Implementing a Resolve Online Prover Using Z3

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

John Bentley, B.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2021

Master's Examination Committee:

Paolo A. G. Sivilotti, Advisor

Neelam Soundarajan

Abstract

The Resolve Online [17] development environment currently supports multiple provers: Dafny, SplitDecision, and variations and combinations thereof. The existing provers used by Resolve Online have some areas of improvement. One downside of the Dafny prover is that it generates Dafny programs to discharge verification conditions, tying it closely to the specific version of the Dafny compiler used. This approach is brittle because new releases of Dafny tend to break the backend prover, making it difficult for the Resolve Online tool to leverage improvements in the Dafny language. Other areas of potential improvement include the number of verification conditions that are automatically discharged and the execution time of the provers.

Z3 [6] is a modern theorem prover developed at Microsoft Research. First released in 2012, Z3 is under active development, with 3 releases in the last year at the time of writing [1] and frequent commits. Z3 is used by Boogie [20], F* [26], Pex [27], and more, as well as tools that indirectly use Z3 through Boogie. This ongoing development could lead to future improvements that will allow for better verification of Resolve programs in the future. In order to leverage Z3 in Resolve Online, we have extended Resolve Online by creating a new prover backend that uses Z3.

A benchmark of Resolve code consisting of 30 examples was used. Each example was run with the three provers: the new Z3 backend, Dafny, and SplitDecision. In all examples Z3 is able to prove at least as many VCs as Dafny, and in 28 examples,

Z3 was able to prove at least as many VCs as SplitDecision. There are 20 examples where Z3 proved more VCs than Dafny and 11 when compared to SplitDecision. A smaller benchmark consisting of 8 complex examples from the full benchmark was selected to be used for analyzing the execution time of the provers. In most cases, Dafny takes substantially longer than SplitDecision and Z3. SplitDecision and Z3 are both close in the execution times. Either one of the two could be faster depending on the example. However, there is one major outlier where Z3 is significantly faster than SplitDecision, running for under a minute while SplitDecition runs for around 3 hours.

Acknowledgments

I would like to thank my advisor, Dr. Paul Sivilotti. His advice and constant guidance have been instrumental in my experience at Ohio State. His helpful direction and criticisms in the completion and analysis benefited this work greatly.

Thanks to Dr. Neelam Soundarajan for serving as a member on my committee and for his class about Programming Languages. I would also like to thank the Reusable Software Research Group for their feedback. Kathryn Reeves was a huge help in dealing with multiple administrative issues. Finally, I would like to thank my parents for their encouragement and support.

Vita

2012 ..... B.S. Computer Science,
Ashland University

2019-2021 ..... Graduate Teaching Assistant,
The Ohio State University.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

Reasoning about software to prove correctness has long been a goal of computer scientists. Verification is the process of proving that a program meets a specification. Hoare logic, [16] developed by Tony Hoare, is a system of techniques that defines axioms and rules of inference for proving that programs meet a specification. Hoare logic is the basis for most modern verification systems. The Resolve [14, 15] language was created for the purpose of writing verifiable code.

There are techniques that can be used to statically detect specific classes of errors. There are many examples of tools and techniques to detect memory leaks [25, 29, 13, 24, 4, 18, 30]. There are also techniques for detecting deadlocks and race conditions [8, 23, 28]. However, while eliminating such errors is useful, these techniques cannot be used to detect logical errors such as a function producing an incorrect result. Full-functional verification, on the other hand, allows for defining a specification and proving that the implementation conforms to it.

Automated verification is a computationally intensive task. Satisfiability, a core component of many provers, is an NP-complete problem [5]. Furthermore, Gödel's incompleteness theorems [12] show that there are fundamental limits to algorithms that are designed to decide whether formulas are true. Despite these limitations, verification is still possible. Since most programs are designed by humans to perform a

practical function, there is more regularity than in an arbitrary program. It is possible to create a prover that is useful for verifying many programs written by humans, without being able to verify every verification condition that could theoretically be needed to verify an arbitrary program.

With more powerful computers, it is now feasible to build and run automatic provers. This means that instead of needing to perform verification by hand, programs can be verified automatically. Resolve Online [17] is a web-based development environment for Resolve programs. It supports multiple prover engines as backends to dispatch verification conditions in order to perform verification.

Resolve was designed to support verification from the beginning. This leads to advantages over existing languages such as C++, Java, and Rust. One advantage is that Resolve uses value semantics to eliminate aliasing. C++ and Java have no compiler support for handling aliasing. These languages allow for copying pointers or references which makes verification difficult. On the other hand, Rust has an ownership model that can be used to prevent mutable aliasing. As long as the user avoids `unsafe` code and escape hatches such as `RefCell`, the ownership model can be utilized for verification.

Resolve also distinguishes between specifications and executable code. This separation allows mathematicians to define the theories used in specifications, while programmers can write the implementations. In systems for verification of existing languages, such as Prusti [9] for Rust and JML [19] for Java, there is no separation. This lack of separation results in extra work to ensure that the functions used in specifications are mathematical functions. That is, that the functions are deterministic, side effect-free, and always terminate. Additionally, the use of a mathematical

representation of the state allows the implementation to represent its internal state in a manner suitable to the algorithm, without effecting the interface.

Resolve enforces separation of specifications from implementations. This abstraction is useful because it allows for multiple implementations of the same specification. Other languages support means of abstraction such as traits and interfaces, but the use of such features is not enforced. This lack of abstraction can result in tight coupling of the specification to the implementation.

The Resolve Online development environment currently supports multiple provers: Dafny, SplitDecision, and variations and combinations thereof. The existing provers used by Resolve Online have some areas of improvement. One downside of the Dafny prover is that it generates Dafny programs to discharge verification conditions, tying it closely to the specific version of the Dafny compiler used. This approach is brittle because new releases of Dafny tend to break the backend prover, making it difficult for the Resolve Online tool to leverage improvements in the Dafny language. Other areas of potential improvement include the number of verification conditions that are automatically discharged and the execution time of the provers.

Z3 [6] is a modern theorem prover developed at Microsoft Research. First released in 2012, Z3 is under active development, with 3 releases in the last year at the time of writing [1] and frequent commits. Z3 is used by Boogie [20], F* [26], Pex [27], and more, as well as tools that indirectly use Z3 through Boogie. This ongoing development could lead to future improvements that will allow for better verification of Resolve programs in the future. In order to leverage Z3 in Resolve Online, we have extended Resolve Online by creating a new prover backend that uses Z3. This

new backend is able to successfully verify more code and, in many cases, perform this verification faster.

Chapter 2: The Z3 Theorem Prover

Z3 [6] is a theorem prover developed at Microsoft Research. It is open source and available under the MIT License. It was developed for the primary purpose of solving problems in formal verification and related problems in software engineering and programming languages. It has been used to perform verification, automatic test generation, model checking, and more.

There are bindings and APIs for multiple languages including the C, C++, Python, .NET/C#, Java, and more. Z3 is used by Boogie [20], F* [26], Pex [27], and more. Boogie is used by Dafny [21], Viper [22], and more, allowing these tools to use Z3 as well.

Z3 uses an SMT (Satisfiability modulo theories) solver as the primary method of solving problems. An SMT solver combines a SAT solver with theories. Z3 recognizes the SMT-LIB [3] format as its syntactic representation of expressions. SMT-LIB expressions are Common Lisp S-expressions.

2.1 SAT Solvers

A SAT solver determines whether there is an assignment of boolean variables such that a given boolean formula is true. Boolean formulas consist of atomic formulas, conjunction ($a \wedge b$, `(and a b)`), disjunction ($a \vee b$, `(or a b)`), and negation ($\neg a$,

(`(not a)`). To determine satisfiability, atomic formulas are assigned either true or false values. If a set of assignments resulting in a true result exists, the result SAT indicates that the expression is satisfiable; if no such set of assignments exists, the result UNSAT indicates that the expression is unsatisfiable. For example, $(a \wedge b) \vee (b \wedge \neg c)$ is SAT because the expression is true when a and b are true. An example of an UNSAT formula is $a \wedge \neg a$ because a cannot be both true and false.

2.2 Quantifiers

Quantifiers are logical operators that bind variables within a formula. Z3 supports both universal quantifiers ($\forall$) and existential quantifiers ($\exists$). SAT solvers do not directly handle quantifiers. Instead, the subformulas involving quantifiers are effectively treated as atomic formulas. Then an separate technique is used for quantifier instantiation.

2.2.1 Universal Quantifiers

A universal quantifier binds variables for a formula that must hold for all values. Universal quantifiers are of the form $\forall x p(x)$. This means for all x, $p(x)$ is true.

2.2.2 Existential Quantifiers

An existential quantifier binds variables for a formula that must hold for some value. Existential quantifiers are of the form $\exists x p(x)$. This means that there exists a value x such that $p(x)$ is true.

2.3 Theories

As implied by the name Satisfiability modulo theories, boolean satisfiability alone is not sufficient for a SMT solver to produce a SAT result. In an SMT solver, atomic formulas can include variables, relations, and functions. The formula must additionally meet the requirements of the theories related to the relations and functions.

2.3.1 Integers

The theory of integers allows for mathematical comparisons and operations related to integers to be used in formulas. The formula $x > 0 \wedge x < 0$ is UNSAT. Under a SAT solver, $x > 0$ and $x < 0$ are considered distinct atomic formulas. However, the theory recognizes that these two formulas cannot be considered independently. The theory of integers understands that the variable x cannot be both positive and negative.

A number of operations are supported including addition (`(+ a b)`), subtraction (`(- a b)`), multiplication (`(* a b)`), division (`(/ a b)` resulting in a `Real` value or `(div a b)` for integer division), and modulo (`(mod a b)`). Equality (`(= a b)`) and inequality (`(< a b)`, `(> a b)`, `(<= a b)`, `(>= a b)`) comparisons are also supported.

2.3.2 Equality

Equality reasoning allows for reasoning about equality of formulas. Expressions representing equality have the form $a = b$. Since Equality is an equivalence relation, the properties of equivalence relations hold.

1. Identity: $\forall a :: a = a$

2. Symmetry: $\forall a, b :: a = b \iff b = a$

3. Transitivity: $\forall a, b, c : a = b \wedge b = c : a = c$

2.3.3 Sequences

Sequences are finite, ordered collections of values. Sequences support a number of operations, some of which are detailed below.

1. `(seq.unit x)` produces a sequence that contains `x`.

2. `(as seq.empty T)` produces an empty sequence.

3. `(seq.++ s1 s2 ...)` concatenates two sequences or more sequences.

4. `(seq.len s)` gets the number of values in the sequence.

5. `(seq.extract s start len)` gets a subsequence from the specified range.

2.3.4 BitVectors

BitVectors are used to represent fixed-size binary data. For example, `(_ BitVec 32)` represents a 32-bit value. Usually this is used to model fixed-size integral values. The `String` sort is defined as `(Seq (_ BitVec 8))`.

2.3.5 Custom Sorts

Z3 supports multiple kinds of custom sorts. This allows for defining sorts that represent types that are not natively supported by Z3.

Records

Records are sorts that contain a fixed set of fields. Each record has a constructor that, given the values of the fields, creates a value of the record sort. Each field of a

record has an accessor function that gets the value of the field from a record value. Listing 2.1 contains an example of a record with two `Int` fields.

```
1  (declare-datatypes () ((Pair (mkpair (first Int) (second Int)))))
2
3  (push)
4  (assert (not (= 4 (first (mkpair 4 5)))))
5  (check-sat)
6  (pop)
7
8  (push)
9  (assert (not (= 5 (second (mkpair 4 5)))))
10 (check-sat)
11 (pop)
```

Listing 2.1: Example of Records in Z3 (SMT-LIB)

Uninterpreted Sorts

Uninterpreted sorts define new sorts without any specific meaning. Instead, the meaning is based on the assertions made about the sort.

2.4 Other Features

Z3 supports other features as well. These features are not used in this work, but may be of interest for other use cases.

Custom types can be defined using discriminated unions. Values of discriminated unions can be constructed using one of multiple constructors.

Z3 supports strategies [7] similar to those found in interactive theorem provers.

Chapter 3: The Resolve Language

Resolve [14, 15] is a language created at The Ohio State University for the purpose of the specification and verification of programs.

3.1 Contracts and Realizations

Resolve separates code into contracts and realizations. This separation enables the consumer of the component to understand its behavior without needing to know how it is implemented. The separation also allows for multiple implementations of a contract. Furthermore, it allows for independent verification of individual functions and procedures.

Interfaces of the code are defined in contracts. Contracts define the supported procedures, functions, math functions, and types. Contracts can also be parameterized by math functions and types. Procedures and functions definitions include the specification which includes the parameter list, precondition, and postcondition. The precondition is specified in a **`requires`** clause and the postcondition in an **`ensures`** clause. Each parameter has a name, type, and a mode.

The **`restores`** mode is used when the parameter will have the same mathematical value after the procedure or function exits. This mode can be used by both procedures and functions. This is essentially equivalent to adding `x = #x` to the ensures clause

where `x` is the name of the parameter. The `#x` syntax represents the original value of `x` when the procedure or function is called.

In addition to the **`restores`** mode, procedures can also use three additional modes. The **`updates`** mode is used when it is expected that the parameter will be modified. The **`replaces`** mode is used when the behavior is the same, regardless of the original value. The **`clears`** mode is used when the value after the procedure or function exits is the initial value of the type of the variable.

Contracts can also enhance other contracts. Enhancements allow additional procedures and functions to be defined for existing contracts.

Implementations are defined in realizations. Realizations define the implementations of procedures and functions. Verification is performed on these implementations in order to ensure that the implementation conforms to the specification.

3.2 Value Semantics

Reference semantics (or pointers) make reasoning about programs more difficult. Consider the Concatenate procedure in Algorithm 1. This procedure moves values from the queue q into p.

However, if queues had pointer semantics, then `p` and `q` could be references to the same queue. In this case, the loop would remove an item from the queue and add it back into the same queue. This would mean the the length of `source` never decreases. As long as the queue is not initially empty, the loop is now infinite.

In order to avoid such problems, Resolve uses value semantics in order to eliminate aliases. This decision has implications that impacts how code is written. One consequence is that Resolve has no copying assignment. The assignment operator can

Algorithm 1 Concatenate Queues

```
1 procedure CONCATENATE(p, q)
2   while ¬IsEmpty(q) do
3     x ← Dequeue(q)
4     Enqueue(p, x)
```

only be used for the return value of a function. Because of this, copying requires a function to copy a value and create a duplicate value. Instead, the swap operator is preferred.

For example, the `Add` procedure for unbounded natural numbers in Listing 3.1 avoids copying the value of `m` is being added. It does this by building up a separate value `p`. Whenever it decrements `m`, it increments `p`. This means that once `m` reaches 0, `p` has the original value of `m`. The values are then swapped using `m :=: p`, restoring the original value of `m`.

```
1  // Contract
2  procedure Add(updates n: UnboundedNaturalBase,
3                restores m: UnboundedNaturalBase)
4    ensures
5      n = #n + m
6
7  // Realization
8  procedure Add (updates n: UnboundedNaturalBase,
9                 restores m: UnboundedNaturalBase)
10   variable p: UnboundedNaturalBase
11   loop
12     maintains n + m = #n + #m and m + p = #m + #p
13     decreases |m|
14   while not IsZero(m) do
15     Decrement(m)
16     Increment(n)
17     Increment(p)
18   end loop
19   m :=: p
20 end Add
```

Listing 3.1: `Add`procedure (Resolve)

3.3 Loop Invariants

Loop invariants are an essential tool for verifying code that uses loops. A loop invariant is a condition that must be true every time the loop condition is tested. Consider the loop invariant found in Listing 3.2, a Resolve implementation of Algorithm 1. The invariant is true initially because `p` and `q` still have their original values. The invariant is also true after a loop iteration because a value is removed from the front of `q` and added to the end of `p`. Because of this, after the loop exits, the loop invariant is known to hold.

```
1  // Contract
2  procedure Concatenate(updates p: Queue,
3                        clears q: Queue)
4    ensures
5      p = #p * #q
6
7  // Realization
8  procedure Concatenate (updates p: Queue,
9                         clears q: Queue)
10   loop
11     maintains p * q = #p * #q
12     decreases |q|
13   while not IsEmpty(q) do
14     variable x: Item
15     Dequeue(q, x)
16     Enqueue(p, x)
17   end loop
18 end Concatenate
```

Listing 3.2: `Concatenate`procedure (Resolve)

3.4 Mathematical Theories

Program types have underlying mathematical theories. The underlying theories include mathematical types which are distinct from program types, as well as associated operations and functions. This separation of program space and math space

allows for mathematicians to develop theories. These theories can then be incorporated into Resolve. Programmers can then use these theories as the basis of program types, procedures, and functions. This allows program types to make design decisions that are useful for programmers while still relying on an underlying theory.

3.4.1 Integers

The `integer` type represents mathematical integers. Integers have inequality comparison operators (`a < b`, `a > b`, `a <= b`, `a >= b`). The theory of integers also includes common operations such as addition (`a + b`), subtraction (`a - b`), multiplication (`a * b`), division (`a / b`), and negation (`-a`). Resolve also has syntax for modulo (`a` **`mod`** `b`) and absolute value (`|a|`). These symbols are an approximation for the mathematical syntax to make it easier to enter in the editor. For example `<=` is used instead of $\leq$, `>=` instead of $\geq$, and `*` instead of $\times$.

In Resolve, division and modulo satisfy the properties

$$a = qb + r \text{ and}$$

$$0 \leq r < |b|$$

where q is the result of `a / b` and r is the result of `a` **`mod`** `b`.

3.4.2 Booleans

The `boolean` type has two possible values: true and false. Boolean theory defines logical operations such as conjunction (`a` **`and`** `b`), disjunciton (`a` **`or`** `b`), negation (**`not`** `a`), and implication (`a implies b`). Universal and existential quantifiers such as **`for all`** and **`there exists`** also have type `boolean`. These operators can be written in mathematical notation using logical operators such as $\wedge$ for conjunction, $\vee$ for

disjunciton, $\neg$ for negation, $\implies$ for implication, $\forall$ for universal quantifiers and $\exists$ for existential quantifiers.

3.4.3 Finite Sets

A `finite set of A` is a finite set of values of type `A`. Set theory defines operations such as cardinality (`|a|`), union (`a union b`), intersection (`a intersection b`), difference (`a \ b`), and membership testing (`x is in a`). These set operations can be written in mathematical notation using $\cup$ for union, $\cap$ for intersection, and $\in$ for membership.

3.4.4 Strings

A `string of A` is a finite sequence of values from an alphabet `A`. String theory has operations such as concatenation (`a * b`), cardinality or length (`|a|`), and substring extraction (`substring(s, start, end)`). There is also an operation for getting the set of values contained in the string (`elements(s)`). Concatenation can be written mathematically as $\circ$.

The `substring(s, i, j)` operation is defined such that, if the indexes are in the range $0 \leq i \leq j \leq |s|$ then the substring contains the characters starting at index i and ending before the character at index j. If the indexes are not in range, then the substring is the empty string.

3.4.5 Characters

The `character` type is a finite type that represents a single code unit. A code unit is a fixed-size value. Code points are encoded as a sequence of code units. A

code point is a number that is the atomic building point of unicode text. Graphemes are visible characters composed of one or more code points.

Resolve defines a simple theory of characters. This is because most operations on characters are not very useful for verification. For example, there is no ordering of characters. This is because an ordering is best defined for graphemes. However, even if ordering is defined on code points, having an ordering for code units will most likely not result in an ordering that and end user would understand.

3.4.6 Tuples

Tuples are types that contain a sequence of values of fixed types. Each element in the tuple has an associated name. Tuples are most commonly used as math subtypes which add a constraint onto the values in the tuple.

In Listing 3.3, we see the definition of `ARRAY_MODEL`. This math type is defined as a subtype of a tuple that consists of three values: a lower bound, an upper bound, and a string. The string models the values in the array. The lower bound models the lowest valid index into the array. The upper bound models the highest valid index into the array. Both of these bounds are inclusive. In the subtype definition the **`exemplar`** introduces a name for the value used in the **`constraint`**. The notation `tuple.field` is used to access the fields of the tuple. In this instance, `a.s` accesses the string of items in the tuple `a`. Similarly, `a.lb` and `a.ub` refer to the values of the two fields for the bounds of the array.

```
1 math subtype ARRAY_MODEL is (lb: integer,
2                               ub: integer,
3                               s: string of Item)
4   exemplar a
5   constraint a.lb <= a.ub + 1 and
6              |a.s| = a.ub - a.lb + 1
```

Listing 3.3: ARRAY_MODELmath subtype of a tuple (Resolve)

Chapter 4: Resolve Toolchain

The Resolve Online IDE [17] is a web based development environment used to write Resolve programs. It includes a collection of base libraries with code for common types such as integers, queues, and more. A tree view browser visualizes the contracts, enhancements, and realizations from the base libraries and that are written by the user. The editor provides syntax highlighting and formatting as seen in Figure 4.1. In order to perform verification, multiple toolchains are supported. The toolchain can be selected using the dropdown seen in Figure 4.2. The main toolchains supported by the IDE are the Dafny prover and the SplitDecision prover.

4.0.1 Verification Conditions

When a verification attempt begins, Resolve Online will generate a list of verification conditions (VCs) that need to be proven. A verification condition consists of a goal that must be proven, a list of givens, and metadata about types and mathematical functions used in the formulas. The IDE can show the VCs that are sent to the prover as seen in Figure 4.3.

Reasoning Tables

Table 4.1 is a reasoning table for the `Concatenate` procedure from Listing 3.2. VCs are generated for each of the state indexes listed in the table. The path conditions

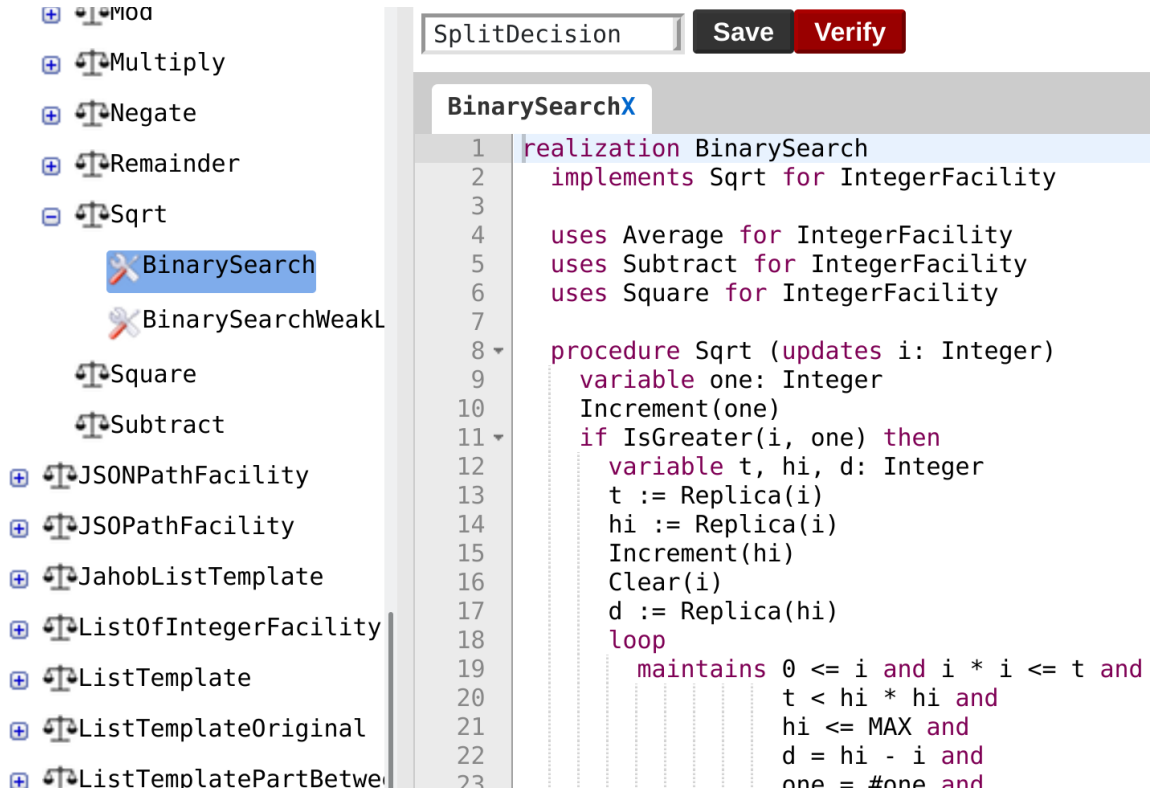

Figure 4.1: Resolve Online IDE editing the BinarySearch realization

list givens that are known because of control flow such as the condition from a loop or if then statement.

For state 0, the facts are empty because `Concatenate` has no **`requires`** clause. The first obligation, $p_0 \circ q_0 = p_0 \circ q_0$ comes from the loop invariant. Because the current state is 0, `p` and `q` are replaced with p_0 and q_0 respectively. `#p` and `#q` are replaced with p_0 and q_0 because the state 0 is the state right before the loop condition. The second obligation, $q_0 \neq \lambda \implies |q_0| > 0$, comes from the termination metric. This obligation is in the from of a implication that indicates that if the loop

State Index	Path Conditions	Facts	Obligations
1	**procedure** Concatenate (**updates** p: Queue, **clears** q: Queue)		
0			$p_0 \circ q_0 = p_0 \circ q_0$ $q_0 \neq \lambda \implies \lvert q_0 \rvert > 0$

3	**loop**
	  maintains p * q = #p * #q
	  decreases |q|
	while not IsEmpty (q) **do**		

1	$q_1 \neq \lambda$	$p_1 \circ q_1 = p_0 \circ q_0$ $\lvert q_1 \rvert \geq 0$	
7	**variable** x: Item		
2	$q_1 \neq \lambda$	$p_2 = p_1$ $q_2 = q_1$ is_initial(x_2)	$q_2 \neq \lambda$
8	Dequeue (q, x)		
3	$q_1 \neq \lambda$	$q_2 = \langle x_3 \rangle \circ q_3$ $p_3 = p_2$	
9	Enqueue (p, x)		
4	$q_1 \neq \lambda$	$p_4 = p_3 \circ \langle x_3 \rangle$ is_initial(x_4) $q_4 = q_3$	$p_4 \circ q_4 = p_0 \circ q_0$ $\lvert q_4 \rvert \geq 0$ $\lvert q_4 \rvert < \lvert q_1 \rvert$
10	**end loop**		
5		$q_5 = \lambda$ $p_5 \circ q_5 = p_0 \circ q_0$	$q_5 = \lambda$ $p_5 = p_0 \circ q_0$
11	**end** Concatenate		

Table 4.1: Reasoning Table for `Concatenate`procedure

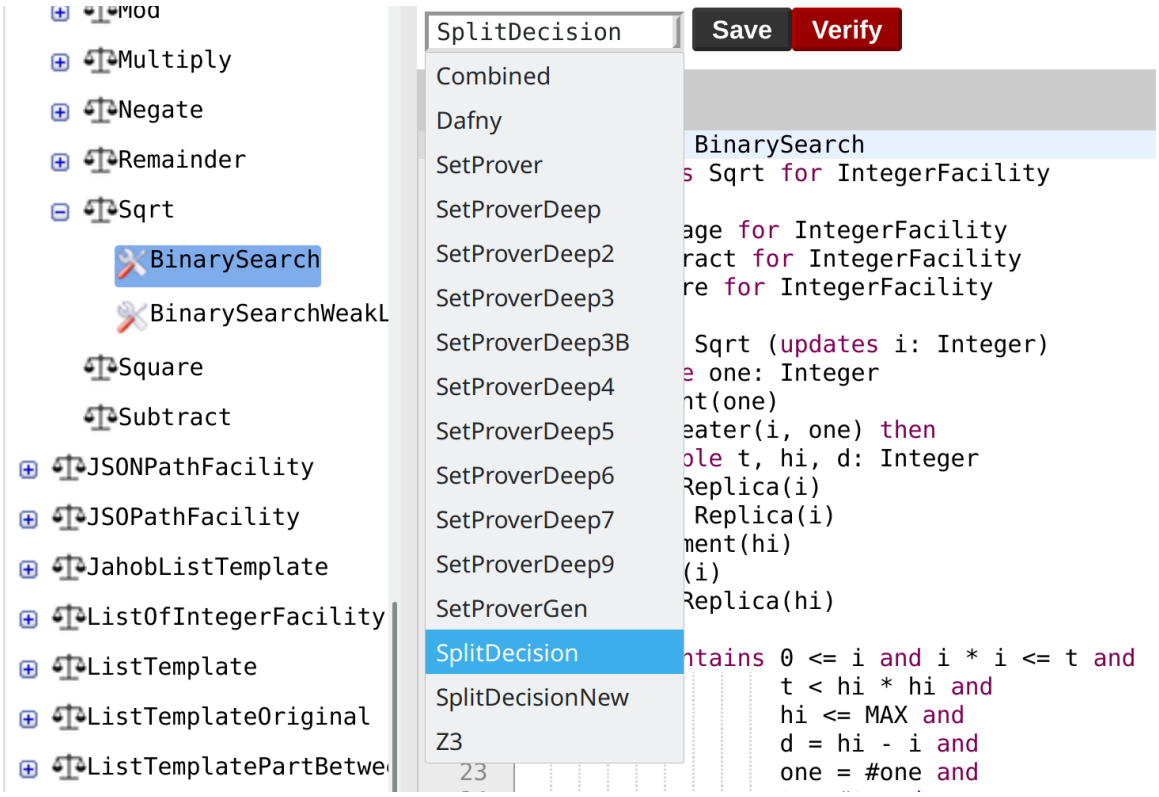

Figure 4.2: Resolve Online IDE toolchain selection

is entered, then the termination metric must have a lower bound of 0. This is needed because without a lower bound on the metric, the loop could run forever.

The obligation $q_1 \neq \lambda$ in state 2 comes from the **`requires`** clause of the `Dequeue` procedure. Additionally, the **`ensures`** clause of `Dequeue` results in the addition of the fact $q_2 = \langle x_3 \rangle \circ q_3$ in state 3.

In state 4, the obligations again come from the loop invariant and termination metric. The same technique is used for the loop invariant; the current state is used for variable references and state 0 is the state before the loop condition. However, the termination metric results in two VCs. One of these, $|q_4| \geq 0$, is to ensure that

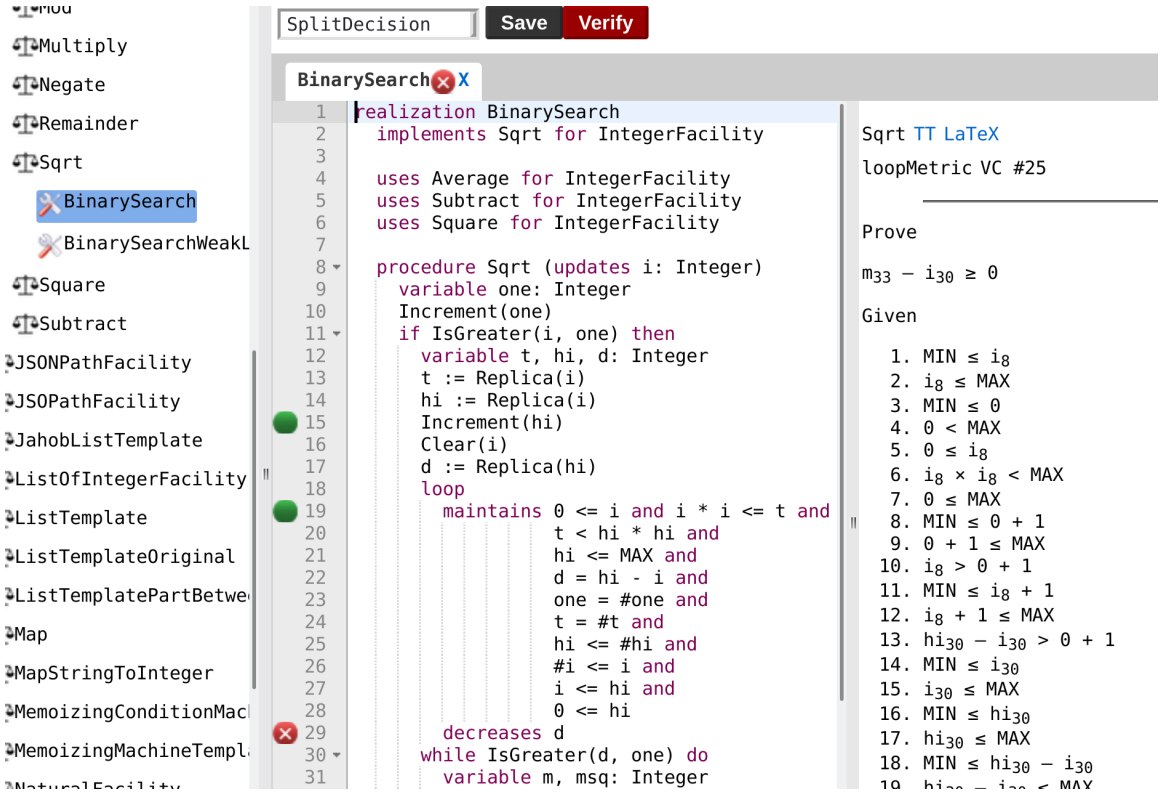

Figure 4.3: Resolve Online IDE VC Viewer

the metric has a lower bound. However, because we are already inside the loop, the implication is no longer used. Additionally, because the loop could exit, the metric is now allowed to be a zero. There is another obligation, $|q_4| < |q_1|$, to show that the metric decreases.

In state 5, the facts come from the loop condition becoming false and the loop invariant. The obligations are from the **`ensures`** clause, including the parameter modes. The obligation $q_5 = \lambda$ is because the parameter `q` has the mode **`clears`** and λ is the initial value of the `Queue` type.

If the Resolve code contains a **confirm** statement, the property that is being confirmed will result in an additional obligation. This obligation will then be added as a fact in the following state.

An if statement will have states in the if branch and the else branch if present. In these states, a path condition is added based on whether it is the if branch or the else branch. After the if statement, facts added in the body will be added as facts with an implication. The left side of the implication is either the condition of the if statement or the negation if the fact originated in the else body.

Before generating VCs from these obligations, the VC generator will make some simplifications. For example, because p_1, p_2, and p_3 all have the same value ($p_1 = p_2 \wedge p_2 = p_3$), these could be collapsed into a single variable p_1. Since `p` is not modified between state 1 and state 3, the reasoning table introduces facts that the value is the same in these three states. This allows these to be collapsed into a single variable.

After collapsing variables, some trivial obligations are eliminated. For example, if the obligation is of the form $a = a$, no VC will be generated. Also, if the obligation is also included verbatim as a given, no VC will be generated.

Generated VCs

Figure 4.4 contains a verification condition generated from the `Concatenate` procedure. This VC comes from state 4 and is derived from the loop invariant. The variables have already been collapsed. For example, the result uses q_3 even though the reasoning table uses q_4 because $q_3 = q_4$. The VC includes additional facts that are not needed to prove the result; only givens 2, 5, and 6 are needed as seen in the proof below.

Prove:

$$p_4 \circ q_3 = p_0 \circ q_0$$

Given:

1. $q_1 \neq \lambda$
2. $p_1 \circ q_1 = p_0 \circ q_0$
3. $|q_1| > 0$
4. is_initial(x_2)
5. $q_1 = \langle x_3 \rangle \circ q_3$
6. $p_4 = p_1 \circ \langle x_3 \rangle$
7. is_initial(x_4)

Figure 4.4: Verification Condition from `Concatenate`

Proof of VC from Figure 4.4. We can see from given 6 that

$$p_4 \circ q_3 = p_1 \circ \langle x_3 \rangle \circ q_3.$$

If we apply given 5, then

$$p_1 \circ \langle x_3 \rangle \circ q_3 = p_1 \circ q_1.$$

Putting this together with given 2, we see that

$$p_4 \circ q_3 = p_1 \circ q_1 = p_0 \circ q_0.$$

□

4.1 Prover Architecture

The Resolve Online IDE supports multiple toolchains. Each toolchain accepts VCs an These toolchains are configured using XML files. The XML files describe how to

configure the prover. Listing 4.1 contains an example of a toolchain configuration file. In this case, the `RemoveProver` class is used. This class uses a SOAP API to communicate with a service that proves VCs.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<toolchain name="Dafny">
  <inputFileExtensions><extension>rr</extension></inputFileExtensions>
  <component>
    <className>edu.osu.cse.resolve.ResolveToolchain.VCGenerator</className>
    <displayName>VCGenerator</displayName>
    <config>
      <config name="TEMP_DIRECTORY">
        <value>/MVEData/workarea</value>
      </config>
    </config>
    <onSuccess>
      <component>
        <className>edu.osu.cse.resolve.WSComponents.RemoteProver</className>
        <displayName>Dafny</displayName>
        <config>
          <config name="endpoint">
            <value>http://dafnyprover</value>
          </config>
        </config>
      </component>
    </onSuccess>
  </component>
  <observer>
    <className>edu.osu.cse.resolve.MVEWeb.server.WebObserver</className>
  </observer>
</toolchain>
```

Listing 4.1: Dafny Toolchain Configuration (XML)

In Figure 4.5, we see that when the user requests verification, the VC Generator will send generated VCs to the RemoteProver. The RemoteProver will communicate with the prover at the configured URL. At the remote URL, the ComponentHost will accept VC requests and dispatch them to the underlying prover.

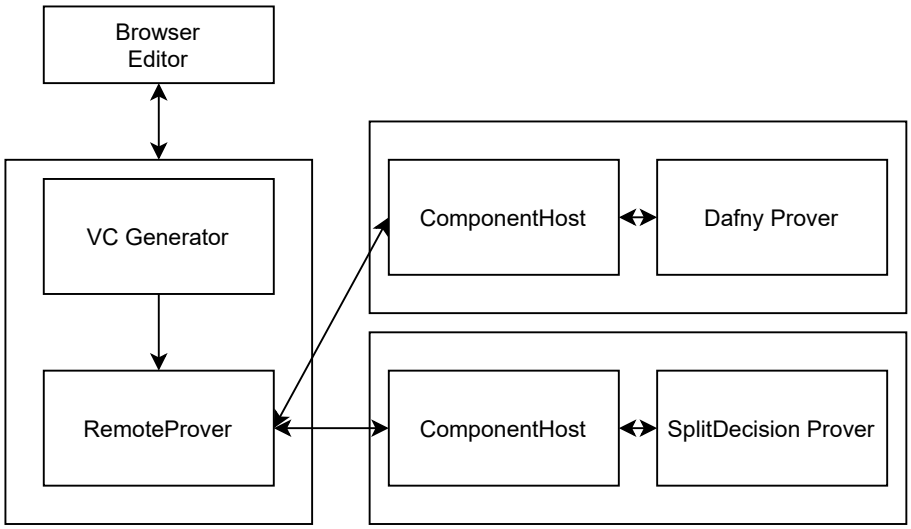

Figure 4.5: Prover Architecture

4.1.1 The Dafny Prover

The Dafny prover backend makes use of the Dafny language to prove VCs. Dafny [21] is a programming language that supports verification. It uses Boogie to interface with the prover, usually Z3, to perform verification. It supports object-oriented constructs such as classes and an ML-style module system. The compiler can generate code in C#, Go, and JavaScript.

The Dafny prover backend takes the VCs generated by Resolve Online and generates Dafny code. It works by generating a Dafny class. If needed, the class is parameterized by types to represent the type parameters of a contract. The versioned variables are turned into fields of the class. A single method is defined in order to perform the actual verification. In this method, assume statements are added for the givens and math functions. This is followed by a single assert statement for the goal of the VC.

There are a number of downsides of using Dafny for verification. First, the performance cost is large. The Dafny compiler is for an entire programming language and many of the features are not utilized, but still are part of the pipeline. Additionally, the Dafny compiler runs on .NET. Because of this, the Dafny compiler is run and the startup cost for .NET is incurred for every VC. Additionally, Dafny loads a large amount of information into Z3. Dafny would typically be able to reuse this, but since each VC is a separate invocation, this must be reloaded for every VC.

Another issue is the fragile nature of generating code. As the Dafny language evolves, sometimes code that was previously valid is no longer allowed. This makes it difficult to upgrade to newer versions of the compiler. Additionally, using Dafny results in additional layers of abstraction on top of the prover. Dafny itself has semantics that influence how the VC is translated into Z3. Boogie is another layer of abstraction between the Resolve VC and the prover.

4.1.2 The SplitDecision Prover

The SplitDecision [2] prover is a prover developed at The Ohio State University for verifying Resolve programs. SplitDecision uses an XML format to represent the input VCs. It is geared toward solving problems involving strings. When proving formulas involving disjunction, it splits into separate VCs.

While SplitDecision is able to prove VCs involving strings, it is more limited for other problems. Notably, SplitDecision has only limited support for problems involving quantifiers. It was noted that SplitDecision is better than Z3 at most problems and that Z3 was primarily only useful for integer problems. However, Z3 has since had over a decade of development and is now one of the most commonly

used SMT solvers. SplitDecision, on the other hand, has not had much development during that same time period.

Chapter 5: Implementing a Resolve Online prover using Z3

In order to address the limitations of the existing provers used by Resolve Online, we created a new prover backend that utilizes Z3 to prove VCs. Z3 configured with an rlimit of 1000000. The rlimit ensures that Z3 will terminate after a machine-independent number of operations, even if it hasn't proven or disproven the VC.

5.1 Translating VCs to Satisfiability

In order to perform proofs using Z3, the VCs must be converted into satisfiability problems. A VC with givens $g_1(\bar{x}), ..., g_n(\bar{x})$ and goal $p(\bar{x})$ can be seen as a single formula with free variables $\bar{x} = x_1, ..., x_m$. (5.1)

$$(g_1(x_1, ..., x_m) \wedge ... \wedge g_n(x_1, ..., x_m)) \implies p(x_1, ..., x_m) \tag{5.1}$$

In this case, the free variables could be substituted for an arbitrary value of the correct type. Effectively, this means that the formula is equivalent to a universally quantified formula with no free variables. (5.2)

$$\forall \bar{x} :: (g_1(\bar{x}) \wedge ... \wedge g_n(\bar{x})) \implies p(\bar{x}) \tag{5.2}$$

However, satisfiability problems attempt to find the existence of an assignment of variables. This means that to translate the formula (5.2) into a satisfiability problem, we need to use an existentially quantified formula. (5.3)

$$\exists y_1, ..., y_k :: f(y_1, ..., y_k) \tag{5.3}$$

In order to convert to an existential quantifier, we negate the formula. (5.4)

$$\neg\exists \bar{x} :: \neg((g_1(\bar{x}) \wedge ... \wedge g_n(\bar{x})) \implies p(\bar{x})) \tag{5.4}$$

Using the law of material implication and DeMorgan's Laws, we can write this without the implication. (5.5)

$$\neg\exists \bar{x} :: (g_1(\bar{x}) \wedge ... \wedge g_n(\bar{x}) \wedge \neg p(\bar{x})) \tag{5.5}$$

Because the existential quantifier is negated, the formula is true if there is no satisfying assignment of the existentially quantified variables. Thus, we can give $(g_1 \wedge ... \wedge g_n \wedge \neg p)$ as input to the SMT solver. If the formula is UNSAT, then the VC is true.

The VC from Figure 4.4 can be written as a formula. For brevity, the unused givens are excluded. (5.6)

$$\begin{gathered}(p_1 \circ q_1 = p_0 \circ q_0 \wedge \\ q_1 = \langle x_3 \rangle \circ q_3 \wedge \\ p_4 = p_1 \circ \langle x_3 \rangle) \implies \\ p_4 \circ q_3 = p_0 \circ q_0\end{gathered} \tag{5.6}$$

Then we make the implicit universal quantifier explicit. (5.7)

$$\forall p_0, q_0, p_1, q_1, x_3, q_3, p_4 :: (p_1 \circ q_1 = p_0 \circ q_0 \wedge \\ q_1 = \langle x_3 \rangle \circ q_3 \wedge \\ p_4 = p_1 \circ \langle x_3 \rangle) \implies \\ p_4 \circ q_3 = p_0 \circ q_0 \tag{5.7}$$

Next, we convert the universal quantifier into a negated existential quantifier. (5.8)

$$\neg\exists p_0, q_0, p_1, q_1, x_3, q_3, p_4 :: \neg((p_1 \circ q_1 = p_0 \circ q_0 \wedge \\ q_1 = \langle x_3 \rangle \circ q_3 \wedge \\ p_4 = p_1 \circ \langle x_3 \rangle) \implies \\ p_4 \circ q_3 = p_0 \circ q_0) \tag{5.8}$$

Then we eliminate the implication using the law of material implication. (5.9)

$$\neg\exists p_0, q_0, p_1, q_1, x_3, q_3, p_4 :: \neg(\neg(p_1 \circ q_1 = p_0 \circ q_0 \wedge \\ q_1 = \langle x_3 \rangle \circ q_3 \wedge \\ p_4 = p_1 \circ \langle x_3 \rangle) \vee \\ p_4 \circ q_3 = p_0 \circ q_0) \tag{5.9}$$

Then, using DeMorgan's Laws we simplify the body of the quantifier. (5.10)

$$\neg\exists p_0, q_0, p_1, q_1, x_3, q_3, p_4 :: p_1 \circ q_1 = p_0 \circ q_0 \wedge \\ q_1 = \langle x_3 \rangle \circ q_3 \wedge \\ p_4 = p_1 \circ \langle x_3 \rangle \wedge \\ \neg(\quad p_4 \circ q_3 = p_0 \circ q_0) \tag{5.10}$$

Now that we have a formula that is in the form of a satisfiability problem, we can remove the quantifier and use (5.11) as the input to the SMT solver.

$$\begin{aligned} & p_1 \circ q_1 = p_0 \circ q_0 \wedge \\ & q_1 = \langle x_3 \rangle \circ q_3 \wedge \\ & p_4 = p_1 \circ \langle x_3 \rangle \wedge \\ & \neg(\quad p_4 \circ q_3 = p_0 \circ q_0) \end{aligned} \tag{5.11}$$

5.2 Mapping Resolve Types to Z3 Sorts

In order to translate Resolve VCs to Z3, there needs to be a mapping from Resolve math types to Z3 sorts. In the case of `integer` and `boolean`, the types are directly mapped to Z3's `Int` and `Bool` sorts. The `string` **`of`** `T` type can be mapped to `(Seq T)`.

The `character` type is mapped to `(_ BitVec 8)`. This means that characters are 8-bit values. This was chosen out of a desire to use UTF-8 encoded strings of characters.

The **`finite set of`** `T` type is mapped to an uninterpreted sort. This is used instead of Z3's `(Set T)` sort because `(Set T)` allows for infinite sets. Instead, a function is defined to map the sets onto Z3 sets.

5.3 Operation Mapping

Many common operations have a direct mapping from Resolve to an equivalent operation in Z3. For example, logical operators such as conjunction, disjunciton, and more all map directly. Similarly, many integer operations such as addition and

multiplication can simply use Z3 addition and multiplication. However, as discussed below, there are some where more work is required.

5.3.1 Absolute Value

Z3 does not define a built-in absolute value function. Because of this, a custom function was used. This function was defined using a conditional. When the value is already non-negative, the original value is used. In the case that the value is negative, the result is the negation of the original value. Listing 5.1 contains a definition of an absolute value function and proves that `abs` always returns a non-negative value.

```
1  (declare-const a Int)
2
3  (declare-fun abs (Int) Int)
4  (assert (forall ((x Int))
5      (= (abs x) (ite (>= x 0) x (- x)))
6  ))
7
8  (assert (not (>= (abs a) 0)))
9
10 (check-sat)
```

Listing 5.1: Proof of Absolute Value properties (SMT-LIB)

5.3.2 Division and Modulo

Unlike other integer operations such as addition, subtraction, and multiplication, integer division is not the same as division on real numbers. This is because using division of real numbers, dividing two integers does not always result in an integer. For example $\frac{1}{2} \notin \mathbb{Z}$ even though $1, 2 \in \mathbb{Z}$. Because of this, integer division is defined to include rounding.

There are two common ways of defining how rounding works: rounding toward zero and rounding toward $-\infty$. Each option has tradeoffs. In both cases the modulo operation is defined to match division such that $a = (a/b) * b + (a \bmod b)$.

When rounding toward $-\infty$, the remainder r is in the range $0 \leq r < |b|$. However, this definition has the disadvantage that $-a/b \neq -(a/b)$. For example, if $a = 1$ and $b = 2$, then $-1/2 = -1$ but $-(1/2) = 0$.

When rounding toward zero, the remainder r is in the range $0 \leq r < b$ when $b > 0$ and $b \leq r \leq 0$ when $b < 0$.

We can see from Listing 5.2 that Z3 uses rounding toward $-\infty$.

```
1  (declare-const a Int)
2  (declare-const b Int)
3  (declare-const q Int)
4  (declare-const r Int)
5
6  (assert (not (= b 0)))
7  (assert (= r (mod a b)))
8  (assert (= q (div a b)))
9  (assert (not (and
10     (= a (+ (* b q) r)) ; a = bq + r
11     (>= r 0)
12     (< r (ite (>= b 0) b (- b))) ; r < |b|
13 )))
14
15 (check-sat)
```

Listing 5.2: Proof of mod properties (SMT-LIB)

5.3.3 Substring

The Resolve `substring` operation does not have a direct correspondence in Z3. The closest equivalent is `seq.extract` which gets a subsequence of a sequence. However, there are two differences in the operations.

The first difference is that `substring` uses a start index and an end index; `seq.extract` uses a length. This can be handled by calculating the length by subtracting the start index from the end index.

The second difference is how out of range indexes are handled. In Resolve, if values of the indexes are not within the expected range $0 \leq \text{start} \leq \text{end} \leq |s|$, the substring is the empty string. In Z3, if start < 0, start $\geq |s|$, or length < 0, then the subsequence is also empty. However, if start + length $> |s|$, the result is the subsequence starting at start until the end of the sequence. In order to turn this case into the empty sequence, we used a conditional. If end $\leq |s|$, then end $-$ start is used as the length. Otherwise, 0 is used as the length. Listing 5.3 contains a definition of a substring function along with proofs that out of range indexes result in the empty string.

5.3.4 Finite Set Operations

Because finite sets of T are mapped to an uninterpreted sort S, we must define custom functions for all set operations. However, in many cases, these operations can leverage underlying Z3 set operations.

1. z3_set : $S \rightarrow \text{Set}(T)$
2. empty : S
3. singleton : $T \rightarrow S$
4. cardinality : $S \rightarrow \text{Int}$
5. elements : $\text{Seq}(T) \rightarrow S$

6. membership : $T \times S \rightarrow$ Bool

7. union : $S \times S \rightarrow S$

8. intersection : $S \times S \rightarrow S$

9. difference : $S \times S \rightarrow S$

The z3_set function defines a mapping from S to Z3 sets. The remaining operations implement Resolve operations. empty is the empty set. singleton is a set with a single element. cardinality gets the number of elements in the set. elements gets the set of values contained in a string. membership tests whether a value is in a set. union, intersection, and difference perform the corresponding operation on the sets.

Axioms are used to define these operations. Where possible, the Z3 set operations are used in the definitions. In these axioms, native Z3 operations are represented using mathematical notation.

Axiom 1 is important because it allows the reasoning Z3 performs on the projected sets to be translated back to finite sets. Axioms 2 through 7 define the operations other than cardinality. Axiom 3 is written as $\emptyset \cup \{x\}$ because Z3 does not have a direct operation for creating a set of one element. Instead, it is implemented by adding x to the empty set.

1. $\forall a, b : \text{z3\_set}(a) = \text{z3\_set}(b) : a = b$

2. $\text{z3\_set}(\text{empty}) = \emptyset$

3. $\forall x :: \text{z3\_set}(\text{singleton}(x)) = \emptyset \cup \{x\}$

4. $\forall \text{seq}, n : 0 \leq n < |seq| : \text{membership}(\text{nth}(\text{seq}, n), \text{elements}(\text{seq}))$

5. $\forall a, b :: \text{z3\_set}(\text{union}(a, b)) = \text{z3\_set}(a) \cup \text{z3\_set}(b)$

6. $\forall a, b :: \text{z3\_set}(\text{intersection}(a, b)) = \text{z3\_set}(a) \cap \text{z3\_set}(b)$

7. $\forall a, b :: \text{z3\_set}(\text{difference}(a, b)) = \text{z3\_set}(a) \setminus \text{z3\_set}(b)$

Axioms 8 and 9 define the elements function in terms of membership. Axiom 8 states that the elements in the string are members of the set. Axiom 9 states that if an element is a member of the set, then it must have been in the string.

8. $\forall \text{seq}, x : \text{membership}(x, \text{elements}(\text{seq})) : (\exists n : 0 \leq n < |\text{seq}| : x = \text{nth}(\text{seq}, n))$

9. $\forall a, x :: \text{membership}(x, a) = x \in \text{z3\_set}(a)$

The remaining axioms, 10 through 22, define cardinality and how it relates to the other operations. Axiom 10 states that all cardinalities are non-negative. Axiom 12 states that all sets of cardinality 0 are the empty set. Axiom 22 handles the case where a single element is being removed from a set. This is included because it is common for loops to remove elements and define the termination metric in terms of cardinality.

Axiom 13 is redundant because it can be proven from other axioms. However, it is useful because it is known to be true and the proof is complex.

Proof of Axiom 13. We can see from instantiating 22 that

$$\text{cardinality}(\text{difference}(\text{singleton}(x), \text{singleton}(x))) = \text{cardanility}(\text{singleton}(x)) - 1.$$

Furthermore, from 2, an instantiation of 7, and mathematical reasoning on sets,

$$\begin{aligned}
&\text{z3_set}(\text{difference}(\text{singleton}(x), \text{singleton}(x))) \\
&\quad = \text{z3_set}(\text{singleton}(x)) \setminus \text{z3_set}(\text{singleton}(x)) \\
&\quad = \emptyset \\
&\quad = \text{z3_set}(\text{empty}).
\end{aligned}$$

Thus, by 1, $\text{difference}(\text{singleton}(x), \text{singleton}(x))) = \text{empty}$. Substituting this and using 11, $0 = \text{cardanility}(\text{singleton}(x)) - 1$. Therefore, $\text{cardanility}(\text{singleton}(x)) = 1$. □

10. $\forall a : \text{cardinality}(a) \geq 0$
11. $\text{cardinality}(\text{empty}) = 0$
12. $\forall a : \text{cardinality}(a) = 0 : a = \text{empty}$
13. $\forall x :: \text{cardinality}(\text{singleton}(x)) = 1$
14. $\forall \text{seq} :: \text{cardinality}(\text{elements}(\text{seq})) \leq |\text{seq}|$
15. $\forall a, b :: \text{cardinality}(\text{union}(a, b)) \geq \text{cardinality}(a)$
16. $\forall a, b :: \text{cardinality}(\text{union}(a, b)) \geq \text{cardinality}(b)$
17. $\forall a, b :: \text{cardinality}(\text{intersection}(a, b)) \leq \text{cardinality}(a)$
18. $\forall a, b :: \text{cardinality}(\text{intersection}(a, b)) \leq \text{cardinality}(b)$
19. $\forall a, b :: \text{cardinality}(\text{difference}(a, b)) \leq \text{cardinality}(a)$
20. $\forall a, b :: \text{cardinality}(\text{difference}(a, b)) \geq \text{cardinality}(a) - \text{cardinality}(b)$

21. $\forall a, b :: \text{cardinality}(\text{union}(a, b)) =$

$$\text{cardinality}(a) + \text{cardinality}(b) - \text{cardinality}(\text{intersection}(a, b))$$

22. $\forall x, a : \text{membership}(x, a) :$

$$\text{cardinality}(\text{difference}(a, \text{singleton}(x))) = \text{cardinality}(a) - 1$$

```
(declare-sort T)

(declare-const s (Seq T))
(declare-const a Int)
(declare-const b Int)

(declare-fun substring ((Seq T) Int Int) (Seq T))
(assert (forall ((str (Seq T)) (start Int) (end Int))
    (= (substring str start end)
        (seq.extract str start
            (ite (<= end (seq.len str))
                (- end start)
                0
        ))
    )
))


(push)
(assert (< a 0))
(assert (not
    (= (as seq.empty (Seq T)) (substring s a b))
))
(check-sat)
(pop)

(push)
(assert (< b a))
(assert (not
    (= (as seq.empty (Seq T)) (substring s a b))
))
(check-sat)
(pop)

(push)
(assert (> b (seq.len s)))
(assert (not
    (= (as seq.empty (Seq T)) (substring s a b))
))
(check-sat)
(pop)
```

Listing 5.3: Proof of out of range properties of Substring (SMT-LIB)

Chapter 6: Results

The Z3 prover backend was evaluated in relation to the other provers on two primary measures: the number of VCs proved and the execution time. These data were captured on computer running Debian Linux with an 8 core AMD Ryzen 7 3700X and 64GB of RAM. The results were captured using a Ruby script that reads the log output of the provers. It counts the number of VCs and tracks the start and end time of the entire set of VCs for the realization being verified. The timing numbers are based on an average of 5 runs.

The realizations that were selected as test cases were chosen by the following criteria. Except for SortableSimple/QuickSortRealization from ArrayAsStringTemplate, all realizations were chosen from the default Resolve Online repository of Resolve code. SortableSimple/QuickSortRealization was written as part of a case study on verifying a QuickSort variation. First, a set of facilities that define a variety of different, commonly used data types was selected. Next, realizations that intentionally contain bugs in the implementation were excluded. These realizations are used to check that the provers are not proving false VCs. Additionally, some realizations that implement trivial operations were excluded.

Table 6.1 contains the number of VCs proved by the Resolve online provers in a variety of code examples. The number of VCs is listed in bold when the number of

VCs proved is the largest among the provers. In the case of a tie, all tied numbers are bolded. The result column includes a checkmark if all VCs were proven by the prover. As seen in this table, in all realizations Z3 is able to prove at least as many VCs as Dafny. There are only two realizations where SplitDecision proves more VCs than Z3. There are 20 realizations where Z3 proved more VCs than Dafny and 11 when compared to SplitDecision. Of these, there are 8 realizations where Z3 proved more VCs than both Dafny and SplitDecision, three of which are fully proven by Z3.

The first case where Z3 proved all VCs, but the other provers did not is Mod-/RemainderRealization from the IntegerFacility. Under the Dafny prover, the VCs that failed involed absolute values. In fact, these VCs failed because the Dafny code generation did not correctly detect that the absolute value function was used. This resulted in invalid Dafny code being generated. In the case of SplitDecition, the two goals that were not proved are found in Equations (6.1) and (6.2). In both cases, SplitDecision appears to have difficulty with proofs involving mod and the absolute value.

$$i_8 \bmod j_8 - |j_8| + j_8 = i_8 \bmod j_8 \tag{6.1}$$

$$i_8 \bmod j_8 - |j_8| = i_8 \bmod j_8 \tag{6.2}$$

The second case where Z3 proved all VCs, but the other provers did not is Sqrt/BinarySearch from the IntegerFacility. Dafny was unable to prove 2 VCs. These goals of these VCs involve multiplication and inequality: $m_{33} \times m_{33} \leq m_{33} \times i_8$ and $m_{33} \times i_8 \leq i_8 \times i_8$. SplitDecision was unable to prove 10 VCs. All of these VCs had goals consisting of inequalities of integral expressions involving variables and subtractions.

		Dafny		**SplitDecision**		**Z3**	
	#VCs	*Proved*	*Result*	*Proved*	*Result*	*Proved*	*Result*
ArrayAsStringTemplate							
Contains/LinearIterative	23	15	-	**23**	✓	17	-
SortableSimple/QuickSortRealization	1419	1410	-	1264	-	**1413**	-
IntegerFacility							
Add/Iterative	17	**17**	✓	**17**	✓	**17**	✓
Add/IterativeOptimized	23	5	-	**23**	✓	**23**	✓
Add/Recursive	18	15	-	**18**	✓	**18**	✓
Mod/RemainderRealization	40	27	-	38	-	**40**	✓
Multiply/Iterative	132	129	-	**132**	✓	**132**	✓
Multiply/IterativeCases	50	**50**	✓	**50**	✓	**50**	✓
Multiply/IterativeNonObvious	24	**24**	✓	**24**	✓	**24**	✓
Multiply/IterativeOneLoop	90	88	-	**90**	✓	**90**	✓
Multiply/IterativeOneLoopOptimized	103	99	-	**103**	✓	**103**	✓
Sqrt/BinarySearch	31	29	-	21	-	**31**	✓
QueueTemplate							
Concatenate/Iterative	5	**5**	✓	**5**	✓	**5**	✓
Concatenate/Recursive	4	**4**	✓	**4**	✓	**4**	✓
Sort/MergeSort	62	39	-	38	-	**43**	-
Sort/QuickSort	23	9	-	7	-	**12**	-
Sort/SelectionSort	22	9	-	9	-	**14**	-
SetTemplate							
Apply/Iterative	16	**15**	-	13	-	**15**	-
Intersect/Iterative	14	13	-	**14**	✓	**14**	✓
Split/Iterative	23	18	-	19	-	**20**	-
Subtract/Iterative	14	**14**	✓	13	-	**14**	✓
Unite/Iterative1	8	7	-	**8**	✓	**8**	✓
Unite/Iterative2	16	14	-	**16**	✓	**16**	✓
UniteAndIntersect/Iterative1	15	14	-	**15**	✓	**15**	✓
UniteAndIntersect/Iterative2	30	27	-	**30**	✓	29	-
StackTemplate							
Reverse/Iterative	5	3	-	**5**	✓	**5**	✓
TextFacility							
SwapSubstring/Recursive1	17	11	-	9	-	**17**	✓
UnboundedIntegerFacility							
Add/Iterative	11	**11**	✓	**11**	✓	**11**	✓
Sqrt/BinarySearch	23	**23**	✓	13	-	**23**	✓
Sqrt/Iterative	9	**9**	✓	**9**	✓	**9**	✓

Table 6.1: Comparison of VCs proved by Resolve Online Toolchains

The third case where Z3 proved all VCs, but the other provers did not is SwapSubstring/Recursive1 from the TextFacility. The VCs that Dafny was unable to prove were the VCs that referenced the math definitions from the SwapSubstring enhancement. These definitions also involve existential quantifiers, making them more difficult to prove. SplitDecision had difficulty with two additional VCs involving integer inequalities.

The first case where SplitDecision proved more VCs than Z3 is the LinearIterative realization of the Contains enhancement of ArrayAsStringTemplate. It makes sense that SplitDecision would do well in this test case as it was designed to handle string problems. The VCs that are failing all have goals or proofs involving givens of the form $x \in \text{elements}(s)$. Unfortunately, this reasoning appears to be difficult for Z3.

The second case where SplitDecision proved more VCs is the Iterative2 realization of the UniteAndIntersect enhancement of SetTemplate as seen in Listing 6.2. In this case it is due to a single VC that fails under Z3. This realization is very similar to Iterative1 from Listing 6.1, except that it swaps so that `s`, the first parameter, is larger. The result of this is that the VC is split between the two branches of the if statement. In the branch without the swap, the VC, $s_{19} \cap t_{19} \setminus \{\text{tmp}_{21}\} = t_4 \cap s_4 \cup \emptyset$, is proved. With the swap, the VC, $s_{19} \cap t_{19} \setminus \{\text{tmp}_{21}\} = t_4 \cap s_4 \cup \emptyset$, is not proved. The givens are equivalent except that t_4 and s_4 are swapped.

The data from Table 6.1 are summarized in Table 6.2. The Average VC Percentages shown are the average of the percentage of VCs proved for each realization. This is done to prevent realizations with larger numbers of VCs from dominating the results. For both Average VC Percentages and Percent Fully Proved, Z3 performed the best, followed by SplitDecition, and then Dafny.

```
variable tmp: Set
loop
    maintains s union t = #s union #t and
             (s intersection t) union tmp =
             (#s intersection #t) union #tmp and
             t intersection tmp = {}
    decreases |t|
while not IsEmpty(t) do
    ...
end loop
t :=: tmp
```

Listing 6.1: `UniteAndIntersect`procedure from `Iterative1`(Resolve)

```
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 = {}
    decreases |t|
while not IsEmpty(t) do
    ...
end loop
t :=: tmp
```

Listing 6.2: `UniteAndIntersect`procedure from `Iterative2`(Resolve)

Table 6.3 contains a selection of realizations that were selected for execution time analysis. Realizations were selected was based on the number of VCs, preferring a larger number of VCs. This results in longer overall verification times.

In most cases, Dafny takes substantially longer than SplitDecision and Z3. In fact, the time that Dafny takes is close to linear to the number of VCs. This is due to the large initial startup time of the Dafny compiler. Dafny runs on .NET and needs to be Just-In-Time compiled before it can be executed. Additionally, the theories

	Dafny	**SplitDecision**	**Z3**
Average % of VCs Proven	83%	88%	**95**%
% of Realizations Fully Proved	30%	63%	**73**%

Table 6.2: Summary of VCs proved by Resolve Online Toolchains

are loaded into Z3. These could be reused in a normal execution of the compiler. However, since each VC is a separate execution, this startup time dominates the execution time of the Dafny prover.

SplitDecision and Z3 are both close in the execution times. SplitDecision is ahead in 3 cases, while Z3 is ahead in 5. In these cases, the verification time is more dependent on the nature of the VCs than just the number of VCs.

One major outlier is SortableSimple/QuickSortRealization. In this case, the execution time of SplitDecision at around 3 hours is around five times that of Dafny. Z3, in contrast, is under a minute. One likely cause of this massive difference is the substantial use of quantifiers in this realization.

	Dafny	SplitDecision	Z3
ArrayAsStringTemplate			
Contains/LinearIterative	00:37.78 ± 00:00.24	00:03.36 ± 00:00.09	00:04.03 ± 00:00.02
SortableSimple/QuickSortRealization	35:17.61 ± 00:24.95	03:01:52.94 ± 06:52.97	00:40.88 ± 00:00.87
IntegerFacility			
Mod/RemainderRealization	00:33.95 ± 00:00.12	00:01.06 ± 00:00.03	00:00.60 ± 00:00.02
Multiply/Iterative	02:27.58 ± 00:00.41	00:03.94 ± 00:00.11	00:02.08 ± 00:00.02
QueueTemplate			
Sort/MergeSort	01:33.97 ± 00:00.18	00:01.98 ± 00:00.09	00:08.99 ± 00:00.09
SetTemplate			
UniteAndIntersect/Iterative2	00:39.19 ± 00:00.14	00:00.66 ± 00:00.08	00:01.47 ± 00:00.02
TextFacility			
SwapSubstring/Recursive1	00:24.12 ± 00:00.10	00:00.67 ± 00:00.03	00:00.44 ± 00:00.08
UnboundedIntegerFacility			
Sqrt/BinarySearch	00:24.99 ± 00:00.04	00:03.62 ± 00:00.01	00:00.45 ± 00:00.01

Table 6.3: Execution times of Resolve Online Toolchains

Chapter 7: Contributions and Future Work

7.1 Future Work

7.1.1 Set Cardanility

The theory of finite sets we defined utilizes Z3's underlying theory of infinite sets. A projection function is defined from the custom finite set sort to Z3's set sort. However, some operations required by Resolve are not directly supported in Z3's theory. As a result, these unsupported operations are defined using assertions.

One notable operation that is not supported by Z3 is the cardinality operation. In fact, SMT-LIB has an operation `set-has-size` which is a predicate that is true when a set has a specific size. There is also a function in the C API, `Z3_mk_set_has_size`, that creates expressions using this predicate. However, support for this predicate was removed from Z3 and the API function now throws an exception. [10] This predicate was removed due to soundness bugs that were discovered that occur when this predicate is used. [11] It is unknown whether `set-has-size` will be supported in the future. If `set-has-size` is fixed in Z3, it would be beneficial to update the theory of sets to utilize this function.

7.1.2 Type Inference for Empty Literals

The VC Generator is expected to annotate expressions with their types. However, in the case of some empty string and set literals, the element type cannot be known without examining the context in which it is used. This is handled in the Z3 prover by recording the first known string and set types. This type is used as the type of the respective empty literal if the element type is unspecified.

7.1.3 Leveraging Unsat Core

Currently, VC Generation is performed before sending any VCs to the provers. One issue is that multiple branches can result in exponential explosion in the number of VCs. Consider the code in Figure 7.1. The confirm statement on line 10 causes a VC to be generated to prove P. However, in order to take the information from the if statement with condition A on line 1, the VC must be split. This means that there will be one VC with A as a given and another with $\neg A$. This is in addition to any givens introduced in the corresponding branch. Repeating this for the next two if statements, results in two more splits. If this pattern continues, then there will be 2^n times as many VCs as there were originally.

```
1  if A then
2      ...
3  end if
4  if B then
5      ...
6  end if
7  if C then
8      ...
9  end if
10 confirm P
```

Listing 7.1: Exponential explosion of VCs

One thing that could help to reduce this explosion is to leverage the Unsat Core generated by Z3. When Z3 generates an UNSAT result, it can be configured to generate an Unsat Core which is a subset of the assertions that is sufficient to prove the result. This could be used to quickly detect whether the other VCs that were split from the proven VC can be shown using Unsat Core, eliminating the need to send them to the prover.

7.2 Contributions

We have seen in Chapter 4 how VCs are generated from Resolve code. We saw how a tracing table represents the way VCs are generated and how givens are introduced. We also saw the existing provers and some of their limitations.

In Chapter 5 we saw how an SMT solver such as Z3 can be used to prove VCs. We saw how VCs are reduced to a satisfiability problem. We also saw how the types are mapped into Z3 sorts and how operations are defined in Z3. Some of the operations without built-in support include the absolute value, substring, and finite set operations.

In Chapter 6 we saw that Z3 compared favorably to both the Dafny and Split-Decision prover backends. Z3 fully proves more realizations from the test cases than either Dafny or SplitDecision. Furthermore, Z3 proves at least as many VCs as the Dafny prover in all realizations. When compared to SplitDecision, Z3 proves at least as many VCs in all but 2 realizations. In terms of the execution time of the provers, Z3 was consistently faster than the Dafny prover due to high startup overhead in the Dafny prover. SplitDecision and Z3 were similar in execution time, with either one taking the lead depending on the realization being verified. However, in one case that

makes extensive use of quantifiers, SplitDecision ran for over 3 hours, while Z3 was under a minute. From this, we see that Z3 is an improvement over existing prover backends in what it is able to prove, while having similar or better verification time.

Appendix A: Execution Time Data

	Dafny	**SplitDecision**	**Z3**
ArrayAsStringTemplate			
Contains/LinearIterative	00:38.21	00:03.51	00:04.02
	00:37.72	00:03.31	00:04.02
	00:37.69	00:03.32	00:04.04
	00:37.61	00:03.31	00:03.99
	00:37.68	00:03.34	00:04.06
SortableSimple/QuickSortRealization	35:55.95	03:03:13.61	00:42.35
	35:03.07	02:50:11.13	00:40.81
	35:29.82	03:02:57.04	00:40.77
	35:01.97	03:08:20.75	00:40.16
	34:57.24	03:04:42.16	00:40.30
IntegerFacility			
Mod/RemainderRealization	00:34.07	00:01.08	00:00.59
	00:33.89	00:01.08	00:00.62
	00:33.79	00:01.05	00:00.61
	00:34.08	00:01.06	00:00.61
	00:33.89	00:01.02	00:00.58
Multiply/Iterative	02:27.04	00:04.11	00:02.08
	02:27.91	00:03.96	00:02.05
	02:27.85	00:03.96	00:02.08
	02:27.24	00:03.81	00:02.09
	02:27.87	00:03.86	00:02.08

Table A.1: Raw Execution Times (Part 1)

	Dafny	**SplitDecision**	**Z3**
QueueTemplate			
Sort/MergeSort	01:33.85	00:02.13	00:08.95
	01:33.90	00:02.02	00:09.12
	01:34.29	00:01.91	00:08.99
	01:33.87	00:01.92	00:09.05
	01:33.94	00:01.95	00:08.88
SetTemplate			
UniteAndIntersect/Iterative2	00:39.23	00:00.74	00:01.49
	00:39.05	00:00.69	00:01.47
	00:39.04	00:00.73	00:01.47
	00:39.37	00:00.58	00:01.45
	00:39.26	00:00.58	00:01.49
TextFacility			
SwapSubstring/Recursive1	00:24.22	00:00.66	00:00.58
	00:24.02	00:00.65	00:00.41
	00:24.11	00:00.63	00:00.41
	00:24.21	00:00.71	00:00.42
	00:24.02	00:00.69	00:00.40
UnboundedIntegerFacility			
Sqrt/BinarySearch	00:25.05	00:03.63	00:00.45
	00:24.98	00:03.62	00:00.44
	00:25.01	00:03.64	00:00.45
	00:24.95	00:03.63	00:00.45
	00:24.98	00:03.60	00:00.45

Table A.2: Raw Execution Times (Part 2)

Appendix B: Repository of Resolve Code

Listing B.1: ArrayAsStringTemplate/ArrayAsStringTemplate.rc

```
1  contract ArrayAsStringTemplate (type Item)
2    uses UnboundedIntegerFacility
3
4    math subtype ARRAY_MODEL is (lb: integer,
5                                 ub: integer,
6                                 s: string of Item)
7      exemplar a
8      constraint a.lb <= a.ub + 1 and
9                  |a.s| = a.ub - a.lb + 1
10
11   type Array is modeled by ARRAY_MODEL
12     exemplar a
13     initialization ensures a = (1, 0, empty_string)
14
15   procedure SetBounds(updates a: Array,
16                       restores lower: Integer,
17                       restores upper: Integer)
18     requires
19       lower <= upper + 1
20     ensures
21       a.lb = lower and a.ub = upper
22
23   procedure SwapItem(updates a: Array,
24                      restores i: Integer,
25                      updates x: Item)
26     requires
27       a.lb <= i and i <= a.ub
28     ensures
29       a.lb = #a.lb and
30       a.ub = #a.ub and
31       substring(a.s, 0, i - a.lb) = substring(#a.s, 0, i - #a.lb) and
32       substring(a.s, i + 1 - a.lb, |a.s|) = substring(#a.s, i + 1 - #a.lb,
             |#a.s|) and
33       substring(a.s, i - a.lb, i + 1 - a.lb) = <#x> and
34       substring(#a.s, i - #a.lb, i + 1 - #a.lb) = <x>
35
36   function LowerBound (restores a: Array) : Integer
37     ensures
38       LowerBound = a.lb
39
40   function UpperBound (restores a: Array) : Integer
41     ensures
42       UpperBound = a.ub
43
```

```
44 end ArrayAsStringTemplate
```

Listing B.2: ArrayAsStringTemplate/Contains/Contains.rc

```
1 contract Contains
2   enhances ArrayAsStringTemplate
3
4   function Contains (restores a: Array,
5                      restores x: Item) : control
6     ensures
7       Contains = x is in elements(a.s)
8
9 end Contains
```

Listing B.3: ArrayAsStringTemplate/Contains/LinearIterative/LinearIterative.rr

```
1 realization LinearIterative (function AreEqual (restores i: Item,
2                                                 restores j: Item) : control
3                                ensures
4                                  AreEqual = (i = j))
5   implements Contains for ArrayAsStringTemplate
6
7   function Contains (restores a: Array,
8                      restores x: Item) : control
9     variable pos: Integer
10    variable ub: Integer
11    pos := LowerBound(a)
12    ub := UpperBound(a)
13    if not IsGreater(pos, ub) then
14      loop
15        maintains a = #a and
16                  x = #x and
17                  a.lb <= pos and
18                  pos <= ub and
19                  ub = a.ub and
20                  Contains = (x is in elements(substring(a.s, 0, pos -
                        a.lb)))
21        decreases ub - pos
22      while not Contains and not AreEqual(pos, ub) do
23        variable y: Item
24        SwapItem(a, pos, y)
25        Contains := AreEqual(x, y)
26        SwapItem(a, pos, y)
27        Increment(pos)
28      end loop
29      if not Contains then
30        variable y: Item
31        SwapItem(a, ub, y)
32        Contains := AreEqual(x, y)
33        SwapItem(a, ub, y)
34      end if
35    end if
36  end Contains
37
38 end LinearIterative
```

Listing B.4: ArrayAsStringTemplate/SortableSimple/QuickSortRealization/QuickSortRealization.rr

```
realization QuckSortRealization (function AreInOrder (restores i: Item,
                                                       restores j: Item) :
                                                          control
                                   ensures
                                     AreInOrder = (ARE_IN_ORDER1(i, j)),
                                 function ItemReplica (restores x: Item) :
                                    Item
                                   ensures
                                     ItemReplica = x)
  implements SortableSimple for ArrayAsStringTemplate

  procedure AverageImpl (updates i: Integer,
                         restores j: Integer)
    decreases j - i
    if IsGreater(j, i) then
      Decrement(j)
      if IsGreater(j, i) then
        Increment(i)
        Average(i, j)
      end if
      Increment(j)
    end if
  end AverageImpl

  procedure Average (updates i: Integer,
                     restores j: Integer)
    if IsGreater(i, j) then
      variable j2: Integer
      j2 := Replica(j)
      AverageImpl(j, i)
      j :=: j2
      i :=: j2
    else
      AverageImpl(i, j)
    end if
  end Average

  procedure ElemReplica (restores a: Array,
                         restores i: Integer,
                         replaces elem: Item)
    variable temp: Item
    SwapItem(a, i, temp)
    elem := ItemReplica(temp)
    SwapItem(a, i, temp)
  end ElemReplica

  function ArrayReplica (restores a: Array) : Array
    variable i: Integer
    variable max: Integer
    variable copy: Array
    i := LowerBound(a)
    max := UpperBound(a)
    SetBounds(copy, i, max)
    Increment(max)
    loop
      maintains a = #a and
                i >= a.lb and
                i <= a.ub + 1 and
                max = a.ub + 1 and
```

```
58                copy.lb = a.lb and
59                copy.ub = a.ub and
60                substring(a.s, 0, i - a.lb) = substring(copy.s, 0, i -
                      copy.lb)
61      decreases 1 + |a.s| - (i - a.lb)
62    while IsGreater(max, i) do
63      variable item: Item
64      ElemReplica(a, i, item)
65      SwapItem(copy, i, item)
66      Increment(i)
67    end loop
68    ArrayReplica :=: copy
69  end ArrayReplica
70
71  procedure FindNewLeftLoopBody (restores a: Array,
72                                 restores mid: Item,
73                                 updates newLeft: Integer,
74                                 updates i: Integer,
75                                 updates foundGT: Boolean)
76    variable curr: Item
77    SwapItem(a, i, curr)
78    if not AreInOrder(mid, curr) then
79      SwapItem(a, i, curr)
80      Increment(i)
81    else
82      newLeft := Replica(i)
83      if not AreInOrder(curr, mid) then
84        SwapItem(a, i, curr)
85        Negate(foundGT)
86      else
87        SwapItem(a, i, curr)
88        Increment(i)
89      end if
90    end if
91  end FindNewLeftLoopBody
92
93  procedure FindNewLeft (restores a: Array,
94                         restores left: Integer,
95                         restores right: Integer,
96                         restores mid: Item,
97                         updates newLeft: Integer)
98    variable i: Integer
99    variable foundGT: Boolean
100   i := Replica(left)
101   loop
102     maintains a = #a and
103               left = #left and
104               right = #right and
105               mid = #mid and
106               left <= newLeft and
107               newLeft <= right and
108               newLeft <= a.ub and
109               left <= i and
110               i <= right and
111               (foundGT implies newLeft = i) and
112               (for all j: Integer
113                    where (left <= j and j < i)
114                  (ARE_IN_ORDER1(ELEM_AT(a, j), mid))) and
115               ARE_IN_ORDER1(mid, ELEM_AT(a, newLeft)) and
116               (#newLeft < right implies newLeft < right)
117     decreases 10 + right - i + TRUE_IS_SMALLER(foundGT)
118   while IsGreater(right, i) and not IsTrue(foundGT) do
```

```
119       FindNewLeftLoopBody(a, mid, newLeft, i, foundGT)
120     end loop
121   end FindNewLeft
122
123   procedure FindNewRightLoopBody (restores a: Array,
124                                   restores mid: Item,
125                                   updates newRight: Integer,
126                                   updates i: Integer,
127                                   updates foundLT: Boolean)
128     variable curr: Item
129     SwapItem(a, i, curr)
130     if not AreInOrder(curr, mid) then
131       SwapItem(a, i, curr)
132       Decrement(i)
133     else
134       newRight := Replica(i)
135       if not AreInOrder(mid, curr) then
136         SwapItem(a, i, curr)
137         Negate(foundLT)
138       else
139         SwapItem(a, i, curr)
140         Decrement(i)
141       end if
142     end if
143   end FindNewRightLoopBody
144
145   procedure FindNewRight (restores a: Array,
146                           restores left: Integer,
147                           restores right: Integer,
148                           restores mid: Item,
149                           updates newRight: Integer)
150     variable i: Integer
151     variable foundLT: Boolean
152     i := Replica(right)
153     Decrement(i)
154     loop
155       maintains a = #a and
156                 left = #left and
157                 right = #right and
158                 mid = #mid and
159                 left - 1 <= newRight and
160                 newRight < right and
161                 left - 1 <= i and
162                 i < right and
163                 (foundLT implies newRight = i) and
164                 (for all j: Integer
165                      where (i < j and j < right)
166                    (ARE_IN_ORDER1(mid, ELEM_AT(a, j)))) and
167                 ARE_IN_ORDER1(ELEM_AT(a, newRight), mid) and
168                 (#newRight >= left implies newRight >= left)
169       decreases 10 + i + TRUE_IS_SMALLER(foundLT)
170     while not IsGreater(left, i) and not IsTrue(foundLT) do
171       FindNewRightLoopBody(a, mid, newRight, i, foundLT)
172     end loop
173   end FindNewRight
174
175   procedure PartitionSwap (updates a: Array,
176                            restores first: Integer,
177                            restores last: Integer,
178                            restores newLeft: Integer,
179                            restores newRight: Integer,
180                            restores mid: Item)
```

```
181     if IsGreater(newRight, newLeft) then
182       variable a2: Array
183       variable temp: Item
184       SwapItem(a, newLeft, temp)
185       SwapItem(a, newRight, temp)
186       SwapItem(a, newLeft, temp)
187     end if
188   end PartitionSwap
189
190   procedure Partition (updates a: Array,
191                        restores first: Integer,
192                        restores last: Integer,
193                        replaces left: Integer,
194                        replaces right: Integer,
195                        replaces mid: Item)
196     variable midIdx: Integer
197     midIdx := Replica(first)
198     Average(midIdx, last)
199     ElemReplica(a, midIdx, mid)
200     left := Replica(first)
201     right := Replica(last)
202     loop
203       maintains a.lb = #a.lb and
204                 a.ub = #a.ub and
205                 first = #first and
206                 last = #last and
207                 midIdx = #midIdx and
208                 mid = #mid and
209                 first <= left and
210                 left <= last and
211                 first <= right and
212                 right <= last and
213                 ((left = first and right = last) or (left > first and right
                        < last)) and
214                 (left = first implies mid = ELEM_AT(a, midIdx)) and
215                 (for all i: Integer
216                      where (first <= i and i < left)
217                    (ARE_IN_ORDER1(ELEM_AT(a, i), mid))) and
218                 (for all i: Integer
219                      where (right <= i and i < last)
220                    (ARE_IN_ORDER1(mid, ELEM_AT(a, i)))) and
221                 substring(a.s, 0, first - a.lb) = substring(#a.s, 0, first
                        - a.lb) and
222                 substring(a.s, last - a.lb, |a.s|) = substring(#a.s, last -
                        a.lb, |a.s|) and
223                 IS_PERMUTATION1(a.s, #a.s)
224       decreases 10 + (a.ub - a.lb) + (right - left)
225     while IsGreater(right, left) do
226       variable newLeft, newRight: Integer
227       if AreEqual(left, first) then
228         newLeft := Replica(midIdx)
229         newRight := Replica(midIdx)
230       else
231         newLeft := Replica(right)
232         newRight := Replica(left)
233         Decrement(newRight)
234       end if
235       FindNewLeft(a, left, right, mid, newLeft)
236       FindNewRight(a, left, right, mid, newRight)
237       if IsGreater(newLeft, newRight) then
238         left :=: newLeft
239         right :=: newRight
```

```
240         Increment(right)
241       else
242         PartitionSwap(a, first, last, newLeft, newRight, mid)
243         left :=: newLeft
244         Increment(left)
245         right :=: newRight
246       end if
247     end loop
248   end Partition
249
250   procedure Sort (updates a: Array,
251                   restores first: Integer,
252                   restores last: Integer)
253     decreases last - first
254     variable i: Integer
255     i := Replica(first)
256     Increment(i)
257     if IsGreater(last, i) then
258       variable a0, a1, a2: Array
259       variable left, right: Integer
260       variable mid: Item
261       Partition(a, first, last, left, right, mid)
262       Sort(a, left, last)
263       Sort(a, first, right)
264     end if
265   end Sort
266
267 end QuckSortRealization
```

Listing B.5: ArrayAsStringTemplate/SortableSimple/SortableSimple.rc

```
1  contract SortableSimple (definition ARE_IN_ORDER1 (a: Item,
2                                                      b: Item)
3                              : boolean
4                              satisfies for all x, y, z: Item
5                                          ((ARE_IN_ORDER1(x, y) or
                                              ARE_IN_ORDER1(y, x)) and
6                                          ((ARE_IN_ORDER1(x, y) and
                                              ARE_IN_ORDER1(y, x)) implies x =
                                              y) and
7                                          (if (ARE_IN_ORDER1(x, y) and
                                              ARE_IN_ORDER1(y, z)) then
8                                             ARE_IN_ORDER1(x, z))))
9   enhances ArrayAsStringTemplate
10
11  uses BooleanFacility
12
13  definition OCCURS_COUNT1 (s: string of Item,
14                            i: Item)
15    : integer
16    satisfies if s = empty_string then
17                OCCURS_COUNT1(s, i) = 0
18              else
19                there exists x: Item,
20                             r: string of Item
21                  ((s = <x> * r) and
22                   (if x = i then
23                      OCCURS_COUNT1(s, i) = OCCURS_COUNT1(r, i) + 1
24                    else
25                      OCCURS_COUNT1(s, i) = OCCURS_COUNT1(r, i)))
26
```

```
27  definition IS_PERMUTATION1 (s1: string of Item,
28                              s2: string of Item)
29    : boolean
30     is
31    for all i: Item
32      (OCCURS_COUNT1(s1, i) = OCCURS_COUNT1(s2, i))
33
34  definition TRUE_IS_SMALLER (b: boolean)
35    : integer
36    satisfies TRUE_IS_SMALLER(true) >= 0 and TRUE_IS_SMALLER(true) <
         TRUE_IS_SMALLER(false)
37
38  definition ELEM_AT (a: Array,
39                      i: integer)
40    : Item
41    satisfies (for all b: Array,
42                       j: integer
43                  where (b.lb <= j and j <= b.ub)
44                (substring(b.s, j - b.lb, j + 1 - b.lb) = <ELEM_AT(b, j)>))
45
46  procedure AverageImpl (updates i: Integer,
47                         restores j: Integer)
48    requires
49      i <= j
50    ensures
51      i + i <= #i + j and #i + j <= i + i + 1
52
53  procedure Average (updates i: Integer,
54                     restores j: Integer)
55    ensures
56      i + i <= #i + j and #i + j <= i + i + 1
57
58  procedure ElemReplica (restores a: Array,
59                         restores i: Integer,
60                         replaces elem: Item)
61    requires
62      a.lb <= i and
63      i <= a.ub
64    ensures
65      elem = ELEM_AT(a, i)
66
67  function ArrayReplica (restores a: Array) : Array
68    ensures
69      ArrayReplica = a
70
71  procedure FindNewLeftLoopBody (restores a: Array,
72                                 restores mid: Item,
73                                 updates newLeft: Integer,
74                                 updates i: Integer,
75                                 updates foundGT: Boolean)
76    requires
77      a.lb <= i and
78      i <= a.ub and
79      not foundGT and
80      ARE_IN_ORDER1(mid, ELEM_AT(a, newLeft))
81    ensures
82      ARE_IN_ORDER1(mid, ELEM_AT(a, newLeft)) and
83      (if foundGT then
84         (i = #i and
85          newLeft = i and
86          not ARE_IN_ORDER1(ELEM_AT(a, #i), mid))
87       else
```

```
88          (i = #i + 1 and
89           (newLeft = #newLeft or newLeft = #i) and
90           ARE_IN_ORDER1(ELEM_AT(a, #i), mid)))
91
92    procedure FindNewLeft (restores a: Array,
93                           restores left: Integer,
94                           restores right: Integer,
95                           restores mid: Item,
96                           updates newLeft: Integer)
97      requires
98        0 <= a.lb and
99        a.lb <= left and
100       left <= newLeft and
101       newLeft <= right and
102       right <= a.ub + 1 and
103       newLeft <= a.ub and
104       ARE_IN_ORDER1(mid, ELEM_AT(a, newLeft))
105     ensures
106       left <= newLeft and
107       newLeft <= right and
108       newLeft <= a.ub and
109       (#newLeft < right implies newLeft < right) and
110       (for all j: Integer
111            where (left <= j and j < newLeft)
112          (ARE_IN_ORDER1(ELEM_AT(a, j), mid))) and
113       ARE_IN_ORDER1(mid, ELEM_AT(a, newLeft))
114
115   procedure FindNewRightLoopBody (restores a: Array,
116                                   restores mid: Item,
117                                   updates newRight: Integer,
118                                   updates i: Integer,
119                                   updates foundLT: Boolean)
120     requires
121       a.lb <= i and
122       i <= a.ub and
123       not foundLT and
124       ARE_IN_ORDER1(ELEM_AT(a, newRight), mid)
125     ensures
126       ARE_IN_ORDER1(ELEM_AT(a, newRight), mid) and
127       (if foundLT then
128          (i = #i and
129           newRight = i and
130           not ARE_IN_ORDER1(mid, ELEM_AT(a, #i)))
131        else
132          (i = #i - 1 and
133           (newRight = #newRight or newRight = #i) and
134           ARE_IN_ORDER1(mid, ELEM_AT(a, #i))))
135
136   procedure FindNewRight (restores a: Array,
137                           restores left: Integer,
138                           restores right: Integer,
139                           restores mid: Item,
140                           updates newRight: Integer)
141     requires
142       0 <= a.lb and
143       a.lb <= left and
144       left < right and
145       right <= a.ub + 1 and
146       left - 1 <= newRight and
147       newRight < right and
148       a.lb <= newRight and
149       ARE_IN_ORDER1(ELEM_AT(a, newRight), mid)
```

```
150     ensures
151       left - 1 <= newRight and
152       newRight < right and
153       a.lb <= newRight and
154       (#newRight >= left implies newRight >= left) and
155       (for all i: Integer
156            where (newRight < i and i < right)
157          (ARE_IN_ORDER1(mid, ELEM_AT(a, i)))) and
158       ARE_IN_ORDER1(ELEM_AT(a, newRight), mid)
159
160   procedure PartitionSwap (updates a: Array,
161                            restores first: Integer,
162                            restores last: Integer,
163                            restores newLeft: Integer,
164                            restores newRight: Integer,
165                            restores mid: Item)
166     requires
167       0 <= a.lb and
168       a.lb <= first and
169       first <= newLeft and
170       newLeft <= newRight and
171       newRight < last and
172       last <= a.ub + 1 and
173       ARE_IN_ORDER1(mid, ELEM_AT(a, newLeft)) and
174       ARE_IN_ORDER1(ELEM_AT(a, newRight), mid) and
175       (for all j: Integer
176            where (first <= j and j < newLeft)
177          (ARE_IN_ORDER1(ELEM_AT(a, j), mid))) and
178       (for all i: Integer
179            where (newRight < i and i < last)
180          (ARE_IN_ORDER1(mid, ELEM_AT(a, i))))
181     ensures
182       a.lb = #a.lb and
183       a.ub = #a.ub and
184       (for all j: Integer
185            where (first <= j and j <= newLeft)
186          (ARE_IN_ORDER1(ELEM_AT(a, j), mid))) and
187       (for all i: Integer
188            where (newRight <= i and i < last)
189          (ARE_IN_ORDER1(mid, ELEM_AT(a, i)))) and
190       substring(a.s, 0, first - a.lb) = substring(#a.s, 0, first - a.lb) and
191       substring(a.s, last - a.lb, |a.s|) = substring(#a.s, last - a.lb,
              |a.s|) and
192       IS_PERMUTATION1(a.s, #a.s)
193
194   procedure Partition (updates a: Array,
195                        restores first: Integer,
196                        restores last: Integer,
197                        replaces left: Integer,
198                        replaces right: Integer,
199                        replaces mid: Item)
200     requires
201       0 <= a.lb and
202       a.lb <= first and
203       first < last and
204       last <= a.ub + 1
205     ensures
206       a.lb = #a.lb and
207       a.ub = #a.ub and
208       first <= right and
209       right <= left and
210       left <= last and
```

```
211       first < left and
212       right < last and
213       (for all i: Integer
214            where (first <= i and i < left)
215          (ARE_IN_ORDER1(ELEM_AT(a, i), mid))) and
216       (for all i: Integer
217            where (right <= i and i < last)
218          (ARE_IN_ORDER1(mid, ELEM_AT(a, i)))) and
219       substring(a.s, 0, first - a.lb) = substring(#a.s, 0, first - a.lb) and
220       substring(a.s, last - a.lb, |a.s|) = substring(#a.s, last - a.lb,
             |a.s|) and
221       IS_PERMUTATION1(a.s, #a.s)
222
223   procedure Sort (updates a: Array,
224                   restores first: Integer,
225                   restores last: Integer)
226     requires
227       0 <= a.lb and
228       a.lb <= first and
229       first <= last and
230       last <= a.ub + 1
231     ensures
232       a.lb = #a.lb and
233       a.ub = #a.ub and
234       (for all i, j: Integer
235            where (first <= i and i < j and j < last)
236          (ARE_IN_ORDER1(ELEM_AT(a, i), ELEM_AT(a, j)))) and
237       substring(a.s, 0, first - a.lb) = substring(#a.s, 0, first - a.lb) and
238       substring(a.s, last - a.lb, |a.s|) = substring(#a.s, last - a.lb,
             |a.s|) and
239       IS_PERMUTATION1(a.s, #a.s)
240
241 end SortableSimple
```

Listing B.6: IntegerFacility/Add/Add.rc

```
1  contract Add
2    enhances IntegerFacility
3
4    procedure Add(updates i: Integer,
5                  restores j: Integer)
6      requires
7        MIN <= i + j and i + j <= MAX
8      ensures
9        i = #i + j
10
11 end Add
```

Listing B.7: IntegerFacility/Add/Iterative/Iterative.rr

```
1  realization Iterative
2    implements Add for IntegerFacility
3
4    procedure Add (updates i: Integer,
5                   restores j: Integer)
6      variable nj, z: Integer
7      loop
8        maintains i + j = #i + #j and nj + j = #nj + #j and z = 0
9        decreases |j|
10     while not AreEqual(j, z) do
```

```
11        if IsGreater(j, z) then
12          Increment(i)
13          Increment(nj)
14          Decrement(j)
15        else
16          Decrement(i)
17          Decrement(nj)
18          Increment(j)
19        end if
20      end loop
21      j :=: nj
22    end Add
23
24  end Iterative
```

Listing B.8: IntegerFacility/Add/IterativeOptimized/IterativeOptimized.rr

```
1  realization IterativeOptimized
2    implements Add for IntegerFacility
3
4    procedure Add (updates i: Integer,
5                   restores j: Integer)
6      variable nj, z: Integer
7      if IsGreater(j, z) then
8        loop
9          maintains i + j = #i + #j and nj + j = #nj + #j and
10                   z = 0 and
11                   j >= 0
12         decreases |j|
13       while not AreEqual(j, z) do
14         Increment(i)
15         Increment(nj)
16         Decrement(j)
17       end loop
18     else
19       loop
20         maintains i + j = #i + #j and nj + j = #nj + #j and
21                   z = 0 and
22                   j <= 0
23         decreases |j|
24       while not AreEqual(j, z) do
25         Decrement(i)
26         Decrement(nj)
27         Increment(j)
28       end loop
29     end if
30     j :=: nj
31   end Add
32
33 end IterativeOptimized
```

Listing B.9: IntegerFacility/Add/Recursive/Recursive.rr

```
1  realization Recursive
2    implements Add for IntegerFacility
3
4    procedure Add (updates i: Integer,
5                   restores j: Integer)
6      decreases |j|
7      variable z: Integer
```

```
8     if IsGreater(j, z) then
9       Increment(i)
10      Decrement(j)
11      Add(i, j)
12      Increment(j)
13    else
14      if IsGreater(z, j) then
15        Decrement(i)
16        Increment(j)
17        Add(i, j)
18        Decrement(j)
19      end if
20    end if
21  end Add
22
23 end Recursive
```

Listing B.10: IntegerFacility/IntegerFacility.rc

```
1  contract IntegerFacility
2    definition MIN
3      : integer
4      satisfies restriction MIN <= 0
5
6    definition MAX
7      : integer
8      satisfies restriction 0 < MAX
9
10   math subtype INTEGERMODEL is integer
11     exemplar i
12     constraint MIN <= i and i <= MAX
13
14   type Integer is modeled by INTEGERMODEL
15     exemplar i
16     initialization ensures i = 0
17
18   procedure Increment (updates i: Integer)
19     requires
20       i < MAX
21     ensures
22       i = #i + 1
23
24   procedure Decrement (updates i: Integer)
25     requires
26       MIN < i
27     ensures
28       i = #i - 1
29
30   function AreEqual (restores i: Integer,
31                      restores j: Integer) : control
32     ensures
33       AreEqual = (i = j)
34
35   function IsGreater (restores i: Integer,
36                       restores j: Integer) : control
37     ensures
38       IsGreater = (i > j)
39
40   function Replica (restores i: Integer) : Integer
41     ensures
42       Replica = i
```

```
43
44   function Min () : Integer
45     ensures
46       Min = MIN
47
48   function Max () : Integer
49     ensures
50       Max = MAX
51
52 end IntegerFacility
```

Listing B.11: IntegerFacility/Mod/Mod.rc

```
1  contract Mod
2    enhances IntegerFacility
3
4    procedure Mod(updates i: Integer,
5                  restores j: Integer)
6      requires
7        j > 0
8      ensures
9        i = #i mod j
10
11 end Mod
```

Listing B.12: IntegerFacility/Mod/RemainderRealization/RemainderRealization.rr

```
1  realization RemainderRealization
2    implements Mod for IntegerFacility
3
4    uses Add for IntegerFacility
5    uses Subtract for IntegerFacility
6    uses Remainder for IntegerFacility
7
8    procedure Mod (updates i: Integer,
9                   restores j: Integer)
10     variable z: Integer
11     Remainder(i, j)
12     if IsGreater(z, i) then
13       if IsGreater(j, z) then
14         Add(i, j)
15       else
16         Subtract(i, j)
17       end if
18     end if
19   end Mod
20
21 end RemainderRealization
```

Listing B.13: IntegerFacility/Multiply/IterativeCases/IterativeCases.rr

```
1  realization IterativeCases
2    implements Multiply for IntegerFacility
3
4    uses IsPositive for IntegerFacility
5    uses Add for IntegerFacility
6    uses Subtract for IntegerFacility
7
8    procedure Multiply (updates i: Integer,
```

```
9                       restores j: Integer)
10    variable p, nj, z: Integer
11    if IsGreater(j, z) then
12      if IsGreater(i, z) then
13        loop
14          maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
15                    i = #i and
16                    z = #z and
17                    j >= 0
18          decreases j
19        while IsPositive(j) do
20          Add(p, i)
21          Increment(nj)
22          Decrement(j)
23        end loop
24      else
25        loop
26          maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
27                    i = #i and
28                    z = #z and
29                    j >= 0
30          decreases j
31        while IsPositive(j) do
32          Add(p, i)
33          Increment(nj)
34          Decrement(j)
35        end loop
36      end if
37    else
38      if IsGreater(i, z) then
39        loop
40          maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
41                    i = #i and
42                    z = #z and
43                    j <= 0 and
44                    nj <= 0
45          decreases -j
46        while not AreEqual(j, z) do
47          Subtract(p, i)
48          Decrement(nj)
49          Increment(j)
50        end loop
51      else
52        loop
53          maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
54                    i = #i and
55                    z = #z and
56                    j <= 0
57          decreases -j
58        while not AreEqual(j, z) do
59          Subtract(p, i)
60          Decrement(nj)
61          Increment(j)
62        end loop
63      end if
64    end if
65    i :=: p
66    j :=: nj
67  end Multiply
68
69 end IterativeCases
```

Listing B.14: IntegerFacility/Multiply/Iterative/Iterative.rr

```
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 + i * j = #p + #i * #j and nj + j = #nj + #j and
                  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 and
                  (if (j < 0 and i > 0) then
                     (nj + j) * i <= p - i) and
                  (if (j < 0 and i < 0) then
                     p - i <= (nj + j) * i)
        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 Iterative
```

Listing B.15: IntegerFacility/Multiply/IterativeNonObvious/IterativeNonObvious.rr

```
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
```

```
13        maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
14                  i = #i and
15                  z = #z and
16                  j >= 0
17        decreases j
18      while IsPositive(j) do
19        Add(p, i)
20        Increment(nj)
21        Decrement(j)
22      end loop
23    else
24      loop
25        maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
26                  i = #i and
27                  z = #z and
28                  j <= 0
29        decreases -j
30      while not AreEqual(j, z) do
31        Subtract(p, i)
32        Decrement(nj)
33        Increment(j)
34      end loop
35    end if
36    i :=: p
37    j :=: nj
38  end Multiply
39
40 end IterativeNonObvious
```

Listing B.16: IntegerFacility/Multiply/IterativeOneLoop/IterativeOneLoop.rr

```
1  realization IterativeOneLoop
2    implements Multiply for IntegerFacility
3
4    uses IsPositive for IntegerFacility
5    uses Add for IntegerFacility
6    uses Subtract for IntegerFacility
7
8    procedure Multiply (updates i: Integer,
9                        restores j: Integer)
10     variable p, nj, z: Integer
11     loop
12       maintains p + i * j = #p + #i * #j and nj + j = #nj + #j and
13                 i = #i and
14                 z = #z and
15                 (if #j > 0 then
16                    j >= 0
17                  else
18                    j <= 0)
19       decreases |j|
20     while not AreEqual(j, z) do
21       if IsGreater(j, z) then
22         Add(p, i)
23         Increment(nj)
24         Decrement(j)
25       else
26         Subtract(p, i)
27         Decrement(nj)
28         Increment(j)
29       end if
30     end loop
```

```
    i :=: p
    j :=: nj
  end Multiply

end IterativeOneLoop
```

Listing B.17: IntegerFacility/Multiply/IterativeOneLoopOptimized/IterativeOneLoopOptimized.rr

```
realization IterativeOneLoopOptimized
  implements Multiply for IntegerFacility

  uses Add for IntegerFacility
  uses Negate for IntegerFacility

  procedure Multiply (updates i: Integer,
                      restores j: Integer)
    variable jSign, iIncr, k, zero, prod: Integer
    Increment(jSign)
    iIncr := Replica(i)
    if IsGreater(zero, j) then // and not AreEqual (i, zero) then
      Negate(jSign)
      Negate(iIncr)
    end if
    // confirm if ( i /= 0 and j /= 0 ) then iIncr = jSign * i
    loop
      maintains jSign = #jSign and iIncr = #iIncr and i = #i and j = #j and
                prod = i * k and
                (if 0 < j then
                   (0 <= k and k <= j)
                 else
                   (j <= k and k <= 0))
      decreases |j - k|
    while not AreEqual(k, j) do
      Add(k, jSign)
      Add(prod, iIncr)
    end loop
    i :=: prod
  end Multiply

end IterativeOneLoopOptimized
```

Listing B.18: IntegerFacility/Multiply/Multiply.rc

```
contract Multiply
  enhances IntegerFacility

  procedure Multiply(updates i: Integer,
                     restores j: Integer)
    requires
      MIN <= i * j and i * j <= MAX
    ensures
      i = #i * j

end Multiply
```

Listing B.19: IntegerFacility/Sqrt/BinarySearch/BinarySearch.rr

```
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
```

Listing B.20: IntegerFacility/Sqrt/Sqrt.rc

```
contract Sqrt
  enhances IntegerFacility

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

```
9
10 end Sqrt
```

Listing B.21: QueueTemplate/Concatenate/Concatenate.rc

```
1 contract Concatenate
2   enhances QueueTemplate
3
4   procedure Concatenate(updates p: Queue,
5                         clears q: Queue)
6     ensures
7       p = #p * #q
8
9 end Concatenate
```

Listing B.22: QueueTemplate/Concatenate/Iterative/Iterative.rr

```
1 realization Iterative
2   implements Concatenate for QueueTemplate
3
4   procedure Concatenate (updates p: Queue,
5                          clears q: Queue)
6     loop
7       maintains p * q = #p * #q
8       decreases |q|
9     while not IsEmpty(q) do
10      variable x: Item
11      Dequeue(q, x)
12      Enqueue(p, x)
13    end loop
14  end Concatenate
15
16 end Iterative
```

Listing B.23: QueueTemplate/Concatenate/Recursive/Recursive.rr

```
1 realization Recursive
2   implements Concatenate for QueueTemplate
3
4   procedure Concatenate (updates p: Queue,
5                          clears q: Queue)
6     decreases |q|
7     if not IsEmpty(q) then
8       variable x: Item
9       Dequeue(q, x)
10      Enqueue(p, x)
11      Concatenate(p, q)
12    end if
13  end Concatenate
14
15 end Recursive
```

Listing B.24: QueueTemplate/QueueTemplate.rc

```
1 contract QueueTemplate (type Item)
2   uses UnboundedIntegerFacility
3
```

```
4   math subtype QUEUE_MODEL is string of Item
5
6   type Queue is modeled by QUEUE_MODEL
7     exemplar q
8     initialization ensures q = empty_string
9
10  procedure Enqueue(updates q: Queue,
11                    clears x: Item)
12    ensures
13      q = #q * <#x>
14
15  procedure Dequeue(updates q: Queue,
16                    replaces x: Item)
17    requires
18      q /= empty_string
19    ensures
20      #q = <x> * q
21
22  function Length (restores q: Queue) : Integer
23    ensures
24      Length = |q|
25
26  function IsEmpty (restores q: Queue) : control
27    ensures
28      IsEmpty = (q = empty_string)
29
30 end QueueTemplate
```

Listing B.25: QueueTemplate/Sort/MergeSort/MergeSort.rr

```
1  realization MergeSort (function AreInOrder (restores i: Item,
2                                               restores j: Item) : control
3                           ensures
4                             AreInOrder = ARE_IN_ORDER(i, j))
5    implements Sort for QueueTemplate
6
7    uses Concatenate for QueueTemplate
8
9    local procedure Split(updates q1: Queue,
10                         replaces q2: Queue)
11     ensures IS_PERMUTATION(q1 * q2, #q1) and
12             |q2| <= |q1| and
13             |q1| <= |q2| + 1
14     variable tmp: Queue
15     Clear(q2)
16     tmp :=: q1
17     loop
18       maintains IS_PERMUTATION(tmp * q1 * q2, #tmp * #q1 * #q2) and
19                 |q2| <= |q1| and
20                 |q1| <= |q2| + 1 and
21                 (tmp /= empty_string implies |q1| = |q2|)
22       decreases |tmp|
23     while not IsEmpty(tmp) do
24       variable x: Item
25       Dequeue(tmp, x)
26       Enqueue(q1, x)
27       if not IsEmpty(tmp) then
28         Dequeue(tmp, x)
29         Enqueue(q2, x)
30       end if
31     end loop
```

```
  end Split

  local procedure Merge(updates q1: Queue,
                        clears q2: Queue)
    requires |q1| > 0 and
             |q2| > 0 and
             IS_NON_DECREASING(q1) and
             IS_NON_DECREASING(q2)
    ensures IS_PERMUTATION(q1, #q1 * #q2) and
            IS_NON_DECREASING(q1)
    variable tmp: Queue
    variable q2Item: Item
    Dequeue(q2, q2Item)
    loop
      maintains IS_PERMUTATION(tmp * q1 * q2 * <q2Item>,
                #tmp * #q1 * #q2 * <#q2Item>) and
                IS_NON_DECREASING(tmp * q1) and
                IS_NON_DECREASING(tmp * <q2Item> * q2)
      decreases |q1 * q2|
    while not IsEmpty(q1) do
      variable q1Item: Item
      Dequeue(q1, q1Item)
      if not AreInOrder(q1Item, q2Item) then
        q1Item :=: q2Item
        q1 :=: q2
      end if
      confirm ARE_IN_ORDER(q1Item, q2Item)
      confirm IS_NON_DECREASING(<q1Item> * q1)
      confirm IS_NON_DECREASING(<q2Item> * q2)
      confirm IS_NON_DECREASING(<q1Item> * q2)
      confirm IS_NON_DECREASING(tmp * <q1Item>)
      Enqueue(tmp, q1Item)
    end loop
    Enqueue(tmp, q2Item)
    Concatenate(tmp, q2)
    q1 :=: tmp
  end Merge

  procedure Sort (updates q: Queue)
    decreases |q|
    variable qLength, one: Integer
    Increment(one)
    qLength := Length(q)
    if IsGreater(qLength, one) then
      variable qSplit: Queue
      Split(q, qSplit)
      Sort(q)
      Sort(qSplit)
      Merge(q, qSplit)
    end if
  end Sort

end MergeSort
```

Listing B.26: QueueTemplate/Sort/QuickSort/QuickSort.rr

```
realization QuickSort (function AreInOrder (restores i: Item,
                                            restores j: Item) : control
                         ensures
                           AreInOrder = ARE_IN_ORDER(i, j))
  implements Sort for QueueTemplate
```

```
6
7   uses Concatenate for QueueTemplate
8
9   local procedure Partition(updates qSmall: Queue,
10                            replaces qBig: Queue,
11                            restores p: Item)
12    ensures IS_PERMUTATION(qSmall * qBig, #qSmall) and
13            IS_PRECEDING(<p>, qBig) and
14            IS_PRECEDING(qSmall, <p>)
15    variable tmp: Queue
16    Clear(qBig)
17    loop
18      maintains IS_PERMUTATION(qSmall * qBig * tmp, #qSmall * #qBig * #tmp)
            and
19                IS_PRECEDING(<p>, qBig) and
20                IS_PRECEDING(tmp, <p>) and
21                p = #p
22      decreases |qSmall|
23    while not IsEmpty(qSmall) do
24      variable x: Item
25      Dequeue(qSmall, x)
26      if AreInOrder(x, p) then
27        Enqueue(tmp, x)
28      else
29        Enqueue(qBig, x)
30      end if
31    end loop
32    qSmall :=: tmp
33  end Partition
34
35  procedure Sort (updates q: Queue)
36    decreases |q|
37    variable qLength, zero: Integer
38    qLength := Length(q)
39    if IsGreater(qLength, zero) then
40      variable partitionElement: Item
41      variable qBig: Queue
42      Dequeue(q, partitionElement)
43      Partition(q, qBig, partitionElement)
44      Sort(q)
45      Sort(qBig)
46      confirm IS_NON_DECREASING(<partitionElement> * qBig)
47      Enqueue(q, partitionElement)
48      confirm IS_NON_DECREASING(q)
49      Concatenate(q, qBig)
50    end if
51  end Sort
52
53 end QuickSort
```

Listing B.27: QueueTemplate/Sort/SelectionSort/SelectionSort.rr

```
1  realization SelectionSort (function AreInOrder (restores i: Item,
2                                                   restores j: Item) : control
3                               ensures
4                                 AreInOrder = ARE_IN_ORDER(i, j))
5   implements Sort for QueueTemplate
6
7   local procedure RemoveMin(updates q: Queue,
8                             replaces min: Item)
9     requires q /= empty_string
```

```
10      ensures IS_PERMUTATION(q * <min>, #q) and
11              IS_PRECEDING(<min>, q)
12      variable tmp: Queue
13      Dequeue(q, min)
14      loop
15        maintains IS_PERMUTATION(tmp * q * <min>, #tmp * #q * <#min>) and
16                  IS_PRECEDING(<min>, tmp)
17        decreases |q|
18      while not IsEmpty(q) do
19        variable x: Item
20        Dequeue(q, x)
21        if not AreInOrder(min, x) then
22          min :=: x
23        end if
24        Enqueue(tmp, x)
25      end loop
26      q :=: tmp
27    end RemoveMin
28
29    procedure Sort (updates q: Queue)
30      variable sorted: Queue
31      loop
32        maintains IS_PERMUTATION(q * sorted, #q * #sorted) and
33                  IS_NON_DECREASING(sorted) and
34                  IS_PRECEDING(sorted, q)
35        decreases |q|
36      while not IsEmpty(q) do
37        variable min: Item
38        RemoveMin(q, min)
39        Enqueue(sorted, min)
40      end loop
41      q :=: sorted
42    end Sort
43
44  end SelectionSort
```

Listing B.28: QueueTemplate/Sort/Sort.rc

```
1  contract Sort (definition ARE_IN_ORDER (x: Item,
2                                          y: Item)
3                   : boolean
4                   satisfies restriction for all z: Item
5                                           ((ARE_IN_ORDER(x, y) or
                                               ARE_IN_ORDER(y, x)) and
6                                           (if (ARE_IN_ORDER(x, y) and
                                               ARE_IN_ORDER(y, z)) then
7                                              ARE_IN_ORDER(x, z))))
8    enhances QueueTemplate
9
10   definition OCCURS_COUNT (s: string of Item,
11                            i: Item)
12     : integer
13     satisfies if s = empty_string then
14                 OCCURS_COUNT(s, i) = 0
15               else
16                 there exists x: Item,
17                              r: string of Item
18                   ((s = <x> * r) and
19                   (if x = i then
20                      OCCURS_COUNT(s, i) = OCCURS_COUNT(r, i) + 1
21                    else
```

```
22                      OCCURS_COUNT(s, i) = OCCURS_COUNT(r, i)))
23
24    definition IS_PERMUTATION (s1: string of Item,
25                               s2: string of Item)
26      : boolean
27       is
28      for all i: Item
29        (OCCURS_COUNT(s1, i) = OCCURS_COUNT(s2, i))
30
31    definition IS_PRECEDING (s1: string of Item,
32                             s2: string of Item)
33      : boolean
34       is
35      for all i,
36               j: Item
37          where (OCCURS_COUNT(s1, i) > 0 and
38                 OCCURS_COUNT(s2, j) > 0)
39        (ARE_IN_ORDER(i, j))
40
41    definition IS_NON_DECREASING (s: string of Item)
42      : boolean
43       is
44      for all a,
45               b: string of Item
46          where (s = a * b)
47        (IS_PRECEDING(a, b))
48
49    procedure Sort(updates q: Queue)
50      ensures
51        IS_PERMUTATION(q, #q) and
52        IS_NON_DECREASING(q)
53
54  end Sort
```

Listing B.29: SetTemplate/Apply/Apply.rc

```
1  contract Apply (definition FUNCTION (x: finite set of Item)
2                    : finite set of Item
3                    satisfies FUNCTION(empty_set) = empty_set and for all s,
4                                                                    t:
                                                                        finite
                                                                        set
                                                                        of
                                                                        Item
5                                  (FUNCTION(s union t) = FUNCTION(s) union
                                      FUNCTION(s)))
6    enhances SetTemplate
7
8    procedure Apply(restores s: Set,
9                    replaces t: Set)
10     ensures
11       t = FUNCTION(s)
12
13 end Apply
```

Listing B.30: SetTemplate/Apply/Iterative/Iterative.rr

```
1  realization Iterative (procedure Apply(restores x: Item,
2                                         replaces y: Item)
3                            ensures
```

```
4                              {y} = FUNCTION({x}))
5   implements Apply for SetTemplate
6
7   procedure Apply (restores s: Set,
8                    replaces t: Set)
9     variable tempSet: Set
10    Clear(t)
11    loop
12      maintains t = FUNCTION(tempSet) and
13                s intersection tempSet = empty_set and
14                s union tempSet = #s union #tempSet
15      decreases |s|
16    while not IsEmpty(s) do
17      variable x, y: Item
18      RemoveAny(s, x)
19      Apply(x, y)
20      if not Contains(t, y) then
21        Add(t, y)
22      end if
23      Add(tempSet, x)
24    end loop
25    s :=: tempSet
26  end Apply
27
28 end Iterative
```

Listing B.31: SetTemplate/Intersect/Intersect.rc

```
1 contract Intersect
2   enhances SetTemplate
3
4   procedure Intersect(updates s: Set,
5                       restores t: Set)
6     ensures
7       s = #s intersection t
8
9 end Intersect
```

Listing B.32: SetTemplate/Intersect/Iterative/Iterative.rr

```
1 realization Iterative
2   implements Intersect for SetTemplate
3
4   procedure Intersect (updates s: Set,
5                        restores t: Set)
6     variable tmp: Set
7     loop
8       maintains (s union tmp) intersection t =
9                 (#s union #tmp) intersection t and
10                s intersection tmp = empty_set and
11                tmp intersection t = tmp and
12                t = #t
13      decreases |s|
14    while not IsEmpty(s) do
15      variable x: Item
16      RemoveAny(s, x)
17      if Contains(t, x) then
18        Add(tmp, x)
19      end if
20    end loop
```

```
    s :=: tmp
  end Intersect

end Iterative
```

Listing B.33: SetTemplate/SetTemplate.rc

```
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 \ {x} and
      xCopy = x

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

  function Size (restores s: Set) : Integer
    ensures
      Size = |s|

end SetTemplate
```

Listing B.34: SetTemplate/Split/Iterative/Iterative.rr

```
realization Iterative (function PRECEEDS (restores x: Item,
                                          restores y: Item) : control
                       ensures
```

```
                              PRECEEDS = IS_PRECEDING({x}, {y}))
  implements Split for SetTemplate

  procedure Split (updates s: Set,
                   restores x: Item,
                   replaces t: Set)
    variable tempSet: Set
    Clear(t)
    loop
      maintains x = #x and
                IS_PRECEDING({x}, t) and
                IS_PRECEDING(tempSet, {x}) and
                (s union t union tempSet = #s union #t union #tempSet) and
                (s intersection t = empty_set) and
                s intersection tempSet = empty_set and
                tempSet intersection t = empty_set
      decreases |s|
    while not IsEmpty(s) do
      variable y: Item
      RemoveAny(s, y)
      if PRECEEDS(x, y) then
        Add(t, y)
      else
        Add(tempSet, y)
      end if
    end loop
    tempSet :=: s
  end Split

end Iterative
```

Listing B.35: SetTemplate/Split/Split.rc

```
contract Split (definition IS_PRECEDING (x: finite set of Item,
                                         y: finite set of Item)
                  : boolean
                  satisfies restriction IS_PRECEDING(x, empty_set) and
                                        IS_PRECEDING(empty_set, x) and
                                        (IS_PRECEDING(x, y) or
                                           IS_PRECEDING(y, x)) and
                                        for all t,
                                                s: finite set of Item
                                            where (y = t union s and
                                                IS_PRECEDING(x, y))
                                          (IS_PRECEDING(x, t) and
                                             IS_PRECEDING(x, s)) and
                                        for all t,
                                                s: finite set of Item
                                            where (x = t union s and
                                                IS_PRECEDING(x, y))
                                          (IS_PRECEDING(t, y) and
                                             IS_PRECEDING(s, y)))
  enhances SetTemplate

  procedure Split(updates s: Set,
                  restores x: Item,
                  replaces t: Set)
    ensures
      #s = s union t and
      s intersection t = empty_set and
      IS_PRECEDING(s, {x}) and
```

```
24        IS_PRECEDING({x}, t)
25
26  end Split
```

Listing B.36: SetTemplate/Subtract/Iterative/Iterative.rr

```
1  realization Iterative
2    implements Subtract for SetTemplate
3
4    procedure Subtract (updates s: Set,
5                        restores t: Set)
6      variable tmp: Set
7      loop
8        maintains s union tmp union t = #s union #tmp union t and
9                  s intersection tmp = empty_set and
10                 tmp intersection t = empty_set and
11                 t = #t
12       decreases |s|
13     while not IsEmpty(s) do
14       variable x: Item
15       RemoveAny(s, x)
16       if not Contains(t, x) then
17         Add(tmp, x)
18       end if
19     end loop
20     s :=: tmp
21   end Subtract
22
23 end Iterative
```

Listing B.37: SetTemplate/Subtract/Subtract.rc

```
1  contract Subtract
2    enhances SetTemplate
3
4    procedure Subtract(updates s: Set,
5                       restores t: Set)
6      ensures
7        s = #s \ t
8
9  end Subtract
```

Listing B.38: SetTemplate/UniteAndIntersect/Iterative1/Iterative1.rr

```
1  realization Iterative1
2    implements UniteAndIntersect for SetTemplate
3
4    procedure UniteAndIntersect (updates s: Set,
5                                 updates t: Set)
6      variable tmp: Set
7      loop
8        maintains s union t = #s union #t and
9                  (s intersection t) union tmp =
10                 (#s intersection #t) union #tmp and
11                 t intersection tmp = {}
12       decreases |t|
13     while not IsEmpty(t) do
14       variable x: Item
15       RemoveAny(t, x)
```

```
16        if not Contains(s, x) then
17          Add(s, x)
18        else
19          Add(tmp, x)
20        end if
21      end loop
22      t :=: tmp
23    end UniteAndIntersect
24
25  end Iterative1
```

Listing B.39: SetTemplate/UniteAndIntersect/Iterative2/Iterative2.rr

```
1  realization Iterative2
2    implements UniteAndIntersect for SetTemplate
3
4    procedure UniteAndIntersect (updates s: Set,
5                                 updates t: Set)
6      variable ss, ts: Integer
7      variable tmp: Set
8      ss := Size(s)
9      ts := Size(t)
10     if IsGreater(ts, ss) then
11       s :=: t
12     end if
13     loop
14       maintains s union t = #s union #t and
15                 (s intersection t) union tmp =
16                 (#s intersection #t) union #tmp and
17                 t intersection tmp = {}
18       decreases |t|
19     while not IsEmpty(t) do
20       variable x: Item
21       RemoveAny(t, x)
22       if not Contains(s, x) then
23         Add(s, x)
24       else
25         Add(tmp, x)
26       end if
27     end loop
28     t :=: tmp
29   end UniteAndIntersect
30
31 end Iterative2
```

Listing B.40: SetTemplate/UniteAndIntersect/UniteAndIntersect.rc

```
1  contract UniteAndIntersect
2    enhances SetTemplate
3
4    procedure UniteAndIntersect(updates s: Set,
5                                updates t: Set)
6      ensures
7        s = #s union #t and t = #s intersection #t
8
9  end UniteAndIntersect
```

Listing B.41: SetTemplate/Unite/Iterative1/Iterative1.rr

```
realization Iterative1
  implements Unite for SetTemplate

  procedure Unite (updates s: Set,
                   clears t: Set)
    loop
      maintains s union t = #s union #t
      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
  end Unite

end Iterative1
```

Listing B.42: SetTemplate/Unite/Iterative2/Iterative2.rr

```
realization Iterative2
  implements Unite for SetTemplate

  procedure Unite (updates s: Set,
                   clears t: Set)
    variable ss, ts: Integer
    ss := Size(s)
    ts := Size(t)
    if IsGreater(ts, ss) then
      s :=: t
    end if
    loop
      maintains s union t = #s union #t
      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
  end Unite

end Iterative2
```

Listing B.43: SetTemplate/Unite/Unite.rc

```
contract Unite
  enhances SetTemplate

  procedure Unite(updates s: Set,
                  clears t: Set)
    ensures
      s = #s union #t

end Unite
```

Listing B.44: StackTemplate/Reverse/Iterative/Iterative.rr

```
realization Iterative
  implements Reverse for StackTemplate

  procedure Reverse (updates s: Stack)
    variable tmp: Stack
    loop
      maintains reverse(s) * tmp = reverse(#s) * #tmp
      decreases |s|
    while not IsEmpty(s) do
      variable x: Item
      Pop(s, x)
      Push(tmp, x)
    end loop
    s :=: tmp
  end Reverse

end Iterative
```

Listing B.45: StackTemplate/Reverse/Reverse.rc

```
contract Reverse
  enhances StackTemplate

  procedure Reverse(updates s: Stack)
    ensures
      s = reverse(#s)

end Reverse
```

Listing B.46: StackTemplate/StackTemplate.rc

```
contract StackTemplate (type Item)
  uses UnboundedIntegerFacility

  math subtype STACK_MODEL is string of Item

  type Stack is modeled by STACK_MODEL
    exemplar s
    initialization ensures s = empty_string

  procedure Push(updates s: Stack,
                 clears x: Item)
    ensures
      s = <#x> * #s

  procedure Pop(updates s: Stack,
                replaces x: Item)
    requires
      s /= empty_string
    ensures
      #s = <x> * s

  function IsEmpty (restores s: Stack) : control
    ensures
      IsEmpty = (s = empty_string)

  function Length (restores s: Stack) : Integer
    ensures
```

```
28        Length = |s|
29
30  end StackTemplate
```

Listing B.47: TextFacility/SwapSubstring/Recursive1/Recursive1.rr

```
1  realization Recursive1
2    implements SwapSubstring for TextFacility
3
4    uses IsPositive for UnboundedIntegerFacility
5
6    procedure SwapSubstring (updates t1: Text,
7                             restores pos: Integer,
8                             restores len: Integer,
9                             updates t2: Text)
10     decreases |t2| + len
11     if IsEmpty(t2) then
12       if IsPositive(len) then
13         variable c: Character
14         variable z: Integer
15         Remove(t1, pos, c)
16         Decrement(len)
17         SwapSubstring(t1, pos, len, t2)
18         Increment(len)
19         Add(t2, z, c)
20       end if
21     else
22       variable c: Character
23       variable z: Integer
24       Remove(t2, z, c)
25       SwapSubstring(t1, pos, len, t2)
26       Add(t1, pos, c)
27     end if
28   end SwapSubstring
29
30 end Recursive1
```

Listing B.48: TextFacility/SwapSubstring/SwapSubstring.rc

```
1  contract SwapSubstring
2    enhances TextFacility
3
4    definition SUBSTRING (s: string of character,
5                          start: integer,
6                          finish: integer)
7      : string of character
8      satisfies if (start < 0) or (start > finish) or (finish > |s|) then
9                  SUBSTRING(s, start, finish) = empty_string
10               else
11                 there exists a,
12                              b: string of character
13                   (s = a * SUBSTRING(s, start, finish) * b and
14                   |a| = start and
15                   |b| = |s| - finish)
16
17   definition SUBSTRING_REPLACEMENT (s: string of character,
18                                     ss: string of character,
19                                     start: integer,
20                                     finish: integer)
21     : string of character
```

```
22      satisfies if (start < 0) or (start > finish) or (finish > |s|) then
23                   SUBSTRING_REPLACEMENT(s, ss, start, finish) = s
24                 else
25                   there exists a,
26                                b,
27                                c: string of character
28                     (s = a * b * c and
29                     |a| = start and
30                     |c| = |s| - finish and
31                     SUBSTRING_REPLACEMENT(s, ss, start, finish) = a * ss * c)
32
33    procedure SwapSubstring(updates t1: Text,
34                            restores pos: Integer,
35                            restores len: Integer,
36                            updates t2: Text)
37      requires
38        0 <= pos and pos + len <= |t1| and len >= 0
39      ensures
40        t1 = SUBSTRING_REPLACEMENT(#t1, #t2, pos, pos + len) and
41        t2 = SUBSTRING(#t1, pos, pos + len)
42
43  end SwapSubstring
```

Listing B.49: TextFacility/TextFacility.rc

```
1   contract TextFacility
2     uses CharacterFacility
3     uses UnboundedIntegerFacility
4
5     definition DIFFER_BY_ONE (t1: string of character,
6                               t2: string of character,
7                               pos: integer,
8                               ch: character)
9       : boolean
10       is
11      there exists a,
12                   b: string of character
13        (t1 = a * b and t2 = a * <ch> * b and |a| = pos)
14
15    type Text is modeled by string of character
16      exemplar t
17      initialization ensures t = empty_string
18
19    procedure Add(updates t: Text,
20                  restores pos: Integer,
21                  restores ch: Character)
22      requires
23        0 <= pos and pos <= |t|
24      ensures
25        DIFFER_BY_ONE(#t, t, pos, ch)
26        // there exists a, b: string of character
27        //     (#t = a * b  and  t = a * <ch> * b  and  |a| = pos)
28
29    procedure Remove(updates t: Text,
30                     restores pos: Integer,
31                     replaces ch: Character)
32      requires
33        0 <= pos and pos < |t|
34      ensures
35        DIFFER_BY_ONE(t, #t, pos, ch)
36        // there exists a, b: string of character
```

```
37        //      (#t = a * <ch> * b  and  t = a * b  and  |a| = pos)
38
39    function Length (restores t: Text) : Integer
40      ensures
41        Length = |t|
42
43    function IsEmpty (restores t: Text) : control
44      ensures
45        IsEmpty = (t = empty_string)
46
47    function AreEqual (restores t1: Text,
48                       restores t2: Text) : control
49      ensures
50        AreEqual = (t1 = t2)
51
52    function Replica (restores t: Text) : Text
53      ensures
54        Replica = t
55
56  end TextFacility
```

Listing B.50: UnboundedIntegerFacility/Add/Add.rc

```
1  contract Add
2    enhances UnboundedIntegerFacility
3
4    procedure Add(updates i: Integer,
5                  restores j: Integer)
6      ensures
7        i = #i + j
8
9  end Add
```

Listing B.51: UnboundedIntegerFacility/Add/Iterative/Iterative.rr

```
1  realization Iterative
2    implements Add for UnboundedIntegerFacility
3
4    procedure Add (updates i: Integer,
5                   restores j: Integer)
6      variable nj, z: Integer
7      loop
8        maintains i + j = #i + #j and nj + j = #nj + #j and z = 0
9        decreases |j|
10     while not AreEqual(j, z) do
11       if IsGreater(j, z) then
12         Increment(i)
13         Increment(nj)
14         Decrement(j)
15       else
16         Decrement(i)
17         Decrement(nj)
18         Increment(j)
19       end if
20     end loop
21     j :=: nj
22   end Add
23
24 end Iterative
```

Listing B.52: UnboundedIntegerFacility/Sqrt/BinarySearch/BinarySearch.rr

```
realization BinarySearch
  implements Sqrt for UnboundedIntegerFacility

  uses Average for UnboundedIntegerFacility
  uses Subtract for UnboundedIntegerFacility
  uses Multiply for UnboundedIntegerFacility
  uses Square for UnboundedIntegerFacility

  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
                  d = hi - i and
                  one = #one and
                  two = #two and
                  t = #t and
                  0 <= hi 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)
        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
```

Listing B.53: UnboundedIntegerFacility/Sqrt/Iterative/Iterative.rr

```
realization Iterative
  implements Sqrt for UnboundedIntegerFacility

  uses Add for UnboundedIntegerFacility
  uses Subtract for UnboundedIntegerFacility
  uses Multiply for UnboundedIntegerFacility
```

```
  uses Divide for UnboundedIntegerFacility
  uses Square for UnboundedIntegerFacility

  procedure Sqrt (updates i: Integer)
    variable x, xsqrd: Integer
    Increment(x)
    Increment(xsqrd)
    loop
      maintains xsqrd = x * x and
                i = #i and
                x + x > 1 and
                (x - 1) * (x - 1) <= i
      decreases i - (x - 1) * (x - 1)
    while not IsGreater(xsqrd, i) do
      Increment(x)
      xsqrd := Replica(x)
      Square(xsqrd)
    end loop
    Decrement(x)
    i :=: x
  end Sqrt

end Iterative
```

Listing B.54: UnboundedIntegerFacility/Sqrt/Sqrt.rc

```
contract Sqrt
  enhances UnboundedIntegerFacility

  procedure Sqrt(updates i: Integer)
    requires
      i >= 0
    ensures
      i * i <= #i and #i < (i + 1) * (i + 1)

end Sqrt
```

Listing B.55: UnboundedIntegerFacility/UnboundedIntegerFacility.rc

```
contract UnboundedIntegerFacility
  type Integer is modeled by integer
    exemplar i
    initialization ensures i = 0

  procedure Increment(updates i: Integer)
    ensures
      i = #i + 1

  procedure Decrement(updates i: Integer)
    ensures
      i = #i - 1

  function AreEqual (restores i: Integer,
                     restores j: Integer) : control
    ensures
      AreEqual = (i = j)

  function IsGreater (restores i: Integer,
                      restores j: Integer) : control
    ensures
```

```
22        IsGreater = (i > j)
23
24    function Replica (restores i: Integer) : Integer
25      ensures
26        Replica = i
27
28  end UnboundedIntegerFacility
```

Bibliography

[1] Z3 releases.

[2] Bruce M. Adcock. *Working Towards the Verified Software Process*. PhD thesis, The Ohio State University, 2010.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[4] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 480–491, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Leonardo de Moura and Grant Olney Passmore. *The Strategy Challenge in SMT Solving*, pages 15–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 237–252, New York, NY, USA, 2003. Association for Computing Machinery.

[9] Matthias Erdin, Vytautas Astrauskas, and Federico Poli. Verification of rust generics, typestates, and traits. Master's thesis, ETH Zürich, 2019.

[10] Levent Erkök. set-has-size disappeared?, 2020.

[11] Levent Erkök. Soundness: Sethassize results in an incorrect unsat answer, 2020.

[12] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, Dec 1931.

[13] David L. Heine and Monica S. Lam. *Static Detection of Leaks in Polymorphic Containers*, page 252–261. Association for Computing Machinery, New York, NY, USA, 2006.

[14] Wayne Heym. *Computer Program Verification: Improvements for Human Reasoning*. PhD thesis, Ohio State University, 1995.

[15] Wayne D. Heym, Timothy J. Long, William F. Ogden, and Bruce W. Weide. Mathematical Foundations and Notation of RESOLVE, Sep 1998.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[17] Dustin Hunter Hoffman. *Techniques for the Specification and Verification ofEnterprise Applications*. PhD thesis, The Ohio State University, 2016.

[18] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, page 131–140, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[20] K. Rustan M. Leino. This is boogie 2. Reference manual for Boogie version 2, June 2008.

[21] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.

[22] P. Müller. Specification and implementation of an annotation language for an object-oriented programming language. Master's thesis, Technische Universität München, 1995. (In German).

[23] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, June 2006.

[24] Maksim Orlovich and Radu Rugina. Memory leak analysis by contradiction. In Kwangkeun Yi, editor, *Static Analysis*, pages 405–424, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[25] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 254–264, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[27] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[28] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 602–629, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[29] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 115–125, New York, NY, USA, 2005. Association for Computing Machinery.

[30] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. Leakchecker: Practical static memory leak detection for managed languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 87–97, New York, NY, USA, 2014. Association for Computing Machinery.