sum which is always equal to the original value of $j$) or above. As we conveniently know $\text{MIN} \leq i \times j \leq \text{MAX}$, we essentially either know $\text{MIN} \leq i \times j \leq p - i$ or $p - i \leq i \times j \leq \text{MAX}$ where necessary.

```plaintext
realization Iterative implements Multiply for IntegerFacility
    uses IsPositive for IntegerFacility
    uses Add for IntegerFacility
    uses Subtract for IntegerFacility

procedure Multiply (updates i: Integer, restores j: Integer)
    variable p, nj, z: Integer
    if IsGreater (j, z) then
        loop
            maintains p \ast i \ast j = \#p \ast \#i \ast \#j \and nj \ast j = \#nj \ast \#j \and
            i = \#i \and z = \#z \and j \geq 0
            decreases j
```

22 In fact, a roomful of Ph.D.s and graduate students were unable to provide a proof for it for 15 minutes.
Listing 4.13: An implementation of \texttt{Iterative} for Integer's \texttt{Multiply} that \textit{could} be verified.

4.2.4 Verification and the user

This (historically accurate) account of an attempt to verify a relatively simple piece of code highlights many points that need to be addressed if we have any hope of asking programmers to correct their code so as to make it valid in the eyes of an automated theorem prover.

0. A programmer should \textit{never} be forced to look at a VC. A typical verification condition includes a collection of mostly useless givens expressed in ways the programmer should not be expected to think.

1. In the case of a VC that cannot be proven, the programmer must at least have some way of understanding \textit{why} the VC was created in the first place, such as for a \texttt{requires} clause, loop invariant, progress metric, or parameter mode. This does \textit{not} mean a conflict with item 0. As we shall see in Section 4.5, while in
the end a VC cryptically contains the information we need, this information can be properly presented to the programmer in a much more usable manner.

2. When possible, the generation of a counterexample is extremely helpful in understanding why code is incorrect. As we shall see, counterexamples provided by many tools are in forms completely foreign to the original problem. A method must exist to take any such (valid) counterexample and restate it in the domain the programmer is using.

3. As much of this as possible should happen automatically, as part of the process of writing the code. However, it must not hinder the creation or editing of code, either by causing unnecessary slowdowns or providing misleading feedback.

4. A plan should be provided in the case when code is actually correct, but current proving technology is unable to show it either through a lack of theory or general deficiency of the prover.

As we shall see, item 0 can and should be observed throughout, and as such can be considered more of a rule for our rules. We will not explore it separately, but instead make it part of the canon within which all solutions should live. Similarly, 3 affects our design decisions, but does not push a specific course of action. As all contemporary attempts at verification involve generating VCs using slightly different particulars, they hold to a general pattern similar to Resolve programming language [10]. As such, we shall discuss these ideas in terms of OSU’s Resolve, but the ideas are in no way restricted to it.
4.3 Tracing a verification condition to its source

A VC can be generated for many reasons when dealing with full, functional verification. The reasons a VC is currently generated for Resolve can be found in Section 1.1.2. There are other reasons we may have more VCs still, for the purpose of simplifying their proofs. We mention them here, as such actions will turn out to be helpful in understanding how to trace a VC back to its cause. Again, the specifics are for OSU’s Resolve, but the ideas are general.

The most obvious and important way is to generate a VC at a point in code for each possible path. That is, if there is an if statement within the code, all VCs after the if block will now be replicated, with one for when tracing through the if case, and one for when tracing through the else case.

The other way more VCs might be generated can come from simplifying logical operators within the VC. Ideally, a VC would be of the form,

$$\left( \bigwedge_{i \in \{1, \ldots, n\}} G_i \right) \Rightarrow C,$$

where $G_i$ and $C$ do not contain the logical connectives $\Rightarrow$, $\iff$, $\land$, or $\lor$. Understanding how they are dealt with aids in understanding why such manipulations are useful. In general, an $\lor$ in a given breaks the VC into two VCs, one for each of the disjuncts. The same is true for an $\land$ in the conclusion we are trying to make; instead of trying to prove all the conjuncts at once, we try to prove each separately. If the conclusion is an implication, this is equivalent to the antecedent being a given, and the consequent being the new conclusion.

We therefore make use of $A \lor B$ being logically equivalent to $\neg A \Rightarrow B$, $A \Rightarrow B$ to $\neg A \lor (A \land B)$, and $\iff$ to either $(A \land B) \lor (\neg A \land \neg B)$ or $(A \Rightarrow B) \land (B \Rightarrow A)$. The final
Table 4.4: Rules for breaking a VC so that givens and the statement to prove do not contain the logical operators $\lor$, $\land$, $\Rightarrow$, or $\iff$. We are considering VCs written in the form $\left(\bigwedge_{i \in \{1, \ldots, n\}} G_i\right) \Rightarrow C$. In the case of looking at the givens, we shall refer to the given to be singled out as $G_j$ for some $j \in \{1, \ldots, n\}$.

<table>
<thead>
<tr>
<th></th>
<th>$C$</th>
<th>$G_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \lor B$</td>
<td>$\left(\neg A \land \bigwedge_{i \in {1, \ldots, n}} G_i\right) \Rightarrow B$</td>
<td>$\left(A \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$ and $\left(B \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>$\left(\bigwedge_{i \in {1, \ldots, n}} G_i\right) \Rightarrow A$ and $\left(\bigwedge_{i \in {1, \ldots, n}} G_i\right) \Rightarrow B$</td>
<td>$\left(A \land B \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$</td>
</tr>
<tr>
<td>$A \Rightarrow B$</td>
<td>$\left(A \land \bigwedge_{i \in {1, \ldots, n}} G_i\right) \Rightarrow B$</td>
<td>$\left(\neg A \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$ and $\left(A \land B \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$</td>
</tr>
<tr>
<td>$A \iff B$</td>
<td>$\left(\bigwedge_{i \in {1, \ldots, n}} G_i\right) \Rightarrow B$ and $\left(B \land \bigwedge_{i \in {1, \ldots, n}} G_i\right) \Rightarrow A$</td>
<td>$\left(A \land B \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$ and $\left(\neg A \land \neg B \land \bigwedge_{i \in {1, \ldots, n}\setminus{j}} G_i\right) \Rightarrow C$</td>
</tr>
</tbody>
</table>
transformations here are shown in Table 4.4. Now that we have the transformations in hand, we may consider their importance in tracing a VC back to its origin.

4.3.1 Origin of a VC

All of this discussion of splitting up a VC may seem pedantic at best, and pointless at worst, but as we shall now see, it is neither. First, to the argument that it is pedantic: one might make the case we already have the information necessary to trace a VC to its source; at the start of Section 4.3 it was shown that every VC can be tied to a particular operation's requires clause, loop invariant, progress metric, confirm statement, parameter mode, ensures clause of the operation being developed, or the convention (representation invariant) and correspondence (abstraction function) of a kernel realization.

To see the usefulness, take, for example, the case of the conclusion being a conjunction of \( n \) clauses (which is broken up into \( n \) VC). The common case for there to be a conjunction in the first place is the presence of a conjunction in a loop invariant, requires clause, ensures clause, or confirm statement. If the single VC fails, which of these smaller clauses is false? How would a programmer make use of that information? Breaking up the VC allows us to examine each smaller clause on its own merits. We took this for granted in the example in Section 4.2.2; the relevant requires clause shown in Listing 4.9 says we should confirm that \( \text{MIN} \leq i + j \leq \text{MAX} \), but the VC shown in Figure 4.3 only checks \( \text{MIN} \leq i + j \) (compound inequalities are expressed with multiple inequalities joined by a \( \land \)).

But then, is it not pointless? A verifier must take a problem and break it down as it needs to, on its own. Most verifiers are not going to say what part of the VC is
actually not valid, as this is harder than proving the VC. Consider the instance of finding a counterexample, as we did in Section 4.2.2. Had this been a conjoined VC, it might take a careful examination of the VC to understand any counterexamples. As we have stated, it must not be necessary for the programmer to look at the VC. One option then is to modify every prover used to highlight what portion of the conjoined VC was problematic. This could work, if every prover did it correctly. Or, we could easily break the VC as demonstrated, and use provers as is. The easier path is clearly to do the latter.

It could be argued that not all such changes to the VC are necessary; indeed, the inclusion of what to do when a conjunction is contained within the givens was merely for completeness, and similarly so describing what to do if the conclusion is an implication. In every other instance, we have either:

- broken the VC into two stronger VCs that allow us to determine which part of the larger VC is not being found valid, or

- in the case of the conclusion being a disjunction, strengthened the givens.

In either situation the givens contain ostensibly new useful facts, so that it is potentially easier to demonstrate validity or invalidity of a VC. Additionally, we can more directly point to where the VC originated.

As a running example, we will be looking at a reasonable enhancement of an array type, shown in Listing 4.14. The model for this array is relatively straightforward. It is made up of three parts: an integer for the lower bound, an integer for the upper bound, and a string representing the actual data kept within the array. Keeping track of the bounds allows us to sidestep the issue of arbitrarily setting the index at 0
or 1, as is the case in most programming languages, and while it makes the model look somewhat more complicated, in reality it turns out to have little bearing on verification, while not pigeonholing the client to a particular design.

```plaintext
contract ArrayAsString2Template (type Item)

uses UnboundedIntegerFacility

math subtype ARRAY_MODEL is (  
   lb: integer,  
   ub: integer,  
   s: string of Item
)

e exemplar a
constraint  
a.lb ≤ a.ub + 1 and  
|a.s| = a.ub - a.lb + 1

type Array is modeled by ARRAY_MODEL

e exemplar a
initialization ensures  
a = (1, 0, empty_string)

procedure SetBounds (updates a: Array,  
   restores lower: Integer,  
   restores upper: Integer)

requires  
lower ≤ upper + 1  
en ensures  
a.lb = lower and a.ub = upper

procedure SwapItem (updates a: Array,  
   restores i: Integer,  
   updates x: Item)

requires  
a.lb ≤ i and i ≤ a.ub  
en ensures  
a.lb = #a.lb and  
a.ub = #a.ub and  
substring (a.s, 0, i - a.lb) = substring (#a.s, 0, i - #a.lb) and  
substring (a.s, i+1 - a.lb, |a.s|) =  
substring (#a.s, i+1 - #a.lb, |#a.s|) and  
substring (a.s, i - a.lb, i + 1 - a.lb) = <#x> and  
substring (#a.s, i - #a.lb, i + 1 - #a.lb) = <x>

function LowerBound (restores a: Array): Integer
en ensures  
LowerBound = a.lb

function UpperBound (restores a: Array): Integer
en ensures  
UpperBound = a.ub

end ArrayAsString2Template
```

Listing 4.14: Contract for ArrayAsString2Template
Note that the design is, of course, modular, in that we have no requirements on the type contained in the \texttt{Array}. The enhancement we will be considering, \texttt{Contains} is in Listing 4.15.

```plaintext
contract Contains enhances ArrayAsString2Template
    function Contains (restores a: Array, restores x: Item): control
        ensures
            Contains = x is in elements(a.s)
end Contains
```

Listing 4.15: Contract for enhancement \texttt{Contains} of \texttt{ArrayAsString2Template}

\texttt{Contains} takes an \texttt{Array}, \texttt{a}, and an \texttt{Item}, \texttt{x}, and returns whether or not there is an \texttt{Item} in \texttt{a} that has the same mathematical value as \texttt{x}. An important point is that both \texttt{a} and \texttt{x} must be restored by the end of the operation. The VC shown in Figure 4.5 is a check of one such situation.

```plaintext
realization LinearIterativeEndConditionBug (    function AreEqual (restores i: Item, restores j: Item): control
        ensures
            AreEqual = ( i = j )
    ) implements Contains for ArrayAsString2Template

    function Contains (restores a: Array, restores x: Item): control
        variable pos: Integer
        variable ub: Integer
        pos := LowerBound (a)
        ub := UpperBound (a)
        if not IsGreater (pos, ub) then
            loop
                maintains
                    a = #a and
                    x = #x and
                    a.lb ≤ pos and
                    pos ≤ ub and
                    ub = a.ub and
                    Contains = (x is in elements(substring (a.s, 0, pos - a.lb)))
                decreases
                    ub - pos
                while not Contains and not AreEqual (pos, ub) do
                    variable y: Item
                    SwapItem (a, pos, y)
                    Contains := AreEqual (x, y)
                    SwapItem (a, pos, y)
                    Increment (pos)
            end loop
        end if
```

85
Figure 4.5: A VC generated for the \texttt{ensures} clause of \texttt{LinearIterativeEndConditionBug}, an enhancement \texttt{Contains} of \texttt{ArrayAsString2Template} shown in Listing 4.16

We may now take what has been discussed in this section to examine the circumstances surrounding the VC, \textit{i.e.}, why was it created. We know from the VC generator that it is trying to prove the \texttt{ensures} clause of \texttt{Contains}, after going through the \texttt{if} branch (and we can likewise trace where each given is generated). The “raw” VC that is generated\footnote{A raw VC has not undergone obvious substitutions such that when presented with a given of $a = b$, all instances of $a$ are replaced with $b$. This has been done manually to maintain some sanity for both the presentation of the VC and also the reader.} can be found in Figure 4.6, and notably contains an $\lor$ and $\leftrightarrow$.

By the rules stated earlier in this section, this VC will in fact be broken up into four separate VCs, as it will be doubling twice.

One such doubling comes from the $\lor$. This comes from the conditional from the \texttt{while} loop: either the loop terminated due to \texttt{Contains} becoming true, or it became...
Prove \( x_0 \in \text{elements} (\text{substring} (a_0.s, 0, pos_{12} - a_0.lb)) \) \(\iff\) \( x_0 \in \text{elements} (a_0.s) \)

Given
\[
|a_0.s| = 1 + a_0.ub - a_0.lb \\
\land \ ( \ x_0 \in \text{elements} (\text{substring} (a_0.s, 0, pos_{12} - a_0.lb)) \\
\lor \ a_0.ub \neq pos_{12} \\
\land \ a_0.lb \neq a_0.ub \\
\land \ pos_{12} \leq a_0.ub \\
\land \ a_0.lb \leq pos_{12} \\
\land \ a_0.lb \leq 1 + a_0.ub 
\]

Figure 4.6: Origin of VC shown in Figure 4.5 before splitting

true because \texttt{AreEqual} (pos, ub) returned true, \(i.e., pos = ub\). In the VC shown in Figure 4.5, the latter is what happens. The VC, therefore, handles the case where we have iterated up to the point where \(pos = ub\) in the array, and says nothing on whether \texttt{Contains} is true. On the other hand, VCs with \texttt{Contains} being true say nothing of whether the loop iterated completely through the array.

The second doubling comes from the \(\iff\) in the conclusion of the original VC. Assuming that \(x_0 \in \text{elements} (a_0.s)\), we must now show

\[
x_0 \in \text{elements} (\text{substring} (a_0.s, 0, pos_{12} - a_0.lb)) .
\]

That is, we are assuming the array given is such that \texttt{Contains} should be \texttt{true}, and now we must demonstrate that it is. Obviously, the other doubling assumes \(x_0 \notin \text{elements} (a_0.s)\), and must likewise show that \texttt{Contains} ends the operation as \texttt{false}.

This VC therefore asks a very interesting question, merely by logical manipulations: given any arbitrary array of items a and item x, where
• x is in a, and
• we have searched up to the upper bound of the array,

can we find our search item? Or, alternatively, is there an array a and search item x such that x is in a, but our search turns up fruitless? The VC is in fact false, and the next section will discuss the process for discovering relevant counterexamples.

4.4 Tracing counterexamples back

After a prover has attempted to show a VC is valid and failed to do so, we must consider the possibility that it is not valid. For example, no prover will (or more accurately, no prover should) be able to prove the VCs shown in Figures 4.3 and 4.5, due to them actually being invalid. We demonstrated this for the former VC by “showing” a counterexample, but how can find that? More importantly, how can we make sure the counterexample is relevant to the programmer who is looking not at VCs but at code that is waiting to be compiled?

The former question can be answered by using tools designed to find such things. For our purposes, we will stick to discussing Contra. Any tool trying to find a counterexample, by definition, must be attempting to come up with assignments to variables so that the VC is false. There are certainly areas for improvement; such a tool might use restrictions on the variables to not bother trying certain assignments. These improvements would certainly aid the speed at finding a counterexample (and in some cases, such as when doing integer linear programming, using brute force to demonstrate there is no counterexample). They do run into the unavoidable issue that a greater number of variables increases the search space, potentially exponentially.
It would, therefore, be potentially helpful to have a way of decreasing the number of variables in the search space as much as possible. While SplitDecision makes no guarantees of decreasing the variable count in processed VCs that it is unable to solve, most of the methods used do in fact work towards that end. Table 4.5 shows a comparison of running SplitDecision and Contra together, versus running just Contra by itself. The VCs chosen are all those in the current Resolve catalog of components, their enhancements, and their realizations that SplitDecision does not find to be valid outright.

Table 4.5: Comparison of finding counterexamples using either both SplitDecision and Contra, or just Contra. Times are in seconds.

<table>
<thead>
<tr>
<th></th>
<th>SD&amp;C</th>
<th>SD&amp;C Times</th>
<th>SD&amp;C Ave.</th>
<th>C</th>
<th>C Times</th>
<th>C Ave.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counterex.</td>
<td>56</td>
<td>9.790</td>
<td>0.175</td>
<td>56</td>
<td>11.023</td>
<td>0.197</td>
</tr>
<tr>
<td>Gave Up</td>
<td>33</td>
<td>30.032</td>
<td>0.910</td>
<td>21</td>
<td>125.445</td>
<td>5.974</td>
</tr>
<tr>
<td>Error</td>
<td>198</td>
<td>35.854</td>
<td>0.181</td>
<td>210</td>
<td>33.522</td>
<td>0.157</td>
</tr>
<tr>
<td>Total</td>
<td>287</td>
<td>75.676</td>
<td>0.264</td>
<td>287</td>
<td>169.990</td>
<td>0.592</td>
</tr>
</tbody>
</table>

In all cases, Contra was allowed to run until it exhausted its search space, running on Intel’s Core i5 750, a CPU with 4 cores running at 2.67 GHz each, and 8 GB of RAM. The combination of SplitDecision and Contra compared to simply Contra was more likely to successfully find a counterexample than to exhaust Contra’s search space (i.e., give up), and more likely to exhaust Contra’s search space than encounter error.

This is mentioned more for completeness, as running either SplitDecision or Contra is typically computationally, not memory, intensive.
Thus we find that, of the 2,035 original VCs, SplitDecision is able to prove 1,748 of them (85.90%). It additionally is able to find a counterexample for 2 (0.10%). Of the remaining 285 VCs, Contra finds counterexamples (either using SplitDecision or not) for 54 (2.65%), including the 2 SplitDecision discovered. So finally, we have 233 VCs (11.45%) currently “unsolved” by either SplitDecision or Contra. We find the situation even better when also employing Z3 and Isabelle. When using all three provers, we prove 1,826 VCs (89.73%), discover counterexamples for 54 (2.65%), thereby leaving only 155 VCs (7.62%) in question as to whether they are true.

It should be clear that processing with a tool such as SplitDecision offers numerous advantages over having a tool to only handle finding a counterexample. What happens, then, if we process the VC in Figure 4.5 with SplitDecision, assuming it is not found to be valid or invalid? As if anticipating this retorical question, we see the results in Figure 4.7.

\[
\text{Prove} \quad \text{strlensim59}_0 \in \text{elements (substrrem9)} \\
\text{Given} \quad \text{true}
\]

Figure 4.7: The result after processing the VC from Figure 4.5 with SplitDecision.

This is quite easy to find a counterexample for; Contra gives \( \text{strlensim58}_0 = 0 \) and \( \text{substrrem9}_1 = \Lambda \) as one such possibility (where Item is arbitrarily decided to be modeled by integers so that Contra may assign a value to it). There is one small

\[25\text{Errors in Contra arise from not yet having a working knowledge of certain mathematical operators, or due to the existence in the VC of locally defined mathematical functions. SplitDecision removes some locally defined mathematical functions but never introduces them, potentially leading to fewer errors. These errors occur as part of a syntactic check.}\]
problem, however: neither of the two variables we have found assignments for exist in
the original VC. Certain rules in SplitDecision cause new variables to be generated, such
as those that remove any mention of substring (i.e., substring removal, or substrrem). These rules are useful for SplitDecision to make progress, so as to be able to apply later rules, but do have the potential problem of increasing the number of variables involved, and introducing new variables in place of older ones.

A general treatment to track how the original variables relate in the modified equation can become quite difficult; in essence, every state change relating to the variable must be logged, as facts irrelevant to the proof but relevant to finding a counterexample might be tossed out. Consider a potential VC in Figure 4.8.

| Prove   | $|\alpha| > 0$ |
|---------|----------------|
| Given   | $|\beta| > 0$ |

Figure 4.8: Example VC where facts irrelevant to the proof are less irrelevant to the counterexample.

There are, of course, many counterexamples to this, as long as $\alpha = \Lambda$ and $\beta$ has a length of one or greater. A prover might decide that the fact $|\beta| > 0$ is irrelevant to the proof; clearly, such an $\alpha$ exists, and further there is no connection between $\alpha$ and $\beta$. Finding a counterexample to the then modified VC, seen in Figure 4.9, leads to a potential counterexample of $\alpha = \langle 0 \rangle$, with either no mention of $\beta$ or saying $\beta = \Lambda$. In either case, these are not valid counterexamples; they do not satisfy all of the givens, so such an assignment is still acceptable as far as the VC is concerned.
Prove \[ |\alpha| > 0 \]
Given \[ true \]

Figure 4.9: VC from Figure 4.8 after potential simplification.

Fortunately, the initial design decisions of \texttt{SplitDecision} neatly sidestep this potential issue. While the VCs in Figures 4.8 and 4.9 are equivalent, a property that \texttt{SplitDecision} maintains in all of its operations, \texttt{SplitDecision} only removes facts for three reasons: duplication, such as \[ |\alpha| > 0 \text{ and } \alpha \neq \Lambda; \] substitution; or being tautological either with mathematical theories, such as \[ 0 \leq |\alpha|, \] or without, like \[ x = x.\] To handle changes of variables, \texttt{SplitDecision} generates a list of all variables mentioned in the VC and then tracks any changes made by substitutions. In the case of the processed VC in Figure 4.7, \texttt{SplitDecision} has kept the following facts:

\[ x_0 = strlensim59_0 \]
\[ a_0 = (a_0_{\text{lb}}, a_0_{\text{lb}} + |\text{substrrem10}|, \text{substrrem10} \circ \langle strlensim59_0 \rangle) \]

In addition to \texttt{strlensim59}_0 and \texttt{substrrem10}_1 mentioned in the processed VC, we also have an integer, \texttt{a}_{0_{\text{lb}}} that is mentioned in neither the original VC nor the processed version. As far as finding a counterexample is concerned, it is of no consequence; any assignment to these variables is correct, and a tool such as \texttt{Contra} may chose its default assignment to that type (in the case of integers, this is 0). By using both \texttt{SplitDecision}

\footnote{A rule for the original string decision procedures removed those of type Val (\(\alpha, i, x\)) \cite{20}, discussed in Chapter 2. While the manner of the rules still left it equivalid, it presents the unwanted situation of finding a correct counterexample to the processed VC, but such that Val (\(\alpha, i, x\)) would not evaluate properly in the original VC. \texttt{SplitDecision} now processes Vals as much as possible to not lose information, and in the case of searching for a counterexample, retains the information completely.}
and Contra combined, a final counterexample is found: $x_0 = 0$ and $a_0 = (0, 0, \langle 0 \rangle)$. That is, if we try to find 0 in the array $(0, 0, \langle 0 \rangle)$, we will not meet the `ensures` clause of `Contains` shown in Listing 4.16.

### 4.5 Failing gracefully (for the programmer)

While a tool such as SplitDecision can find most VCs (1,748 of 2,035 total VCs generated in the Resolve component catalog, or roughly 85.9%) to be either valid or invalid, this leaves many (285 of all VCs, or roughly 14.0%) undecided. Adding in other provers such as Isabelle or Z3, and a tool such as Contra, we are left with 7.6% (155 VCs) that we are unable to say anything about the validity of. When we consider that many of the components in the Resolve catalog are attempts to be correct code, or are broken in specific known ways, it should be clear that in general a programmer must deal frequently with VCs whose validity is unknown. Yet, in Section 4.2.4, we stated as an uppermost goal that the programmer should never be forced to look at a VC in order to verify their code. We will address this in the following section.

#### 4.5.1 An example of failure

The dilemma can be better seen if we try to verify the code in Listing 4.18, an attempted implementation of the enhancement `UniteAndIntersect` (contract in Listing 4.17) for the `SetTemplate` component (contract in Listing 1.1). It would be a poor example indeed if all the VCs were verified, but we are in luck: of the 28 VCs, there are 2 that are not automatically verified (nor can Contra find a counterexample for them).

```plaintext
contract UniteAndIntersect enhances SetTemplate

procedure UniteAndIntersect (updates s: Set, updates t: Set)
ensures
```

93
\[ s = \#s \text{ union } t \quad \text{and} \quad t = \#s \text{ intersection } t \]

end UniteAndIntersect

Listing 4.17: A useful enhancement to SetTemplate.

realization Iterative2Bug implements UniteAndIntersect for SetTemplate

procedure UniteAndIntersect (updates s: Set, updates t: Set)
  variable ss, ts: Integer
  variable tmp: Set
  ss := Size (s)
  ts := Size (t)
  if IsGreater (ts, ss) then
    s :=: t
  else
    loop
      maintains s union t = \#s union \#t
      (s intersection t) union tmp =
      (\#s intersection \#t) union \#tmp and
      t intersection tmp = {}
      decreases |t|
      while not IsEmpty (t) do
        variable x: Item
        RemoveAny (t, x)
        if not Contains (s, x) then
          Add (s, x)
        end if
      end loop
      t :=: tmp
    end loop
  end if
end UniteAndIntersect

end Iterative2Bug

Listing 4.18: An attempted implementation of UniteAndIntersect

To understand why not showing the VC to the programmer in this situation is so important, the two VCs in question are shown in Figures 4.10 and 4.11. Lest one mistakenly think that looking at a VC after it has been processed by SplitDecision is perhaps more helpful, the processing of the first VC is shown in Figure 4.12. The VC has been made “simpler” by the metrics SplitDecision uses, but hardly looks any closer to being solved, and certainly is in no better shape for a programmer to translate back to their code.\(^\text{27}\)

We may start by making use of the what we learned about VC generation and splitting in Section 4.3.1. First, in both cases the VCs were generated to

\(^{27}\)The processing of the second VC is even less illuminating.
Prove \[ t_0 \cap s_0 = \text{tmp}_8 \cup (s_8 \cap (t_8 \setminus x_{10})) \]

Given \[ \text{tmp}_8 \cap t_8 = \emptyset \]
\[ \land t_0 \cap s_0 = \text{tmp}_8 \cup (t_8 \cap s_8) \]
\[ \land t_8 \cup s_8 = t_0 \cup s_0 \]
\[ \land \text{is}\_\text{initial}(x_9) \]
\[ \land |t_8| \geq 0 \]
\[ \land |t_0| > |s_0| \]
\[ \land t_8 \neq \emptyset \]
\[ \land x_{10} \in t_8 \]
\[ \land x_{10} \in s_8 \]

Figure 4.10: One of two unproven VCs generated from code in Listing 4.18.

Prove \[ t_0 \cap s_0 = \text{tmp}_8 \cup (s_8 \cap (t_8 \setminus x_{10})) \]

Given \[ \text{tmp}_8 \cap t_8 = \emptyset \]
\[ \land t_0 \cap s_0 = \text{tmp}_8 \cup (t_8 \cap s_8) \]
\[ \land t_8 \cup s_8 = t_0 \cup s_0 \]
\[ \land \text{is}\_\text{initial}(x_9) \]
\[ \land |t_8| \geq 0 \]
\[ \land |t_0| > |s_0| \]
\[ \land t_8 \neq \emptyset \]
\[ \land x_{10} \in t_8 \]
\[ \land x_{10} \in s_8 \]

Figure 4.11: Second of two unproven VCs generated from code in Listing 4.18.
Prove  \( t_0 \cap s_0 = \text{tmp}_8 \cup (s_8 \cap a) \)

Given  \( \text{tmp}_8 \cap a = \emptyset \)
\[ \land \begin{align*} & t_0 \cap s_0 = \text{tmp}_8 \cup \{x_{10}\} \cup (s_8 \cap a) \\ & t_0 \cup s_0 = s_8 \cup a \\ & \text{is\_initial}(x_9) \\ & x_{10} \notin a \\ & x_{10} \notin \text{tmp}_8 \\ & |s_6| < |t_0| \\ & x_{10} \in t_0 \\ & x_{10} \in s_8 \\ & x_{10} \in s_0 \end{align*} \]

Figure 4.12: VC from Figure 4.10 after processing by \text{SplitDecision}.

prove that the code meets a loop invariant, from within the loop after an iteration. And, most tellingly, both VCs are for the same loop invariant, specifically \((s \cap t) \cup \text{tmp} = (\#s \cap \#t) \cup \#\text{tmp}\). Loop invariants can be particularly difficult to examine, in that they can be not just wrong, but subtly trying to claim something too weak or too strong. Ultimately, they are up to the programmer, so what tools can be provided to help the programmer determine the problem?

If the loop invariant is too strong, it is possible it may be unprovable before even entering the loop. As all other VCs are valid in this example, this is not the case. If the loop invariant is too weak, it is possible VCs after the loop cannot be proved. Again, this is not the case here. Even though both possibilities neither rule out an invariant as being too weak nor too strong, it is still important that the details we have looked into are easily presented to the programmer.
Another possibility is the loop invariant is not really a valid invariant. Before the removal of VCs that are trivially correct, the VC generator finds there are 36 items to prove (eight of which are found by SplitDecision to be trivial). Of these, 28 have as a given some variation of the loop invariant. It would be helpful to know how many of the VCs require the given to be found valid. There is a straightforward way to check; one can modify all the VCs and remove that specific given, to see if the VCs are still found to be correct. To check this, we may use an option in SplitDecision to “prune hypotheses”, which in the case of a valid VC iterates through the givens, removing them one by one, to see if the VC is still valid. Using this method, we find that, of the 26 VCs (two are the VCs we cannot verify) that could make use of the invariant in question, four do: there are two paths that use the loop invariant at the beginning of the loop to prove it at the end; and every path uses it to prove part of the ensures clause at the end of the operation, that \( t = \#s \text{ intersection } \#t \). Another easy check is if, perhaps, the invariant causes a contradiction in the givens, by seeing if we can use the collection of facts to prove false.\(^{28}\) For this problem, there is no VC with this property.

Next we can look at the circumstances surrounding the two unsolvable VCs, and those of similar ones. In this case, no VCs were split due to an errant ⇔, ∨, ∧, or ⇒. There are four VCs to prove the loop invariant in question after an iteration of the loop, generated for different paths. One split is from the if statement before the loop; on the one hand, we assume IsGreater (ts, ss) evaluates to true, and on the other not. The second split is within the while loop, for whether or not Contains (s, x) =

\(^{28}\)If the facts were contradictory in the original VC, they should continue to be contradictory. Otherwise, the conclusion we are trying to prove did indeed depend on the facts, and thus there was no contradiction.
evaluates to true. We find that the VCs we cannot solve are exactly those that go through the `else` path within the `while` loop.

Without specifically showing a VC to a programmer, we are able to show to the programmer:

1. Everything verifies, with the exception of 2 out of 6 places relating to a specific loop invariant.

2. The places in question are inside an iteration of the loop, so if the loop invariant is too strong, it is not too strong from the perspective of the code outside of the loop.

3. Depending on the path inside the loop, the loop invariant can be proven, again suggesting that the loop invariant is not too strong.

4. The loop invariant is used as a relevant fact in proofs after the loop, suggesting that the loop invariant is strong enough.

5. Depending on the path inside the loop, the loop invariant is used within the loop, again weakening the case that the loop invariant is not strong enough.

6. There are 4 paths leading to trying to prove the loop invariant in question after an iteration of the loop. The paths we are unable to prove are exactly the ones going through one of the branches of one `if` statement.

If the programmer uses these facts and takes into account the intended purpose of the loop invariant in question, that `tmp` is to hold elements of `s` and `t` that are duplicates, *i.e.*, are in the intersection, it becomes clear that there is a missing portion
to the code. A corrected implementation is shown in Listing 4.19. We see that it was
necessary to flesh out the else branch inside the loop.

```plaintext
realization Iterative2 implements UniteAndIntersect for SetTemplate

procedure UniteAndIntersect (updates s: Set, updates t: Set)
  variable ss, ts: Integer
  variable tmp: Set
  ss := Size(s)
  ts := Size(t)
  if IsGreater(ts, ss) then
    s :=: t
  end if
  loop
    maintains s union t = #s union #t and
    (s intersection t) union tmp =
    (#s intersection #t) union #tmp and
    t intersection tmp = {} and
    decreases |t|
    while not isEmpty(t) do
      variable x: Item
      RemoveAny(t, x)
      if not Contains(s, x) then
        Add(s, x)
      else
        Add(tmp, x)
      end if
    end loop
  end loop
  t :=: tmp
end UniteAndIntersect

end Iterative2
```

Listing 4.19: A corrected implementation of UniteAndIntersect

### 4.5.2 A framework for failure

We have now suggested a pattern for trying to develop verified software. For present purposes, finding all the VCs valid is uninteresting, so consider code where not all VCs are proved. One option, as we now well know from Section 4.4, is to find a counterexample. It is advantageous to try to find a counterexample and lift the counterexample back to the original variables mentioned in the unprocessed VC, which are themselves related to the mathematical models of variables at particular states. Even with a counterexample in hand, programmers may have difficulty understanding where the code went wrong, and so we must glean as much information from relevant VCs as possible.
This information will come in many forms. First, we may make use of the origin of the VC. We know not only the location it was created for, but why it was created. We also know the path it was created along, and how the VC was simplified. Further, we may use the origin of the givens. This lets us know where a relevant fact is from, or where a problematic fact is from. Finally, we have the automated provers themselves. By manipulating the VCs and examining the outcome of the new proof attempts, we are able to determine useful details about the code.

It is important to realize that no matter what information we provide, a programmer attempting to develop verified software will, by necessity, require a different skill set than what most currently have. Most current programmers do not think in terms of loop invariants, and yet the example in Section 4.5.1 shows that the programmer is required to consider whether it is too weak or too strong. Working in a verified software process demands this change in the programmer’s workflow, no matter what interface is presented.

4.6 A new interface for debugging

We demonstrate a standard workflow for the desired new interface for debugging in Section 4.6.1, and expound upon the details in Section 4.6.2. The example below is a modification of actual work to debug a realization, adapted to show how this could have been done with a more appropriate interface.

4.6.1 An example workflow

As a programmer we have been enlisted to write a correct implementation of Sqrt for bounded integers using a binary search algorithm, the contract for which is shown in Listing 4.20. The idea for the algorithm is simple. Given an Integer i, we start
with a search space, in this case ranging from 0 to \( i \). For any integer value \( m \) ranging from 0 to \( i \), if \( m^2 \leq i \) then all values less than \( m \) are too small. Similarly, if \( m^2 > i \) then all values greater than \( m \) are too large. We thus do the standard binary search method: make \( m \) be the midpoint of the search space, and rule out half of our search space until there is only one possible value.

```plaintext
contract Sqrt enhances IntegerFacility

  procedure Sqrt (updates i: Integer)
  requires
      0 \leq i and i * i < MAX
  ensures
      i * i \leq #i and #i < (i + 1) * (i + 1)

end Sqrt
```

Listing 4.20: Contract for Sqrt, an enhancement to IntegerFacility.

Unfortunately, something is rotten in the Sqrt as implemented. On attempting to verify, three errors are found, as shown in Figure 4.13. The two in red with thicker borders are quite serious. In the case of the first, related to the call to Add, exactly one path reaches that point, with the requires clauses being violated; likewise there are two paths that reach the ensures clause for Sqrt, and at least one of them violates it. We show the number of such paths compared to the total number of paths that reach that point (so, for example, 2 paths reach the ensures clause of Sqrt, but only 1 has a problem). In both cases, a counterexample is found and so a box in red is shown next to the location where there is a problem, with a line drawn to demonstrate at what program state the problem occurs. One or more paths reaches any highlighted violation or potential violation. Additionally, that the code meets the requires clause for Square is neither provable nor disprovable, and so this is shown with an orange-colored box. There is only one path to the requires clause, and it is causing the difficulty. Finally, we see that in the cases of calls to Add and Sqrt, not every obligation has a counterexample, which is shown by using the dotted line.
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
  variable one: Integer
  Increment (one)
  if IsGreater (i, one) then
    variable two, t, hi, d: Integer
    Increment (two)
    t := Replica (i)
    hi := Replica (i)
    Increment (hi)
    Clear (i)
    d := Replica (hi)
    loop
      maintains 0 <= i and i * i <= t and
      t < hi * hi and hi <= MAX and
      d = hi - i and one = #one and
      two = #two and t = #t
      decreases d
      while IsGreater (d, one) do
        variable m, msq, r: Integer
        m := Replica (i)
        Add (m, hi)
        Divide (m, two, r)
        msq := Replica (m)
        Square (msq)
        if IsGreater (msq, t) then
          hi := m
        else
          i := m
        end if
        d := Replica (hi)
        Subtract (d, i)
      end loop
    end if
  end Sqrt
end BinarySearch

Figure 4.13: Attempted binary search implementation of Sqrt in Listing 4.20.
In the case of \textit{Square}, every conjunct of the \texttt{requires} clause is causing difficulty, denoted with the solid line.

This would all make more sense if we could be reminded of the pertinent contracts applicable to our code. While an implemented debugging interface would not display all the contracts at once, Figure 4.14 shows the different ways for the intrepid programmer to be reminded of contracts, for both code being called as a client, and for the contract for the operation in question.\footnote{The contract for \texttt{Add} can be found in Listing 4.9; \texttt{Subtract} in Listing 4.12; \texttt{Divide}, \texttt{Square}, and \texttt{Sqrt} are shown in Figure 4.14 itself; all other component operations are in Listing 4.6.} For the contracts that have been violated or potentially violated, the appropriate portion is highlighted; again red for a counterexample and orange for neither provable nor disprovable. Note also that the names of formal parameters in the contract are replaced with the actuals used by the client; thus making it easier to understand.

Going back to the original report of what went wrong, we find that it would be most helpful to get a better breakdown of the specific provability issues. Figure 4.15 demonstrates this breakdown. Each obligation at the states in question is shown, with an additional breakdown by execution path. Interestingly, we see that the \texttt{ensures} clause of \texttt{Sqrt} has two conjuncts to it, and there are also two paths to reach it. Taking one path, one part of the \texttt{ensures} clause will be met while the other is violated; taking the other path, both conjuncts will be met. This is shown with the oblong shapes underneath. First, the numbers enumerate the paths. Being green with a “✓” shows the code is correct under these conditions, orange with a “?” demonstrates uncertainty, and red with a “C” means there is a counterexample. From here, programmers should be able to check the paths in question. As there is a counterexample available, the counterexample will be shown to us automatically, so we will move on.
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
variable one: Integer
Increment (one)
if Is Greater (i, one) then
variable two, t, hi, d: Integer
Increment (two)
t := Replica (i)
hi := Replica (i)
Increment (hi)
Clear (i)
d := Replica (hi)
loop
maintains 0 <= i and i * i <= t and
   t < hi * hi and hi <= MAX and
   d = hi - i and one = #one and
   two = #two and t = #t
decreases d
while Is Greater (d, one) do
variable m, msq, r:
m := Replica (i)
Add (m, hi)
Divide (m, two, r)
msq := Replica (m)
Square (msq)
if Is Greater (msq, t) then
   hi :=: m
else
   i :=: m
end if
d := Replica (hi)
Subtract (d, i)
end loop
end if
end Sqrt
end BinarySearch

Figure 4.14: Showing some contracts to operations used in Listing 4.13.
procedure Sqrt (updates i: Integer)
  variable one: Integer
  Increment (one)
  if IsGreater (i, one) then
    variable two, t, hi, d: Integer
    Increment (two)
    t := Replica (i)
    hi := Replica (i)
    Increment (hi)
    Clear (i)
    d := Replica (hi)
    loop
      maintains 0 <= i and i * i <= t and
      t < hi * hi and hi <= MAX and
      d = hi - i and one = #one and
      two = #two and t = #t
      decreases d
      while IsGreater (d, one) do
        variable m, msq, r: Integer
        m := Replica (i)
        if MLEq (m, hi) then
          Add (m, hi)
        end if
        if MLEq (m * hi, MAX) then
          Divide (m, two, r)
        end if
        if MLEq (msq * msq, MAX) then
          Square (msq)
          if IsGreater (msq, t) then
            hi :=: m
          else
            i :=: m
          end if
        end if
        d := Replica (hi)
        Subtract (d, i)
      end loop
    end loop
  end if
end Sqrt

Figure 4.15: Explaining the different paths and requirements.
It is worth noting that we are highlighting a possible error with the use of \texttt{Square}, despite it relying on code we know to be defective previously (since the earlier call of \texttt{Add} has a counterexample). It should be up to the user to decide whether or not this is shown; here it is shown for completeness. The case for not showing it is that \texttt{Add} has potentially undefined behavior, thus we lose all trust in results relying on a correct use of \texttt{Add} being likewise correct. It might, however, still show a different error further in the code, or, potentially, be along a path where \texttt{Add} is called properly, and so the user should have the option to show potential later errors if they would like. The violated \texttt{ensures} clause for \texttt{Sqrt} would still be highlighted in any case, as it relies on the loop invariant, not the body of the loop.

If not for counterexamples being available, especially after the \texttt{while} loop, we might consider that some of the loop invariants are too strong. Other statements, such as \texttt{confirm} statements or \texttt{ensures} clauses for operations, might be suspected at some point. In any case, it is useful to ask where a fact is relevant. If it is not relevant where we think it ought to be, this can help point us to the problem. Figure 4.16 demonstrates an examination of where the loop invariant $0 \leq i$ is used.

Next to each place where the fact is used, or could be used, a $\star$ is placed. A solid $\star$ indicates the fact was definitely used here. Internally, this is easy to check; for any theorem prover that found the relevant VC valid at this point, we may give the same VC, minus the appropriate given or givens. If we still find it to be valid, the fact is not necessary (perhaps due to other similar facts), whereas otherwise it is.

In the case where a VC is left unsolved, or a counterexample is found, we may use a similar trick. The process is shown in Figure 4.17. Instead of showing that it is still true, which we were unable to show originally and so we would still not be able to,
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
    variable one: Integer
    Increment (one)
if IsGreater (i, one) then
    variable two, t, hi, d: Integer
    Increment (two)
    Increment (two)
    t := Replica (i)
    hi := Replica (i)
    Increment (hi)
    Clear (i)
    d := Replica (hi)
loop
    maintains 0 <= i <= t and i * i <= t and
    t < hi * hi and hi <= MAX and
    d = hi - i and one = #one and
    two = #two and t = #t
    decreases d
while IsGreater (d, one) do
    variable m, msq, r: Integer
    m := Replica (i)
    Add (m, hi) \( \star \)
    Divide (m, two, r)
    msq := Replica (m)
    Square (msq) \( \star \)
    if IsGreater (msq, t) then
        hi := m
else
    i := m
end if
    d := Replica (hi)
    Subtract (d, i) \( \star \)
end loop
end if
end Sqrt
end BinarySearch

Figure 4.16: Selecting code shows where the facts are used, to help the programmer determine whether it is correct or necessary.
we may show that we produce the same processed VC, up to variable naming. Ergo, the fact removed is irrelevant to the validity of the VC. It is not possible to show relevancy, so in the case where we get a different result we must still include it. We denote this possibility with a ★ that is hollow.

![Diagram](image)

Figure 4.17: Process to see if a fact is unnecessary in an unsolved VC.

We thus see that \( 0 \leq i \) is relevant in showing... \( 0 \leq i \); that is, the loop invariant is a necessary given to show that the loop invariant holds at the end of an iteration of the loop. It was also integral in proving the requires clauses for both Add and Subtract. It cannot be ruled out for the requires clause of Square, or for the ensures clause of Sqrt. Not demonstrated here, but multiple ★s would denote that there were multiple paths or clauses where a fact is definitely or possibly relevant.

The color of the ★ helps to describe further where a fact is used. Green goes hand-in-hand to a solid ★; orange implies the fact or facts are possibly relevant in an unsolved case; red with a thick border is when there is a counterexample. This helps
to highlight that with $\texttt{Add}$, $0 \leq i$ is irrelevant for the case of a counterexample, but is relevant for proving $\text{MIN} \leq m + hi$.

It is now time to attempt to fix the code. A user could ask for a symbolic tracing table of the code; this would look very similar to the other symbolic tracing tables we have seen, and so there is little utility in showing another one here. Instead, we shall ask for a tracing table with the counterexample for the \texttt{ensures} clause of $\texttt{Sqrt}$, shown in Figure 4.18. Here, the code not along the relevant path is deemphasized. At each relevant state, the mathematical models for every variable (including the constants \texttt{MAX} and \texttt{MIN}) are shown. When one of them changes between states, it is highlighted. As there may be multiple counterexamples to consider, the relevant one is shown. In this case we are looking at the first path that fails to show $\#i < (i+1) \times (i+1)$.

There are numerous curious things about this counterexample. The first is that, clearly, the \texttt{ensures} clause is not met. Equally important is that $hi = -2$ at this point in the code; the description of algorithm for binary search should imply $0 \leq \#i \leq i \leq hi \leq \#hi \leq \#i + 1$ at the end of an iteration of the loop, or, more directly, that both $hi \geq 0$ and $hi \geq i$. It appears that what we know about $hi$ does not fit this criterion.

Before attempting to make a full-fledged statement of what is wrong, we shall look at the second counterexample, shown in Figure 4.19. We see that again, our presumed invariants based on the algorithm are violated, in this case, that $hi \leq \#hi$. It appears that things we know to be invariant, specifically $0 \leq hi$, $i \leq hi$, and $hi \leq \#hi$, are not being used; they certainly are not mentioned as loop invariants.

A useful question does come up, however: why are counterexamples found to other parts of the code? For instance, the \texttt{ensures} clause for $\texttt{Sqrt}$ has two components to it,
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
MAX = 5, MIN = -2, i = 3
variable one: Integer

MAX = 5, MIN = -2, i = 2, one = 0
Increment (one)

MAX = 5, MIN = -2, i = 2, one = 1

if IsGreater (i, one) then
variable two, t, hi, d: Integer

MAX = 5, MIN = -2, i = 2, one = 1, two = 0, t = 0, hi = 0, d = 0

Increment (two)

MAX = 5, MIN = -2, i = 2, one = 1, two = 1, t = 0, hi = 0, d = 0
Increment (two)

MAX = 5, MIN = -2, i = 2, one = 1, two = 2, t = 0, hi = 0, d = 0
Increment (two)

MAX = 5, MIN = -2, i = 2, one = 1, two = 2, t = 2, hi = 0, d = 0
hi := Replica (i)

MAX = 5, MIN = -2, i = 2, one = 1, two = 2, t = 2, hi = 2, d = 0
Increment (hi)

MAX = 5, MIN = -2, i = 2, one = 1, two = 2, t = 2, hi = 3, d = 0
Clear (i)

MAX = 5, MIN = -2, i = 0, one = 1, two = 2, t = 2, hi = 3, d = 3

loop

maintains 0 <= i and i * i <= t and
t < hi * hi and hi <= MAX and
d = hi - i and one = #one and
two = #two and t = #t
decreases d

while IsGreater (d, one) do
variable m, msq, r: Integer

m := Replica (i)
Add (m, hi)
Divide (m, two, r)
msq := Replica (m)
Square (msq)
if IsGreater (msq, t) then
hi :=: m

else
i :=: m
end if

r :=: m
end if

d := Replica (hi)
Subtract (d, i)

end loop

end Sqrt

end BinarySearch

Figure 4.18: Examining one of the counterexamples showing that the code is defective.
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
MAX = 5, MIN = 0, i = 2
variable one: Integer
MAX = 5, MIN = 0, i = 2, one = 0
Increment (one)
MAX = 5, MIN = 0, i = 2, one = 1
if IsGreater (i, one) then
variable two, t, hi, d: Integer
MAX = 5, MIN = 0, i = 2, one = 1, two = 0, t = 0, hi = 0, d = 0
Increment (two)
MAX = 5, MIN = 0, i = 2, one = 1, two = 1, t = 0, hi = 0, d = 0
Increment (two)
MAX = 5, MIN = 0, i = 2, one = 1, two = 2, t = 0, hi = 0, d = 0
t := Replica (i)
hi := Replica (i)
MAX = 5, MIN = 0, i = 2, one = 1, two = 2, t = 2, hi = 0, d = 0
Increment (hi)
MAX = 5, MIN = 0, i = 2, one = 1, two = 2, t = 2, hi = 2, d = 0
Clear (i)
MAX = 5, MIN = 0, i = 0, one = 1, two = 2, t = 2, hi = 3, d = 0
d := Replica (hi)
MAX = 5, MIN = 2, i = 0, one = 1, two = 2, t = 2, hi = 3, d = 3
loop
maintains 0 <= i and i * i <= t and
t < hi * hi and hi <= MAX and
d = hi - i and one = #one and
two = #two and t = #t
decreases d
while IsGreater (d, one) do
MAX = 5, MIN = 2, i = 1, one = 1, two = 2, t = 2, hi = 5, d = 2
variable m, msq, r: Integer
MAX = 5, MIN = 2, i = 1, one = 1, two = 2, t = 2, hi = 5, d = 2, m = 0, msq = 0, r = 0
m := Replica (i)
MAX = 5, MIN = 2, i = 1, one = 1, two = 2, t = 2, hi = 5, d = 2, m = 1, msq = 0, r = 0
MIN <= m + hi
Add (m, hi)
M * hi <= MAX
Divide (m, two, r)
msq := Replica (m)

Figure 4.19: Examining the second counterexample showing that the code is defective.
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
    variable one: Integer
    Increment (one)
    if IsGreater (i, one) then
        variable two, t, hi, d: Integer
        Increment (two)
        t := Replica (i)
        hi := Replica (i)
        Increment (hi)
        Clear (i)
        d := Replica (hi)
    loop
        maintains 0 <= i and i * i <= t and
        t < hi * hi and hi <= MAX and
        d = hi - i and one = #one and
        two = #two and t = #t
        decreases d
    while IsGreater (d, one) do
        variable m, msq, r: Integer
        m := Replica (i)
        Add (m, hi)
        Divide (m, two, r)
        msq := Replica (m)
        Square (msq)
        if IsGreater (msq, t) then
            hi := m
        else
            i := m
        end if
        d := Replica (hi)
        Subtract (d, i)
    end loop
end Sqrt
end BinarySearch

Figure 4.20: Examining what in the code was necessary to prove one of the requirements.
only one of which has a counterexample when going down the path through the if branch. What did it rely upon? We, the user, can find out by selecting the relevant path and clause as shown in Figure 4.20. A tool such as SplitDecision can take the relevant VC, and incrementally remove facts to check if they are necessary, similar to what was used in Figure 4.16. Here we find that we know can show \( i \times i \leq \#i \) because \( t = \#i \), and \( i \times i \leq t \) at all times. This, if anything, should make us more confident that strengthening the loop invariants will help.

The loop invariants we will add in this example are \( h_i \leq \#h_i \), \( \#i \leq i \), and \( i \leq h_i \). These are not the specific invariants the counterexamples highlighted as being a problem, but those invariants follow by transitivity. The result is shown in Figure 4.21. Square is still a problem, as is Add, but for Add no counterexample is found. In this case the possible error in Square should be shown. We do not know whether Add is correct; perhaps if we had a better mathematical theory we could show it to be correct, but it would still be the case that Square is potentially defective. Our interface, therefore, should show both Add and Square to have potential defects.

For now, we will concentrate on Add. Examining the requirements to be proved in the same way we did in Figure 4.15, we see that again, we are having difficulty proving \( \text{MIN} \leq m + h_i \). Given that we know the value of \( h_i \) is at most one more than the original incoming value of \( i \), and \( i \times i < \text{MAX} \), it certainly seems unlikely that this is a problem, but could we prove it ourselves? More importantly, would we expect another programmer to be able to prove it?

Here it would be best to step back and consider what we are trying to achieve with the call to Add. Ostensibly, we are taking two numbers, adding them together, and then dividing by 2. Indubitably, the code averages to find the midpoint used for our
realization BinarySearch implements Sqrt for IntegerFacility

uses Add for IntegerFacility
uses Subtract for IntegerFacility
uses Multiply for IntegerFacility
uses Divide for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
variable one: Integer
Increment (one)
if IsGreater (i, one) then
variable two, t, hi, d: Integer
Increment (two)
Increment (two)
t := Replica (i)
hi := Replica (i)
Increment (hi)
Clear (i)
d := Replica (hi)
loop
maintains 0 <= i and i * i <= t and
t < hi * hi and hi <= MAX and
d = hi - i and one = #one and
two = #two and t = #t and
hi <= #hi and
#i <= i and i <= hi

decreases d
while IsGreater (d, one) do
variable m, msq, r: Integer
m := Replica (i)

msq := Replica (m)

if IsGreater (msq, t) then
hi :=: m
else
i :=: m
endif

d := Replica (hi)
Subtract (d, i)
end loop
endif
end Sqrt
end BinarySearch

Figure 4.21: Second attempt at binary search implementation of Sqrt in Listing 4.20.
binary search routine. Averaging is, as mentioned earlier in Section 4.2.2, potentially fraught with overflow and underflow errors when dealing with bounded integers, a lesson that seems to be learnt time and again. There are certainly correct ways to do it—only doing the “standard” averaging if the signs differ, and otherwise noting \( \frac{a+b}{2} = b + \frac{a-b}{2} \), where \(|a| > |b|\). In any case, we will be making our realization much more complicated if we place this inline with the rest of our code. Fortunately, we may make use of the Average enhancement to IntegerFacility, shown in Listing 4.21. The contract states that twice the average must always be less than or equal to the sum of the two incoming values, and only one fewer at worst (which would happen if one value was even and one odd).

```plaintext
contract Average enhances IntegerFacility

procedure Average (updates i: Integer, restores j: Integer)
  ensures
    i + j \leq #i + j and #i + j \leq i + i + 1
end Average
```

Listing 4.21: Contract for Average enhancement to IntegerFacility.

While adding the uses statement for Average, we notice there are uses statements we do not need anymore. At this point, we may ask the editor what code is currently unnecessary. The resulting code and the highlighted unnecessary uses statements are shown in Figure 4.22. We have no need to have the operations Add or Divide. Also, as is often the case, there is a somewhat antediluvian reference to operation Multiply never even used in this iteration of the code, probably from before there was a mention of the operation Square.

It also occurs to us that the variable two seems of little use now that we do not need to call Divide with it. On selecting two, we may ask to be shown where facts about two used. Unsurprisingly, we discover that knowledge of the value of two is
realization BinarySearch implements Sqrt for IntegerFacility
    uses Add for IntegerFacility
    uses Average for IntegerFacility
    uses Subtract for IntegerFacility
    uses Multiply for IntegerFacility
    uses Divide for IntegerFacility
    uses Square for IntegerFacility
procedure Sqrt (updates i: Integer)
    variable one: Integer
    Increment (one)
    if IsGreater (i, one) then
        variable two t, hi, d: Integer
        Increment (two)
        Increment (two)
        t := Replica (i)
        hi := Replica (i)
        Increment (hi)
        Clear (i)
        d := Replica (hi)
        loop
            maintains 0 <= i and i * i <= t and
            t < hi * hi and hi <= MAX and
            d = hi - i and one = #one and
            two = #two and t = #t and
            hi <= #hi and
            #i <= i and i <= hi
            decreases d
            while IsGreater (d, one) do
                variable m, msq: Integer
                m := Replica (i)
                Average (m, hi)
                msq := Replica (m)

                if IsGreater (msq, t) then
                    hi := m
                else
                    i := m
                end if
                d := Replica (hi)
                Subtract (d, i)
            end loop
        end if
    end Sqrt
end BinarySearch

Figure 4.22: Third attempt at binary search implementation of Sqrt in Listing 4.20.
needed for its calls of Increment (so that we know the increased value will be in bounds), and for the loop invariant two = #two. As it is never used to talk about anything else, we may safely remove it.

The simplified version of the code is shown in Figure 4.23. Given that Average has no requires clause, it should be unsurprising that there is now no problem calling the operation. Unfortunately, there is still an issue with Square, and an examination shows that we are unable to prove (or disprove) that $m \times m \leq \text{MAX}$ always. This is, perhaps, both unsurprising and somewhat disheartening. Unfortunately, most provers are limited to linear arithmetic, and balk at anything else. Before throwing in the towel, however, it is worth considering why we even think it is true.

The incoming value, $m$ (since msq has its value by a call to Replica (m)) is the average of $i$ and $hi$. That average should be no more than $t$, which is holding i’s original value. Further, all values are non-negative, so the average is non-negative as well. We know then that $0 \leq m \leq t$, which should mean $m \times m \leq t \times t$. And since $t \times t < \text{MAX}$, we appear to have what we want. If any of the tools have some form of transitivity (such as SplitDecision), then we may very well be able to prove the requires clause for Square.

Through the use of confirm statements, we can help a prover break the problem up in to chunks that it can better handle. Our attempt is shown in Figure 4.24. In using multiple confirm statements, we are trying to document our reasoning above, which serves the purpose of both aiding in the proof, and also providing useful documentation as to why the code should be correct.30

30 Incidentally, while we are using two confirm statements, we could easily have done it with three, in which case the proof of $m \leq t$ would have had the fact from $0 \leq m$ to rely upon. For the same reason, we would not try this in just one statement, as then proving $m \times m \leq t \times t$ would not be able to rely on the other two statements.
realization BinarySearch implements Sqrt for IntegerFacility

uses Average for IntegerFacility
uses Subtract for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
variable one: Integer
Increment (one)
if IsGreater (i, one) then
variable t, hi, d: Integer
t := Replica (i)
hi := Replica (i)
Increment (hi)
Clear (i)
d := Replica (hi)
loop
  maintains 0 <= i and i * i <= t and
t < hi * hi and hi <= MAX and
d = hi - i and one = #one and
t = #t and hi <= #hi and
  #i <= i and i <= hi
  decreases d
while IsGreater (d, one) do
  variable m, msq: Integer
  m := Replica (i)
  Average (m, hi)
  msq := Replica (m)
  if IsGreater (msq, t) then
    hi :=: m
  else
    i :=: m
  end if
  d := Replica (hi)
  Subtract (d, i)
end loop
end if
end Sqrt

end BinarySearch

Figure 4.23: Cleaning up the cleaned up third attempt binary search implementation of Sqrt in Listing 4.20.
realization BinarySearch implements Sqrt for IntegerFacility

    uses Average for IntegerFacility
    uses Subtract for IntegerFacility
    uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
    variable one: Integer
    Increment (one)
    if IsGreater (i, one) then
        variable t, hi, d: Integer
        t := Replica (i)
        hi := Replica (i)
        Increment (hi)
        Clear (i)
        d := Replica (hi)
        loop
            maintains 0 <= i and i <= t and
            t < hi and hi <= MAX and
            d = hi - i and one = #one and
            t = #t and hi <= #hi and
            #1 <= i and i <= hi and
            0 <= hi
            decreases d
            while IsGreater (d, one) do
                variable m, msq: Integer
                m := Replica (i)
                Average (m, hi)
                confirm 0 <= m and m <= t
                confirm m * m <= t * t
                msq := Replica (m)
                Square (msq)
                if IsGreater (msq, t) then
                    hi :=: m
                else
                    i :=: m
                end if
                d := Replica (hi)
                Subtract (d, i)
            end loop
        end if
    end Sqrt
end BinarySearch

Figure 4.24: Fourth attempt at binary search implementation of Sqrt in Listing 4.20, now with confirm statements.
What happens to the generated VCs upon adding these statements? We are able to prove the first confirm, so it is indeed true that both \( 0 \leq m \) and \( m \times m \leq t \times t \). Further, knowing that \( m \times m \leq t \times t \) makes short work of the requires clause of Square. Amazingly, we are unable to prove the statement \( m \times m \leq t \times t \), which would seem to be the easy part in all of this, given what we already know. To finally push the code through, we must resort to a little bit of ingenuity.

Our initial attempt to show that \( m^2 \leq \text{MAX} \) was to find a value \( t^2 \) inclusively between \( m^2 \) and MAX. The problem lies in showing \( m^2 \leq t^2 \), so we will concentrate our efforts there. We have already shown at this point that \( 0 \leq m \leq t \), and of course know \( 1 < t \). If we take \( m \leq t \) and multiply each side by \( m \), we have that \( m^2 \leq m \cdot t \). Similarly, if we multiply each side of \( m \leq t \) by \( t \), we have \( m \cdot t \leq t^2 \). Putting those together shows us that \( m^2 \leq m \cdot t \leq t^2 \), which is exactly what we want to know. Automated prover willing, confirm statements of \( m \times m \leq m \times t \) and \( m \times t \leq t \times t \) will be enough to completely verify the code.

This revision is shown in Figure 4.25. At this point, the final confirm statement worked.\(^{31}\) The code is now verified, and this is indicated to the user with a golden checkmark. An interesting question, and one that allows a programmer to understand the strengths of the tools being used, is to ask which automated provers solved which part of the code. Asking such a question reveals Figure 4.26.

If the relevant VC is so obvious it would normally be tossed out, nothing is shown. Otherwise, each tool that returned first on that VC is what is shown, with a thicker, darker border if it is was the only tool to return with valid. Sometimes, there might be

\(^{31}\)While it may seem unfortunate that we had to go to great lengths with confirm statements, there should be some solace in knowing that advancements in automated theorem provers will only aid in the ability to eliminate them over time. A programmer might choose to leave the original two confirm statements in, however, as they provide useful documentation as to why the code is correct.
realization BinarySearch implements Sqrt for IntegerFacility

uses Average for IntegerFacility
uses Subtract for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
    variable one: Integer
    Increment (one)
    if IsGreater (i, one) then
        variable t, hi, d: Integer
        t := Replica (i)
        hi := Replica (i)
        Increment (hi)
        Clear (i)
        d := Replica (hi)
        loop
            maintains 0 <= i and i * i <= t and
                t < hi * hi and hi <= MAX and
                d = hi - i and one = #one and
                t = #t and hi <= #hi and
                #i <= i and i <= hi and
                0 <= hi
            decreases d
            while IsGreater (d, one) do
                variable m, msq: Integer
                m := Replica (i)
                Average (m, hi)
                confirm 0 <= m and m <= t
                confirm m * m <= m * t and m * t <= t * t
                confirm m * m <= t * t
                msq := Replica (m)
                Square (msq)
                if IsGreater (msq, t) then
                    hi :=: m
                else
                    i :=: m
                end if
                d := Replica (hi)
                Subtract (d, i)
            end loop
        end if
    end Sqrt
end BinarySearch

Figure 4.25: Final attempt at binary search implementation of Sqrt in Listing 4.20.
realization BinarySearch implements Sqrt for IntegerFacility

uses Average for IntegerFacility
uses Subtract for IntegerFacility
uses Square for IntegerFacility

procedure Sqrt (updates i: Integer)
  variable one: Integer
  Increment (one)
  if IsGreater (i, one) then
    variable t, hi, d: Integer
    t := Replica (i)
    hi := Replica (i)
    Increment (hi)
    Clear (i)
    d := Replica (hi)
    loop
      maintains 0 <= i <= t and
      t < hi * hi and hi <= MAX and
      d = hi - i and one = #one and
      #i <= i and hi <= #hi
      #t <= t and
      0 <= hi
      decreases d

while IsGreater (d, one) do
  variable m, msq: Integer
  m := Replica (i)
  Average (m, hi)
  confirm 0 <= m <= #t and
  confirm m * m <= t <= #t * #t
  confirm m * m <= t * t
  msq := Replica (m)
  Square (msq)
  if IsGreater (msq, t) then
    hi :=: m
  else
    i :=: m
  end if
  d := Replica (hi)
  Subtract (d, i)
end loop
end Sqrt
end BinarySearch

Figure 4.26: Showing the user which prover was involved at each point in the code for verifying our implementation of Sqrt.
multiple things to prove, such as the \texttt{decreases} clause of the loop, and so there is a possibility of multiple tools being shown. Of particular interest to this problem is how \texttt{Isabelle}, \texttt{SplitDecision}, and \texttt{Z3} are required to handle all of the relevant \texttt{decreases} requirements. Additionally, we see that both \texttt{Isabelle} and \texttt{SplitDecision} are in use to prove all the \texttt{confirm} statements. In fact, \texttt{SplitDecision} is necessary to handle the case of $m \times t \leq t \times t$.\textsuperscript{32}

4.6.2 Final details

The suggested interface for debugging makes no claims to being the \textit{best} one; rather, it claims to present the details necessary to aid a programmer who is trying to verify their code, without forcing them to look at the VCs themselves. For example, the method of invoking any specific view is left unsaid, as this is particular to the overall interface used. One can imagine one interface handling it all by menu actions, or another through appropriately placed contextualized buttons.

In the example in Section 4.6.1, it was not demonstrated how to handle code that is being actively developed. It started with mostly complete code as its base. How should the code be checked for validity while writing? In essence, what would be the equivalent to incremental compilation when writing code to be verified? We should note that, as compilation does not create a compiled product unless the code is completely valid, there is ostensibly no point in modifying the code once it has been verified.\textsuperscript{33} We instead need to worry about handling incremental parsing [21, 58].

\textsuperscript{32}A similar enhancement \texttt{Sqrt} exists for unbounded integers, minus the restrictions due to bounds, and of course a similar realization exists, with the \texttt{confirm} statements to prove the bounds. Interestingly, in this implementation, both \texttt{Z3} and \texttt{SplitDecision} show themselves to be necessary to prove all of the code by proving VCs that no other prover can. \texttt{Isabelle} turns out not to be necessary.

\textsuperscript{33}There is the possibility of trying to write a new, optimized, version with the original verified code as a base. However, unless the programmer is willing to maintain correctness with each new line of
Assuming that it is possible to incrementally generate VCs, or at least generate VCs very quickly, which problems with verification should be shown, and how?

One advantage to OSU’s Resolve is being able to say there actually is a counterexample.\(^{34}\) If, for some path, we are able to produce a counterexample, there certainly is no reason to continue to produce VCs and attempt to verify them. The indication that there is a counterexample in this case should be innocuous, however. Unlike regular syntax checking, where fixing the syntax error is a relatively local fix, writing a realization to a contract would be the whole point of the programming session. Aggressively reminding the programmer they have not finished would only serve to annoy them; we hardly want the programmer to write the equivalent of “\texttt{assume false}” before the error to make it go away.\(^{35}\) We might think we should highlight the case when all obligations at a point in the code are met; this presents a situation where the programmer is unlikely to want to change the code, and this should be easily understood by the programmer. Such highlighting implies it is “correct”, which we have no way of saying until seeing the whole. A programmer can, and certainly should, be able to look at what is provable when the code is in a certain state, but any attempt to do this by default merely presents a false sense of security to the programmer.

code in the change, this is a highly unlikely scenario to require \textit{also} allowing conventional incremental compilation. Attempts at optimized versions should be encouraged to just be new realizations of the same contract.

\(^{34}\)Tools such as Z3, or Boogie which relies on Z3, provide a “counterexample”, but make no guarantees that it actually is a counterexample. Instead, they assign values to variables in some way based on the final state of the VC after processing.

\(^{35}\)This could be accomplished by invoking a dummy operation with an \texttt{ensures} clause of \texttt{false}. Such an operation would be unimplementable, but it would make the code in question “correct”.

124
One useful feature for writing code is to also make available a symbolic tracing table to the user as they write (similar to what is done with VeriFast’s IDE [31]), either around the portion of code they are currently working on, or for the realization as a whole. Had a programmer been presented with feedback such as what is shown in Figure 4.27, the initial problem of weak loop invariants would likely have been caught sooner.

![Figure 4.27: Giving a symbolic trace while programming, along with a found counterexample.](image)

### 4.7 Future work

We can break up the future work in to two parts, that of the behind-the-scenes tools (VC generation, automated provers, etc.), and that of a front-end editor.

#### 4.7.1 Front end

While we have demonstrated features of what information a front-end should provide, the best way to present that information is still unknown. The next steps are
to build a prototype interface, and determine where improvements in the presentation are needed.

Additional work also includes examining new features that can be presented to the user to aid in writing clean, correct, and efficient code. Some of this was hinted at in the previous section; by presenting the user with every location where facts about a variable are used, the programmer may decide whether or not the variable is even needed. It would be helpful to automate the process of determining if a variable is necessary, and present it as an audit, along with any unnecessary dependencies.

Showing the user the location of verifiably dead code is another task that would be helpful to automate. Conventional compilers do not have automated theorem provers at hand to aid them in this task, but we do. Dead code—an \texttt{if} branch that is never taken, for example—has the characteristic that any VCs purporting to go through those paths has contradictory (\textit{i.e.}, inconsistent) givens. It is not the only way contradictory givens can exist, however. For example, the breaking of VCs into "simpler" VCs can potential generate "useless" VCs; that is, by essentially breaking a problem up in to different cases, some of those cases might never happen and could appear as false positives in dead code detection efforts. This problem needs further examination. Indeed, it might turn out to be a boon, as this information could present us with the opportunity to let users know of impossible cases. There might be a desire to document this, with a \texttt{confirm} statement perhaps, or suggest to the programmer there is something wrong. Future ideas are expected to come through use of the prototype interface. Since so much is known statically through the generation of VCs, the biggest difficulty is not in being able to comb through the VCs and give feedback to the user; rather, it is determining what is the pertinent information to provide.
Additional work is necessary to better identify problematic specifications, either in
the contracts being realized, contracts being relied upon by using an operation as a
client, or annotations such as confirm statements or loop invariants. It is currently
unclear as to what form these should take.

4.7.2 Back end

Currently produced VCs with OSU’s Resolve do not keep track of why a fact is
generated; this is not a technical problem, but is not implemented merely because
it was not necessary originally. These facts persist along much of the pipeline that
produces VCs, and are lost only towards the end of the generation. As so much of the
feedback depends on this, it will be useful to implement.

A useful technique employed by SplitDecision is to selectively remove givens from a
valid VC and check if it can still be found valid. SplitDecision does this internally, but
externally modifying the VCs and comparing answers with tools such as Isabelle or
Z3 would provide the same effect. For VCs that SplitDecision is unable to solve but
another is able to, we would then have the ability to give better feedback. For VCs
that multiple tools are able to solve, we can attempt to further refine which facts are
necessary; it is possible that a given is rendered superfluous by mathematical theory
possessed by another tool. Finally, more research is necessary to determine how the
order in which givens are removed impacts the utility of the feedback. No guarantee is
made that the final collection of givens is the smallest possible, and more work needs
to be done to get a tighter bound on the number of givens. Some of this work might
include starting from no givens, and progressively adding them.
4.8 Related work

The initial suggestion of debugging when confronted with unsolved VCs in Resolve comes from [25]. The work of Hermenegildo et al. [26] addresses the idea of an IDE for debugging, but specifically avoids true verification attempts since they consider it to be “not realistic” to do full-functional verification, and as such is more about generating test cases. Much work exists on using formalisms in order to generate test cases [1]. Intel has done some work on the generation of counterexamples to formal specifications for hardware design as a way of static debugging [12].

In terms of IDE work and theorem provers, Mulhern et al. [41] attempt to describe an IDE for those working on the proofs directly, as opposed to processing information out of them. Other IDE research has focused on how to best interact with theorem provers [6].

One interesting line of work is to check if a counterexample generated can actually occur from the code [22]. It unfortunately places stringent rules on the contracts of operations. Without these restrictions, it is not clear whether a loop invariant is too weak or strong, or that the code is incorrect. Work to strengthen loop invariants, specifically by generating new ones, is one active area of research [37].

Pex [55] is a tool by Microsoft Research designed to have high code coverage, by using symbolic execution and examining constraints to try and cover as many paths as possible. We differ in this by doing complete code coverage through the generation of VCs, a benefit of using formal specifications (which Pex does not use).

Some work exists on using exhaustive search of inputs up to a specified bound in order to find counterexamples [11]. As our counterexamples are for the mathematical model, not the concrete implementation, a direct comparison is difficult. The examples
of bounded integers, for example, might be difficult for such a tool to find a counterexample for, in part if they are not allowed to specify MIN or MAX. Model checking can of course generate counterexamples [49], but due to its focus on checking that specific properties hold, it is unclear that similar methods are usable for full-functional verification.


in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), DEFECTS ’09, pages 16–20, New York, NY, USA, 2009. ACM.


