12-2011

Mechanical and Modular Verification Condition Generation for Object-Based Software

Heather Harton
Clemson University, hkeown@g.clemson.edu

MECHANICAL AND MODULAR VERIFICATION CONDITION GENERATION FOR OBJECT-BASED SOFTWARE

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Heather Keown Harton
December 2011

Accepted by:
Dr. Murali Sitaraman, Committee Chair
Dr. Jason Hallstorm
Dr. Joan Krone
Dr. David Jacobs

ABSTRACT

The foundational goal of this work is the development of mechanizable proof rules and a verification condition generator based on those rules for modern software. The verification system will be modular so that it is possible to verify the implementation of a component relying upon only the specifications of underlying components that are reused. The system must enable full behavioral verification. The proof rules used to generate verification conditions (VCs) of correctness must be amenable to automation. While automation requires software developers to annotate implementations with assertions, it should not require assistance in the proofs. This research has led to a VC generator that realizes these goals. The VC generator has been applied to a range of benchmarks to show the viability of verified components. It has been used in classrooms at multiple institutions to teach reasoning principles.

A fundamental problem in computing is the inability to show that a software system behaves as required. Modern software systems are composed of numerous software components. The fundamental goal of this work is to verify each independently in a modular fashion, resulting in full behavioral verification and providing an assurance that components meet their specifications and can be used with confidence to build verified software systems. Of course, to be practical, such a system must be mechanical. Although the principles of verification have existed for decades, the basis for a practical verification system for modern software components has remained elusive.

DEDICATION

To Mom, Dad, Billy, Robyn, and Levi for being there for me, and believing I could finish. I couldn't have done it without you.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Murali Sitaraman, for his help and support in continuing in school. Without his support and guidance, I would never have been able to finish this work. I would also like to thank Bill Ogden and Joan Krone in working through each iteration of the proof rules. I would never have arrived at the final versions which are so integral to this work without their help. I would also like to thank my dissertation committee for providing me with valuable feedback and for serving on my committee. This research has been funded in part by NSF grants CCF0811748, DMS0701187, and DUE-1022941.

TABLE OF CONTENTS

Page

ABSTRACT ii
DEDICATION iii
ACKNOWLEDGMENTS iv
TABLE OF CONTENTS v
LIST OF TABLES vii
LIST OF FIGURES viii
1. INTRODUCTION 1
Organization 2
Problem Statement 3
Research Approach and Contributions 3
Thesis 5
Scope of Research 13
2. RELATED RESEARCH 18
Verification Research 18
Alternative Verification Condition Generation Approaches 22
RESOLVE Background 29
3. MECHANIZATION AND SEMANTICS OF PROOF RULES 41
Semantic Foundations 42
Relational Semantics 44
Formal Semantics 46
Proof Rules for RESOLVE Language Constructs 48
VC Simplification 82
4. COMPONENT-LEVEL VERIFICATION 83
Object-Based Component Specification, Implementation, and VCs 83
Enhancement Specification, Implementation, and VCs 100
Facility Declarations and VCs 103

5. EXPERIMENTAL EVALUATION 109
Benchmarks 109
An Exercise with Auxiliary Variables 127
6. EDUCATIONAL USES 130
User Feedback 136
7. CONCLUSIONS AND FUTURE DIRECTIONS 138
APPENDICES 141
A. Proof Rules 142
B. Simplification Rules 152
C. Additional Specifications 153
D. Alternate Stack Specifications with implementations and VCs 158
E. Examples of VCs from Component-Level Verification 164
F. Benchmark VCs 201
G. Recursive Educational Example Verification Conditions 290
REFERENCES 301

LIST OF TABLES

Page

Table 1: Related Works table 22
Table 2: Reasoning Table 24
Table 3: An application of the goal oriented approach 28
Table 4: VC Comparison when Using Auxiliary Code 289

LIST OF FIGURES

Page

Figure 1. The architecture for a verifying compiler 3
Figure 2: Modularity of Verification 9
Figure 3: An example of Assertive Code 23
Figure 4: Stack Template 31
Figure 5: Relationship of Correspondence, Constraints, and Conventions 35
Figure 6: Array Based Stack Implementation 36
Figure 7: Stack Flip Specification and Realization 38
Figure 8: Stack Facility enhanced with Reversal Capability 39
Figure 9: Implications of Relational Operation and Correspondence Specifications 45
Figure 10: Semantics of Basic Statements 47
Figure 12: Flip Assertive Code after the Application of Procedure Declaration Rule (Version 1) 50
Figure 13: Assertive Code after the Application of the Swap Rule (Version 2) 53
Figure 14: Hoare's Original Proof Rule 58
Figure 15: Application of the Mechanizable While Loop Rule (Version 3 – Part 1) 65
Figure 16: Application of the Mechanizable While Loop Rule (Version 3 – Part 2) 65
Figure 17: Flip Example (Version 4) 71
Figure 18: Assertive Code after the application of Assume Rule (Version 5) 72
Figure 19: Assertive Code after the application of Confirm Rule (Version 6) 74
Figure 20: Assertive Code just before the Application of Variable Declaration Rule (Version 7) 75
Figure 21: Assertive Code After the Application of Variable Declaration Rule (Version 8) 76
Figure 22: Flip Example (Version 9) 77
Figure 24: Queue Template 85
Figure 25: Queue Array Implementation 86
Figure 26: Example Concept Specification and Realization 90
Figure 27: Sorting Capability for a Queue 100
Figure 28: Selection Sort of a Queue 103
Figure 29: Sort Facility 105
Figure 30: Two VCs for QF Facility Declaration 107
Figure 31: VCs to show Operation is valid for QF Facility Declaration 108
Figure 32: Add and Multiply Example 111
Figure 33: Wrong Binary Search Example 113
Figure 34: Binary Search Specification 115
Figure 35: Binary Search Implementation 116

Figure 36: Search and Store Specification 118
Figure 38: Example Search and Store VC 120
Figure 39: Search and Store Implementation 121
Figure 40: Location Linking Template 124
Figure 41: Unbounded Queue 124
Figure 42: Queue Implementation with a Linked List 126
Figure 43: List Implementation of Unbounded Queue 127
Figure 44: Queue Rotate with Alternate Specification 128
Figure 45: Queue Rotate with Auxiliary Variables Specification and Implementation 129
Figure 46: Queue Recursive Append Specification and Implementation 132
Figure 47: Example VC for Recursive Append 134
Figure 48: Invalid Queue Recursive Append implementation 134
Figure 49: Example VC for Recursive Append with Wrong Decreasing Clause 135
Figure 50: Another Invalid Queue Recursive Append Implementation 135
Figure 51: Example VC for Recursive Append with wrong implementation 136

CHAPTER ONE

INTRODUCTION

A verification system to validate the correctness of software was first presented by James C. King in 1969 [1] [2]. The same year, Tony Hoare introduced proof rules that formed a basis for program verification [3] . Nearly 40 years later, no such verifier exists. Acknowledging this state of computing, Tony Hoare has proposed a *Grand Challenge* to the Computer Science community in 2003: development of a verifier that will automatically and mechanically prove through mathematical logic a program's correctness [4].

Why verify? Because even with thorough testing, there is no guarantee that a program is error-free. Most commercial software endures extensive testing until testing reveals no serious errors and the customers choose to accept the reliability of the resulting code. However, unknown and hidden errors remain. For example, in 2006, Joshua Bloch reports in a Google blog an error in the *binary search algorithm* that arises in searching large arrays [5]. This algorithm that has been in use for decades fails if the sum of the low value and the high value is greater than the value of the maximum positive integer. Amazingly, this simple error has remained hidden in a common piece of code. If this fairly simple and widely-used code has an error, it is likely that nearly all present-day software—including safety-critical ones—have similar errors. Joshua Bloch notes that in his class at CMU, Bentley (author of *Programming Pearls*) proved the binary search algorithm to be correct. Actually, what Bentley did was give only a typical, "informal" argument, not a formal proof. If integer bounds are specified and the code undergoes verification through a verifier (such as the one envisioned by Hoare and as shown through the generated proofs from my prototype Verification Condition (VC) generator), the error would have been caught in a straightforward manner. A key goal is to replace informal proofs with formal, automated proofs. The binary

search example shows that a verification system must consider all aspects of correctness – including checking that variables stay within their specified bounds. My goal is to provide a VC generator that will process modern object-oriented software systems and provide VCs that, if proven, establish correctness of components, independently, to achieve *scalability*.

Organization

The rest of this chapter presents the motivation for this work and discusses the complexities of generating verification conditions. Chapter 2 provides a detailed description of related research. This includes work by other organizations in verification, other approaches to verification, and a background of the RESOLVE language. Chapter 3 describes the semantics of RESOLVE and provides proofs of soundness and completeness for key proof rules. Operation-level proof rules are provided. Chapter 4 explains how RESOLVE VC generation provides modular and scalable software design using module-level proof rules. Chapter 5 provides a listing of examples and benchmarks demonstrating the capabilities of the VC generator. Chapter 6 briefly discusses educational uses of the RESOLVE VC generator. Finally, Chapter 7 will provide a conclusion and list future issues and directions to be researched.

There are also numerous appendices containing proof rules, source code, and generated VCs. The proof rules are provided in Appendix A. Appendix B contains simplification rules that were considered when implementing the VC generator. Appendix C, Appendix D, Appendix E, Appendix F and Appendix G all contain source code and VCs for examples. These will be detailed in later chapters.

Problem Statement

Overall, I see three challenges in developing a verification system (such as the one Hoare envisions):

• It must be scalable so that each component can be proven correct independently using only the specifications for any subcomponents that the given component relies on;

• It must enable full verification to guarantee that the implementation fulfills the completely specified behavior; and

• It must be mechanical, requiring programmers to supply assertions, but developers should not need to interact with the prover.

Research Approach and Contributions

To address the verification challenge, we envision an architecture for a verifier similar to the one shown below.

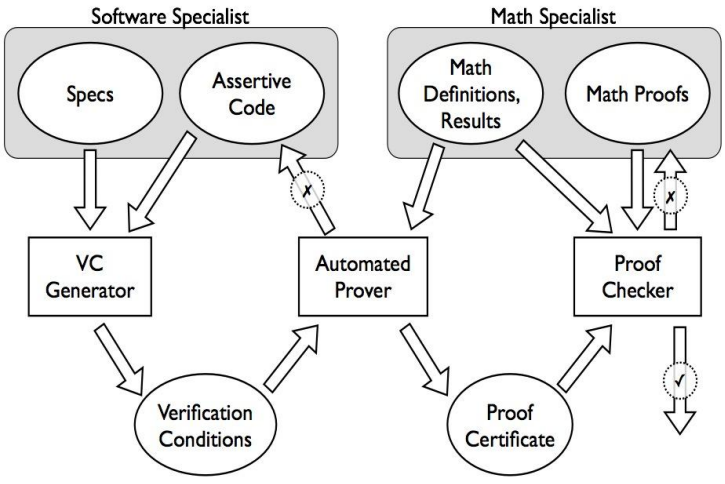


Figure 1. The architecture for a verifying compiler

In Figure 1, the *Verification Condition Generator* (a prototype of which I have developed) applies applicable proof rules to an implementation. To begin the process, a software specialist, who is assumed to be a competent programmer, will write the specification and implementation (annotated as necessary for generating the assertive code). The programmer may require assistance from the mathematician in writing the specifications. Using the specification of the new code and the specifications of components used by the implementation, the VC Generator mechanically forms mathematical clauses equivalent to the correctness of the program. The implementations will need to be annotated by internal assertions, such as loop invariants, progress metrics, and representation invariants and abstraction relations for objects. These assertions are not just annotations but are part of the language syntax that distinguishes RESOLVE from many other languages. The requirement for additional annotations by the developer is not unique to the RESOLVE verification system. In a recent set of benchmarks provided by various verification systems at VSTTE 2010, every solution with a loop had a loop invariant supplied by the programmer [6]. In addition, a recent paper has shown that when compared to a variety of systems, RESOLVE requires only minimal assertions [7].

The verification conditions (VCs) are sent to the *Automated Prover* that uses pre-established theorems from the mathematical units to prove the obligations correct. The mathematical units exist in a library of theorems provided by the mathematical specialist. The theorems in the math units must be supported by proofs which will be checked for correctness by the *Proof Checker*. The expectation is that the proof of program obligations needs to be fully automated, but only the ability to validate proofs provided by the mathematicians is necessary for non-trivial mathematical theorems.

There are several research problems that concern general verification. These include techniques for construction of specifications that allow for easy mechanization, yet are human-understandable, handling of higher-ordered assertions (e.g., such as those that arise in proving generic operations such as sorting), and mechanical handling of existential quantifiers using suitable language mechanisms and adjunct variables in programs. These issues and others must be considered when automating and implementing the proof rules used to generate the VCs. In order to achieve the ultimate goal of a verifying compiler that is useful for substantial software in languages like C++, Java or RESOLVE, my goal is to work towards modular verification of imperative, object-oriented software that is scalable as a result of the verification of individual components. This research will focus on the RESOLVE language because both C++ and Java are highly dependent upon aliasing which make reasoning for verification more difficult. Because of this design goal of modularity, additional research problems must be considered such as extensions, minimization of aliasing, handling of unavoidable aliasing that occurs with pointers, and the correct processing of global variables. My research will address these and other research issues, and establish the basis for a scalable and automated verification system for full behavioral correctness. A specific goal is to demonstrate the abilities of the VC generator by experimentally validating the correctness of a set of benchmarks.

Thesis

Challenges and Complexities

If verification were a simple goal, automated verification systems would already exist and verification would likely be a normal part of software development. The verification problem is still an outstanding grand challenge because several research questions remain to be addressed. Some of these challenges are common to all software systems. For example, as with any

programming language, a verification system must be scalable in order for it to be usable. There are of course other challenges that are specific to the verification goal.

The generation of clauses equivalent to the full behavioral verification is the primary goal of our VC generator. This goal, when achieved, will provide confidence that an implementation fulfills the complete behavioral specification. It is to be noted that specifications can be provided in varying levels of detail. If full verification is unnecessary, the specification can be written with only the required details.

To provide any level of verification with confidence, the verification system must be sound and complete. A sound and complete verification system is dependent upon these qualities also existing in the proof rules used by the VC generator. With proper proof rules, the VC Generator can create VCs that if proved will guarantee full behavioral correctness of a software system with respect to its specification.

For a verification system to be practical, it must be scalable. This requires a mechanism for the verification of components. It is important that the verification process can focus on verifying a single component with the assumption that any supporting components have already been verified or will be verified separately. Developers should be able to create generic components. This allows data abstractions to be verified in such a way that they are correct independent of their parameters. For example, the ability to implement and verify the operations on a list of generic items as opposed to a list of integers will provide a more usable object. Component-based verification requires an implementation of a component, a specification of the component, and the specification of any used components to generate the VCs. Proving the generated VCs will also require mathematical units to complete the proofs. Scalability in verification is based on the same

ideas as in the traditional object oriented programming languages – where scalability is a result of component-based systems.

Mechanization is vital to the verifying compiler. In order for the verification system to be used by software developers, it needs to work, in most ways, like a normal compiler. This is achieved by modifying proof rules to be automatable and generating VCs that can be proved or disproved by a prover with no human interaction. Previous proof rules were designed for a theoretical VC generator that was not suitable for implementation and required modifications in order to mechanically generate VCs. The automation of the proof rules while maintaining soundness and completeness is an important contribution provided by this research. Of course, the VC generator still requires the developer to write specifications that must describe the behavior of the implementation. In addition, internal assertions may be required for more complex language mechanisms – such as **while** loops and recursion. However, no human interaction should be required for performing the proofs of the generated VCs. If the proofs are unable to be completed, it may be that new mathematical theorems are required. A mathematician will be needed to update the theorems. Of course, at other times, the VCs will be unproved as a result of flaws in the implementation which the developer will need to correct.

Scalability, full verification, and mechanization are three major issues that must be addressed by the RESOLVE verification system, and my work provides the clause generation for the system.

Scalability

Scalability is a necessary attribute of a language used to develop a large software system. Thus, in order for RESOLVE to be a practical alternative to current languages, RESOLVE must be scalable. RESOLVE is based on modular components. In object oriented languages, developers

build large systems by using components that they have not implemented and many times may not even have access to the implementation. For example, Java has a large library of components including lists, stacks, and queues which developers build upon in any major software system with access only to the specification. Similarly, RESOLVE developers will use components that have been implemented and verified previously and will build software in the already trusted object oriented method, using built-in and user defined components. An important distinction is that in RESOLVE developers will have formal specifications to use when building upon existing components instead of relying upon information descriptions provided in some other languages. Some developers may question whether the specifications will become overly complex and difficult to create as components get more complicated. However, just as object oriented languages today prevent excessive complexity by using previously written components, RESOLVE specs will remain similarly understandable through the use of suitable mathematical abstractions.

In order to provide scalable verification, the verification process must provide a method to reason about and to verify individual components. Figure 2 demonstrates the elements of concern when reasoning about correctness of components in a RESOLVE system. To prove the correctness of a new component realization, it is only necessary to use the specs for the components used by the new component, A and B. The corresponding implementations are not taken into consideration. In order to provide this separation of concerns, it is necessary that the specifications for each component contain enough information to reason about correctness. Each level of a software system can be individually verified. This allows implementations to be written and proved independently. The system as a whole is proved correct when each subsystem has been verified.

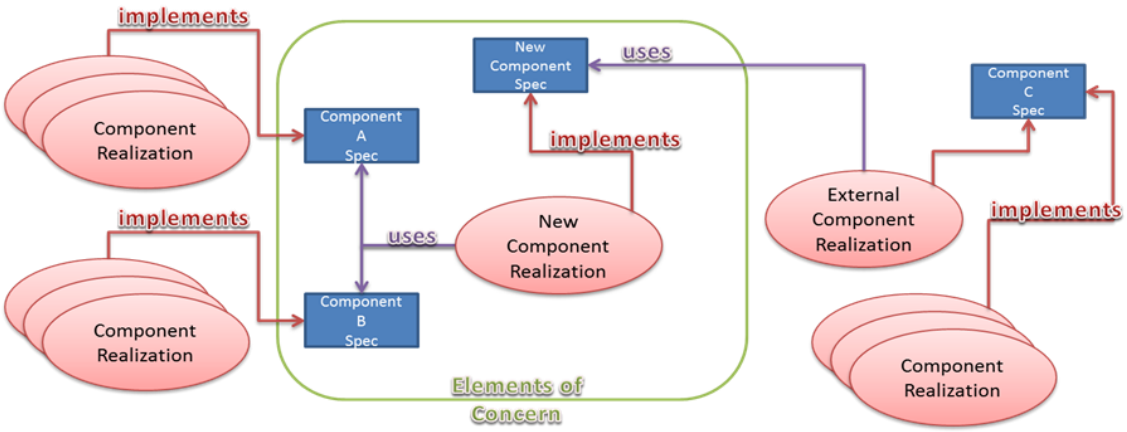


Figure 2: Modularity of Verification

Another element that is important to a verification system is the scalability of the proof rules themselves. It is not possible to have a VC Generator that can handle the majority of code if it is necessary to have a unique proof rule for each data type or different situation. It is important that using a common set of proof rules, VCs can be generated for code that both uses built-in components, like pointers and arrays, as well as user-built components. In order for this to be plausible, built-in types must be specified similar to how user-defined components are specified. Because of this requirement of built-in types, different implementations of pointers or arrays could be plugged into the system without the need to re-verify any software built on these specifications. Thus, in RESOLVE, the proof rules are uniformly applied to all data types. The same proof rules are used independent of the type of module it is used in: a component, an enhancement, or a facility.

Full Verification vs Verification of Properties

In order to provide full verification, the proof system for the RESOLVE language has been designed to be both sound and complete. We distinguish full verification from "lightweight

verification" that is based on lightweight specifications with the intent of checking only certain characteristics of the code. Lightweight verification can be used to check for common, specification-independent programming errors such as dereferencing null pointers [8] or unexpected cycles in pointer-based data structures [9]. Lightweight verification does not require specifications or internal assertions to prove the absence of these class of errors. Thus, other errors may remain. So, to prove the correctness of software (that the realization implements a complete specification of the behavior of the component), full specification and verification are necessary.

However, full behavioral verification does not mean that the specifications and assertions are going to be complicated. In fact, RESOLVE allows the specs to be as simple as the software developer desires. For example, a developer could write a spec for an operation such that the ensures clause is "**ensures** true." Then, no matter what is done within the operation, as long as the pre-conditions to any operations that are called are provable (which could be non-existent) and the procedure terminates, RESOLVE will verify that any implementation is correct. However, this obviously reduces any benefits of using RESOLVE to verify software. Thus, a practical method for developing specifications is to write them only as detailed as required. For example, a developer could create an implementation of a stack which does not require that the actual stack behavior is verified. It may be important however, that too many items are not added to the stack removed from the stack (i.e., it may be important to show the bounds of the stack are respected). In this example, as opposed to modeling the stack as a string of entries, the stack could be modeled as a number which corresponds to the number of items in the stack. Details of how a stack may be specified will be provided later.

Steps Toward Mechanization

In order for verification to be practical and reliable, it must be mechanical. As seen in the example of the binary search algorithm, non-mechanical verification (because of the many details it assumes that may or may not hold, when the ideas are mechanized) is prone to human error, even though the central ideas behind the code may be correct "in principle". Given an implementation annotated with suitable assertions, corresponding specifications, and appropriate theorems from mathematics, an automated verification system will postulate a correctness assertion, mechanically. The implementation will be deemed correct if and only if the correctness assertion can be proved, also mechanically.

The "principles" for a verification condition generator have also been around for numerous years, beginning with the work of Hoare [3] and King [1]. These principles have been enhanced in later research for verification code beyond simple procedures, such as the work of Krone [10] and Heym [11]. Just like there is a large gap between the idea of a binary search and its mechanical verification, there is a large gap between the previously known principles of verification and actual development of a mechanical VC generation system. Heym's dissertation that proposes the forward reasoning system (a complementary approach to the one taken in this dissertation) goes to considerable trouble just to establish the names of variables in different states, for example. So the proof rules in the efforts of Hoare and Krone are useful starting points, mechanization of VC generation practically requires re-development of every single rule. A significant contribution of this dissertation is in development of mechanizable rules that are also sound and relatively complete. This is an iterative process. For every rule, a new sound, mechanizable rule had to be devised, and when that mechanization wasn't adequate the rule had to be revamped and mechanized again.

There is a context in which every VC generation step takes place. For example, this context contains type information, specification of all operations within scope, and mathematical definitions (to be used by the prover). The proof rules and corresponding implementation must be developed to provide access to all information within the context. Several iterations of each proof rule were required for automation. The **If/Else** and **While** loop rules, for example, are discussed in detail in a later chapter. Among others, the condition in these statements must be converted from the condition statement in RESOLVE code to a mathematical statement in the VC, pre-conditions for nested function calls must be generated, and termination of loops and recursion must be proved all while maintaining soundness and completeness in the proof rules. In other circumstances, verification variables must be generated to indicate different states, auxiliary variables (specification variables defined by developers) must be processed to allow easier-to-complete proofs, and generic objects allowed for maintaining re-usability. These are but some of the complications involved in the automation of proof rules.

While there is no verification of the VC generator itself at this point and it is an important topic for future research, the generator has gone through considerable experimentation by us and others as detailed in this dissertation.

The RESOLVE VC generator expands upon a previously implemented RESOLVE-to-Java translator. Both the translator and assertion generator are based upon the visitor pattern and implemented in Java. RESOLVE is an object-based language and thus there are objects (concepts and corresponding concept specifications). In addition, there are facilities (where objects are instantiated) and enhancements (which allow a programmer to add extra operations to a concept that do not require access to private variables). After performing special processing depending on the source of the RESOLVE implementation and corresponding specification (concept versus

enhancement versus facility), the appropriate proof rules are applied in a goal-oriented fashion until all code is processed and a list of assertions are generated.

Currently, the verification condition generator mechanically generates and outputs assertive code that needs to be proved in order to show correctness. In addition to the generated assertions, an automated prover will use theorems located in mathematical libraries to prove the assertions. An earlier version of the VC generator produced Isabelle-friendly VCs. Isabelle is powerful and can prove many of the VCs, but it is intended to be used as a proof assistant, not as an automated prover. Thus, the focus of the current VC generator has been on the prover that has been specialized to work with the RESOLVE VCs. The VC Generator will output the assertions in a Goal-Given form that lists the Goal of the VC followed by all the assumptions that can be used in the proof under a header Givens. This Goal/Givens format makes it easy to see what needs to be proved. The VC generator also includes a verbose mode that shows each intermediate step in VC generation.

Thesis Statement

It is possible to generate provable verification conditions of correctness mechanically for full behavioral verification of object-based software in a modular fashion, one component at a time.

Scope of Research

The implementation of the VC generator has been a large effort and has included integrating many subtle details into the system. The basis of the VC generator is the mechanized version of proof rules for each language construct. Versions of these proof rules have been created from the non-mechanized versions which have been modified and implemented to develop the VC generator.

Although it is not a part of this work to attempt to verify the VC Generator, it is important to show that the process is reliable. First, to ensure the technical validity of proposed mechanizable proof rules we have shown the soundness and completeness of key proof rules in Chapter 3. To evaluate the implementation of the VC generation system based on the rules, VCs have been generated for a large library of reusable components, including a series of benchmarks (benchmarks #1 - #5) for verification discussed in [12].

The focus of the current research is on the verification condition generation process. The diagram in Figure 1 is the overall verification vision. However, this research will focus only on the VC generation process. This process will apply proof rules to annotated code and corresponding specifications to generate VCs of correctness. The VC Generator will be able to interface with the prover to provide the VCs, but will otherwise not be involved in the details of the proof process. The prover is specialized to process the VCs and currently the VCs that could be simplified by the VC generator were already easily proved by the prover. The VC Generator should behave in a similar way to a compiler in that it will provide details to allow the programmer to identify the location of errors when the prover is unable to prove a VC.

This VC generation process is based on the creation of sound, complete, and mechanized proof rules. This research will build upon the principles for creating proof rules for a RESOLVE-like language in [10]. As previously discussed, there is a wide research gap between principles of VC generation to the development of actual mechanizable proof rules for a language like RESOLVE. The process of mechanizing the current proof rules is an iterative process which occurred for each proof rule. The contributions of this dissertation are both research and development of mechanizable proof rules for object-based software verification, through several iterations of design and experimentation. Specifically, mechanizable proof rules for several specification and

implementation constructs, including the following, have been devised, implemented, and employed on a wide range of software components.

- Module-Level Verification
  - Handing of Generic Parameters
    - Types
    - Operations
    - Mathematical Definitions
  - Concept and Enhancement Specifications
    - Conceptual requirements
    - Provided types and constraints
    - Conceptual global variables
    - Operations
  - Realizations
    - Facility Declarations
    - Type representations
    - Global Variables
    - Representation invariants
    - Abstraction functions
    - Local operations
  - Facility Declarations
    - Instantiations with and without enhancements
    - Handling of different kinds of facility parameters
    - Long facility declarations

- Specification-Based Operation-Level Verification
  - Operation Implementation Declarations
  - Various specification parameter modes
  - Iterative and Recursive Procedures
  - Object Declarations
  - Control Constructs
  - Auxiliary Variables and Code
  - Calls to Operations with Functional and Relational Specifications

The remainder of this chapter contains a summary of research issues that arise for both module-level and operation-level VC Generation. For module-level verification, one such issue involves showing that the correspondence between the mathematical model and the implementation is valid. Also it is important to show that any parameters to the realization meet requirements for the parameter defined in the specification. At the operation level, some of the issues that are encountered include the process of context enrichment and code annotated with specification-only variables.

It is also important to handle global variables in the RESOLVE Verifier. As might be imagined global variables do cause difficulty in modularizing component-based verification. However, RESOLVE does permit the use of global variables. Each operation must specify if a global variable is updated. A realization that uses pointers and exemplifies the use of global variables will be shown in a later chapter. It is important that global variables are handled by the VC Generator, but there are still some issues that need to be researched concerning the use of global variables. The question of how best to handle global variables is still not resolved completely.

Some of the other complications that arise with verification are a result of how the specifications are written. Ideally, suitable mathematical developments will make it possible to minimize or avoid quantifiers in specifications. However, sometimes specifications may need to use

quantifiers. When existential quantifiers are unavoidable, there are programming constructs provided by RESOLVE to make automated verification possible. Auxiliary variables (in conjunction with auxiliary code) are specification-only variables (and non-executable code). These programming variables coincide with the mathematical variable in the existential quantification. The auxiliary code is used to set the value of the auxiliary variable which is then used to generate easier-to-prove VCs. Thus, the VC prover will not have to "guess" at the value of the specification variable.

While this research will indeed cover a vast territory in object-based software verification, by no means will it be exhaustive. For example, it will not cover all possible looping constructs or mutual recursion. Nor does it cover object-oriented features such as subtype-supertype relationships and related polymorphism. The treatment of global variables here is merely a starting point.

CHAPTER TWO

RELATED RESEARCH

Verification Research

This chapter summarizes specification and verification efforts that share some of the goals of the proposed research. A detailed survey of verification efforts may be found in [13] . Corresponding specification languages have been surveyed in [14].

"Why" is a software verification tool [15]. It is directed towards construction of functional programs with assertions, though imperative constructs such as iteration are available. "Why" focuses on typically built-in types, such as arrays rather than modularization or generic data abstractions. The "Why" system provides a mechanism to verify C, ML, and Java programs. The programs are converted into a "Why" internal language which is similar to ML. For example, Java programs annotated in JML are translated into "Why" by a program called Krakatoa. Krakatoa therefore represents another verification system for Java programs with JML specs [16]. "Why" generates VCs for various existing proof tools including Coq, PVS, Mizar, and HOL Light. "Why" also uses the Simplify [17] and haRVey [18] decision procedures to perform proofs. In general, "Why" disallows aliasing; however, Krakatoa models the Java heap so that once converted to "Why", there is no aliasing. There are portions of Java which are not handled by Krakatoa [19]. "Why" does not allow aliasing and only allows base types, arrays, and pointers, but no user defined types.

Some verification efforts are integrated whereas some others address specific aspects of verification. An integrated method of verification is based on refinement [20]. This process consists of refinement between levels of abstraction that are based on abstraction relations.

Starting from higher levels of abstraction (written as a specification) through refinement a correct lower level result (such as an implementable solution) is developed. Verification then becomes the process of checking the correctness of the refinement steps. The Vienna Development Model (VDM) is based on this process [20] [21]. Each step of refinement creates proof obligations that show the refinement process does not alter the meaning of the original specification. PVS is both a specification language and a theorem prover [22] [23]. The included specification language is based on higher order logic and provides a type system. The specification language is closely accompanied by an interactive proof system that together provides the ability to complete verification of large systems.

Model checking is often used as an alternative to full verification of behavior. Typically, the goal is to check whether (in the context of verification) the model (or implementation) has certain properties (the specification) [24]. Property verification is an area of model checking that verifies that certain specific characteristics (or properties) are evident in the implementation. An excellent summary of model checking efforts as well a specific system for model checking Java programs using JPF (Java Path Finder) can be found in [25]. Symbolic execution principles have been employed in SLAM, a model checking system for C programs [26]. Verification of safety specifications is an area of ongoing research in property verification.

Research into verification of existing languages must deal with situations, such as aliasing, that greatly complicate modular reasoning. Using Isabelle, a theorem prover [27], Verisoft provides an integrated system for full verification of C0 programs, a subset of the C language [28]. By design, C0 precludes several inherent verification difficulties that exist with the C language, such as aliasing. Correct pointer manipulation, on the other hand, is one of the goals of the ESC-Java effort [8]. Because Verisoft uses Isabelle, the proof process is interactive.

There is also much research on modular verification of object-oriented programs. Leino and Muller have both dealt with verification of pointer behavior for object-oriented programs [8] [29]. JML, short for Java Modeling Language, is a specification language for Java. In JML, subclasses must have stronger specifications (stronger post-conditions, weaker pre-conditions) than those of their superclass [30] [31]. Though the initial focus of JML has been on specification and run-time assertion checking, more recent efforts include verification. ESC/Java2 is used to perform static checks for JML [32]. In addition to ESC, there are other JML Static Verification Tools [33] [34] [16]. In order to provide soundness and completeness while using Java and dealing with aliasing, several approaches have been taken. ESC/Java discourages aliasing. JML, however, uses a model that only allows the "owner" of an object to modify the object. These approaches primarily result in the software developer being limited to a subset of Java.

A precursor to JML is Larch. Larch provides a two-tiered style of specification that requires specifications written in two languages: the Larch Interface Language and the Larch Shared Language [35]. Some programs specified using Larch have been checked using LP, the Larch Prover. LSL specifications are algebraic. The Larch project is no longer actively being maintained.

Jahob is a verification system where programs are written in a subset of the Java programming language using Isabelle as a specification language [36]. Jahob's goal is to provide static verification and ensure that a class of errors will not occur. Jahob uses shape analysis to assist in automatically determining invariants [37], [38]. Proofs are based on a variety of decision procedures and automated provers. Jahob attempts to choose the best procedure or prover for the proof at hand. As with other languages that use Isabelle, Jahob often requires interactive proofs.

A summary of many important verification efforts is given in Table 1. The table highlights how the RESOLVE effort differs from other verification systems. It is important to notice that only some of the languages allow for higher-order specifications. In order to reason about generic, user-defined types, it is necessary for a language to allow higher-ordered specifications. This allows specifications to take a mathematical function as a parameter and to be used in specifications. Of the languages that do allow higher-order specifications and provide full verification, none other than RESOLVE are automatic. In order to support full verification without restricting the language, the language needs to be defined with verification in mind. The characteristics of a language suitable for verification include maintaining efficiency and performance while supporting verification, handling the complexities that generics introduce, and the difficulties that come from aliasing. Many other languages that support verification have difficulties handling some of these challenges because the language was not designed for verification.

Effort	Implementation Language	Specification Language	Verification Type	Prover	Automation	Logic
Compcert	Clight, subset of C	Coq	Restrictive*	Coq	*Interactive*	Higher-Order
Dafny	Dafny (Research Language)	Dafny	Restrictive*	Boogie/Z3	Automatic	*First-Order*
Eiffel	Eiffel	Eiffel	*Verification of Properties*	*(Currently only runtime checks)*	Automatic	*First-Order*
ESC/Java	Java	JML	*Verification of Properties*	Simplify	Automatic	Higher-Order
Jahob	Subset of Java	Subset of Isabelle	Restrictive*	Isabelle, Others	*Interactive*	Higher-Order
JML/ FSPV	Java	JML	Restrictive*	Isabelle/Simpl	*Interactive*	Higher-Order
Larch	Various Languages	Larch	Restrictive*	Larch Prover	*Interactive*	*First-Order*
PVS	PVS	PVS	Full	PVS	*Interactive*	Higher-Order
SPARK Ada	Subset of Ada	SPARK	*Verification of Properties*	SPADE	*Interactive*	Higher-Order
Spec#/ Boogie	Spec #, Extension of C#	Spec #	*Verification of Properties*	Boogie/Z3	Automatic	*First-Order*
VDM-SL/ VDM++	Various Languages	VDM	Full	VDM	*Interactive*	Higher-Order
Verisoft	Subset of C, C0	Isabelle	Restrictive*	Isabelle	*Interactive*	Higher-Order
Why	C, Java, ML	Why	Restrictive*	Isabelle and Coq	*Interactive*	*First-Order*
πVC	C-like language, PI	πVC	Restrictive*	πVC	Automatic	*First-Order*
RESOLVE	RESOLVE	RESOLVE	Full	RESOLVE or Isabelle auto	Automatic	Higher-Order

*These languages are listed as restrictive instead of full because they have restricted the capabilities of the language in order to provide full verification.

Table 1: Related Works table

Alternative Verification Condition Generation Approaches

Two different approaches to generating VCs have been examined in the literature [10] [11]. Until now the research has been primarily theoretical with no actual mechanized VC generator. A brief discussion of the two approaches will follow.

The Hoare-style, goal-oriented, approach in [10] begins with the goal and modifies the goal with each program statement until the ultimate the goal (or assertion) to be proved is generated. The other method is a tabular method in which verification assumptions and obligations are

documented for each statement of code [11]. The VC generator described in this proposal implements the goal-oriented approach. A RESOLVE verification system implementation in development at The Ohio State University is based on the tabular generation method. [11]

The simple example in Figure 3 will be used to demonstrate the differences. The only explanation that should be necessary to understand this code is to describe the :=: symbol. This symbol is the 'swap' statement and will swap the values in the two variables. Further details on the swap statement and the RESOLVE language will be explained later in this chapter.

```
Assume true;
If J > I then
   I:=:J
end;
If K > I then
   I:=:K
end;
Confirm I >= J ^ I >= K;
```

Figure 3: An example of Assertive Code

Tabular Reasoning Approach

The tabular reasoning approach forms as its name implies a table. The table is split into states and shows how the variables change by numbering them in each state. As seen in Figure 3, the table is formed to include the assumptions, requirements, and path conditions. One of the research problems that develop from this approach is how to best eliminate unnecessary statements and simplify out extra variables. The principles behind this approach are described in Heym's thesis [11].

The tabular method is straight-forward and thus may be the preferred method to use when introducing verification to students. When reading the table below the variable numbers are based on the value of at a certain state (or line number) in the code. So, for example, 1 is the state of the

variable after the first line of code. At line two, the table shows that the assumption is "I2=J1 and J2=I1 and K2=K1." This is because the value of K did not change but I and J were swapped. These assumptions are key as they are used to prove the obligations. The obligations must be proved to show correctness of the code. Obligations result from pre-conditions, invariants, and of course as in this example, the final Confirm clause (or goal), "I6 >= J6 ^ I6 >= K6".

State	Path Conditions	Assumptions	Obligations
0		True	
	If J > I **then**		
1	J1 > I1	J1=J0 and I1 = I0 and K1=K0	
	I:=:J		
2	J1 > I1	I2=J1 and J2=I1 and K2=K1	
	End;		
3.1	J1 > I1	I3=K2 and K3=I2 and J3=J2	
3.2	~(J1 > I1)	I3=I0 and K3=K0and J3=J0	
	If K > I **then**		
4	K4 > I4	J4=J3 and I4 = I3 and K4=K3	
	I:=:K		
5	K4 > I4	J5=J4 and I5 = K4 and K5=I4	
	End;		
6.1	K4 > I4	J6=J5 and I6 = I5 and K6=K5	I6 >= J6 and I6 >= K6
6.2	~(K4 > I4)	J6=J3 and I6 = I3 and K6=K3	I6 >= J6 and I6 >= K6

Table 2: Reasoning Table

The final assertions to be proved by the tabular method are as follows:

- (((J1 > I1) => (J1=J0 and I1 = I0 and K1=K0)) and ((J1 > I1) => (I2=J1 and J2=I1 and K2=K1)) ^ ((J1 > I1) => (I3=K2 and K3=I2 and J3=J2)) and (~(J1 > I1) => (I3=I0 and K3=K0and J3=J0)) and ((K4 > I4) => (J4=J3 and I4 = I3 and K4=K3)) and ((K4 > I4) => (J5=J4 and I5 = K4 and K5=I4)) and ((K4 > I4) => (J6=J5 and I6 = I5 and K6=K5))) => (I6 >= J6 and I6 >= K6)

- (((J1 > I1) => (J1=J0 and I1 = I0 and K1=K0)) and ((J1 > I1) => (I2=J1 and J2=I1 and K2=K1)) and ((J1 > I1) => (I3=K2 and K3=I2 and J3=J2)) and (~(J1 > I1) => (I3=I0 and K3=K0 and J3=J0)) and ((K4 > I4) => (J4=J3 and I4 = I3 and K4=K3)) and ((K4 > I4) => (J5=J4 and I5 = K4 and K5=I4)) and (~(K4 > I4) => (J6=J3 and I6 = I3 and K6=K3))) => (I6 >= J6 ^ I6 >= K6)

When using the tabular approach, there is no need to generate new names for variables that are not affected by a certain statement. If this aspect is mechanized, then unnecessary names would disappear, simplifying the VCs. This tabular approach appears ideal for teaching about verification condition generation. It is easy to explain, understand, and perform manually. However, as this approach is not focused on the actual goal to be proved, it does not simplify out assumptions that will be unnecessary.

Goal-Oriented Reasoning Approach

The goal-oriented approach forms an assertion by modifying the goal based on each line of code moving in the opposite order from which the code will be executed. A proof system consists of proof rules for each statement or construct in a language. Given the goal and code of an implementation, the verifier applies proof rules (which replace code with mathematical assertions) and then simplifies the assertions with the objective of reducing the final assertion to "true." For example, consider the following piece of assertive code (a combination of code, facts, and goals), also called a Hoare triple. In the example, S and T are two Stack variables. The swap statement exchanges the values of the participating variables, without introducing aliasing. All code is written and verified within a context, and the Context (indicated in the proof rule with C/)

here includes mathematical String_Theory, the Stack_Template specification, as well as declarations of Stack variables. It is not explicitly listed in this proposal.

```
C\
Assume S = empty_string;
T :=: S;
Confirm T = empty_string;
```

To simplify the assertive code, a proof rule for the swap statement needs to be applied. In the rule shown below, it is necessary and sufficient to prove what is above the line to prove what follows below the line. This is the typical format of a formal proof rule. In the rule, *C* stands for Context. The notation, RP[x⇝y, y⇝x], means that concurrently, every x is replaced with y and every y is replaced with x. Intuitively, this rule means that to confirm what follows after the swap statement, the same assertion needs to be confirmed before the swap statement, but with x and y exchanged in the assertion. For the verifier to apply the rule mechanically, the swap statement in the rule is preceded by "code" which denotes 0 or more statements.

```
Proof Rule for the Swap Statement:
C\ code; Confirm RP[x⇝y, y⇝x];
_____________________________
C\ code; x :=: y; Confirm RP;
```

After the application of the swap rule, the following assertive code remains:

```
Assume S = empty_string;
Confirm S = empty_string;
```

The next statements to be processed by the verifier are **Assume** and **Confirm** clauses. The rule for removing the **Assume** clause has the effect of making the resulting assertion an implication. The rule for handling the **Confirm** clause is simply syntactic: eliminate the keyword **Confirm**.

```
Assume Rule:
C\ code; Confirm IP implies RP;
_____________________
C\ code; Assume IP; Confirm RP;

Confirm Rule:
C\ RP;
_____________________
C\ Confirm RP;
```

In our example, after the application of the **Assume** rule, we have the following assertion:

```
Confirm S = empty_string implies S = empty_string.
```

Subsequently following the application of the **Confirm** rule produces the final assertion:

```
S = empty_string implies S = empty_string.
```

Since this implication is true based on mathematical logic, the assertion can be simplified to:

```
true.
```

Thus, we can see that our final assertion is true; therefore, assuming the soundness of the proof rules we have employed, we can conclude that the original assertive code is correct.

To return to our previous example which we used to demonstrate the tabular method, the following table shows the process of generating the VCs using the goal-oriented method.

Step 1: Apply If_Then Rule			
Assume true; **If** J > I **then** I:=:J **end If;** **Assume** K > I I:=:K **Confirm** I >= J ^ I >= K;		**Assume** true; **If** J > I **then** I:=:J **end If;** **Assume** ~(K > I) **Confirm** I >= J ^ I >= K;	
Step 2: Apply Swap and Assume again			
Assume true; **If** J > I **then** I:=:J **end If;** **Confirm** (K > I) => (K >= J ^ K >= I);		**Assume** true; **If** J > I **then** I:=:J **end If;** **Confirm** ~(K > I) => (I >= J ^ I >= K);	
Step 2: Apply If_Then Rule again			
Assume true; **Assume** J > I I:=:J **Confirm** (K > I) => (K >= J ^ K >= I);	**Assume** true; **Assume** ~(J > I) I:=:J **Confirm** (K > I) => (K >= J ^ K >= I);	**Assume** true; **Assume** J > I I:=:J **Confirm** ~(K > I) => (I >= J ^ I >= K);	**Assume** true; **Assume** ~(J > I) I:=:J **Confirm** ~(K > I) => (I >= J ^ I >= K);
Step 3: Apply Swap Rule:			
Assume true; **Assume** J > I **Confirm** (K > J) => (K >= I ^ K >= J);	**Confirm** true => (~(J > I) => ((K > I) => (K >= J ^ K >= I)));	**Assume** true; **Assume** J > I **Confirm** ~(K > J) => (J >= I ^ J >= K);	**Confirm true** => (~(J > I) => (~(K > I) => (I >= J ^ I >= K)));
Step 4: Apply Swap Rule:			
Confirm true => ((J > I) => ((K > J) => (K >= I ^ K >= J)));		**Confirm** true => ((J > I) => (~(K > J) => (J >= I ^ J >= K)));	

Table 3: An application of the goal oriented approach

The final assertions from the goal-oriented approach to be proved are as follows:

- `true => ((J > I) => ((K > J) => (K >= I ^ K >= J)));`
- `true => (~(J > I) => ((K > I) => (K >= J ^ K >= I)));`
- `true => ((J > I) => (~(K > J) => (J >= I ^ J >= K)));`
- `true => (~(J > I) => (~(K > I) => (I >= J ^ I >= K)));`

One advantage of the goal-oriented approach is in the cases where the goals that need to be proved are weaker than what can be proved. The VCs can be generated in such a way as to simplify the proof process based on possibly modifying the goal with each step leading to an easier proof as opposed to appending more assumptions as in the tabular approach. Krone's dissertation explains the principles behind this approach of VC Generation [10].

RESOLVE Background

The RESOLVE Verification Condition Generator has benefitted from many years of previous verification research by the RESOLVE Research Community. RESOLVE is an object-based language designed to support verification [39]. In order to verify software, the code includes mathematical notations which specify the behavior of the software. There are several important characteristics of RESOLVE which provide the ability to generate verification conditions which are strong enough to provide full verification, yet not so complicated that they are too difficult for a proof system to prove. These characteristics include the design of the specification language and facilities for object-based design. RESOLVE also provides "clean" semantics [40] which allows programmers to avoid aliasing. To be clear the use of the term object-based instead of object oriented is to distinguish RESOLVE in some specific aspects. Expressly, RESOLVE does not claim to solve issues concerning inheritance or polymorphism. In addition, RESOLVE only provides static typing.

RESOLVE allows (but does not require) programmers to provide a specification of the results of every implementation. RESOLVE requires that these specifications are written in a formal notation. The ultimate goal of writing the detailed specification is to prove that an implementation meets the specification. With these formal specifications, the confusion that comes with informal specifications can be avoided and mechanical methods can be used to process the specification and an implementation.

Concept Specifications and Implementations

RESOLVE has been designed as an object-based specification and programming language to provide a scalable verification solution. A few example modules are useful to demonstrate the RESOLVE language. A commonly used object in many object-oriented languages is the stack, so we begin with a specification of the Stack_Template concept in Figure 4 . An actual realization of Stack_Template is in a different file and there could actually be multiple implementations that could meet this specification. An array-based implementation of Stack_Template is also provided below in Figure 6.

A concept specification provides a mathematical model for the type specified by the concept and formally defines the behavior of each operation defined in the concept used to manipulate variables of the defining type. In Figure 4, the Stack type is modeled by a mathematical string of entries. In Stack_Template, S is introduced as an exemplar which is used to demonstrate the behavior of a generic Stack variable in this type portion of the specification. A concept provides initialization details, constraints for the variables of the type, and specifications for each operation. The initialization statement defines the initial state of a variable of the concept's type. The spec defines that a Stack is initially empty. Understanding that Stack is modeled as a string, the concept defines the initial value of Stack as the empty string. The constraint clause formally

describes that every Stack will always be constrained to be within bounds. This bound is based on the parameter Max_Depth which must be provided upon instantiation of the Stack_Template. The constraint requires that the length of the string must be less than Max_Depth.

```
Concept Stack_Template(type Entry;
                       evaluates Max_Depth: Integer);
    uses Std_Integer_Fac, String_Theory;
    requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = empty_string;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures  S = <#E> o #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires |S| /= 0;
        ensures #S = <R> o S;

    Operation Depth(restores S: Stack): Integer;
        ensures Depth = (|S|);

    Operation Rem_Capacity(restores S: Stack): Integer;
        ensures Rem_Capacity = (Max_Depth - |S|);

    Operation Clear(clears S: Stack);

end Stack_Template;
```

Figure 4: Stack Template

A Stack is specified as a mathematical string of entries and this is used to define the requirements for the various Stack_Template operations. A few more details are useful in understanding the specification. The pound (#) sign on a variable in the ensures clause indicates the value passed into the operation. A variable without the pound sign refers to the variable when the operation returns. The parameter modes (i.e., updates) which precede each variable type and name in the operation definition is a part of the specification which describes how each parameter will (or will not) change during the operation. The specifications are supported by mathematical theories.

RESOLVE requires a math library that provides the mathematical definitions needed to prove the generated assertions. The specification of arrays can be found in Appendix B.

Appendix D contains the specification of a stack modeled as a natural number and a stack specified as a string of entries. In addition, the code and the VCs generated for a stack reversal based on a stack modeled as a natural number as well as the stack modeled as a string can be found in this appendix. The VCs generated for the stack modeled as a natural number are much easier to prove than the VCs generated for the stack modeled as a string.

One aspect of the specification that deserves a little more detailed explanation is the parameter modes. RESOLVE supports several parameter modes each of which is useful in certain types of specifications. These ease human comprehension, and in some cases they make specifications simpler because proof obligations are automatically generated based on the modes of each parameter without the programmer having to add additional conjuncts to the specifications. The *updates* parameter mode is probably the most 'general' parameter mode. It means that the value of the parameter coming into the operation may be updated in some way. If important, the precise manner in which the parameter is updated will be defined by the ensures clause. Other parameter modes are *evaluates*, *replaces*, *restores*, *preservers*, *alters*, and *clears*. *Restores* and *Preserves* seem initially to define the same behavior; the distinction is that *preserves* is a statically checked mode that will prevent the variable from being modified by the implementation whereas *restores* will allow the value to change as long as it is always restored back to the original value. *Clears* requires that at the end of the operation the value is set to the initial value for the variable type. *Replaces* indicates that the incoming value of the variable is not important and cannot be guaranteed. *Alters* indicates that the outgoing value of the parameter is not important and cannot

be guaranteed. Finally, *evaluates* indicates that the incoming parameter is an expression that can be evaluated. If no mode is specified *alters* will be assumed.

An operation's specification should be viewed as a contract between the client and implementer. The pre-condition (or requires clause) must be true prior to a call of any operation. In this example, the Push operation requires that there is room in the stack for another element. Similarly, in order to guarantee correct functionality, the Pop operation requires that there is at least one element in the Stack. The operation's implementation must satisfy the post-condition (or ensures clause) at the end of the procedure if the pre-condition holds. In Stack_Template, the ensures clause for Push provides the guarantee that S is updated so that it becomes the original value of E (a parameter of Push) concatenated with the original value of S. Pop removes the top entry from the parameter S and replaces the parameter R with the top entry.

An implementation of Stack_Template realized with static arrays follows in Figure 6. The array is initialized to the maximum size of the stack and there is an additional variable, top, which indexes the top of the stack in the array. The specification for the RESOLVE array is provided in Appendix C. The :=: syntax in the realization is the swap operator. Operations to permit swapping an entry with an entry that is in an array are provided in the Static_Array_Template. Although these operations are indicated using the normal swapping syntax (:=:) these are actually replaced by a call to Swap_Entry (also defined in the Static_Array_Template). Swap_Entry is then processed as a normal operation with normal pre and post-conditions. This differs from the behavior of the default swap operator which swaps any two objects of the same type.

Swapping is an important aspect of making RESOLVE a clean language. A clean language allows users to reason about components modularly without the problems that come with aliasing. Excluding references and aliasing from the semantics of a language simplifies the reasoning

process and allows for true abstraction. Kulczycki's dissertation details the use of clean semantics in the RESOLVE language [40]. RESOLVE implements swapping which has allowed the removal of pointers and references without losing the benefits of pointers. In addition, swapping is the mechanism used for passing parameters to an operation. Primarily, pointers are used to implement efficient copying of components in regards to both time and space. However, copying pointers creates reasoning difficulties by changing values of other variables when only one variable is modified. Often, programmers do not really want or need to copy an object, but need to move it to a different location (into a different variable). Thus, RESOLVE incorporates swapping. Swapping allows the programmer to swap two variables which maintains the efficiency of pointers, but retains the ability to reason about the two variables as two distinct entities, i.e. changing one will not change the other. Further discussion of the importance of swapping in designing verified software can be found in [41].

The **conventions** clause of this realization of Stack_Template documents the *representation invariant*s. The invariants must be true at the boundary of each external operation (i.e., an operation that is specified in the concept), including at the end of initialization and beginning of finalization. For the inductive proof that the given conventions are indeed representation invariants, the conventions are first proved to be true at the end of initialization for the base case. For the inductive case, the conventions are assumed to be true at the start of each (external) operation and will be verified to be true at the end. The VC generator will produce a proof obligation to establish the conventions for each external operation as a part of the proof process. The conventions for this implementation indicate that S.Top is less than Max_Depth. Thus, S.Top must be a valid index into the array.

Given the conventions defining which representation values in the implementation are valid, the **correspondence**, specified in the realization, relates them to the mathematical model. The correspondence assertion may be an *abstraction function* or *relation* between the representation space and the specification space.This relationship must be well founded. This means that the correspondence must relate all legitimate representation values (i.e., values that satisfy to conventions) to legitimate abstract values (i.e., values that satisfy the constraints on the abstract model). Figure 5 models the conventions and correspondence relationship. The VC generator will generate a proof obligation to establish that the correspondence is well founded. Here, the correspondence specifies that the conceptual (specification) version of the Stack, Conc.S is equal to the reverse of the concatenation of each element of the array from the first element to S.Top. The implementation is intuitive in that push increments S.Top and then adds the element to that index in the array. Pop removes the top entry from the array and then decrements S.Top.

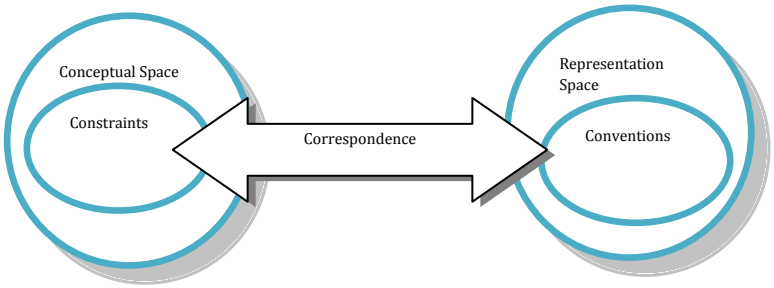


Figure 5: Relationship of Correspondence, Constraints, and Conventions

```
Realization Array_Realiz for Stack_Template;

    Type Stack is represented by Record
        Contents: Array 1..Max_Depth of Entry;
        Top: Integer;
    end;
    convention
        0 <= S.Top <= Max_Depth;
    correspondence
        Conc.S = Reverse(Concatenation i: Integer
            where 1 <= i <= S.Top, <S.Contents(i)>);

    Procedure Push(alters E: Entry; updates S: Stack);
        S.Top := S.Top + 1;
        E :=: S.Contents[S.Top];
    end Push;

    Procedure Pop(replaces R: Entry; updates S: Stack);
        R :=: S.Contents[S.Top];
        S.Top := S.Top - 1;
    end Pop;

    Procedure Depth(preserves S: Stack): Integer;
        Depth := S.Top;
    end Depth;

    Procedure Rem_Capacity(preserves S: Stack): Integer;
        Rem_Capacity := Max_Depth - S.Top;
    end Rem_Capacity;

    Procedure Clear(clears S: Stack);
        S.Top := 0;
    end Clear;

end Array_Realiz;
```

Figure 6: Array Based Stack Implementation

RESOLVE is an object-based language in that it provides a modular process to support verification. Thus, with only the specification of a component, a programmer can implement a second component relying only on the specification of the first component. This provides a scalable way to verify software (just as object oriented languages provide a scalable manner to implement large software systems).

Enhancement Specifications and Implementations

Another aspect of RESOLVE is the enhancement of a concept (or an object), which is a form of specification inheritance. A language needs a mechanism to extend the primary or core set of operations. Secondary operations are not integral to the concept and can be realized in terms of the primary set of operations. Enhancements to a concept provide the ability to define secondary operations.

An example enhancement is one for a stack reversal. The following implementation and specification of a reversal builds upon the operation in the Stack_Template without requiring knowledge of or allowing access to implementation details. VCs will be generated for this Stack_Template enhancement. The verifier may assume the Stack constraints and requirements but are not reliant on an actual implementation of Stack_Template. The specification contains the signature for the operation Flip. Flip takes one Stack as parameter, S, and the ensures clause states the resulting Stack is equal to the reverse of the incoming Stack.

This implementation of Flip contains a **While** loop in which the operation iteratively pops an item off the incoming Stack, S, and pushes it onto S_Flipped, another Stack. In order to support full verification, **While** loops require programmer-provided assertions in order to determine the behavior of the loop and show termination of the loop. The loop invariant is provided in the maintaining statement. In this example, the developer is stating that each time through the loop the reverse of S_Flipped when concatenated with the current Stack, S, will be equal to the original value of S. VCs will be generated to show the invariant is true in the base case (the first time through the loop) and in subsequent cases. The changing statement is used to simply the invariant. In a situation where some variables are not changed by the loop, by leaving these variables out of the changing clause, the developer does not have include the fact that these

variables are not modified in the maintaining statement. The decreasing clause requires an ordinal that will decrement each time through the loop. The decreasing clause is required to show termination of the loop. A VC will be generated to show that the decreasing clause does decrease with each iteration.

```
Enhancement Flipping_Capability for Stack_Template;
   Operation Flip(updates S: Stack);
         ensures S = reverse(#S);
end Flipping_Capability;

Realization Obvious_Flip_Realiz for Flipping_Capability of
Stack_Template;
   uses Std_Boolean_Fac;
   Procedure Flip(updates S: Stack);
         Var S_Flipped: Stack;
         Var Next_Entry: Entry;
         While (Depth(S) /= 0)
               changing S, S_Flipped, Next_Entry;
               maintaining #S = reverse(S_Flipped) o S;
               decreasing |S|;
         do
               Pop(Next_Entry, S);
               Push(Next_Entry, S_Flipped);
         end;
         S_Flipped :=: S;
   end Flip;
end Obvious_Flip_Realiz;
```

Figure 7: Stack Flip Specification and Realization

Facility Specifications and Implementations

Before a client can use the Stack_Template or an enhancement, they must first instantiate it. Shown below is an example facility declaration. VCs must be generated for facility declarations to show that parameters to the concepts, enhancements, and their realizations meet the defined requirements.

```
Facility Rev_Stack_Fac is Stack_Template(Integer, 4)
      realized by Array_Realiz;
  enhanced by Reversal_Capability
   realized by Obvious_Rev_Realiz;
```

Figure 8: Stack Facility enhanced with Reversal Capability

In addition to facility declaration, RESOLVE has a facility construct that allows construction of one-of-a-kind software modules (that combine specifications and implementations), and are not designed for reuse. Such a construct is also useful to develop a 'main' module found in other languages. An example facility is below.

```
Facility Stack_Flip_Facility;
    uses Std_Boolean_Fac, Std_Character_Fac,
         Std_Integer_Fac, Std_Char_Str_Fac;

    Facility Stack_Fac is Stack_Template(Integer, 4)
            realized by Array_Realiz
         enhanced by Flipping_Capability
       realized by Obvious_Flip_Realiz;

    Operation Main();
    Procedure
       Var S: Stack_Fac.Stack;
       Var C, D, E, F: Integer;

       C := 150;
       D := 300;
       E := 7;
       F := 9;

       Push(C, S);
       Push(D, S);
       Push(E, S);
       Push(F, S);

       Flip(S);

       Pop(C, S);
       Pop(D, S);
       Pop(E, S);
       Pop(F, S);

    end Main;
end Stack_Flip_Facility;
```

In this example, the “long” facility Stack_Flip_Facility creates a (short) facility of stack of Integers, enhanced with the stack Flip operation.

CHAPTER THREE

MECHANIZATION AND SEMANTICS OF PROOF RULES

In order to generate verification conditions (VCs), there must be mechanizable proof rules that convert the RESOLVE software implementation and specification into assertions which correspond to the correctness of the software. If all the VCs are provable, the software is correct. If one of the VCs is unprovable, then software correctness is not established. It is important to emphasize that the VC generation process for RESOLVE is unique, because unlike other programming languages, RESOLVE includes mathematics as a part of the language and was developed with verification and efficiency in mind using principles of software engineering. Also, it handles modularity, generics, and typically built-in data abstractions such as arrays and pointers in a unique component-centric way, so the challenges of generating VCs in RESOLVE are different in comparison with other VC generators.

It is well known that the general verification problem of software correctness is undecidable. Provability of the VCs relies on mathematical theories that are undecidable. So the goal of verification of correctness of all software is unachievable. However, in practice, the reason for why verification of VCs generated to prove software fails has little to do with undecidability [42]. In fact, for much of the software we will ever write, verification fails because of poor code or annotations, not because of undecidability; in the remaining chapters, we will demonstrate practical verification condition generation on numerous examples over a variety of mathematical spaces.

Two necessary characteristics of the verification system are soundness and completeness. Soundness is defined to mean that if the verifier reports the code is correct, that the code is in fact valid (where validity is defined by the semantics). Completeness means that if the code is valid,

the verifier will report that the code is correct. In our verification system, the actual goal can only be relative completeness, because of the incompleteness of any non-trivial mathematical theory upon which the proofs will rely. In order to show soundness and (relative) completeness for a basic set of mechanized proof rules, the semantics of RESOLVE are presented in this chapter.

Semantic Foundations

The RESOLVE semantics formally describe the meaning of RESOLVE software beyond what implementations do when they are executed. The semantics capture the behavior of the program by describing transformations from one (or more) state(s) to other state(s). There are "normal" states and special "stuck" states. In a "normal" state, the semantics provide a function from variable names to their values. The "stuck" states are special states from which the program cannot move to a normal state. Thus, once in a "stuck" state, the state is "stuck" in the special state. We use three stuck states, named, Manifestly Wrong (**MW**), Vacuously Correct (**VC**), and Bottom (⊥). To distinguish the abbreviation for Vacuously Correct from Verification Condition, **VC** usage will be bold when indicating Vacuously Correct. For consistency, **MW** is a "bad" state that indicates the code is wrong. An example that would cause a transformation of a calling code into the **MW** state is when a pre-condition of a called operation is violated in that code. **VC** is a state where the code is correct because of, for example, a false assumption. This is because we are assuming something that is false, so based on the principles of logic the result is true. For example, in a calling code if a called operation's implementation does not satisfy its post-condition, then the calling code will reach the **VC** state. The bottom state is entered when the code does not terminate.

Another important concept to understand is what it means for code to be valid. This is based on the type or strength of the post-condition defined in the operation. The semantics can either respect or ensure the post-condition. If an operation respects the post-condition then if the code terminates, it must satisfy the post-condition. In this case, code is defined as valid if the program does not end in **MW**. Using the normal notion of validity, which requires that the code terminates, validity is defined as not ending in either **MW** or ⊥. Now, we are ready to discuss RESOLVE semantics in more formal terms.

```
Def. St: Id→Set∪{MW, VC, ⊥} =
( Id→( Var_Dom) ) ∪ {MW, VC, ⊥} );

Def. env: Id -> ( Var_Dom ⋃ Oper_Dom ⋃ Facility_Dom ⋃
                  Realization_Domain ⋃ Concept_Domain) )
…
```

In the description above, St is a state which provides a mapping from the variable names (the set Id) to the corresponding set of values as well as the three special stuck states. ∪ means the union in which all sets are completely not intersecting.

The second definition in the semantics above defines the environment (env) which includes the union of the domains of all variables, procedures, facilities, realizations, and concepts. In classical semantics, only the domain of variables is included. Because RESOLVE uses the specifications components and operations instead of the actual components and operations to define the behavior of an operation, these extra domains are necessary. This allows the verification system defined by RESOLVE to maintain modularity by not relying on implementation of underlying components but only the specifications. The environment corresponds to the context that is maintained by the proof rules.

Relational Semantics

Any language in which specifications of operations are allowed to be relational (i.e., one of many outputs is possible for a given input), relational semantics are necessary. Because RESOLVE allows operations with such relational specifications, each statement is potentially a relation that relates a state (or set of states) to another state (or set of states). If any of the states in which a piece of code may end in is **MW**, the code is invalid. The following example should demonstrate why relations are necessary. In this example, the Push procedure call alters Next_Entry; the state before and after the call to Push are connected by a relation. This is because the value of Next_Entry varies based on the implementation of Pop and Push. Based on the specification, Next_Entry can have any value. We have used the Push example here only because we have already discussed its specification. Any operation that is specified to return one of several values (e.g., one that captures an optimization problem) will serve the purpose.

```
// Excerpt from a Stack Flip
While (Depth(S) /= 0)
      ...
do
      Pop(Next_Entry, S);
      Push(Next_Entry, S_Flipped);
end;
```

In the same way operation specifications may be relational, the correspondence of a data abstraction in RESOLVE may also be an abstraction relation [43] [44]. The correspondence relation, specified in the implementation, relates the concrete representation space (in the implementation) with the abstract conceptual space (in the specification). Figure 9 shows the implications of relational correspondences and operation specifications on the semantics. In the diagram, the value in the realization space, R, is a concrete value before the operation P is called. In the conceptual space, there are multiple abstract values to which R may be related to,

according to the correspondence relation. All the related values in the conceptual space must meet this pre-condition. The code for the operation may have relational behavior, denoted by Sem_R[code] in the figure. So it is possible that R will be related to one of multiple implementation values after the call. In this example, suppose that the actual value after execution of the code for P is S. Once again, because the correspondence is relational, there may be multiple corresponding values in the conceptual space. For the code to be correct, any possible value, say D, after the operation is called must satisfy the post-condition of P. Furthermore, all such values must be within the set of values that follow from the abstract pre-state of operation P; in particular, the precursor for D should be C or C' that satisfies the precondition of P and is within the envelope of values allowed by the correspondence from the concrete representation value R. This dissertation does not attempt to formalize this relational semantics for RESOLVE, but it is an important future direction.

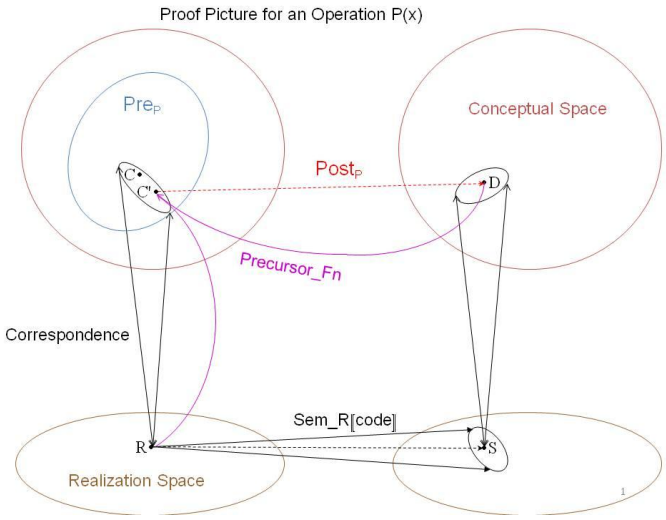


Figure 9: Implications of Relational Operation and Correspondence Specifications

Formal Semantics

The semantics of RESOLVE are defined using an inductive definition on a sequence of statements in the description below. The base case, case (i) below, considers the condition when the sequence is empty. In this case, the state does not change; the state before the sequence of statements, S, is the same as the state after the statements, T, and they are related by the semantic relation, denoted by Sem_R. Next, we will consider the case when the sequence contains valid RESOLVE statements.

Now we discuss case (ii). To be clear, the state R lies before all the assertive code. S is the state before the statement of interest and T is the 'current' state after the statement of interest. In the case where valid assertive code lies between states R and S, *if* S is already a stuck state, *then* the state T will be the same state as S. However, if S is not in stuck state, the modifications to the state vary based upon the statement type. This is the focus of the rest of the discussion.

Inductive Definition on code: Stmnt_Seq **of**
(S: St) Sem_R⟦code⟧ (T: St): 𝔹 **is**

i. S Sem_R⟦Λ⟧ T = (T = S)

ii. **if** R Sem_R⟦code⟧ S **then if** S ∈ {**MW**, **VC**, ⊥} **then** T = S
else

(a) R Sem_R⟦code; x :=: y⟧ T = (∀ ξ: Id, **if** ξ ∉ {x, y}
then
T(ξ) = S(ξ) **and**
T(x) = S(y) **and**
T(y) = S(x));

(b) R Sem_R⟦code; **If** BE **then** code1 **else** code2⟧ T = (
If not Invk_Cond(BEx),
then T = **MW**
Else[Val(BE, S) **and** S Sem_R⟦code1⟧ T **or**
¬Val(BE, S) **and** S Sem_R⟦code2⟧ T**]**);

(c) R Sem_R⟦code; x := f(y, exp)⟧ T = (
If not Invk_Cond(f(y, exp)),
then T = **MW**
Else[∀ ξ: Id, **if** ξ ≠ x **then** T(ξ) = S(ξ)
and T(x) = Val(f(y, exp), S) **]**);

(d) R Sem_R⟦code; op(**updates** y)⟧ T = (
If not Invk_Cond(op(y)),
then T = **MW**
Else[∀ ξ: Id, **if** ξ ≠ y **then** T(ξ) = S(ξ)
and T(y) Op_Rel S(y) **]**);

(e) R Sem_R⟦code; **Confirm** Q⟧ T = ($\begin{cases} T = S & \textbf{if } Val(Q,S) \\ T = \textbf{MW} & \textbf{otherwise} \end{cases}$);

(f) R Sem_R⟦code; **Assume** Q⟧ T = ($\begin{cases} \textbf{VC} & \textbf{if } \neg Val(Q,S) \\ T = S & \textbf{otherwise} \end{cases}$);

(g) R Sem_R⟦code; **While** BE **do** body **end**⟧ T = (…);

⋮

Figure 10: Semantics of Basic Statements

In order to show that each proof rule of the RESOLVE verification system is sound and complete, it is necessary to have proofs of soundness and completeness for the proof rule for each language construct. A sample of the soundness and completeness proofs will be included in this chapter as the semantics and proof rules are provided and explained.

Proof Rules for RESOLVE Language Constructs

Simple Operation and Procedure Declaration Rules

Before we discuss the semantics and rules for executable statements that affect the state, we first discuss the basic operation and procedure declaration rules that serve as a useful starting point that sets up the assertive code for processing. To illustrate the rules, we return to the simple example to reverse a stack.

```
Enhancement Flipping_Capability for Stack_Template;
   Operation Flip(updates S: Stack);
         ensures S = reverse(#S);
end Flipping_Capability;

Realization Obvious_Flip_Realiz for Flipping_Capability of
Stack_Template;
   uses Std_Boolean_Fac;
   Procedure Flip(updates S: Stack);
         Var S_Flipped: Stack;
         Var Next_Entry: Entry;
         While (Depth(S) /= 0)
               changing S, S_Flipped, Next_Entry;
               maintaining #S = reverse(S_Flipped) o S;
               decreasing |S|;
         do
               Pop(Next_Entry, S);
               Push(Next_Entry, S_Flipped);
         end;
         S_Flipped :=: S;
   end Flip;
end Obvious_Flip_Realiz;
```

Figure 11: Stack Flip Specification and Realization

```
CDP = Operation P( updates t: T1);
         requires Pre_t_\;
         ensures Post_#t, t _\;
```

The Operation Declaration rule for the operation, P, will add the specification of P to the context.

$$\frac{C \cup \{CDP\} \setminus}{C \setminus CDP;}$$

The Procedure Declaration rule for the operation, P, will generate assertive code to check the correctness of the implementation.

```
Procedure Declaration Rule:
   C ⋃{CDP}\ Assume Pre ∧ T1.Constraint(t);
             Remember;
               body;
             Confirm Post;

   C ⋃{CDP}\ code; Confirm RP;
   ____________________________________________________________
   C ⋃{CDP}\ Proc P(updates t: T1); body; end P; code;
               Confirm RP;
```

A simplified version of the operation declaration rule is above. This version of the rule only contains one parameter with a parameter mode of updates. The rule adds the specification of the parameter to the context and generates the assertive code that will be processed by the statement-level rules. The rules assume the pre-condition and appropriate constraints. In Figure 12, the first **Assume** statement includes the constraints and requires clauses from the included templates. Specifically, the constraints from Integer_Template((min_int <= 0) and (0 < max_int)), the constraints from Character_Template(Last_Char_Num > 0), and the requires from Stack_Template (Max_Depth > 0). We also assume the constraint on the parameter to Flip, S, from Stack_Template (|S| <= Max_Depth) is true. The body of the operation is inserted into the assertive code after the pre-condition is assumed and before the post-condition is confirmed. One new verification statement (that has no effect on the program state) that is introduced in this rule is the **Remember** statement that simply is used to note that "#" symbol denotes the value of the variable at this place. This is inserted prior to the first line of code from the procedure and it serves as a reminder to the verifier that variable values preceded by the distinguishing symbol (#) are the same as their values without that prefix at the point of remembrance. Initially, only one

goal is created, the ensures clause of the operation. In addition, there are no givens (or assumptions) that can be used to prove the goal. However, as the rest of the assertive code is processed, givens will be generated. The context is not shown in this excerpt of assertive code.

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
      While (Depth(S) /= 0)
         maintaining #S = (Reverse(S_Flipped) o S);
         decreasing |S|;
         changing S, S_Flipped, Next_Entry;
      do
               Pop(Next_Entry, S);
               Push(Next_Entry, S_Flipped);
      end;
      S_Flipped :=: S;
Confirm S = Reverse(#S);
```

Figure 12: Flip Assertive Code after the Application of Procedure Declaration Rule (Version 1)

Swap Statement

Now, let us consider the semantics for swap, the very last statement in the Flip code. It is necessary to understand the semantics of swap before proving that the swap proof rule used is sound and complete. I have reproduced below the semantics for the swap statement. As noted before, if the initial state is one of the stuck states, no statement will change the state. Otherwise, the state transformation depends on the type of statement. Now, we will address the questions of soundness and completeness for the swap rule.

```
Inductive Definition on code: Stmnt_Seq of
      (S: St) Sem_R[code] (T: St): 𝔹  is
i.  S Sem_R[Λ] T  = ( T = S )
ii. if R Sem_R[code] S then if S ∈ {MW, VC, ⊥} then T = S
    else
 (a) R Sem_R[code; x :=: y] T  = ( ∀ ξ: Id,
           if ξ ∉ {x, y} then
                 T(ξ) = S(ξ) and
                 T(x) = S(y) and T(y) = S(x));
 (b)…
```

In state T, all Ids other than x and y have the same value as in state S. However, for the Id x, in state T, the value is equal to the value of Id y in state S. The value of Id y, in state T, is equal to the value of Id x in state S. The swap rule has been supplied previously, but it is duplicated below for understanding this proof. For the verifier to apply the rule mechanically, the swap statement in the rule is preceded by "code" which denotes 0 or more statements.

```
Semantics of Swap:
   if Γ = (x :=: y), then ∀ ξ: Id∉{x, y}, U(ξ) = T(ξ)
   and U(x) = T(y) and U(y) = T(x)

Swap Rule:
   C\ code; Confirm RP[ x⇝y, y⇝x ];
_____________________________________________
   C\ code; x :=: y; Confirm RP;
```

First, we will prove that the swap rule is sound. Soundness means that if the code is provable, then it is valid. In the RESOLVE context, if the code is correct, the verifier will never report that it is incorrect. We will do a proof by contradiction. Thus, we assume the code is invalid but the generated VCs are provable.

Another way of saying this is that we assume the assertive code on the top line of the Swap statement is provable but the assertive code underneath the line is not valid.

Because we assume the code is invalid, we know that we end in the state **MW**. We assume that state σ is prior to the swap statement.

```
                          σ(y)  if id = x
σ´ = λ id:Id,   {         σ(x)  if id = y
                          σ(id),otherwise
```

From the semantics, if σ is the state before the swap statement, then because of the semantics of the Swap statement, `RP[x⇝>y, y⇝x]` is false in σ. But this is a contradiction because we assumed that the assertive code above the line (which is state σ) "`code; Confirm RP[ x⇝y, y⇝x ]`" is provable.

Now, let us consider a proof to show completeness of the swap rule. Completeness means that if the code is valid, the generated VCs must be provable. In the RESOLVE context, if the code is incorrect, the verifier will never report that it is correct. One important caveat is that because of the inherent incompleteness in any non-trivial mathematical theory, such as number theory, upon which proofs of programs are based, completeness can be only relative. We will do a proof by contradiction. Thus, we assume the code is valid but the generated VCs are not provable.

Another way of saying this is that we assume the assertive code below the line of the Swap statement rule is valid but the assertive code above the line is not provable.

We assume that state σ is prior to the swap statement.

```
                          σ(y)  if id = x
σ´ = λ id:Id,   {         σ(x)  if id = y
                          σ(id),otherwise
```

From the semantics, if σ is the state before the swap statement, then because of the semantics of the Swap statement, `RP[x⇝>y, y⇝x]` is true in σ. But this is a contradiction because we assumed that the assertive code above the line (which is state σ) "`code; **Confirm** RP[ x⇝y, y⇝x ]`" is not provable.

We conclude this discussion with an application of the Swap proof rule. In the assertive code generated for the Flip procedure (as shown in Figure 12), the last statement in the assertive code was a swap statement, so we apply the swap statement proof rule to this assertive code. Figure 13 shows the modified assertive code after an application of the swap rule. Note that in the final **Confirm** statement, S was modified to S_Flipped. If any instances of S_Flipped had existed in the goal, they would have become S.

```
Assume (((min_int <= 0) and (0 < max_int)) and
          (Last_Char_Num > 0) and ((Max_Depth > 0) and
          (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
      While (Depth(S) /= 0)
         maintaining #S = (Reverse(S_Flipped) o S);
         decreasing |S|;
         changing S, S_Flipped, Next_Entry;
      do
               Pop(Next_Entry, S);
               Push(Next_Entry, S_Flipped);
      end;
Confirm S_Flipped = Reverse(#S);
```

Figure 13: Assertive Code after the Application of the Swap Rule (Version 2)

If Statement

Next we consider the proof rule for the **If/Else** construct. Because of the complexity of this proof rule and the many changes that were necessary to mechanize the rule, let's consider the original,

simple version of the rule first. Then, we will explain the steps that were necessary to create the current version of the proof rule. The initial version of the rule follows. The **If/Else** proof rule will generate assertive code that first confirms any pre-conditions to the condition. The assertive code will be split into two sections. One section will process the **If** section of Code. The second will process the **Else** section of code. Several updates are necessary to mechanize the classical version of the rule that looks something like the one below:

Version 1:

```
Assume BE; code1; Confirm RP;
Assume ¬ BE; code2; Confirm RP;
_____________________________________
If BE then code1 else code2 end_if;
   Confirm RP;
```

An important aspect of this rule that is missing is the context (often abbreviated to C\ in the proof rules). The **If/Else** statement does not occur in isolation. There is a context around it that is needed by the proof system to attempt verification of the code. The context includes the mathematical model (that is described in the specification) which describes the mathematical version of each variable and the specifications for the operation used in the code. For example, arrays and queues are often modeled as mathematical strings. In the Stack_Template specification, our stacks are modeled as strings and that is a necessity in order to understand the syntax (e.g., concatenation = o) and the mathematical meanings of the assertions. The context also includes relevant concept and enhancement specifications (added through the application of declaration proof rules). This observation leads us to following proof rule:

Version 2:

```
Context\ code; Assume BE; code1; Confirm RP;
Context\ code; Assume ¬ BE; code2; Confirm RP;
_____________________________________________________
Context\ code; If BE then code1 else code2 end_if;
               Confirm RP;
```

Another aspect of this proof rule that is necessary is the conversion from programming notation to mathematical notation. This provides proper processing of the conditional statement in the If condition. We cannot simply assume that the If condition statement true because the statement is in code not in a mathematical form. The proof system will not understand programmatic function calls. Consider an example (like the stack reversal) where we have an If statement that is based on the depth of a stack. Depth(S) > 0 does not have any mathematical meaning. Thus, this needs to be converted to its proper mathematical form that can be used by the VC generator. This is denoted with Math(BE). Math(BE) in this case would result in |S| > 0. Thus we need to convert our condition statements into a mathematical form before sending them to a mathematical proof system. Math(BE) will generate the mathematical statements that must be true to enter the If portion of the code.

Version 3:

```
Context\ code; Assume Math(BE);
               code1; Confirm RP;
Context\ code; Assume ¬ Math(BE); code2; Confirm RP;
_____________________________________________________
Context\ code; If BE then code1 else code2 end_if;
               Confirm RP;
```

In the example of Flip, neither Depth(S) nor the greater than operation have a pre-condition. However, it is possible that the code in the condition could have a pre-condition. If there is a pre-

condition for the condition statement, Invk_Cond(BE) will generate the goal that must be proved. Let us assume that BE is a complex statement that could include pre-conditions. Thus, prior to each iteration of the loop, we must also confirm that the pre-condition (or possibly multiple pre-conditions depending on the complexity of the condition) are true. For example, if the condition was (I * ((I + 1) – J)) > 0, there would be several conditions to prove. First, the addition operator requires that I + 1 is within the bounds of integer (greater than or equal to min_int and less than or equal to max_int). The subtraction operation requires that (I+1) – J is not greater than max_int or less than min_int. Finally, the multiplication operator requires that (I * ((I + 1) – J)) is also within integer bounds. Thus, all of these pre-conditions must be proved and will be generated by the Invk_Cond statement. The final version of the **If/Else** rule follows:

Version 4:

```
Context\ code; Confirm Invk_Cond(BE); Assume Math(BE);
               code1; Confirm RP;
Context\ code; Assume ¬ Math(BE); code2; Confirm RP;
_____________________________________________________
Context\ code; If BE then code1 else code2 end_if;
               Confirm RP;
```

An example demonstrating the behavior of the **If/Else** rule will be shown in the discussion of the **While** Loop Rule.

The **If/Else** proof rule aligns closely with the semantics. The semantics for the **If** Statement describe the behavior of the RESOLVE language when processing an **If/Else** statement. If the condition has a pre-condition and if that condition is false, then the transformation will move into the **MW** state. Otherwise, depending on the condition, S and T are related according to the code in either the If or the Else section based on the value of BE in S.

```
   Inductive Definition on code: Stmnt_Seq of
         (S: St) Sem_R[code] (T: St): 𝔹  is
   i.  S Sem_R[Λ] T  = ( T = S )
   ii. if R Sem_R[code] S then if S ∈ {MW, VC, ⊥} then T = S
else
    (a) …
    (b) R Sem_R[code; If BE then code1 else code2] T = (
         If not Invk_Cond(BEx),
                    then T = MW
         Else[Val(BE, S) and S Sem_R[code1] T or
               ¬Val(BE, S) and S Sem_R[code2] T] );
    (c) …
```

To show soundness through a proof by contradiction, we assume the code is invalid but the generated VCs are provable. Another way of saying this is that we assume the assertive code on the top line of the **If/Else** statement is provable but the assertive code underneath the line is not valid. Because we assume the code is invalid, we know that we end in the state **MW**. We assume that state σ is prior to the **If/Else** statement. Because of our assumption, then we know that beneath the line the code is invalid. Then in σ either RP is false or Invk_Cond(BE) is false in order to be in state **MW**. However this is a contradiction because we assumed that the assertive code above the line (which is state σ′) is provable. This means that Invk_Cond(BE) is true and that RP is true.

To show completeness, i.e., for proving that if the code is valid, the generated VCs must be provable, we do a proof by contradiction. Thus, we assume the code is valid but the generated VCs are not provable. Another way of saying this is that we assume the assertive code below the line of the **If/Else** statement rule is valid but the assertive code above the line is not provable.

We assume that state σ is prior to the **If/Else** statement. Because of our assumption, then we know that beneath the line the code is provable. Then in σ both RP is true and Invk_Cond(BE) is true in order for this to be valid. However this is a contradiction because we assumed that the

assertive code above the line (which is state σ′) is probable. This means that Invk_Cond(BE) is false or that RP is false.

Developmental Steps for a While Loop Proof Rule

In 1969, Hoare put forth rules that have provided a basis for showing the correctness of software. Hoare's proof rules however are not mechanical. Thus the conversion from Hoare's original proof rules to automatable rules that can be used to create a verifier is an important process. Hoare's original rules are significantly different from the mechanical versions. The following will demonstrate the process of converting an example proof rule, the **While** loop, from Hoare's original rule to the implementable version. Following the same approach as was used for the **If/Else** statement, I will introduce the **While** loop rule by describing the conversion from the original rule, one modification at a time. This example helps highlight the iterative process for mechanizing a proof rule. Most importantly, the iterative process must maintain soundness and completeness.

```
Inv ^ B {S} Inv
_____________________________
Inv {While B do S} not(B) ^ Inv
```

for the while loop code with the invariant "inv":

```
While B
do
   S;
end;
```

Figure 14: Hoare's Original Proof Rule

Hoare's rule attempts to prove the invariant is true after the completion of the **While** loop rule (when it is true before the loop). In order to prove an invariant true, we will assume that the

invariant and conditional statement are true before an iteration of the loop and prove (based on that assumption and the body of the loop) that the invariant is true at the end of an iteration. If we can prove this invariant is true after an iteration of the loop, the proof rule shows that the invariant is true after all iterations.

Proof Rule: Version 2

In order to use this rule in a mechanical system, a few initial modifications are needed. First, a mechanical verification system needs to be able to prove (or try to prove) generic assertions, i.e. The invariant may be more specific than the general assertion the verification system may be attempting to prove. Thus, we need to modify our rule to prove assertions other than the invariant. Secondly, we need to add a syntactic slot for our invariant in the **While** statement. The programmer will include the invariant when writing the code. The invariant should be detailed enough to capture the "reason" for the **While** loop. If not provided, the invariant will be assumed true.

The second iteration of the **While** loop rule follows:

```
code; Confirm Inv;
Assume Inv ^ B; body; Confirm Inv;
Assume Inv ^ not(B); Confirm RP;
_____________________________________
code; While B maintaining Inv; do Body end; Confirm RP
```

This version of the proof rule will convert the previous example into three assertions. In order to do this, however, we must first know what we are trying to prove.

Proof Rule: Version 3

As discussed when the **If/Else** rule was introduced, the context is necessary to perform proofs and reason about the VCs. Thus, this version includes the context in the proof rule.

```
Context/ code; Confirm Inv;
Context/ Assume Inv ^ B; body; Confirm Inv;
Context/ Assume Inv ^ not(B); Confirm RP;
_____________________________________
Context/ code; While B maintaining Inv; do Body end;
                                          Confirm RP
```

Proof Rule: Version 4

Another aspect of this proof rule that needs modification is the handling of the conditional statement in the **While** loop. This is the same statements that were used in the **If/Else** rule. The proof system would not understand the calls to Depth(S). Thus we will convert our condition statements into a mathematical form before sending them to a mathematical proof system. Actually, this was necessary for the first multiplication example, as well. For the condition X > Y, The '>' sign in the code is actually a call to a "Greater Than" operation from the Integer facility. On the other hand, the '>' sign when it's shown in the assertion, is a math symbol.

Thus the following proof rule becomes the next iteration of our mechanical **While** loop proof rule:

```
C/ code; Confirm Inv;
C/ Assume Inv ^ Math(B); body; Confirm RP;
C/ Assume Inv ^ not(Math(B)); Confirm RP;
_____________________________________
C/ code; While B maintaining Inv; do Body end; Confirm RP
```

Proof Rule: Version 5

In order to motivate the next rule modifications, we will use an excerpt from the Stack Flip example.

```
Var S_Reversed: Stack;
Var Next_Entry: Entry;
While (Is_Not_Zero(Depth(S)))
      maintaining #S = Rev(S_Reversed) o S;
do
      Pop(Next_Entry, S);
      Push(Next_Entry, S_Reversed);
end;
```

The three assertions created by the above version of the **While** proof rule are as follows:

```
1) Assume (((min_int <= 0) and (0 < max_int)) and
      (Last_Char_Num > 0) and ((Max_Depth > 0) and
      (min_int <= Max_Depth) and (Max_Depth <= max_int))));
   Assume (|S| <= Max_Depth);
   Var S_Reversed: Stack;
   Var Next_Entry: Entry;
   Confirm #S = Rev(S_Reversed) o S;

2) Assume #S = Rev(S_Reversed) o S ^ (Depth(S) != 0)
      Pop(Next_Entry, S);
      Push(Next_Entry, S_Reversed);
   Confirm #S = Rev(S_Reversed) o S;

3) Assume #S = Rev(S_Reversed) o S ^  not(Depth(S) != 0)
   Confirm S_Reversed = Reverse(S);
```

It can be observed that by splitting the rule into three parts, we lose assumptions (seen in #1 above) found in the beginning of the assertive code that may be needed to prove the VCs. previous code. Assertive code two and three do not assume the constraints. Assumptions about the constraints of the various variables are placed prior to the implementations by the procedure declaration rule. These constraints originate from the specification of the variable type. Thus,

when proving the invariant, we will assume the constraints on the various variables are true. As can be seen in this reversal example when trying to prove the assertion created by the **While** loop rule, we call both the Push and Pop operations. The VCs need be able to show the pre-condition of Push and Pop are true.

Thus, in this example, to prove the pre-condition of push (|S| < Max_Depth), we will need the constraint |S| <= Max_Depth (in addition to the fact that we just popped an element off the top of the stack) to prove the pre-condition of push. The pre-condition of pop, does not require the constraints to prove. It can be shown by the assumption of the condition of the **While** loop. Thus, the new version of the **While** loop rule is:

```
C/ code; Confirm Inv;
C/ Assume Inv ^Math(B) ^ Are_Constraints_Compliant(P, T);
        body; Confirm RP;
C/ Assume Inv ^ not(Math(B)); Confirm RP;
_____________________________________
C Var P:Type1, Var T:Type2/ code; While B maintaining Inv; do
Body end; Confirm RP;
```

Proof Rule: Version 6

There is a distinction between proofs that show partial and total correctness. If code is only partially correct, there is a guarantee of correctness only if the code terminates. Total correctness additionally requires a proof that the code will terminate. In order to guarantee total correctness with our proof rules, we must also show that the **While** loop terminates. The code can be modified to include an ordinal that is will decrease every time through the loop. The proof rule can, then, be modified to make use of that ordinal to show total correctness. The rule making use of that ordinal is shown below:

```
C/ code; Confirm Inv;
C/ Assume Inv ^Math(B) ^ Are_Constraints_Compliant(P, T) ^
        P_Val=P_Exp; body; Confirm P_Exp < P_Val ^ RP;
C/ Assume Inv ^ not(Math(B)); Confirm RP;
_____________________________________
C  Var P:Type1, Var T:Type2/ code; While B maintaining Inv;
decreasing P_Exp; do Body end; Confirm RP;
```

Proof Rule: Version 7

Another modification to the **While** rule concerns the pre-conditions of function operations invoked in condition B in the code, as discussed in the case of the If-then-else statement.

```
C/ code; Confirm Inv;
C/ Confirm Invk_Cond(B); Assume Inv ^Math(B) ^
Are_Constraints_Compliant(P, T) ^ P_Val=P_Exp; body;
   Confirm P_Exp < P_Val ^ RP;
Context/ Assume Inv ^ not(Math(B)); Confirm RP;
_____________________________________
C  Var P:Type1, Var T:Type2/ code; While B maintaining Inv;
decreasing P_Exp; do Body end; Confirm RP;
```

Proof Rule: Version 8

The final iteration of the rule is primarily to simplify the rule. This rule converts a **While** loop to an If statement. This obviously allows for reuse of the **If/Else** rule which reduces the possibility of implementation errors and ideally makes the rule more "understandable." Positive side-effects of converting the **While** loop to an **If/Else** statement include the removal of Are_Constraints_compliant, Invk_Cond(conditional statement), and Math(conditional statement) forming a much simpler rule.

```
C/ code; Confirm Inv; Change Vlist; Assume Inv ^ NQV(RP,
P_Val)= P_Exp; If BE then body; Confirm Inv ^ P_Exp < NQV(RP,
P_Val); else Confirm RP end_if; Confirm True;
_____________________________________
C/ code; While B maintaining Inv; decreasing P_Exp; changing
VList; do Body end; Confirm RP
```

In addition with this simplification, we will add another annotation to the **While** loop implementation. A list of all variables that the loop may change can be listing in **changing**. As a result of the formation of the rule as one unit (instead of the three previously discussed) and to simplify invariants, the verifier-introduced **Change** statement is necessary. The Change statement differentiates between variables that are altered in the loop and variables that the loop leaves unaltered. Without it, the verifier would assume that in the inductive case, when the "**Assume** inv **and** BE" clause is processed, that each of these are unaltered variables that are modifiable by the application of rules on the code prior to the **While** loop. The statement has the effect of introducing new names for each of the variables listed in the changing clause by aging them with the Next Quotation Mark (NQV) variable and by replacing each variable X with X' in subsequent assertions. In the case when a variable has been already aged and X' is found in subsequent assertions, the verifier will introduce X'' and so on, as necessary. So, all verification-introduced variables will be preceded by one or more question marks and they are all universally quantified. The proof rule uses NQV(RP, X) to state the need to find the instance of X in RP preceded by the most question marks and use one more question mark where NQV is used. NQV is also used with P_Val to insure that nested loops do not try to "use" the same P_Val.

After applying the **While** rule and the **If/Else** rule to the example in Figure 13, the following assertive code is generated.

```
Part One of Assertive Code after Loop Rule:
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
Confirm #S = (Reverse(S_Flipped) o S);
Change S:Modified_String_Theory.Str(Entry),
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry,
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry;
Assume (#S = (Reverse(S_Flipped) o S) and P_val' = |S|);
Confirm true;
Assume |S| /= 0;
      Pop(Next_Entry, S);
      Push(Next_Entry, S_Flipped);
Confirm (#S = (Reverse(S_Flipped') o S) and (|S| < P_val'));
```

Figure 15: Application of the Mechanizable While Loop Rule (Version 3 – Part 1)

```
Part Two of Assertive Code after Loop Rule:
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
Change S:Modified_String_Theory.Str(Entry),
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry,
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry;
Assume #S = (Reverse(S_Flipped) o S);
Confirm true;
Assume |S| = 0;
Confirm S_Flipped = Reverse(#S);
```

Figure 16: Application of the Mechanizable While Loop Rule (Version 3 – Part 2)

While Statement

The following list explains the semantics of the **While** loop annotations in RESOLVE.

- If a variable that is not in the "changing" list is used in a way it might be changes (e.g., a non-restores mode argument of an operation or a swap statement), then the program is syntactically incorrect. If the changing list is omitted, then all variables in scope are considered to be "changing". In this case, and in general, if the changing list has something listed that doesn't change, then the implicit invariant may be too weak and we may not be able prove the VCs, but the code would still be valid. Here, unprovability results from a lack of proper documentation, so it is not a soundness or completeness problem.
- If the programmer-supplied invariant is false, then the code will be invalid (**MW**). If invariant is omitted, it is assumed to be true. In this case and in general, if the invariant is weaker than necessary to establish code correctness, we cannot prove the code but it is still valid. Again, unprovability results from only from a lack of proper documentation because it is always possible to construct an invariant for correct code; so this is not a soundness or completeness problem either.
- If the decreasing clause is wrong, then the code will be invalid (**MW**). If it is omitted and the procedure is expected to terminate with an ensures clause, then the code is invalid (**MW**) as well. The clause is not necessary if the code only 'respects' its guarantees; here the semantics will become $\perp$ and would be still valid. There is no such thing as a too weak decreasing clause. It is possible to construct a decreasing clause for every terminating loop.

To show this proof rule is sound and complete, we use the standard approach of reducing it to a version of the proof rule that has already been proved sound and complete. In many ways, this process is the reverse of the steps we just made to mechanize the proof rule. The following proof rule is the current version:

```
C/ code; Confirm Inv; Change Vlist; Assume Inv ^ NQV(RP,
P_Val)= P_Exp; If BE then body; Confirm Inv ^ P_Exp < NQV(RP,
P_Val); else Confirm RP end_if; Confirm True;
_____________________________________
C/ code; While B maintaining Inv; decreasing P_Exp; changing
VList; do Body end; Confirm RP
```

By removing the Changing statement because that is simply an aid to simply the invariant and removing the decreasing clause which is only necessary if showing termination, the current proof rule reduces to the following proof rule.

```
C/ code; Confirm Inv; Assume Inv; If BE then body; Confirm
Inv; else Confirm RP end_if; Confirm True;
_____________________________________
C/ code; While B maintaining Inv; do Body end; Confirm RP;
```

By applying the **If/Else** rule to the **If/Else** portion of the proof rule, our proof rule reduces to this more simplified version.

```
C/ code; Confirm Inv; Assume Inv; Assume BE; body; Confirm
Inv;
C/ code; Assume Inv; Assume ¬BE; Confirm RP;
_____________________________________
C/ code; While B maintaining Inv; do Body end; Confirm RP;
```

This is the classical version of the proof rule (modulo context and code preceding loop) which is already known to be sound and complete.

Function/Expression Assignment and Operation Call Rules

Before we present the more complex operation call rule, we will first present a classical function assignment rule. The semantics for the reassignment statement, or function call statement, are below. It is necessary to show that any pre-conditions of the function are true. This could include pre-conditions for multiple functions. For example ((X + Y) – Z) would need to show the pre-condition to plus and minus are both true. Invk_Cond(f(y,exp)) will determine if all pre-conditions for this function call are true. Unlike in a general operation call statement, function call parameters are not changeable. Thus, the only value that can change by an application of the function call statement is 'x' the variable that is being assigned the value of f(y, exp). All other Ids in state T remain the same as in state S. T(x) is the value of f(y,exp) in the state S.

```
Inductive Definition on code: Stmnt_Seq of
      (S: St) Sem_R⟦code⟧ (T: St): 𝔹  is
i.  S Sem_R⟦Λ⟧ T  = ( T = S )
ii. if R Sem_R⟦code⟧ S then if S ∈ {MW, VC, ⊥} then T = S
    else
 (a) …
 (c) R Sem_R⟦code; x := f(y, exp)⟧ T  = (
                  If not Invk_Cond(f(y, exp)),
                        then T = MW
                  Else[∀ ξ: Id, if ξ ≠ x then T(ξ) = S(ξ)
                        and T(x) = Val(f(y, exp), S)]);
 (d)…
```

The corresponding proof rule is:

```
C\ code; Confirm Invk_Cond(exp) ∧ RP[x⇝Math(exp)];
_____________________________________________________
C\ code; x := exp; Confirm RP;
```

As with our previous proofs of soundness we will show soundness through a proof by contradiction. We assume the code is invalid and we know that we end in the state **MW**. We

assume that state σ is prior to the reassignment statement. Because of our assumption, then we know that beneath the line the code is invalid. Then in σ either RP is false or Invk_Cond(f(y,exp)) is false in order to be in state **MW**. However this is a contradiction because we assumed that the assertive code above the line (**Confirm** Invk_Cond(exp) ∧ RP[x⇝Math(exp)];)is provable.

To show completeness with a proof by contradiction, we assume the assertive code below the line of the function assignment statement rule is valid but the assertive code above the line is not provable. We assume that state σ is prior to the function assignment statement. Because of our assumption, then we know that beneath the line the code is provable. Then in σ both RP is true and Invk_Cond(f(y,exp)) is true in order for this to be valid. However this is a contradiction because we assumed that the assertive code above the line (which is state σ´) is provable.

Now, let us consider the operation call rule. The semantics of the operation call statement exemplifies the need for relational semantics. RESOLVE permits a result of an operation to be within a set of acceptable choices instead of something particular. For example, an ensures clause could state that an operation ensures for an updated Integer "I > 0". Any implementation that provides an I greater than zero will be a valid implementation. Another example is the alters parameter mode. When this parameter mode is used, no return value is specified of this value. Thus any value an implementation may supply is acceptable.

The semantics for the operation call rule state that if the pre-condition of the operation is not true, then the state of T is **MW**. However, if the pre-condition is true, then all Ids in state T are the same, except any parameters to the operation. Op_Rel is used to define the relation between the value of y in state T and in state S. Op_Rel is based on the post-condition of the operation.

```
Inductive Definition on code: Stmnt_Seq of
      (S: St) Sem_R⟦code⟧ (T: St): 𝔹  is
i.  S Sem_R⟦Λ⟧ T  = ( T = S )
ii. if R Sem_R⟦code⟧ S then if S ∈ {MW, VC, ⊥} then T = S
    else
 (a) …
 (d) R Sem_R⟦code; op(updates y)⟧ T  = (
                  If not Invk_Cond(op(y)),
                        then T = MW
                  Else[∀ ξ: Id, if ξ ≠ y then T(ξ) = S(ξ)
                        and T(y) Op_Rel S(y) )  ]);
 (e) …
```

From the pre-condition a VC to prove it is generated. The post-condition is assumed and is used to prove existing **Confirm** assertions. Variables that are changed by the operation will need to be updated in the existing VCs. This is done using NQV which works the same as explained earlier for the **While** Loop Rule.

This is a simplified version of the operation invocation rule which processes an example operation with an *updates* and *alters* parameter. Note that the specification of operation P is already in the context. Before applying this rule, the assertive code needs to confirm RP which may contain the variables a and b which are arguments to the operation P. The rule will then confirm the pre-condition of P and assume the post-condition of P while updating each statement to use the actual variables instead of specification variables. To preserve the distinction between states, NQV will generate the next verification variable for the specified variable.

```
CDP =  Operation P( updates t: T1; alters u: T2);
          requires Pre_t, u_;
          ensures Post_ #t, #u, t_;
```

Operation Invocation Rule:

```
C ⋃{CDP}\ code; Confirm Invk_Cond(P(a,b));
   Assume Post[t⇝NQV(RP2, a), #t⇝a, #u⇝b];
   Confirm RP[a⇝NQV(RP2, a)];
_____________________________________________________________
C ⋃{CDP}\ code; P( a, b); Confirm RP/_ a, b _\ ;
```

Returning to the Flip example in Figure 15, we will apply the operation invocation rule to the following code. Note that the parameter modes in this example match the parameter modes for the push operation.

```
Operation Push(alters E: Entry; updates S: Stack);
   requires |S| < Max_Depth;
   ensures  S = <#E> o #S;
```

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
Confirm #S = (Reverse(S_Flipped) o S);
Change S:Modified_String_Theory.Str(Entry),
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry,
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry;
Assume (#S = (Reverse(S_Flipped) o S) and P_val' = |S|);
Confirm true;
Assume |S| /= 0;
      Pop(Next_Entry, S);
Confirm (|S_Flipped| < Max_Depth);
Assume S_Flipped' = (<Next_Entry> o S_Flipped);
Confirm (#S = (Reverse(S_Flipped') o S) and (|S| < P_val'));
```

Figure 17: Flip Example (Version 4)

Assume Rule

In the assertive code provided above, the next proof rule needed is a rule to process the **Assume** statement. The **Assume** rule is a basic rule that processes **Assume** statements generated by other proof rules.

```
Assume Rule:

   C\ code; Confirm exp => RP;
   ___________________________________________
   C\ code; Assume exp; Confirm RP;
```

The results of an application of the **Assume** rule to the assertive code in Figure 18 with minor simplifications follow.

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
Confirm #S = (Reverse(S_Flipped) o S);
Change S:Modified_String_Theory.Str(Entry),
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry,
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry;
Assume (#S = (Reverse(S_Flipped) o S) and P_val' = |S|);
Confirm true;
Assume |S| /= 0;
      Pop(Next_Entry, S);
Confirm (|S_Flipped| < Max_Depth);
Confirm (#S = (Reverse((<Next_Entry> o S_Flipped)) o S) and
            (|S| < P_val'));
```

Figure 18: Assertive Code after the application of Assume Rule (Version 5)

Notice that this **Assume** statement was not added as a given! This might be surprising based on the proof rule. However, because the assumption is an equality of the form variable = value, then

we are able to replace any instances of the variable with the value, if the variable exists in the goal. This is done in an effort to simplify the VCs and reduce the number of givens.

The semantics state that the value of state T is the same as the value of state S except when Q is not valid in state S. In that case, the state T enters the **VC** state, because the assumption on which the code is based is not true. The proof rule for the **Assume** statement modifies the goal in the same manner as the semantics update the state. The assumption implies the current goal and if false, the code will be vacuously true. Otherwise, the assumption is used as an additional fact to prove the goals.

Inductive Definition on code: Stmnt_Seq **of**
(S: St) Sem_R⟦code⟧ (T: St): 𝔹 **is**
i. S Sem_R⟦Λ⟧ T = (T = S)
ii. **if** R Sem_R⟦code⟧ S **then if** S ∈ {**MW, VC,** ⊥} **then** T = S
else
(a) …
(f) R Sem_R⟦code; **Assume** Q⟧ T = ($\begin{cases} \textbf{VC} & \textbf{if } \neg \text{Val(Q,S)} \\ \text{T= S} & \textbf{otherwise} \end{cases}$);
(g) …

Confirm Rule

The semantics for the **Confirm** rule require that the **Confirm** statement be true or the state enters **MW**. Otherwise, the state stays the same.

Inductive Definition on code: Stmnt_Seq **of**
(S: St) Sem_R⟦code⟧ (T: St): 𝔹 **is**
i. S Sem_R⟦Λ⟧ T = (T = S)
ii. **if** R Sem_R⟦code⟧ S **then if** S ∈ {**MW, VC,** ⊥} **then** T = S
else
(a) …
(e) R Sem_R⟦code; **Confirm** Q⟧ T = ($\begin{cases} \text{T = S} & \textbf{if } \text{Val(Q,S)} \\ \text{T = } \textbf{MW} & \textbf{otherwise} \end{cases}$);
(f) …

These semantics align with the proof rule which adds the statement to be confirmed as another goal that must be true if the code is valid. The **Confirm** rule is typically used to process **Confirm** statements generated by other proof rules, though a programmer may introduce explicit **Confirm** clauses in code occasionally as hints to assist an automated prover.

```
Confirm Rule:

   C\ code; Confirm IP ∧ RP;
   ____________________________________________
   C\ code; Confirm IP; Confirm RP;
```

After applying the **Assume** rule for the post-condition of Push, the next rule to statement to be processed is the **Confirm** statement. The **Confirm** rule updates the **Confirm** statement with a new goal that must be proved. At this point in time, we cannot prove that |S_Flipped| < Max_Depth but after processing of the prior statements, this should be provable when the VCs are sent to the prover.

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
   Var Next_Entry:Entry;
Confirm #S = (Reverse(S_Flipped) o S);
Change S:Modified_String_Theory.Str(Entry),
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry,
   S_Flipped:Modified_String_Theory.Str(Entry),
   Next_Entry:Entry;
Assume (#S = (Reverse(S_Flipped) o S) and P_val' = |S|);
Confirm true;
Assume |S| /= 0;
      Pop(Next_Entry, S);
Confirm ((|S_Flipped| < Max_Depth) and
            (#S = (Reverse((<Next_Entry> o S_Flipped)) o S)
            and (|S| < P_val')));
```

Figure 19: Assertive Code after the application of Confirm Rule (Version 6)

Variable Declaration Rule

The Variable declaration rule processes each variable declaration. In general, a variable may be initialized to one of many values (i.e., its specification is relational). So RESOLVE uses a predicate to capture whether a given value is an initial value. For some types, there may be a single initial value (e.g., Stack). If the initial value is known, each instance of the Variable may be replaced in the **Confirm** statement.

```
Variable Declaration Rule:

C\ code; Assume T.is_initial(v); Confirm RP;
_____________________________________________
C\ code; code; Var v: T; Confirm RP;
```

Once again, returning to the Flip example, we will use the variable declaration of S_Flipped as an example. The following assertive code assumes proof rules have been applied up until processing the variable declaration for S_Flipped.

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
   Var S_Flipped:Stack;
Confirm  (#S = (Reverse(S_Flipped) o S) and
         (#S = (Reverse(S_Flipped') o S'') implies
         (|S''| /= 0 implies
         (|S''| /= 0 and
         (S'' = (<Next_Entry'> o S') implies
         ((|S_Flipped'| < Max_Depth) and
         ((Reverse(S_Flipped') o S'') =
          (Reverse((<Next_Entry'> o S_Flipped')) o S') and
         (|S'| < |S''|))))))))
```

Figure 20: Assertive Code just before the Application of Variable Declaration Rule (Version 7)

After processing the variable declaration for S_Flipped, the assertive code in Figure 21 is generated. Note that every instance of S_Flipped in the final **Confirm** was replaced with empty_string.

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Remember;
Confirm  (#S = (Reverse(empty_string) o S) and
         (#S = (Reverse(S_Flipped') o S'') implies
         (|S''| /= 0 implies
         (|S''| /= 0 and
         (S'' = (<Next_Entry'> o S') implies
         ((|S_Flipped'| < Max_Depth) and
         ((Reverse(S_Flipped') o S'') =
          (Reverse((<Next_Entry'> o S_Flipped')) o S') and
         (|S'| < |S''|))))))))
```

Figure 21: Assertive Code After the Application of Variable Declaration Rule (Version 8)

Remember Rule

The **Remember** rule is used to convert the old variables in the **Confirm** statement to the normal version. The old variables distinguish incoming and outgoing variables while processing the assertive code. The **Remember** rule is placed so that the variables no longer need that distinction for the remaining proof rules to be applied and the proofs to be completed.

```
C\ code; Confirm RP[#s⇝s, #t⇝t];
_____________________________________________________
C\ code; Remember; Confirm RP/_ s, #s, t, #t, ⋯ _\;
```

After application of the **Remember** rule to the assertive code in Figure 21, the assertive code would appear as below. At this point, the VC generator will apply the **Assume** rule twice and then supply the final VCs to the prover.

```
Assume (((min_int <= 0) and (0 < max_int)) and
         (Last_Char_Num > 0) and ((Max_Depth > 0) and
         (min_int <= Max_Depth) and (Max_Depth <= max_int))));
Assume (|S| <= Max_Depth);
Confirm  (S = (Reverse(empty_string) o S) and
         (S = (Reverse(S_Flipped') o S'') implies
         (|S''| /= 0 implies
         (|S''| /= 0 and
         (S'' = (<Next_Entry'> o S') implies
         ((|S_Flipped'| < Max_Depth) and
         ((Reverse(S_Flipped') o S'') =
          (Reverse((<Next_Entry'> o S_Flipped')) o S') and
         (|S'| < |S''|))))))))
```

Figure 22: Flip Example (Version 9)

At this point, we have discussed all the rules for verifying the code for Flip procedure. Repeated application of the rules leads to the VCs given in Appendix D. Each VC has a goal and one or more givens. Some VCs are simple and some others are require knowledge of mathematical theorems to establish. Two interesting VCs follow. Both VCs rely on the definition of Reverse. In order to prove VC 0_1, the prover must be able to determine that Rev(empty_string) is still the empty_string that that the empty_string concatenated with S is the same as S. VC 0_3 also requires understanding of Reverse but in addition uses the givens to prove the correctness. If the goal is updated to apply the Reverse operator to <Next_Entry> o S_Flipped, the goal will become (Reverse(S_Flipped') o S") = (Reverse(S_Flipped') o <Next_Entry'> o S'). Then because of the assumption that S'' = <Next_Entry'> o S', this VC can be proven true.

```
VC: 0_1:
Base Case of the Invariant of While Statement in Procedure
Flip modified by Variable Declaration rule:
Obvious_Flip_Realiz.rb(10)

Goal:
S = (Reverse(empty_string)  o S)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Max_Depth  > 0)
5: (min_int <= Max_Depth) and (Max_Depth <= max_int)
6: (|S|  <= Max_Depth)


VC: 0_4:
Inductive Case of Invariant of While Statement in Procedure
Flip modified by Variable Declaration rule:
Obvious_Flip_Realiz.rb(10)

Goal:
(Reverse(S_Flipped')  o S'') = (Reverse((<Next_Entry'>  o
S_Flipped'))  o S')

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Max_Depth  > 0)
5: (min_int <= Max_Depth) and (Max_Depth <= max_int)
6: (|S|  <= Max_Depth)
7: S = (Reverse(S_Flipped')  o S'')
8: |S''| /= 0
9: S'' = (<Next_Entry'>  o S')
```

General Procedure Declaration and Call Rules

We conclude this chapter with general rules for operation calls and procedure bodies, involving all different parameter modes. The example operation P, defined by the name CDP, will be used to demonstrate these proof rules.

```
CDP =  Operation P( updates t: T1; evaluates u: T2;
         replaces v: T3; restores w: T4; preserves x: T5;
         alters y: T6; clears z: T7);
            requires Pre_t, u, w, x, y, z, _\;
            ensures Post_ #t, u, w, x, #y, #z, t, v, _\;
```

First let us consider the general operation realization rule. For the operation P each parameter has a different mode and the rule defines how each mode affects the VCs generated. This rule must assume the constraints of all the parameters to the operation (along with the pre-condition). In addition, for a parameter that is defined to use the replaces parameter as with parameter v, the incoming value is assumed to be the initial value of the type, T3. The **Remember** statement and body of the assertive code are generated in the same manner as with the simpler rule. The final **Confirm** must also take into account the different parameter mode. If the parameter mode is restores, as with w, the assertive code will confirm that the final value of w is the same as the starting value. For the parameter mode clears, the final value of z is assumed to be the initial value of type, T7.

Procedure Declaration Rule:

```
C ⋃{CDP}\ Assume Pre ∧ T1.Constraint(t) ∧ T2.Constraint(u) ∧
      T3.Is_Init(v) ∧ T4.Constraint(w) ∧ T5.Constraint(x) ∧
      T6.Constraint(y) ∧ T7.Constraint(z);
      Remember;
            body;
      Confirm Post ∧ w = #w ∧ T7.is_initial(z);

C ⋃{CDP}\ code; Confirm RP;
_____________________________________________________________
C ⋃{CDP}\ Proc P(… ); body; end P; code;
            Confirm RP;
```

The general operation call rule provides the same functionality as the simplified version of the call rule but defines the behavior for all parameter modes. For certain parameter modes, the pre-condition or post-condition can be modified. For example, in P, z (of type T7) is cleared. The actual parameter in the operation call is g. Thus, the proof rule adds an additional **Assume** clause that T7.**is_initial**(NQV(RP, g)) after the operation call. Of course, the formal parameters must be replaced with the actual parameters. In the post-condition, each parameter is replaced based on its parameter mode. Any parameter that may have a different final value than the initial value will use NQV (as defined for previous rules) to generate new variable names. So the resulting value of any parameter with updates and replaces mode are replaced by the NQV value of the actual arguments. The **Assume** statement (that assumes the post-condition) is also updated so that any instance of the actual parameter is replaced with the NQV value. For updates and replaces modes, any variables with the # sign (variables referring to the incoming values) are replaced by the actual parameter. This distinguishes the incoming and outgoing values of the operation. For example, if the actual variable is A and the formal parameter is T, then the specification may refer to #T and T, but the assertive code would updated those references to A and A'. Any parameters that are evaluated are replaced by the mathematic equivalent of the expression. The other

parameters are replaced by the actual argument used when the operation is invoked. There is no need to use NQV since the variable does not change. Similar modifications must be made to the pre-condition (replacing the parameters used in the specification with the actual values) but this logic is handled by Invk_Cond. Invk_Cond, just as with the function call rule, will provide the combined pre-conditions of the operation being called in addition to any pre-conditions for functions being used as parameters to the operation.

Operation Call Rule:

```
C ⋃{CDP}\ code; Confirm Invk_Cond(P(a,exp,b,c,d,e,f));
   Assume ( T1.Constraint(t) ∧ T3.Constraint(v) ∧
   T6.Constraint(y) ∧ Post) [t⇝NQV(RP, a), #t⇝a, u⇝Math(exp),
   v⇝NQV(RP, b), w⇝c, x⇝d, #y⇝e, #z⇝f] ∧ T7.is_initial(NQV(RP,
   f));
   Confirm RP[a⇝NQV(RP, a), b⇝NQV(RP, b), e⇝NQV(RP, e),
   f⇝NQV(RP, f)] );
_____________________________________________________________________
C ⋃{CDP}\ code; P( a, exp, b, c, d, e, f);
            Confirm RP/_ a, b, c, d, e, f, g, h, ⋯ _\ ;
```

One complication in verification of operation calls concerns repeated arguments. If an operation takes multiple parameters and the same variable is passed more than once, what is the behavior of the operation? An answer to this question is important in defining the behavior of the software under all circumstances. For RESOLVE, this behavior has been defined in [45]. To implement this semantics, the current compiler is being augmented with a pre-processor that introduces necessary intermediate variables so that the resulting code processed by the VC generator is such that there are no operation calls with repeated arguments. This makes it possible to apply the proof rules in this section directly, and thus handle calls with repeated arguments indirectly.

VC Simplification

The VC generator contains simplification rules for reducing the verification conditions. Between each application of a proof rule, the VC generator can simplify each VC if possible. A list of basic simplification rules is provided in Appendix B. While the rules do eliminate VCs, the VCs that are eliminated could be easily proved by the RESOLVE prover.

The simplification process has to be done carefully, for otherwise incompleteness could be introduced. Consider the following example where the VCs could be oversimplified to understand the reasoning for not adding additional simplification steps. At initial inspection, in the following example, it appears that we can simplify the following example. The fact that Max_Length > 0 seems to have no impact on proving the value of X is greater than -1.

```
assertive_code;
Assume Max_Length > 0;
Confirm X > -1;
```

However, because we do not know what facts the assertive_code that comes before the assumption contains (when processing the **Assume** rule), one must be careful in the simplification process. For example, if the assertive code stated: X > Max_Length, this fact is necessary to prove our goal.

```
Assume X > Max_Length;
Assume Max_Length > 0;
Confirm X > -1;
```

Thus, it is currently thought to be too risky to perform certain simplification until the entire VC generation process is completed. Sound simplifications are among the future directions for research.

CHAPTER FOUR

COMPONENT-LEVEL VERIFICATION

In order to build and verify a large software system, it is important to be able to verify each component independently. So this chapter builds on the previous discussion of verification of a procedure to complications of verification of components. To illustrate how a component-based system can be designed and verified in RESOLVE, we consider a detailed example here. In the process, module-level proof rules required to verify each component of a software system are presented in this chapter.

Object-Based Component Specification, Implementation, and VCs

For software verification to be viable, it must be scalable. Software implementations must be component-based and the components must be designed to allow for verification of one component at a time. The verification process should not require re-verification of components, even when they are generic (i.e., parameterized). Given that a component-based system often consists of several components, it is important for the verification to take place within an environment that uses only the specifications of other components to provide modularity.

The diagram in Figure 23 demonstrates the idea of developing components using only the specification of the underlying components. For example, the selection sort code will not use details of the queue implementation but will rely only on the queue specification. The queue facility will actually choose which implementation of the queue and of the sort it needs. Although any implementation of the queue or sort should behave as specified, they may have different characteristics in regards to performance and memory usage.

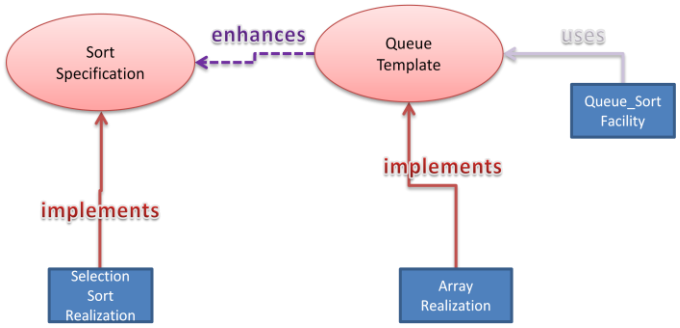


Figure 23: A UML Diagram for a Simple Component-Based System

The Queue_Template specification is in many ways similar to the Stack_Template specification that has already been discussed. Queue_Template requires two arguments, the type of item in the queue and the maximum size of the queue which must be provided whenever Queue_Template is instantiated. The mathematical model of the queue is a string of entries. It is specified that the queue is initialized to the empty string. In the version where Q is the exemplar, the length of Q, |Q|, is constrained to be no larger than the Max_Length parameter. These constraints must be maintained by the implementation of the queue but may be assumed by client code.

Queue_Template provides the expected queue operations: Enqueue and Dequeue. In addition, the current size of the queue and the remaining capacity of the queue may be accessed with the Length and the Rem_Capacity operations.

```
Concept Queue_Template(type Entry; evaluates Max_Length:
Integer);
    uses Std_Integer_Fac, Modified_String_Theory;
    requires Max_Length > 0;

    Type Family Queue is modeled by Str(Entry);
    exemplar Q;
    constraint |Q| <= Max_Length;
    initialization ensures Q = empty_string;

    Operation Enqueue(alters E: Entry; updates Q: Queue);
        requires |Q| < Max_Length;
        ensures Q = #Q o <#E>;
```

```
    Operation Dequeue(replaces R: Entry; updates Q: Queue);
        requires |Q| /= 0;
        ensures #Q = <R> o Q;

    Operation Length(restores Q: Queue): Integer;
     ensures Length = (|Q|);

    Operation Rem_Capacity(restores Q: Queue): Integer;
        ensures Rem_Capacity = (Max_Length - |Q|);

    Operation Clear(clears Q: Queue);

end Queue_Template;
```

Figure 24: Queue Template

Obviously in order to make use of Queue_Template, an actual implementation of the queue specification is necessary. The implementation of Queue_Template using arrays can be found in Figure 25. It is important to note that developers can provide an implementation for a new component prior to realizing underlying components because verification relies on just the specifications of used components. The implementation is fairly straight-forward. The array is of the same size as Max_Length, the maximum size of the queue. Two integers, front and length, keep track of the location of the queue in the array. The front variable defines the index into the array where the top entry in the queue is located. The length variable defines the size of the queue (or the number of consecutive elements in the array used for the queue). For efficiency sake, the queue will not always begin at the first element in the array. Thus the queue may loop around the end of the array.

```
Realization Circular_Array_Realiz for Queue_Template;
      Type Queue = Record
            Contents: Array 0..Max_Length - 1 of Entry;
            Front, Length: Integer;
      end;

   convention
         0 <= Q.Front < Max_Length and
         0 <= Q.Length <= Max_Length;
   correspondence
         Conc.Q = (Concatenation i: Integer
               where Q.Front <= i <= Q.Front + Q.Length - 1,
               <Q.Contents(i mod Max_Length)>);

   Procedure Enqueue(alters E: Entry; updates Q: Queue);
        Q.Contents[(Q.Front + Q.Length) mod Max_Length] :=: E;
        Q.Length := Q.Length + 1;
   end Enqueue;

   Procedure Dequeue(replaces R: Entry; updates Q: Queue);
        Q.Contents[Q.Front] :=: R;
        Q.Front := (Q.Front + 1) mod Max_Length;
        Q.Length := Q.Length -1;
   end Dequeue;

   Procedure Length(restores Q: Queue): Integer;
        Length := Q.Length;
   end Length;

   Procedure Rem_Capacity(restores Q: Queue): Integer;
         Rem_Capacity := Max_Length - Q.Length;
   end Rem_Capacity;

   Procedure Clear(clears Q: Queue);
         Q.Front := 0;
         Q.Length := 0;
   end Clear;

end Circular_Array_Realiz;
```

Figure 25: Queue Array Implementation

In addition to the proof rules that must be generated to show each operation in the queue is correct, a few special concept-level proof rules are required. In order to understand these proof rules, we will use the example concept specification and implementation in Figure 26.

This is an example template only and includes nearly all possible characteristics of a concept template. In order to better understand this template, let us consider what it means. As in earlier example and rules, the following notation means that the ‘ExampleStatement’ can possibly contain each variable within the brackets: ExampleStatement/_ ?, ? _\. The example template, CT, defines a component CN which has three arguments: T, a type; n, a variable of type U; and R, a mathematical definition. Within the template, additional definitions are provided: a global variable (gv), a math definition (s), and a math defines (f). Each of these (and other definitions) can be used in various statements throughout the specification. For example, the concept requires that CPC is true. CPC can refer to n and R. This template defines a new type, TF, which is exemplified with x and is constrained by TC. These example types, constraints, and requires can all be used in the proof rule to demonstrate the behavior.

Suppose a Concept Template is specified by:

```
CT = Concept CN( type T; eval n: U; def R: T×V→B );
              uses AFac, BTh;
              requires CPC/_ n, R _\;
         Definition S: W→B = ( DExp );
         Defines f: W→T;
         constraint DC/_ f, n, R _\;
         Var gv: X1;
         constraint VC/_ gv, f, n, R _\;

         Facility_Initialization
              ensures GIC/_ gv, f, S, n, R _\;

         Type Family TF is modeled by MTE;
              exemplar x;
              constraint TC/_ x, n, R, gv, f _\;
              initialization
                   updates gv;
                   ensures IC/_ x, gv, f, n, R _\;
              finalization[1]


                   updates gv;
                   ensures FC1/_#x, gv, f, n, R_\;

         Oper P( updates x: TF; evaluates z: U );
              updates gv;
              requires Pre/_x, y, gv, f, n, R_\;
              ensures Post/_x, #x, y, gv, #gv, f, n, R_\;
   end CN;
```

First, the specifications provided in the concept declaration must be added to the context. The rule below shows that the definition of the concept is included in the context.

Concept Declaration Rule:

$\mathcal{C} \cup \{CT\} \setminus$ code; **Confirm** RP;

$\mathcal{C} \setminus$ CT; code; **Confirm** RP;

[1] The proof rules in this dissertation do not address variable finalization or facaility finalization clauses; straightforward extensions are necessary to handle them.

Suppose a concept implementation is specified by the following realization template. This template is setup similar to the concept template in that any of the variables or definitions that can be used in a statement are included within the brackets. The implementation, RN, takes four parameters: rn, of type RU; RR, a definition; F_Realiz, a realization of an operation; and RP, the specification of an operation. This example instantiates a facility, F, defines both implementation and auxiliary variables (rg1, and ra), and provides the correspondence (Cor_Fn_Exp), conventions (RC), and initialization (I_body) for type TF.

```
RT =  Realization RN( eval rn: RU; def RR: RT×RV→B;
                  Realization F_Realiz(eval e:T3);
                  Procedure RP( updates rx: RT2 );
                        requires preRP/_ rx, rn _\;
                        ensures postRP/_ rx, #rx, rn _\)
      for CN;
            uses RAFac, RBTh, GTy, R_C;
            requires RPC/_ rn, RR _\;)
            Definition RS: RW→B = ( RDExp );
            Definition f: W→T = (F_Exp);
            constraint RDC/_ rn, RR _\;

   Facility F is R_C( f_exp/_ rx, rn _\, GTy, RR, RS)
      realized_by F_Realiz(f_exp/_ rx, rn _\, RR, RS);

   (* Treatment of rules does not include realization globals *)
   (*
   Var rg1: RX;
   Aux_Var ra: Tm;
      convention RGC//_ ra, rg, f, rn, RR, RS _\;
      correspondence CR_Exprg//_ gv, ra, rg1, rn, RR, RS _\;
      Facility_Initialization GI_body; end;
   *)
```

```
   Type TF = RT;
      conventions RC/_ x, rg1, f, rn, RR, RS_\;
      correspondence[2] conc.x = Cor_Fn_Exp/_x, gv, rg, rg, ra,
                                              f, n, rn, R, RR, S,
                                          RS_\;
      Facility_Initialization I_body; end;


   Procedure P (updates x: TF; evaluates z: U);
      p_body;
   end P;

end RN;
```

Figure 26: Example Concept Specification and Realization

The current VC generator does not handle global variables in realizations. So, the realization declaration rule below is a simplified version.

Realization Declaration Rule:

C ∪{R_C}\ Fac_Instantiation_Hyp;
C ∪{CT}\ Well_Def_Corr_Hyp;
C ∪{CT}\ T_Init_Hyp;
C ∪{CT}\ Correct_Op_Hyp;
C ∪{CT}∪{R_Heading}\ code; **Confirm** Q;

C ∪{CT}\ RT; code; **Confirm** Q;

The facility instantiation hypothesis is discussed in the context of long facility declarations in the section titled Facility Declarations and VCs. It is necessary to show that the correspondence is well-defined. This rule will only handle a functional correspondence. We must show the

[2] The correspondence may be a relation of the form Cor_Exp /**conc**.x, x, ...\. In this case, instead of confirming TC[x ⇝ Cor_Fn_Exp] in the well defined correspondence hypothesis on the next page, we will assume Cor_Exp /**conc**.x, x, ...\ and confirm TC[x ⇝ **conc**.x]. Similarly, the rules for type intialization hypothesis and correct operation hypothesis will also change.

convention is true when the exemplar is replaced with the correspondence expression. The VC will assume any concept specification or concept realization requires clauses in addition to any defined conventions.

```
Well_Def_Corr_Hyp shows that the correspondence is well
defined.
   Assume CPC/_ n, R _\ ∧ RPC/_ rn, RR_\ ∧
            RC/_ rg, f, rn, RR, RS_\;
         // Assumes Concept Level Requires, Concept
         // Realization level requires and convention
   Confirm TC[x ⇝ Cor_Fn_Exp];
```

The VCs generated to show the correspondence of the queue is well defined follows. The correspondence defines the relationship between the conceptual (mathematical) space and the representational (implementation) space. We must show that the correspondence relates all legitimate representation values to legitimate abstract values. The VC is formed by generating assertive code that is then processed. The assertive code assumes any concept level requirements, concept realization requirements, and conventions. Then the constraints must be proved. The constraints are updated when generated to replace the conceptual value with the correspondence. This VC is showing that the constraints clause of the queue (|Q| <= Max_Length) is true for this implementation. The VC can be proved because of the givens providing information about the size of Q.Front and Q.Length. This goal will generate a string by concatenate the entry at Q.Contents(i%Max_Length) a total of Q.Length -1 times. Because Q.Length <= Max_Length, VC 1_1 is true.

```
VC: 1_1: Correspondence Rule for Queue:
Circular_Array_Realiz.rb(49)

Goal:
(|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
<= Max_Length)

Given:
1: (Max_Length  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Last_Char_Num  > 0)
5: (0  <= (Max_Length - 1  + 1))
6: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
7: (min_int <= 0) and (0 <= max_int)
8: (0 <= Q.Front) and (Q.Front < Max_Length)
9: (0 <= Q.Length) and (Q.Length <= Max_Length)
```

Now we will consider the type initialization. It must be shown that the initialization code in the realization initializes the type(s) according to the specification. The type initialization rule follows:

```
T_Init_Hyp establishes that the type initialization is done
correctly.

Assume CPC ∧ RPC;
var x1: RT;
I_Body;
Confirm RC;
Confirm IC/_ x ⇝ Cor_Func_Exp, gv, f, n, R _\;
```

The VCs below are generated to show the type initialization for the queue concept. The assertive code for the initialization is generated by assuming the concept level requires clause and concept realization requires clause. Then a temp variable of the defining type is generated with the same name as the exemplar. The body of the initialization code is added to the assertive code. The conventions then added to the assertive code as a **Confirm** statement because the convention must be true after initialization. Finally, the initialization clause must also be proved. Any instance of the exemplar in the initialization ensures clause are replaced by the functional

correspondence. (If the correspondence is relational, this would become an **Assume** correspondence clause followed by an ensures **Confirm** clause.) First it is necessary to show that the convention is true after initialization. VC 2_1 shows that Q.Front is less than Max_Length and greater than (or equal to) zero after initialization. VC 2_2 shows the same is true for Q.Length. VC 2_3 shows that the initialization statement Q = empty_string is true after initialization. Because the array is empty after initialization, this VC is also true. The string generated by the concatenation is empty because the ending index is less than the starting index.

```
VC: 2_1:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal: (0 <= 0) and (0 < Max_Length)

Given:
1: for all i:Z, Entry.Is_Initial(Q.Contents(i))
2: (Max_Length  > 0)


VC: 2_2:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal: (0 <= 0) and (0 <= Max_Length)

Given:
1: for all i:Z, Entry.Is_Initial(Q.Contents(i))
2: (Max_Length  > 0)


VC: 2_3:
Initialization Rule for Queue: Circular_Array_Realiz.rb(4)

Goal: Concatenation i:Integer where (0 <= i) and
   (i <= ((0 + 0) - 1)),
   <Q.Contents((i mod Max_Length))> = empty_string

Given:
1: for all i:Z, Entry.Is_Initial(Q.Contents(i))
2: (Max_Length  > 0)
3: Conc.Q = Concatenation i:Integer where
    (0 <= i) and (i <= ((0 + 0) - 1)),
    <Q.Contents((i mod Max_Length))>
```

In general, to show correctness of a component, it is also necessary to show the initialization of global variables is valid. This is not necessary for this example because there are no global variables. This rule can be found inAppendix A.

Finally, VCs must be generated for each operation to verify the component. There is a key difference between verifying a external procedure (with its operation specification in the concept interface) in a component versus an internal procedure with a local operation specification. That the convention must be proved true at the end of the operation, assuming it is true at the start of the operation.

```
Correct Operation Hypothesis Rule

   Assume CPC ∧ DC ∧ VC ∧ RPC ∧ RDC ∧
          RC ∧ TC ∧ Pre[x ⇝ Cor_Fn_Exp];
   Remember;
   Body;
   Confirm RC/_ x, rg, f, rn, RR, RS_\
           ∧ Post[x ⇝ Cor_Fn_Exp];
```

If the procedure is *local* and its specification and code are both inside the realization, then the rule does not use the correspondence or conceptual constraints. It does not assume conventions at the beginning of the code or confirm conventions at the end. The rule is similar to the one given previously in Chapter 3.

The VCs for the Dequeue Operation are provided on the next several pages. VC 4_1 and 4_2 show that the pre-condition to Swap_Entry is true prior to the operation call. VC 4_3 shows that the pre-condition to the mod operation is true. VC 4_4 shows that the subtract pre-condition is true. In order to show that the queue convention is true at the end of the operation, VCs 4_5 and 4_6 are generated. Finally VC 4_7 shows that post-condition to Dequeue is true at the end of the

operation. VC 4_7 is the most interesting and most difficult VC to prove. The string that is generated from the Q.Contents array must be equal to the concatenation of the first item in the Q.Contents array and the string generated from all but the first item in the Q'.Contents array. This goal is true because of the 13th assumption which states that Q'.Contents is equal to Q.Contents except at Q.Front where the value differs.

```
VC: 4_1: Requires Clause of Swap_Entry in Procedure Dequeue:
Circular_Array_Realiz.rb(22)

Goal: (0  <= Q.Front)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
```

```
VC: 4_2: Requires Clause of Swap_Entry in Procedure Dequeue:
Circular_Array_Realiz.rb(22)

Goal: (Q.Front  <= Max_Length - 1)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0


VC: 4_3: Requires Clause of Q.Front + 1 % Max_Length in
Procedure Dequeue: Circular_Array_Realiz.rb(23)

Goal: Max_Length /= 0

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})
```

```
VC: 4_4: Requires Clause of Q.Length - 1 in Procedure Dequeue:
Circular_Array_Realiz.rb(24)

Goal: (min_int <= (Q.Length-1)) and ((Q.Length-1) <= max_int)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})


VC: 4_5: Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal: (0 <= ((Q.Front + 1) mod Max_Length)) and (((Q.Front +
1) mod Max_Length) < Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise }})
```

```
VC: 4_6: Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal: (0 <= (Q.Length - 1)) and ((Q.Length - 1) <= Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise }})
```

```
VC: 4_7: Ensures Clause of Dequeue:
Circular_Array_Realiz.rb(25)

Goal: Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>
= (<Q.Contents(Q.Front)>  o Concatenation i:Integer where
(((Q.Front + 1) mod Max_Length) <= i) and (i <= ((((Q.Front +
1) mod Max_Length) + (Q.Length - 1)) - 1)), <Q'.Contents((i
mod Max_Length))>)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})
```

Enhancement Specification, Implementation, and VCs

In order to sort the queue provided by Queue_Template, an enhancement to Queue_Template which provides the ability to sort is used. The Sorting_Capability has been provided in Figure 27. First, notice that the sorting spec does not define what sort algorithm must be used, just that the queue is sorted by the operation. Next, Queue_Template doesn't specify what sort order must be used in the implementation. Because Queue_Template could contain any type of object, it is impossible to sort based on an unknown type. Thus, the spec is parameterized. A definition, LEQV, which defines the sort order is required. The only requirement for the definition is that it is a total preordering (as stated by the Is_Total_Preording requirement). This requires that LEQV is both total and transitive. The sort is then specified to return a permutation of the incoming queue (Is_Permutation(#Q,Q)) that confirms to the ordering defined by the LEQV definition (Is_Conformal_With(LEQV, Q)).

```
Enhancement Sorting_Capability(Definition LEQV(x,y : Entry):B)
   for Queue_Template;
   uses Modified_String_Theory;
   requires Is_Total_Preordering(LEQV);

   Operation Sort(updates Q : Queue);
         ensures Is_Conformal_With(LEQV, Q) and
                 Is_Permutation(#Q, Q);

end Sorting_Capability;
```

Figure 27: Sorting Capability for a Queue

The enhancement declaration rule, similar to the concept declaration rule, simply adds the enhancement specification to the context.

```
Enhancement Enh(..)
   for CT;
   uses …;
   requires …;
```

```
        Operation Op(…);
              ensures …;

    end Enh;
```

Enhancement Declaration Rule:

$$\frac{C \cup \{Enh\} \setminus}{C \setminus Enh;}$$

An example sort implementation can be seen in Figure 28. The sort implementation is also parameterized for similar reasons. An operation, Compare, is required which ensures that the result of Compare is equal to the outcome of LEQV. An actual operation specific to the type of entries in the queue will need to be provided to perform an actual sort. The specification for the parameter operation will need to be at least as strong as the specification of Compare. The sort realization uses the selection sort algorithm based upon the operation provided. This implementation has a more complex loop invariant than we've encountered thus far. The invariant for the sort loop states that in each iteration Q and Sorted_Queue are a permutation of the initial Q (stated via the Is_Permutation definition), Sorted_Queue is ordered based on LEQV (stated via the Is_Conformal_with definition), and the entirety of Sorted_Queue is less than (as defined by LEQV) Q (stated via the Is_Universally_Related definition). These definitions are provided in the Modified_String_Theory math theory. Because sort is an enhancement and not an internal queue operation, the sort implementation cannot access the internal implementation variables. In fact, the actual realization of the queue concept which is used is not required by the sort enhancement code.

```
Realization Selection_Sort_Realization(
   Operation Compare(restores E1, E2 : Entry) : Boolean;
         ensures Compare = LEQV(E1, E2);)
   for Sorting_Capability of Queue_Template;
   uses Modified_String_Theory;

   Procedure Sort(updates Q : Queue);
         Var Sorted_Queue : Queue;
         Var Lowest_Remaining : Entry;

         While (Length(Q) > 0)
               changing Q, Sorted_Queue, Lowest_Remaining;
               maintaining
                     Is_Permutation(Q o Sorted_Queue, #Q) and
                     Is_Conformal_With(LEQV, Sorted_Queue) and
                     Is_Universally_Related(LEQV,
                                       Sorted_Queue, Q);
               decreasing |Q|;
         do
               Remove_Min(Q, Lowest_Remaining);
               Enqueue(Lowest_Remaining, Sorted_Queue);
         end;
         Q :=: Sorted_Queue;
   end Sort;
```

```
    Operation Remove_Min(updates Q : Queue;
                         replaces Min : Entry);
          requires |Q| /= 0;
          ensures Is_Permutation(Q o <Min>, #Q) and
                Is_Universally_Related(LEQV, <Min>, Q) and
                |Q| = |#Q| - 1;
    Procedure
          Var Considered_Entry : Entry;
          Var New_Queue : Queue;
          Dequeue(Min, Q);
          While (Length(Q) > 0)
                changing Q, New_Queue, Min, Considered_Entry;
                maintaining Is_Permutation(
                            New_Queue o Q o <Min>, #Q) and
                      Is_Universally_Related(LEQV,
                                  <Min>, New_Queue);
                decreasing |Q|;
          do
                Dequeue(Considered_Entry, Q);
                if (Compare(Considered_Entry, Min)) then
                      Min :=: Considered_Entry;
                end;
                Enqueue(Considered_Entry, New_Queue);
          end;
          New_Queue :=: Q;
    end Remove_Min;

end Selection_Sort_Realization;
```

Figure 28: Selection Sort of a Queue

A special enhancement realization rule is not necessary. For any procedure in the enhancement realization, the procedure declaration rule discussed in the last chapter is used. All of the 31VCs for this implementation are provable (actually, automatically, by the RESOLVE minimalist prover [46]) and can be found in Appendix E.

Facility Declarations and VCs

Now that VCs have been generated – and are all true – for the queue and sort code, a RESOLVE facility can instantiate and use these components. The following diagram depicts how this was implemented.

Consider the design for a system that must process a variety of prioritized jobs sequentially. A container object will be required to hold and process the jobs. A developer should first determine what kind of data type should be used to handle the jobs. For this example, let us assume the jobs will be processed in a first come first serve priority. Thus, a queue may make the most sense. Let's also remember that each job has a priority. Occasionally, the program should change the order of jobs to ensure high priority jobs are not held up behind low priority jobs. This will require the ability to sort the queue.

Figure 29 shows a sample RESOLVE facility. In this code, a facility is created where the Queue_Template is realized by the Circular_Array_Realiz and enhanced by the Sorting_Capability which is realized by Selection_Sort_Realization. Sorting_Capability is parameterized with the Priority_LEQV which prioritizes the jobs based on their priority. The corresponding operation, Priority_Order, which orders the job based on the priority as well.

```
Facility Sort_Job_Queue;
   uses Std_Boolean_Fac, Std_Integer_Fac,
            Std_Char_Str_Fac, Queue_Template;

    Type Job_Info = Record
        Name: Char_Str;
        Priority: Integer;
    end;

    Definition Priority_LEQV(S1,S2: Job_Info):
               Boolean = (S1.Priority <= S2.Priority);

    Operation Priority_Order(restores S1, S2: Job_Info):
                                                  Boolean;
        ensures Priority_Order = (Priority_LEQV(S1, S2));
      Procedure
        Priority_Order := (S1.Priority <= S2.Priority);
    end Priority_Order;
```

```
    Facility QF is Queue_Template(Job_Info, 10)
       realized by Circular_Array_Realiz
       enhanced by Sorting_Capability(Priority_LEQV)
         realized by
Selection_Sort_Realization(Priority_Order);

    Operation Main();
    Procedure
        Var S1, S2, S3, Temp: Job_Info;
        Var Q: QF.Queue;

        S1.Priority := 2;
        S2.Priority := 3;
        S3.Priority := 1;

        Enqueue(S1, Q);
        Enqueue(S2, Q);
        Enqueue(S3, Q);
        Sort(Q);
        Dequeue(Temp, Q);
        Write_Line(Temp.Priority);
        Dequeue(Temp, Q);
        Write_Line(Temp.Priority);
        Dequeue(Temp, Q);
        Write_Line(Temp.Priority);
    end Main;
end Sort_Job_Queue;
```

Figure 29: Sort Facility

There are a few interesting aspects in developing a proof rule and generating VCs for a facility declaration. When the queue facility is created, VCs must be created to show that the parameters provided to the facility meet the specifications. The correctness of facility declaration must be verified. So we present a facility declaration rule next. In this rule, whereas the facility instantiation hypothesis ensures that the actual definitions and operations passed as arguments satisfy the requirements of the formals, the facility initial expression is concerned with assuming that the global variables in the declared facility, if any, are properly initialized.

```
Suppose a Facility instantiation takes the form:

F_Instn = Facility FN is CN(IT, n_exp, IR)
                      realized_by RN(rn_exp, IRR, IRCR, IRP);

   where Operation IRP has the specification:
         Operation IRP (updates irx: RT2);
               requires preIRP /_rn_exp, irx_\;
               ensures postIRP /_rn_exp, #irx, irx _\;
```

Facility Instantiation Rule:

```
   Fac_Instantiation_Hyp;
   C ⋃{CT, RT} ⋃ {F_I_Spec} \ Assume I_Exp; code; Confirm RP;
   ____________________________________________________________
   C ⋃{CT, RT} \ F_Instn; code; Confirm RP;

where
   F_I_Spec is Facility Instantiation Specification

   I_Exp is GIC[S⇝ DExp, f⇝ F_Exp[rn⇝ rn_exp, RR⇝ IRR]
               [ n⇝ n_exp, R⇝ IR, T⇝ IT];

   Fac_Instantiation_Hyp is
         (RPC[rn⇝ rn_exp, RR⇝ IRR] ∧ CPC)[ n⇝ n_exp, R⇝ IR]
         ∧ ( preRP [rn⇝ rn_exp, rx⇝ irx] implies preIRP )
         ∧ ( postIRP implies postRP[rn⇝ rn_exp, #rx⇝ #irx,
                                                rx⇝ irx] );
```

For the present example, Max_Length must be greater than 0 (because of the requires clause in the concept Queue_Template), so a VC is generated to show that 10 > 0. A VC must also be generated to show that Priority_LEQV is a total pre-ordering (because of the requires clause in the Sorting_Capability enhancement). Figure 30 shows these two VCs.

```
VC: 3_1: Requirement for Facility Declaration Rule for QF:
Sorting_Capability.en(5)

Goal:Is_Total_Preordering(Priority_LEQV)
Given:

VC: 3_2: Facility Declaration Rule: Queue_Template.co(42)

Goal: (10  > 0)
Given:
1: true
```

Figure 30: Two VCs for QF Facility Declaration

In the facility declaration, the Sorting_Capability enhancement is realized by Selection_Sort_Realization. For this realization, Priority_Order is passed as an argument. Thus VCs must be generated to show that the Priority _Order operation is strong enough to be used as the Compare operator (given in the realization of Selection_Sort_Realization). One VC must show that the requires clause of Compare is strong enough to show that the requires clause of Order is true. This is easy because both operations have no requires clause. The resulting VC will simply have a goal that we need to prove 'true' with an assumption 'true'. There will be another group of VCs that must show that the ensures clause of Priority_Order implies the ensures clause of Compare. These are more intricate to generate but simple to prove and require replacing the parameters with the actual values so that it can be proven. This must also take into account the parameter modes. Both Priority_Order and Compare restore their parameters and a VC must show that to be the case in Priority_Order since Compare restores the parameters. These VCs are shown in Figure 31. The VCs generated are simple and all of them can be automatically proven by a minimalist prover.

```
VC: 2_1: Ensures from QF: Sort_Job_Queue.fa(13)

Goal: Priority_LEQV(S1, S2) = Priority_LEQV(S1, S2)

Given:
1: Priority_Order = Priority_LEQV(S1, S2)
2: #S1 = S1
3: #S2 = S2


VC: 2_2: Ensures from QF: Sort_Job_Queue.fa(13)

Goal: S1 = S1

Given:
1: Priority_Order = Priority_LEQV(S1, S2)
2: #S1 = S1
3: #S2 = S2


VC: 2_3: Ensures from QF: Sort_Job_Queue.fa(13)

Goal: S2 = S2

Given:
1: Priority_Order = Priority_LEQV(S1, S2)
2: #S1 = S1
3: #S2 = S2
```

Figure 31: VCs to show Operation is valid for QF Facility Declaration

CHAPTER FIVE

EXPERIMENTAL EVALUATION

In order to evaluate the VC generator, many VCs have been generated for a variety of component implementations. The evaluation process includes generating VCs and determining the provability of the generated VCs. The provability of the VCs has been checked both manually and, where possible, using the minimalist RESOLVE VC prover.

Benchmarks

Eight benchmarks are provided in [12]. Five of these benchmarks will be discussed in this chapter. Although various benchmarks for verification have been provided, this set of benchmarks was chosen because they show the ability to handle a variety of scenarios, including the ability to handle both built-in and user-defined types, and the ability to handle layered components to demonstrate the scalability of the system. The benchmarks that will not be presented involve topics not addressed in this dissertation, such as iterators (a construct that is not necessary and hence, absent in RESOLVE) and input/output formatting, specification of which requires further specification research. For each of the benchmarks shown, VCs have been generated and manually checked for provability.

Benchmark #1: Adding and Multiplying Numbers

Problem Requirements: Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive. [12]

Solution: An add and multiple enhancement has been written in RESOLVE. The enhancement specification and enhancement realization follow.

```
Enhancement Add_And_Multiply for Integer_Template;

    Operation Add(evaluates i: Integer;
                  evaluates j: Integer): Integer;
        requires (min_int <= i + j) and (i + j <= max_int);
        ensures  Add = ( i + j );

    Operation Multiply(evaluates I, J: Integer): Integer;
         requires (min_int <= -I) and (-I <= max_int) and
                  (min_int <= I * J) and (I * J <= max_int);
         ensures Multiply = I * J;


end Add_And_Multiply;


Realization Add_And_Multiply_Realiz for Add_And_Multiply of
               Integer_Template;
    uses Std_Boolean_Fac;

   Recursive Procedure Add(evaluates i: Integer;
                           evaluates j: Integer): Integer;
         decreasing |j|;
         Var zero:Integer;
         Add := Replica(i);
         If j > zero then
               Increment(Add);
                     Decrement(j);
                     Add  := Add(Add, j);
                     Increment(j);
         else
               If zero > j then
                     Decrement(Add);
                     Increment(j);
                     Add := Add(i, j);
                     Decrement(j);
               end;
         end;
   end Add;
```

```
    Procedure Multiply(evaluates I, J: Integer): Integer;
          Var nj, zero: Integer;
          Multiply := Replica(zero);
          If (J >= zero) then
                While (J > zero)
                      changing Multiply, nj, J;
                      maintaining Multiply + (I * J) = #I * #J
                            and  nj + J = #J;
                      decreasing J;
                do
                      Multiply := Add(Multiply, I);
                      Increment(nj);
                      Decrement(J);
                end;
          else
                While (J /= zero)
                      changing Multiply, J, nj;
                      maintaining Multiply - (I*J) = - #I*#J
                            and nj + J = #J  and J <= 0;
                      decreasing -J;
                do
                      Multiply := Add(Multiply, I);
                      Decrement(nj);
                      Increment(J);
                end;
                Multiply := Negate(Multiply);
          end;

          J :=: nj;
       end Multiply;
    end Add_And_Multiply_Realiz;
```

Figure 32: Add and Multiply Example

The VCs for this example have been generated and are located in Appendix F.

Benchmark #2:Binary Search in an Array

Problem Requirements: Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order. [12]

Solution: Two solutions are provided for this example. The first solution will demonstrate a basic binary search enhancement for a Static Array which has no post-condition. The VCs that will be generated here are VCs that are required to show that the pre-condition for each operation

call is true. This example will show that the VCs for the original binary search algorithm (mentioned to be erroneous in Bloch's blog [5]) in the introduction are indeed not provable.

```
Enhancement Simple_Search_Capability(definition LEQ(x,y:
Entry): B)
            for Static_Array_Template;
    uses Std_Boolean_Fac;
   requires Is_Total_Preordering(LEQ);

    Operation Is_Present(restores key: Entry;
                         restores A: Static_Array): Boolean;
         ensures true;

end Simple_Search_Capability;



Realization Simple_Binary_Search_Realiz(
      Operation Are_Ordered(restores x,y: Entry): Boolean;
         ensures Are_Ordered = (LEQ(x,y));
      )for Simple_Search_Capability of Static_Array_Template;
    uses Std_Boolean_Fac;

    Operation Are_Equal(restores x, y: Entry): Boolean;
         ensures Are_Equal = (x = y);
    Procedure
         Are_Equal := And(Are_Ordered(x, y),
                          Are_Ordered(y, x));
    end;
```

```
    Procedure Is_Present(restores key: Entry;
                         restores A: Static_Array): Boolean;
        Var low, mid, high: Integer;
        Var midVal, lowVal, highVal: Entry;


        Is_Present := False();
        low := Replica(Lower_Bound);
        high := Replica(Upper_Bound);
        mid := low;

        While (low <= high)
              changing low, mid, high, A, midVal, Is_Present;
              maintaining true;
              decreasing (high - low);
        do
              mid := high + low;
              Divide(mid, 2, mid);

              Swap_Entry(A, midVal, mid);
              if (Are_Equal(midVal, key)) then
                    Is_Present := True();
                    low := high + 1;
              else
                    if (Are_Ordered(midVal, key)) then
                          low := mid + 1;
                    else
                          high := mid - 1;
                    end;
              end;
              Swap_Entry(A, midVal, mid);
        end;

    end Is_Present;

end Simple_Binary_Search_Realiz;
```

Figure 33: Wrong Binary Search Example

Full VCs for the valid binary search example will be provided later. However, for this example, let us just consider a VC that show this example is invalid. The following VC will need to be true before the call to mid := high + low. The pre-condition for plus is that the sum of the two numbers is not larger than max_int. However, we can't prove this. If high and low were for example both equal to max_int, this code would fail, because of the following unprovable VC.

```
VC: 1_1:
Requires Clause of high + low in Procedure Is_Present:
Simple_Binary_Search_Realiz.rb(30)

Goal:
(min_int <= (high' + low')) and ((high' + low') <= max_int)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: true
8: (low'  <= high')
```

A correct binary search enhancement for Static Arrays has been written in RESOLVE. The enhancement specification and enhancement realization follow in Figure 34 and Figure 35. The specification requires that an ordering is provided, LEQ. LEQ must be a total, transitive, and symmetric. This is a different requirement than was used in the sorting example, and it guarantees that LEQ(x, y) and LEQ(y, x) implies x = y. The operation, Is_Present, returns a Boolean that is true if the key exists in the array and false if the key does not exist in the array. The array must be ordered based on LEQ.

```
Enhancement MySearch_Capability(definition LEQ(x,y: Entry): B)
            for Static_Array_Template;
    uses Std_Boolean_Fac;
    requires Is_Total(LEQ) and Is_Transitive(LEQ) and
              Is_Symmetric(LEQ);

    Definition Is_Ordered(A: Static_Array, From: Z,To: Z): B=
         (For all i: Z, if (From <= i and i < To)
                     then LEQ(A(i), A(i+1)));

    Definition Exists_Between(E: Entry, A: Static_Array,
                           From: Z, To: Z): B =
         (There exists i: Z such that
          (From  <= i and i <= To) and A(i) = E);

    Operation Is_Present(restores key: Entry;
                         restores A: Static_Array): Boolean;
      requires Is_Ordered(A, Lower_Bound, Upper_Bound) and
                     Upper_Bound + 1 <= max_int;
      ensures Is_Present =
                Exists_Between(key, A, Lower_Bound,
                               Upper_Bound);

end MySearch_Capability;
```

Figure 34: Binary Search Specification

```
Realization MyBinarySearchRealiz(
      operation Are_Ordered(restores x,y: Entry): Boolean;
         ensures Are_Ordered = (LEQ(x,y));
      )for MySearch_Capability of Static_Array_Template;
    uses Std_Boolean_Fac;

    Operation Are_Equal(restores x, y: Entry): Boolean;
         ensures Are_Equal = (x = y);
    Procedure
         Are_Equal := And(Are_Ordered(x,y),Are_Ordered(y,x));
    end;

    Definition Exists_Between(E: Entry, A: Static_Array,
            From: Z, To: Z): B = (There exists i: Z such that
               (From  <= i and i <= To) and A(i) = E);

    Procedure Is_Present(restores key: Entry;
                         restores A: Static_Array): Boolean;
         Var low, mid, high: Integer;
         Var midVal, lowVal, highVal: Entry;
         Is_Present := False();
         low := Replica(Lower_Bound);
         high := Replica(Upper_Bound);
         mid := low;
         While (low <= high)
               changing low, mid, high, A, midVal, Is_Present;
               maintaining Is_Present =
               (Exists_Between(key, A, Lower_Bound, low-1) or
                Exists_Between(key, A, high+1, Upper_Bound))
                and Lower_Bound <= low and high <= Upper_Bound
                and A = #A;
               decreasing (high - low);
         do
               mid := high - low;
               Divide(mid, 2, mid);
               mid := low + mid;
               Swap_Entry(A, midVal, mid);
               if (Are_Equal(midVal, key)) then
                     Is_Present := True();
                     low := high + 1;
               else
                     if (Are_Ordered(midVal, key)) then
                           low := mid + 1;
                     else
                           high := mid - 1;
                     end;
               end;
               Swap_Entry(A, midVal, mid);
         end;

    end Is_Present;
end MyBinarySearchRealiz;
```

Figure 35: Binary Search Implementation

Benchmark #3:Sorting a Queue

Problem Requirements: Specify a user-defined FIFO ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order. [12]

Solution: An enhancement to sort a generic queue has already been provided and VCs generated in Chapter 4.

Benchmark #4:Layered Implementation of a Map ADT

Problem Requirements: Verify an implementation of a generic map ADT, where the data representation is layered on other built-in types and/or ADTs. [12]

Solution: A search and store object has been specified and implemented in RESOLVE. This concept is parameterized by a Key type, and it allows the user to store keys in the store. A Store is modeled mathematically as a function from Key to Booleans. The user can then check to see if a specific key exists, remove a specific key or remove any key. The user can also clear the store. The parameters to the template are the type of key in the store and the maximum size of the store. The constraints of the store state that the maximum number of keys that can be added to the store is the Max_Capacity of the store. The store is initialized to having no keys.

```
Concept Search_Store_Template
         (Type Key; evaluates Max_Capacity: Integer);
   uses Std_Integer_Fac, Std_Boolean_Fac;
   requires Max_Capacity > 0;

   Definition Key_Ct (S: Store): N = |||{ k: Key, S(k)}|||;

   Type Family Store is modeled by (Key -> B);
         exemplar S;
         constraint Key_Ct (S) <= Max_Capacity;
            initialization ensures Key_Ct (S) = 0;

   Oper Add (restores k: Key; updates S: Store);
         requires    Key_Ct (S) < Max_Capacity and not S(k);
         ensures     S(k) and
               (for all k1: Key, if k1 /= k then
                     S(k1) = #S(k1));

   Oper Remove (restores k: Key; updates S: Store);
         requires    S(k);
         ensures     not S(k) and
               (for all k1: Key, if k1 /= k then
                           S(k1) = #S(k1));

   Oper Remove_Any (replaces k: Key; updates S: Store);
         requires    Key_Ct (S) > 0;
         ensures     #S(k) and not S(k) and
               (for all k1: Key, if k1 /= k then
                     S(k1) = #S(k1));

   Oper Is_Present(restores k: Key; restores S: Store):
                                                   Boolean;
         ensures     Is_Present= (S(k));

   Oper Key_Count (restores S: Store): Integer;
         ensures     Key_Count = Key_Ct (S);

   Oper Rem_Capacity (restores S: Store): Integer;
         ensures     Rem_Capacity = Max_Capacity - Key_Ct (S);

   Oper Clear (clears S: Store);
end Search_Store_Template;
```

Figure 36: Search and Store Specification

The store is implemented with a preemptable queue (a variation that has additional operations to manipulate queues.), and a specification of this concept is given in Appendix C. Each key is entered in the queue. No key may be added to the store (or queue) more than once. This example was originally implemented by students in an undergraduate software engineering course. A

UML diagram depicting this implementation is shown in Figure 37. Updates to the implementation to make the correspondence functional and remove the quantifiers in the conventions were made to the example. The conventions to the implementation of Search_Store_Template require that there are no duplicates in the queue. The correspondence maps each item that is in the preemptable queue to being in the store.

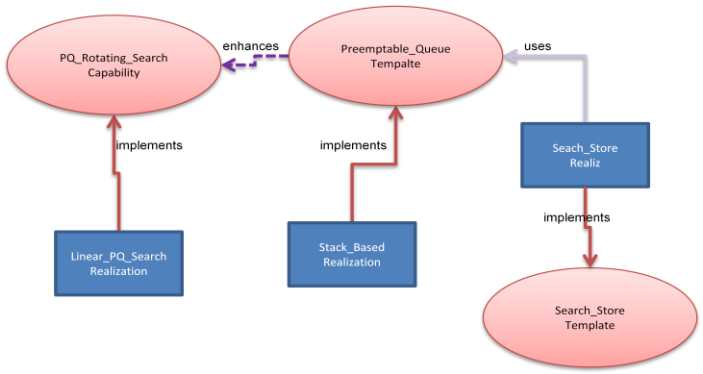


Figure 37: A UML Diagram for the Search and Store System

VCs for this example have been generated and manually checked to be correct. An VC from this example shows that the correspondence is well-defined and can be found in . All other expected VCs were also generated and can be located in Appendix F. This implementation uses an enhancement for the preemptable queue, which is also included in Appendix C.

```
VC: 1_1: Correspondence Rule for Store:
Search_Store_Realiz.rb(25)

Goal:
(Key_Ct(lambda k: Key (Is_Substring(<k>, S.Contents)))  <=
Max_Capacity)

Given:
1: (Max_Capacity  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Last_Char_Num  > 0)
5: (Max_Capacity  > 0)
6: (|S.Contents|  <= Max_Capacity)
7: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
8: Is_Duplicate_Free(S.Contents)
```

Figure 38: Example Search and Store VC

```
Realization Search_Store_Realiz (
       operation Are_Equal_Keys(evaluates i, j: Key): Boolean;
           ensures Are_Equal_Keys = (i = j);;
       operation Copy_Key(replaces Copy: Key;restores Orig: Key);
           ensures Copy = Orig;)
       for Search_Store_Template;
       uses Preemptable_Queue_Template, Std_Char_Str_Fac,
            Std_Integer_Fac, Std_Boolean_Fac,
         Modified_String_Theory;

       Facility PQ_Fac is Preemptable_Queue_Template(Key,
                        Max_Capacity)
           realized by Stack_Based_Realiz
         enhanced by Searching_Capability
           realized by Searching_Realiz(Are_Equal_Keys);

       Type Store is represented by Record
           Contents: PQ_Fac.P_Queue;
       end;
       Conventions Is_Duplicate_Free(S.Contents);
       Correspondence
            Conc.S = lambda k2:Key.(Is_Substring(<k2>,
                                    S.Contents));

       Procedure Add (restores k: Key; updates S: Store);
           Var t: Key;
           Copy_Key(t,k);
           Enqueue(t,S.Contents);
       end Add;


      Procedure Remove (restores k: Key; updates S: Store);
```

```
        Var b: Boolean;
        Search_and_Move(k,S.Contents,b);
        Dequeue(k,S.Contents);
    end Remove;

    Procedure Remove_Any (replaces k: Key; updates S: Store);
        Dequeue(k,S.Contents);
    end Remove_Any;

    Procedure Is_Present(restores k: Key;
                         restores S: Store): Boolean;
        Var b: Boolean;
        Search_and_Move(k,S.Contents,b);
        Is_Present := b;
    end Is_Present;

    Procedure Key_Count (restores S: Store): Integer;
        Key_Count := Length(S.Contents);
    end Key_Count;

    Procedure Rem_Capacity (restores S: Store): Integer;
        Rem_Capacity := PQ_Fac.Rem_Capacity(S.Contents);
    end Rem_Capacity;

    Procedure Clear (clears S: Store);
        PQ_Fac.Clear(S.Contents);
    end Clear;
end Search_Store_Realiz;
```

Figure 39: Search and Store Implementation

Benchmark #5:Linked-List Implementation of a Queue ADT

Problem Requirements: Verify an implementation of the queue type specified in benchmark #3, using a linked data structure for the representation. [12]

Solution: It is important that the VC Generator demonstrate the ability to generate VCs for commonly used structures, even if they are not the recommended development mechanism in the RESOLVE language. This ability will demonstrate that RESOLVE is not just a 'toy' language and that VCs can be generated for complex concepts. Specifically, although pointers are not generally recommended in RESOLVE, there may be times when they are necessary. Thus, we

will consider how pointers could be specified and verified in RESOLVE. More details for how pointers can be handled in RESOLVE are provided in [47].

The specification used to demonstrate how pointers can be handled in RESOLVE is Location_Linking_Template_1. The technical aspects of this specification are detailed in [45] and summarized in [47]. The present specification is incomplete in many respects and is presented here only as a proof of concept example. It is expected that the implementation of this concept will be hard-coded in the RESOLVE language, similar to arrays. Figure 40 specifies the behavior. Location_Linking_Template_1 has one parameter, a type that defines the type, Info, of information stored in each location. This is a simplified version of this template, in that only one link is permitted from each location. The more complex version would require a second parameter defining the number of links from a given location which would provide the ability to create more interesting data structures. This specification contains two global variables: Ref and Content. Ref is the link to the next Location. Contents is a variable of type Info which contains the data at the current Location. Each operation that updates one of these global variables must state then with the **updates** clause in the specification. Any operation that does not update is assumed to restore the global variable.

```
Concept Location_Linking_Template_1(type Info);
   uses Std_Integer_Fac, Std_Boolean_Fac,
         Modified_String_Theory;

   Definition Void: Z;
   Var Content: Z -> Info;
   Var Ref: Z -> Z;
   Facility_Initialization ensures for all L: Z,
         Info.Is_Initial(Content(L)) and Ref(L) = Void;

   Type Family Position is modeled by Z;
         exemplar P;
         initialization ensures P = Void;

   Operation Take_New_Location(updates P: Position);
         ensures P /= Void;

   Operation Relocate_to(updates P: Position;
                         preserves Q: Position);
         ensures (P = Q);

   Operation Follow_Link(updates P: Position);
         requires P /= Void;
         ensures P = Ref(#P);

   Operation Relocate_To_Target(updates P: Position;
               preserves Q: Position);
         requires Q /= Void;
         ensures P = Ref(Q);

   Operation Redirect_Link(preserves P: Position;
                           updates Q: Position);
         updates Ref;
         requires P /= Void;
         ensures Ref = lambda L:Z.(
                     {{#Q if L = P; #Ref(L) otherwise;}}) and
                     Q = #Ref(P);

   Operation Redirect_To_Target(updates P: Position;
                     preserves Q: Position);
         updates Ref;
         requires P /= Void and Q /= Void;
         ensures Ref = lambda L:Z.(
                     {{Ref(Q) if L = P; #Ref(L) otherwise;}});

   Operation Redirect_and_Update(preserves P: Position;
               updates Q: Position);
         updates Ref;
         requires P /= Void;
         ensures Ref = lambda L:Z.(
               {{#Ref(#Q) if L = P; #Ref(L) otherwise;}}) and
               Q = #Ref(P);
```

```
    Operation Is_At_Void(preserves P: Position): Boolean;
              ensures Is_At_Void = (P = Void);

    Operation Reset_To_Void(clears P: Position);

    Operation Swap_Info(preserves P: Position;
                              updates I: Info);
              updates Content;
              requires P /= Void;
              ensures I = #Content(P)
                    and Content = lambda L:Z.(
                          {{#I if L = P;
                          #Content(L) otherwise;}});
end;
```

Figure 40: Location Linking Template

We consider code here that uses the Location_Linking_Template to implement Queue_Template. This queue specification is slightly modified from the previous queue specification. Figure 41 shows a specification for an unbounded queue. This specification should not necessitate a detailed explanation. The only difference from the previous version is the lack of bounds.

```
Concept Unbounded_Queue_Template(type Entry);
    uses Std_Integer_Fac, Modified_String_Theory;

    Type Family Queue is modeled by Str(Entry);
    exemplar Q;
    initialization ensures Q = empty_string;

    Operation Enqueue(alters E: Entry; updates Q: Queue);
        ensures Q = #Q o <#E>;

    Operation Dequeue(replaces R: Entry; updates Q: Queue);
        requires Q /= empty_string;
        ensures #Q = <R> o Q;

    Operation Is_Empty(restores Q: Queue): Boolean;
     ensures Is_Empty = (Q = empty_string);

    Operation Clear (clears Q: Queue);
end Unbounded_Queue_Template;
```

Figure 41: Unbounded Queue

The following implementation of a queue uses the Location_Linking_Template. Because this realization defines the type of the queue as a position, a facility is used to instantiate the

Location_Linking_Template_1 with the type, Entry. The correspondence of this queue requires the string that represents the queue is equal to the string created by concatenation each entry of the linked list. The conversions require that there are no loops – that void is reachable. This implementation is not complete in many respects [47]. For example, it is missing conventions, such as no two queues share locations. It is mainly intended to show the use of global variables in VC generation and how the verification machinery is suitable for handling code with pointer behavior.

```
Realization Queue_Location_Linking_Realiz for
         Unbounded_Queue_Template;
   uses Location_Linking_Template_1;

   definition Is_Reachable(first: Z, last: Z,
         refContext : Z -> Z) : B = {{(true) if first = last;
         Is_Reachable(refContext(first), last, refContext)
                     otherwise;}};

   definition Str_Info(first: Z, refContext : Z -> Z,
               contentsContext : Z -> Entry): Str(Entry) =
         {{empty_string if first = Void;
           <contentsContext(first)> o
               Str_Info(refContext(first), refContext,
               contentsContext) otherwise;}};

    Facility Entry_Ptr_Fac is
          Location_Linking_Template_1(Entry)
        realized by Std_Location_Linking_Realiz;

    Type Queue is represented by Record
         Front: Entry_Ptr_Fac.Position;
         Back: Entry_Ptr_Fac.Position;
    end;

    convention Is_Reachable(Q.Front, Q.Back, Ref) and
               (Ref(Q.Back) = Void) and
               (Q.Back = Void iff Q.Front = Void);
    correspondence Conc.Q = Str_Info(Q.Front, Content, Ref);

    Procedure Dequeue(replaces R: Entry; updates Q: Queue);
         Var Temp: Position;

         Swap_Info(Q.Front, R);
         Follow_Link(Q.Front);
         If (Is_At_Void(Q.Front)) then
               Reset_To_Void(Q.Back);
```

```
        end;
    end Dequeue;

    Procedure Enqueue(alters E: Entry; updates Q: Queue);
        Var Temp: Position;

        Take_New_Location(Temp);
        Swap_Info(Temp, E);
        If (Is_At_Void(Q.Front)) then
              Relocate_to(Temp, Q.Back);
              Relocate_to(Q.Back, Q.Front);
        else
          Redirect_Link(Q.Back, Temp);
        end;
    end Enqueue;

    Procedure Is_Empty(restores Q: Queue) : Boolean;
        Var Temp: Position;

        Is_Empty := Is_At_Void(Q.Front);
    end Is_Empty;

    Procedure Clear(clears Q: Queue);
        Reset_To_Void(Q.Front);
        Reset_To_Void(Q.Back);
    end Clear;
end Queue_Location_Linking_Realiz;
```

Figure 42: Queue Implementation with a Linked List

VCs for each of these operations have been generated and are available in Appendix F. To further motivate why pointers are not the desired mechanism to implement data structures in RESOLVE, the following example demonstrates a much simpler realization of Unbounded_Queue_Template. The code in Figure 43 is much shorter than the linked implementation in Figure 42, allowing less opportunity for bugs to be introduced and less VCs that must be proved. The linked list implementation generated 39 VCs whereas the list implementation only generated 11.

```
Realization UnboundedQueue_List_Realiz for
               Unbounded_Queue_Template;
    uses Unbounded_List_Template, Boolean_Template;

    Facility List_Fac is Unbounded_List_Template(Entry)
        realized by Std_Unbounded_List_Realiz;

    Type Queue is represented by List_Fac.List;

    convention true;
    correspondence Conc.Q = Q.Prec o Q.Rem;

    Procedure Dequeue(replaces R: Entry; updates Q: Queue);
        Reset(Q);
        Remove(R, Q);
    end Dequeue;

    Procedure Enqueue(alters E: Entry; updates Q: Queue);
        Advance_to_End(Q);
        Insert(E, Q);
    end Enqueue;

   Procedure Is_Empty(restores Q: Queue): Boolean;
       Is_Empty := And(Is_Prec_Empty(Q), Is_Prec_Empty(Q));
    end Is_Empty;

    Procedure Clear(clears Q: Queue);
       Clear(Q);
    end Clear;

end UnboundedQueue_List_Realiz;
```

Figure 43: List Implementation of Unbounded Queue

An Exercise with Auxiliary Variables

We conclude this chapter by considering an example that contains an existential quantifier in the specification of the operation. In general, developers should be able to write specifications without existential quantifiers so that auxiliary variables are not required in implementations. For example, if an operation was specified to move one element from the front of a queue to the back of a queue, an existential quantifier is not necessary. Rotating_Capability, an enhancement to Queue_Template, can be written using the Prt_Btwn definition. Prt_Btwn is defined in String_Theory and returns a portion of the string where the first argument indicates the starting position, the second argument indicates the ending position, and the third argument indicates the

source string. Prt_Btwn assumes the string is indexed beginning at 0. Figure 44 provides a specification with a simple ensures clause that would be a good option and would provide VCs that are easy to prove. However, to demonstrate the use of auxiliary variables, we will also consider how this could be specified with existential quantifiers.

```
Enhancement Rotating_Capability for Queue_Template;
   Operation Rotate( updates Q: Queue );
         requires |Q| /= 0;
         ensures Q = Prt_Btwn(0,1,#Q) o Prt_Btwn(1,|#Q|,#Q)
end Rotating_Capability;
```

Figure 44: Queue Rotate with Alternate Specification

So suppose instead that Rotate is specified using existential quantifiers as in Figure 45. In writing an implementation of a specification with such quantifiers, RESOLVE provides the ability to define verification variables or auxiliary variables; also, termed ghost or adjunct variables in [1], [2] and [10]. These are not used to translate and execute the code but are used to help the VC generator create VCs that are more easily proved. These auxiliary variables coincide with the mathematical variable in the existential quantification. The auxiliary code is used to set the value of the auxiliary variable which is then used to generate easier-to-prove VCs. Thus, the VC prover will know the value of the existential variables and won't have to "guess" at the value. This can be most easily explained with an example.

The specification in Figure 45 states that there exist two entries, E and R. There are also two auxiliary variables, the entry E and the Queue R. The auxiliary variables, E and R, are set using the auxiliary code (Queue_Replica and Entry_Replica). Auxiliary code is ignored by the translator but the VC generator treats the Auxiliary variables and Auxiliary code just like normal variables and code. The only difference is that there won't be actual realizations for the auxiliary operations!

```
Enhancement Rotating_Capability for Queue_Template;

   Operation Rotate( updates Q: Queue );
         requires |Q| /= 0;
         ensures there exists E: Entry,
                 there exists R: Str(Entry),
                     #Q = <E> o R and Q = R o <E>;

end Rotating_Capability;

Realization Obvious_Rotate_Realiz for
               Rotating_Capability of Queue_Template;
   Aux Operation Entry_Replica (restores E: Entry): Entry;
         ensures Entry_Replica = E;

   Aux Operation Queue_Replica (restores P: Queue): Queue;
         ensures Queue_Replica = P;

   Procedure Rotate( updates Q: Queue );
         Var TE: Entry;
         Aux_Var E: Entry;
         Aux_Var R: Queue;
         Dequeue( TE, Q );
         Aux_Code
               E := Entry_Replica(TE);
               R := Queue_Replica(Q);
         end;
         Enqueue( TE, Q );

   end Rotate;
end Obvious_Rotate_Realiz;
```

Figure 45: Queue Rotate with Auxiliary Variables Specification and Implementation

Consider Appendix F that shows the VCs generated for Queue Rotate with the auxiliary code and without the auxiliary code. VC 1 and VC 2 contain only minor differences between the two versions. However VC 3 (and 4) are much simpler to prove in the example using auxiliary code. Each of the VCs for the auxiliary code were automatically proved by the RESOLVE prover.

CHAPTER SIX

EDUCATIONAL USES

One important use of the RESOLVE verifier is its value in the classroom. It has been used at multiple universities to provide a foundation for logical reasoning about software design. All too often students, just like other software developers, write bug-filled code with an attitude that most bugs will be found during the testing process. Often that wrong assumption leads to a lack of proper understanding of their own code. Using RESOLVE to teach students demonstrates both the need and viability of creating provably correct software and leads students to create software that has been properly designed based on the corresponding specifications. [48]

There are several different ways the web interface for the RESOLVE verifier is currently being used in the classroom. The most obvious use of RESOLVE is in a software engineering course. However, it has also been used at a few schools in a theory and programming language course. Students interacted with the VC generator to varying degrees at different schools. The web-interface for the VC generator was used by the students at the school in the list below.

- Clemson University
    - Junior/Senior-Level Software Engineering Course
    - Graduate-Level Software Engineering Course
- Cleveland State University
    - Junior/Senior-Level Software Engineering Course
    - Graduate-Level Software Engineering Course
- Denison University
    - Junior/Senior-Level Software Engineering Course
    - Theory and Programming Language Course
- Depauw university
    - Theory and Programming Language Course
- Ramapo College
    - Theory and Programming Language Course
- University of Alabama
    - Junior/Senior-Level Software Engineering Course

- VT Northern Virginia Campus
    - Junior/Senior-Level Software Engineering Course
    - Graduate-Level Software Engineering Course
- Western Carolina University
    - Junior/Senior-Level Software Engineering Course

To understand the educational benefits provided by the VC generator, let us consider how it has been used at Clemson in a junior-level software engineering course. The students have been assigned a team project for the past several semesters used to teach, by example, how helpful formal specifications and verification are in contract-based software design. With each group using the same specifications, the implementations from various groups were combined and executed. Because full verification was not yet possible, these did not always work perfectly, but students saw that most of the time the process worked. This demonstrated the feasibility of the using contracts to design and build software.

At other universities (Clemson, Alabama, Western Carolina, DePauw, and Cleveland State) VCs were generated for examples using the web interface during classtime. Students were able to edit the code to attempt various 'What If' scenarios. This allowed students to see and use a VC generator to understand the purpose and usefulness of verification.

Recursive Queue Append Example

This example shows the type of 'What If' scenarios that can be demonstrated in undergraduate courses. RESOLVE provides support for recursion and thus must be able to generate VCs to show termination of the recursion in addition to the normal VCs required for showing that the post-condition holds. Similar to the **While** loop, recursive procedures require a decreasing clause to show termination. The example in Figure 46 shows an operation, Append, which accepts two queues as parameters and will append the second queue, Q, to the first queue, P. The queue, Q, will be empty after the operation completes. With each recursive call by append the queue, Q,

will decrease in length (by one) with Append removing one entry from Q and then recursively calling Append on the resulting Q. As can be seen, the implementation of Append defines the length of Q as the decreasing clause. So it must be shown that this is true before each recursive call. After the recursive call to Append, the entry removed from Q before the call is added to P.

```
Enhancement Append_Capability for Queue_Template;
    Operation Append(updates P: Queue; clears Q: Queue);
        requires |P| + |Q| <= Max_Length;
        ensures P = #P o #Q;
end Append_Capability;

Realization Recursive_Append_Realiz for Append_Capability
         of Queue_Template;
        uses Std_Boolean_Fac;
    Recursive Procedure Append(updates P: Queue;
                                       clears Q: Queue);
        decreasing |Q|;

      Var E: Entry;
      If (Length (Q) /= 0) then
          Dequeue(E,Q);
          Enqueue(E,P);
          Append(P,Q);
      end;
    end Append;
end Recursive_Append_Realiz;
```

Figure 46: Queue Recursive Append Specification and Implementation

When generating VCs for recursive operations, it is necessary to update the operation call rule and procedure declaration rule. Each of these will be modified to use and maintain P_Val based on the decreasing clause (similar to the **While** loop). The procedure declaration and operation call rule with updates to support recursion follow. The italicized portions of the proof rules have been added to handle recursion. These proof rules will construct VCs such that there is an assumption that P_Val is equal to the decreasing statement at the start of the operation. However, before the recursive call, we must show that the value the decreasing statement has decreased.

Assuming an example operation, P, with one parameter being updated and one parameter being cleared, the modified, recursive procedure declaration and call rules follow.

```
CDP =  Operation P( updates t: T1; clears u: T2);
            requires Pre/_ t, u _\;
            ensures Post/_ #t, #u, t _\;

Recursive Procedure Declaration Rule:

C ⋃{CDP}\ Assume Pre ∧ T1.Constraint(t) ∧ T2.Constraint(u);
      Remember;
      Assume P_Val = P_Exp;
            body;
      Confirm Post ∧ T2.is_initial(u);
C ⋃{CDP}\ code; Confirm RP;
_____________________________________________________________________
C ⋃{CDP}\ Recursive Proc P(… ); decreasing P_Exp; body; end P;
            code; Confirm RP;

Recursive Procedure Invocation Rule:

C ⋃{CDP}\ code; Confirm Invk_Cond(P(a,b)) and P_Exp < P_Val;
   Assume Post[t⇝NQV(RP, a), #t⇝a, #u⇝b] and T2.is_initial(b);
   Confirm RP[a⇝NQV(RP, a)];
_____________________________________________________________________
C ⋃{CDP}\ code; P( a, b); Confirm RP/_ a, b _\ ;
```

Each of the VCs generated by this example are provable. There are 8 generated VCs that are generated for the Append operation. Three VCs are generated from pre-conditions to operation calls, one will show the recursion terminates, and one will show the final goal is achieved.

VC 0_3 is the most interesting of these VCs at it shows termination. The goal of the VCs is that the size of Q decreases before the recursive call. This VC is provable based on assumption 10 which exists because of the Dequeue of E from Q. The other VCs are located in Appendix G.

```
VC: 0_3:  Show Termination of Recursive Call:
          Recursive_Append_Realiz.rb(5)
Goal: (|Q'| < |Q|)
Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
12: Q = (<E'> o Q')
```

Figure 47: Example VC for Recursive Append

Another interesting example is one where a programmer supplies an incorrect decreasing clause. In that case, although the append will still work correctly, the code will be invalid since we cannot show termination. Consider Figure 47. In this example, we have indicated that the size of the Queue, P, is decreasing. However, the size of the Queue, P, is in fact increasing with each level of recursion.

```
Realization Recursive_Append_Realiz for Append_Capability
         of Queue_Template;
        uses Std_Boolean_Fac;
    Procedure Append(updates P: Queue; clears Q: Queue);
        decreasing |P|;

      Var E: Entry;
      If (Length (Q) /= 0) then
          Dequeue(E,Q);
          Enqueue(E,P);
          Append(P,Q);
      end;
    end Append;

end Recursive_Append_Realiz;
```

Figure 48: Invalid Queue Recursive Append implementation

The VCs generated from the example in Figure 48 are nearly identical to the VCs generated from Figure 46 except for the VC that shows termination. The complete list of VCs are included in Appendix G. All of the VCs are still provable except for VC 0_3 shown in Figure 49 This VC is not provable because |(P o <E'>)| is greater than |P| which is generated because we are appending an entry to P with each level of recursion. Thus we cannot show that the recursion ends.

```
VC: 0_3:
Show Termination of Recursive Call:
Recursive_Append_Realiz.rb(5)

Goal:
(|(P o <E'>)| < |P|)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0
12: Q = (<E'> o Q')
```

Figure 49: Example VC for Recursive Append with Wrong Decreasing Clause

Let's consider another variation of the recursive Append example in Figure 50.

```
Realization Recursive_Append_Realiz for Append_Capability of
Queue_Template;
        uses Std_Boolean_Fac;
    Procedure Append(updates P: Queue; clears Q: Queue);
        decreasing |Q|;

      If (Length (Q) /= 0) then
          Append(P,Q);
      end;
    end Append;

end Recursive_Append_Realiz;
```

Figure 50: Another Invalid Queue Recursive Append Implementation

As can be seen in this example, we have supplied the |Q| as the decreasing clause again. The previous example in Figure 48 was a correct implementation but invalid because of the wrong specification. However, this is an example where the VCs are not going to be provable because it's simply a wrong implementation. Similar to the previous example, only the VC that shows termination is unable to be proved. Because the size of Q does not decrease before calling Append, we cannot prove VC 0_1 in Figure 51 that would show the recursion is finite. These VCs are also located in Appendix G.

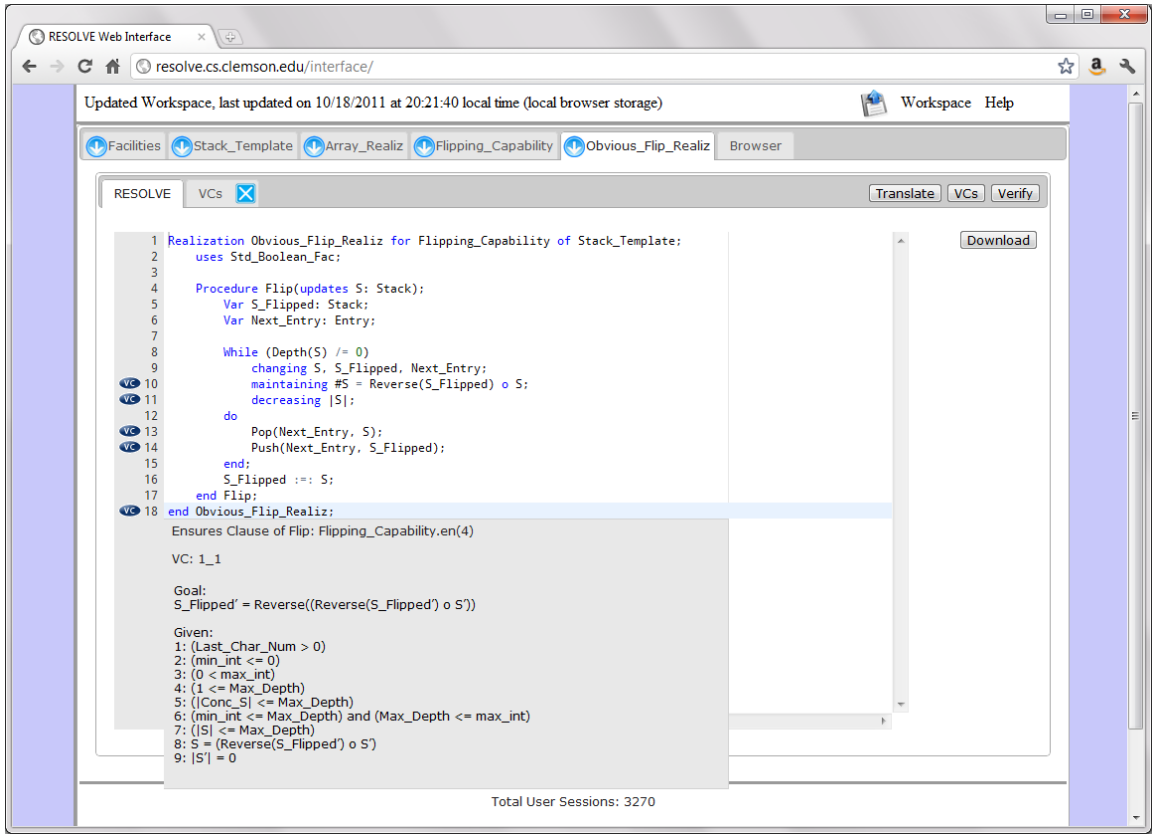
```
VC: 0_1:
Show Termination of Recursive Call:
Recursive_Append_Realiz.rb(5)

Goal:
(|Q| < |Q|)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
```

Figure 51: Example VC for Recursive Append with wrong implementation

User Feedback

The VC generator also needs to be able to provide appropriate user feedback. Similar to a compiler, users should be able to identify which portions of code generated a VC so they know where to begin to debug. This can be done by relating line numbers to each VC. The current VC generator can specify where the goal of an assertion originates and thus debugging based on unprovable assertions will become an easier process. The current VC

generator can also relate each assumption to a line number, though it is currently unclear if that will be a commonly used option. When each assertion is formed, the line number of the code is connected to the related portion of the code. The current web interface, designed by Clemson University students, has made use of this information to create a very useful tool [48].



CHAPTER SEVEN

CONCLUSIONS AND FUTURE DIRECTIONS

This research has shown that automated VC generation is not only possible but practical, by designing mechanizable proof rules and realizing them within a full-fledged object-based language, bridging the large gap between theoretical principles and actual verification. These proof rules are scalable, allowing verification of all data types in many data constructs. The system also allows for modular generation of VCs, verifying each component independently. VCs have been generated for a wide variety of examples in addition to a set of benchmarks demonstrating the goal of verification – the Grand Challenge provided by Tony Hoare [4]– is achievable.

Several directions for further research remain. The proof rules and the VC generator in this dissertation cover only a subset of RESOLVE, albeit a large subset. Specifically, the treatment of global variables is minimal at best. But the work here provides a solid theoretical and practical basis to build on. Similarly, only initial semantics for the RESOLVE language have been formalized and only examples of the soundness of mechanizable rules have been presented in this dissertation. Ultimately, more complete semantics for the RESOLVE language are needed, including formal semantics for data abstraction. Additionally, details about the relational semantics need to be formalized.

One reason to formalize the semantics is so that the VC generator itself can be verified. Much like Pascal once demonstrated that a compiler could be written in a high-level language and compiled, the verifier should be written in RESOLVE that supports full verification and verified. The present verifier can be used for this bootstrapping. Many questions remain as to how this should

be done. It must be determined which components are best suited for this task and how each of these components should be specified and verified.

There is much more work to be done to have a completely automated verification system. Currently, work is continuing on the RESOLVE minimalist prover. The prover is now able to automatically prove many of the VCs generated by the VC generator. Many others cannot be proved. A key reason for this is simply the result of incomplete mathematical theories. As the theories are used and expanded and the prover is improved, the abilities to prove more software correct will also grow. Currently, the prover is a completely independent step which follows the VC generation process. It should be considered whether the two steps should be intertwined to provide a more efficient and capable verifier.

Another area of research is verification of performance. Performance profiles can be written to complement the specifications of functional behavior (concepts). These profiles will specify the time and space requirements of the software. Thus, not only will the verifier be able to prove that the software is correct in functionality but can also prove that the software runs within the specified time constraints and resource constraints. These will be modular and scalable, and can be layered on the mechanizable proof rules and VC generator, conceived in this dissertation.

In conjunction with the online RESOLVE web-interface, the VC generator has provided the ability to allow many students and researchers to use RESOLVE. This allows students to both learn to reason about the software and shows that verification is an attainable goal. For example, the VC generator has been used for a PayPal type system for research at Virginia Tech, Northern Virginia Center and for verifying a piece of sensor network code at Clemson University. Other software needs to be developed with the VC generator to test and demonstrate its ability to verify large software applications.

As the verifier develops into a fully automatic system, students should learn to use the verifier instead of a debugger to develop software. This will provide students with a better understanding of the code they have written, and should lead to stronger developers in the workplace, skilled in mathematical reasoning, even if they may never use a verifier.

APPENDICES

Appendix A

Proof Rules

In the following rules,

- `RP` is the result predicate
- `code` refers to the statements preceding the construct
- `C` is the context
- `Invk_Cond(exp)` conjoins all pre-conditions for all the programming functions in *exp*
- `Math(exp)` composes the mathematical expressions for all the programming functions in *exp*.
- `BE` is a Boolean valued programming expression.
- *P_Exp* is the ordinal valued progress metric expression, the system variables $?^k$**P_Val** hold progress metric values
- `NQV( RD, x )` produces a next question marked variable name of the form $?^m x$ such that *m* is the smallest value for which $?^m x$ doesn't occur in *RD*.
- `CExp/_t_\` is the constraint expression for *T*.

Assume Rule:

```
C\ code; Confirm exp => RP;
_____________________________________________
C\ code; Assume exp; Confirm RP;
```

Swap Rule:

```
C\ code; Confirm RP[ x⇝y, y⇝x ];
_____________________________________________________
C\ code; x :=: y; Confirm RP;
```

Function Call/Expression Reassignment Rule:

```
C\ code; Confirm Invk_Cond(exp) ∧ RP[x⇝Math(exp)];
_____________________________________________________
C\ code; x := exp; Confirm RP;
```

If/Else Rule:

```
C\ code; Confirm Invk_Cond(BE); Assume Math(BE); code1;
   Confirm RP;
C\ code; Assume ¬ Math(BE); code2; Confirm RP;
_____________________________________________________
C\ code; If BE then code1 else code2 end_if; Confirm RP;
```

If Rule:

```
C\ code; If BE then code1 else end_if; Confirm RP;
_____________________________________________________
C\ code; If BE then code1 end_if; Confirm RP;
```

Confirm Rule:

```
C\ code; Confirm exp ∧ RP;
_____________________________________________________
C\ code; Confirm exp; Confirm RP;
```

While Rule:

```
C/ code; Confirm Inv; Change Vlist; Assume Inv ^
      NQV(RP, P_Val)= P_Exp;
      If BE then body; Confirm Inv ^ P_Exp < NQV(RP, P_Val);
      else Confirm RP end_if;
      Confirm True;
_____________________________________________________________
C/ code; While B
            maintaining Inv;
            decreasing P_Exp;
            changing VList;
         do
            body
         end;
         Confirm RP;
```

Change Rule:

```
(The context indicates that x is of type T.)
C ⋃{NQV(RP, x): T}\ code;
Confirm RP[x⇝NQV(RP, x)];
_____________________________________________________________
C\ code; Change x; Confirm RP;
```

Remember Rule:

```
C\ code; Confirm RP[#s⇝s, #t⇝t];
_____________________________________________________________
C\ code; Remember; Confirm PR/_ s, #s, t, #t, u, v, ⋯ _\;
```

The following rules will use the example operation template for P given below.

```
CDP =  Operation P( updates t: T1; evaluates u: T2;
         replaces v: T3; restores w: T4; preserves x: T5;
         alters y: T6; clears z: T7);
            requires Pre/_t, u, w, x, y, z, _\;
            ensures Post/_ #t, u, w, x, #y, #z, t, v, _\;
```

Operation Declaration Rule:

```
C ⋃{CDP} \
_____________________________________________________________________
C \ CDP;
```

Operation Invocation Rule:

```
C ⋃{CDP}\ code; Confirm Invk_Cond(P(a,exp,b,c,d,e,f));
   Assume ( T1.Constraint(t) ∧ T3.Constraint(v) ∧
   T6.Constraint(y) ∧ Post) [t⇝NQV(RP, a), #t⇝a, u⇝Math(exp),
   v⇝NQV(RP, b), w⇝c, x⇝d, #y⇝e, #z⇝f] ∧
   T7.is_initial(NQV(RP, f));
   Confirm RP[a⇝NQV(RP, a), b⇝NQV(RP, b), e⇝NQV(RP, e),
   f⇝NQV(RP, f)] );
_____________________________________________________________________
C ⋃{CDP}\ code; P( a, exp, b, c, d, e, f);
            Confirm RP/_ a, b, c, d, e, f, g, h, ⋯ _\;
```

Procedure Declaration Rule:

```
C ⋃{CDP}\ Assume Pre ∧ T1.Constraint(t) ∧ T2.Constraint(u) ∧
      T3.Is_Init(v) ∧ T4.Constraint(w) ∧ T5.Constraint(x) ∧
      T6.Constraint(y) ∧ T7.Constraint(z);
      Remember;
            body;
      Confirm Post ∧ w = #w ∧ T7.is_initial(z);
C ⋃{CDP}\ code; Confirm RP;
_____________________________________________________________________
C ⋃{CDP}\ Proc P(… ); body; end P; code;
            Confirm RP;
```

Suppose a Concept Template is specified by:

```
CT = Concept CN( type T; eval n: U; def R: T×V→B );
              uses AFac, BTh;
              requires CPC/_ n, R _\;
         Definition S: W→B = ( DExp );
         Defines f: W→T;
         constraint DC/_ f, n, R _\;
         Var gv: X1;
         constraint VC/_ gv, f, n, R _\;

         Facility_Initialization
              ensures GIC/_ gv, f, S, n, R _\;

         Type Family TF is modeled by MTE;
              exemplar x;
              constraint TC/_ x, n, R, gv, f _\;
              initialization
                   updates gv;
                   ensures IC/_ x, gv, f, n, R _\;
              finalization
                   updates gv;
                   ensures FC1/_#x, gv, f, n, R_\;

         Oper P( updates x: TF; evaluates z: U );
              updates gv;
              requires Pre/_x, y, gv, f, n, R_\;
              ensures Post/_x, #x, y, gv, #gv, f, n, R_\;
   end CN;
```

Concept Declaration Rule:

```
C ⋃{CT}\ code; Confirm Q;
_____________________________________________________________
C\ CT; code; Confirm Q;
```

Suppose a Concept implementation is specified by:

```
RT =  Realization RN( eval rn: RU; def RR: RT×RV→B;
                  Realization F_Realiz(eval e:T3);
                  Procedure RP( updates rx: RT2 );
                        requires preRP/_ rx, rn _\;
                        ensures postRP/_ rx, #rx, rn _\)
      for CN;
            uses RAFac, RBTh, GTy, R_C;
            requires RPC/_ rn, RR _\;)
            Definition RS: RW→B = ( RDExp );
            Definition f: W→T = (F_Exp);
            constraint RDC/_ rn, RR _\;

   Facility F is R_C( f_exp/_ rx, rn _\, GTy, RR, RS)
      realized_by F_Realiz(f_exp/_ rx, rn _\, RR, RS);

   (* Treatment of rules does not include realization globals *)
   (*
   Var rg1: RX;
   Aux_Var ra: Tm;
      convention RGC//_ ra, rg, f, rn, RR, RS _\;
      correspondence CR_Exprg//_ gv, ra, rg1, rn, RR, RS _\;
      Facility_Initialization GI_body; end;
   *)

   Type TF = RT;
      conventions RC/_ x, rg1, f, rn, RR, RS_\;
      correspondence[3] conc.x = Cor_Fn_Exp/_x, gv, rg, rg, ra,
                                          f, n, rn, R, RR, S,
                                    RS_\;
      Facility_Initialization I_body; end;

   Procedure P (updates x: TF; evaluates z: U);
      p_body;
   end P;

end RN;
```

Please see footnote #1 for how the rule changes if the correspondence is relational.

Realization Declaration Rule:

```
C ⋃{R_C}\ Fac_Instantiation_Hyp;
C ⋃{CT}\ Well_Def_Corr_Hyp;
C ⋃{CT}\ T_Init_Hyp;
C ⋃{CT}\ Correct_Op_Hyp;
C ⋃{CT}⋃{R_Heading}\ code; Confirm Q;
_____________________________________________________________________
C ⋃{CT}\ RT; code; Confirm Q;
```

Well_Def_Corr_Hyp shows that the correspondence is well defined.

```
Assume CPC/_ n, R _\ ∧ RPC/_ rn, RR_\ ∧
            RC/_ rg, f, rn, RR, RS_\;
            // Assumes Concept Level Requires, Concept
            // Realization level requires and convention
Confirm TC[x ⇝ Cor_Fn_Exp];
```

T_Init_Hyp establishes that the type initialization is done correctly.

```
Assume CPC ∧ RPC;
var x1: RT;
I_Body;
Confirm RC;
Confirm IC/_ x ⇝ Cor_Func_Exp, gv, f, n, R _\;
```

Correct_Op_Hyp establishes the correctness of each procedure.

```
Assume CPC ∧ DC ∧ VC ∧ RPC ∧ RDC ∧
      RC ∧ TC ∧ Pre[x ⇝ Cor_Fn_Exp];
Remember;
      p_body;
Confirm RC/_ x, rg, f, rn, RR, RS_\
             ∧ Post[x ⇝ Cor_Fn_Exp];
```

Suppose a Facility instantiation takes the form:

```
F_Instn = Facility FN is CN(IT, n_exp, IR)
                         realized_by RN(rn_exp, IRR, IRCR, IRP);

      where Operation IRP has the specification:
            Operation IRP (updates irx: RT2);
                  requires preIRP /_rn_exp, irx_\;
                  ensures postIRP /_rn_exp, #irx, irx _\;
```

Facility Instantiation Rule:

```
Fac_Instantiation_Hyp;
C ⋃{CT, RT} ⋃ {F_I_Spec} \ Assume I_Exp; code; Confirm RP;
_____________________________________________________________
C ⋃{CT, RT} \ F_Instn; code; Confirm RP;

   where
      F_I_Spec is Facility Instantiation Specification

      I_Exp is GIC[S⇝ DExp, f⇝ F_Exp[rn⇝ rn_exp, RR⇝ IRR]
                  [ n⇝ n_exp, R⇝ IR, T⇝ IT];

Fac_Instantiation_Hyp is
      (RPC[rn⇝ rn_exp, RR⇝ IRR] ∧ CPC)[ n⇝ n_exp, R⇝ IR] ∧
      ( preRP [rn⇝ rn_exp, rx⇝ irx] implies preIRP ) ∧
      ( postIRP implies postRP[rn⇝ rn_exp, #rx⇝ #irx, rx⇝ irx]);
```

Suppose an explicit facility is specified by:

```
F = Facility FN;
            uses AFac, BRealiz, CCon;

    var gv: X;
    initialization
          ensures XI/_ gv_\;                    } Global_Info;

    procedure;
       G_I_Body;
    end;
        Type T = PTE;
           exemplar x;
           Def. D: W → B = ( DE /_ x, gv _\);
           constraints TCE /_ x, gv_\;          } Type_Info;
           initialization
           ensures TIE /_ x, gv_\;
       procedure;
             T_I_Body;
       end;


    Operation P1(updates X: T; updates gv: X);
         requires Pre/_x, gv _\;                } Op_Info1
         ensures Post/_x, #x, gv, #gv _\;


    procedure P1;
             P1_body;
       end P1;
       ⋮


    end FN;

    F_Spec = {Global_Info, Type_Info, Op_Info1, ⋯ }
```

Explicit **Facility** Declaration Rule:

```
C \ G_I_F_Hyp;
C ⋃{Global_Info} \ T_I_F_Hyp;
C ⋃{Type_Info} \ Correct_Op_Hyp;
C ⋃ F_Spec \ code; Confirm RP;
_____________________________________________________________
C \ F; code; Confirm RP;
```

Appendix B

Simplification Rules

```
Confirm X;
```

```
Assume true; Confirm X;
```

```
Confirm X;
```

```
Confirm true -> X;
```

```
Confirm X;
```

```
Confirm X; Confirm true;
```

```
Confirm true;
```

```
Confirm A -> true;
```

```
Confirm (A ^ B) -> C;
```

```
Confirm A -> B -> C;
```

```
Confirm A -> B;
```

```
Confirm A -> (A ^ B);
```

```
Confirm A -> B;
```

```
Confirm A -> (B ^ A);
```

```
Confirm A;
```

```
Confirm A ^ true;
```

```
Confirm A;
```

```
Confirm true ^ A;
```

Appendix C

Additional Specifications

Array Specification

```
Concept Static_Array_Template(type Entry; evaluates
Lower_Bound, Upper_Bound: Integer);
    uses Std_Integer_Fac;
    requires (Lower_Bound <= Upper_Bound);

    Type Family Static_Array is modeled by (Z -> Entry);
        exemplar A;
     constraint true;
     initialization ensures
         for all i: Z, Entry.Is_Initial(A(i));

    Operation Swap_Entry(updates A: Static_Array;
                    updates E: Entry; evaluates i: Integer);
         requires Lower_Bound <= i  and i <= Upper_Bound;
         ensures E = #A(i) and
               A = lambda j: Z.({{#E if j = i;
                                 #A(j) otherwise;}});

    Operation Swap_Two_Entries(updates A: Static_Array;
                               evaluates i, j: Integer);
        requires Lower_Bound <= i  and i <= Upper_Bound and
                   Lower_Bound <= j  and j <= Upper_Bound;
        ensures A = lambda k: Z.({{#A(j) if k = i;
                                   #A(i) if k = j;
                                   #A(k) otherwise;}});

end Static_Array_Template;
```

```
Concept Integer_Template;
    uses Integer_Theory, Std_Boolean_Fac;

    defines min_int: Z;
    defines max_int: Z;

    Constraint min_int <= 0 and 0 < max_int;

    Type Family Integer is modeled by Z;
        exemplar i;
        constraint min_int <= i <= max_int;
        initialization ensures i = 0;

    Operation Is_Zero(evaluates i: Integer): Boolean;
        ensures      Is_Zero = ( i = 0 );

    Operation Is_Not_Zero(evaluates i: Integer): Boolean;
        ensures      Is_Not_Zero = ( i /= 0 );

    Operation Increment(updates i: Integer);
        requires i + 1 <= max_int;
        ensures      i = #i + 1;

    Operation Decrement(updates i: Integer);
        requires min_int <= i - 1;
        ensures      i = #i - 1;

    Operation Less_Or_Equal(evaluates i, j: Integer): Boolean;
        ensures      Less_Or_Equal = ( i <= j );

    Operation Less(evaluates i, j: Integer): Boolean;
        ensures Less = ( i < j );

    Operation Greater(evaluates i, j: Integer): Boolean;
        ensures Greater = ( i > j );

    Operation Greater_Or_Equal(
                evaluates i, j: Integer): Boolean;
        ensures Greater_Or_Equal = ( i >= j );

    Operation Sum(evaluates i, j: Integer): Integer;
        requires min_int <= i + j <= max_int;
        ensures      Sum = ( i + j );

    Operation Negate(evaluates i: Integer): Integer;
        requires min_int <= -i <= max_int;
        ensures Negate = ( -i );
```

```
    Operation Difference(evaluates i, j: Integer): Integer;
        requires min_int <= i - j <= max_int;
        ensures Difference = ( i - j );

    Operation Product(evaluates i, j: Integer): Integer;
        requires min_int <= i * j <= max_int;
        ensures Product = ( i * j );

    Operation Power(evaluates i, j: Integer): Integer;
        requires min_int <= i**j <= max_int;
        ensures Power = ( i**j );

    Operation Divide(evaluates i, j: Integer;
                     replaces q: Integer);
        requires if (j <= 0) then
               (j*(max_int + 1) < i and i < j*(min_int -1));
        ensures (|j*q| <= |i|) and (|i - j*q| < |j|);

    Operation Mod(evaluates i, j: Integer): Integer;
        requires j /= 0;
        ensures Mod = ( i mod j );

    Operation Rem(evaluates i, j: Integer): Integer;

    Operation Quotient(evaluates i, j: Integer): Integer;

    Operation Div(evaluates i, j: Integer): Integer;
        requires j /= 0;
        ensures Div = ( i/j );

    Operation Are_Equal(evaluates i, j: Integer): Boolean;
        ensures Are_Equal = (i = j);

    Operation Are_Not_Equal(evaluates i, j: Integer): Boolean;
        ensures Are_Not_Equal = (i /= j);

    Operation Replica(restores i: Integer): Integer;
        ensures Replica = (i);

    Operation Read(replaces i: Integer);

    Operation Write(evaluates i: Integer);

    Operation Write_Line(evaluates i: Integer);

    Operation Max_Int(): Integer;
        ensures Max_Int = max_int;

    Operation Min_Int(): Integer;
        ensures Min_Int = min_int;

    Operation Clear(clears i: Integer);

end Integer_Template;
```

Preemptable Queue Specification

```
Concept Preemptable_Queue_Template(type Entry;
                                   evaluates Max_Length: Integer);
    uses Std_Integer_Fac, Modified_String_Theory;
    requires Max_Length > 0;

    Type Family P_Queue is modeled by Str(Entry);
    exemplar Q;
    constraint |Q| <= Max_Length;
    initialization ensures Q = empty_string;

    Operation Enqueue(alters E: Entry; reassigns Q: P_Queue);
        requires |Q| < Max_Length;
        ensures Q = #Q o <#E>;

    Operation Inject(alters E: Entry; updates Q: P_Queue);
        requires |Q| < Max_Length;
        ensures Q = <#E> o #Q;

    Operation Dequeue(replaces R: Entry; updates Q: P_Queue);
        requires |Q| > 0;
        ensures #Q = <R> o Q;

    Operation Swap_Last_Entry(updates E: Entry;
                              updates Q: P_Queue);
        requires |Q| > 0;
        ensures there exists Pre: Str(Entry) such that
         #Q = Pre o <E> and Q = Pre o <#E>;

    Operation Length(restores Q: P_Queue): Integer;
        ensures Length = (|Q|);

    Operation Rem_Capacity(restores Q: P_Queue): Integer;
        ensures Rem_Capacity = (Max_Length - |Q|);

    Operation Clear(clears Q: P_Queue);

end Preemptable_Queue_Template;
```

Search Enhancement to Preemptable Queue

```
Enhancement Searching_Capability for
         Preemptable_Queue_Template;
   uses Std_Boolean_Fac;

   Operation Search_and_Move(restores E: Entry;
                            updates Q: P_Queue;
                            replaces Result: Boolean);
         ensures Is_Permutation(Q, #Q) and
         (Is_Substring(<E>, #Q) iff Result = true) and
         (Result = true implies Is_Prefix(<E>, Q));
end Searching_Capability;

Realization Searching_Realiz
(
     Operation Entries_Are_Equal
               (restores E, F: Entry) : Boolean;
          ensures Entries_Are_Equal = (E = F);
)
 for Searching_Capability of Preemptable_Queue_Template;
     uses Std_Boolean_Fac;

     Procedure Search_and_Move (restores E: Entry;
                                updates Q: P_Queue;
                                replaces Result: Boolean);
          Var T: Entry;
          Var R: P_Queue;
          Result := False();
          While(Length(Q) > 0)
               changing Q, Result, T, R;
               maintaining Is_Permutation(Q o R, #Q) and
                 (Is_Substring(<E>, R) iff Result = true) and
                 (Result = true implies Is_Prefix(<E>, R));
               decreasing |Q|;
          do
               Dequeue(T, Q);
               if (Entries_Are_Equal(E, T)) then
                    Inject(T, R);
                    Result := True();
               else Enqueue(T, R);
               end;
          end;
          Q :=: R;
     end Search_and_Move;

end Searching_Realiz;
```

Appendix D

Alternate Stack Specifications with implementations and VCs

Example of a Stack Specified as a Natural Number

```
Concept Stack_Template(type Entry; evaluates Max_Depth:
Integer);
    uses Std_Integer_Fac;
    requires Max_Depth > 0;

    Type Family Stack is modeled by N;
        exemplar S;
        constraint S <= Max_Depth;
        initialization ensures S = 0;

    Operation Push(alters E: Entry; updates S: Stack);
        requires S < Max_Depth;
        ensures  S = #S + 1;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires S /= 0;
        ensures #S = S + 1;

    Operation Depth(restores S: Stack): Integer;
        ensures Depth = (S);

    Operation Rem_Capacity(restores S: Stack): Integer;
        ensures Rem_Capacity = (Max_Depth - S);

    Operation Clear(clears S: Stack);

end Stack_Template;

Enhancement Reversal_Capability for Stack_Template;
   Operation Reverse(updates S: Stack);
         ensures S = #S;
end Reversal_Capability;


Realization Obvious_Rev_Realiz for Reversal_Capability of
Stack_Template;
   uses Std_Boolean_Fac, Std_Integer_Fac;

   Procedure Reverse(updates S: Stack);
         Var S_Reversed: Stack;
         Var Next_Entry: Entry;

         While (Is_Not_Zero(Depth(S)))
               changing S, S_Reversed, Next_Entry;
```

```
            maintaining #S = S_Reversed + S;
            decreasing S;
         do
            Pop(Next_Entry, S);
            Push(Next_Entry, S_Reversed);
         end;
         S :=: S_Reversed;
   end Reverse;
end Obvious_Rev_Realiz;
```

Stack Flip VCs generated with alternative Stack Spec:

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Obvious_Rev_Realiz.rb
// on:         Tue Mar 09 21:14:08 EST 2010
//

Free Variables: Max_Depth:Z, min_int:Z, max_int:Z, S:N,
?Next_Entry:Entry, ?S_Reversed:N, ?S:N, ??S:N,
Next_Entry:Entry, S_Reversed:N


VC: 0_1

(((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(S <= Max_Depth))
======================>
S = (0 + S)


VC: 0_2

((((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(S <= Max_Depth)) and
(S = (?S_Reversed + ??S) and
(??S /= 0 and
??S = (?S + 1))))
======================>
(?S_Reversed < Max_Depth)


VC: 0_3

((((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
```

```
(S <= Max_Depth)) and
(S = (?S_Reversed + ??S) and
(??S /= 0 and
??S = (?S + 1))))
======================>
(?S_Reversed + ??S) = ((?S_Reversed + 1) + ?S)


VC: 0_4

((((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(S <= Max_Depth)) and
(S = (?S_Reversed + ??S) and
(??S /= 0 and
??S = (?S + 1))))
======================>
(?S < ??S)


Free Variables: Max_Depth:Z, min_int:Z, max_int:Z, S:N,
?P_val:N, ?S:N, ?S_Reversed:N, Next_Entry:Entry, S_Reversed:N


VC: 1_1

(((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
((S <= Max_Depth) and
(S = (?S_Reversed + ?S) and
?S = 0)))
======================>
?S_Reversed = (?S_Reversed + ?S)
```

Example of a Stack Specified as a String of Entries

```
Concept Stack_Template(type Entry; evaluates Max_Depth:
Integer);
    uses Std_Integer_Fac, String_Theory;
    requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = empty_string;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures  S = <#E> o #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires |S| /= 0;
        ensures #S = <R> o S;

    Operation Depth(restores S: Stack): Integer;
        ensures Depth = (|S|);

    Operation Rem_Capacity(restores S: Stack): Integer;
        ensures Rem_Capacity = (Max_Depth - |S|);

    Operation Clear(clears S: Stack);

end Stack_Template;


Enhancement Flipping_Capability for Stack_Template;

   Operation Flip(updates S: Stack);
         ensures S = Reverse(#S);

end Flipping_Capability;

Realization Obvious_Flip_Realiz for Flipping_Capability of
Stack_Template;
   uses Std_Boolean_Fac;

   Procedure Flip(updates S: Stack);
         Var S_Flipped: Stack;
         Var Next_Entry: Entry;
         While (Depth(S) /= 0)
               changing S, S_Flipped, Next_Entry;
               maintaining #S = Reverse(S_Flipped) o S;
               decreasing |S|;
         do
               Pop(Next_Entry, S);
               Push(Next_Entry, S_Flipped);
```

```
         end;
         S_Flipped :=: S;
   end Flip;
end Obvious_Flip_Realiz;
```

Stack Flip VCs generated with normal Stack Spec:

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Obvious_Rev_Realiz.rb
// on:         Mon Mar 08 20:29:54 EST 2010
//

Free Variables: Max_Depth:Z, min_int:Z, max_int:Z,
S:String_Theory.Str(Entry), ?Next_Entry:Entry,
?S_Reversed:String_Theory.Str(Entry),
?S:String_Theory.Str(Entry), ??S:String_Theory.Str(Entry),
Next_Entry:Entry, S_Reversed:String_Theory.Str(Entry)



VC: 0_1

(((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(|S| <= Max_Depth))
======================>
S = (Rev(empty_string) o S)


VC: 0_2

((((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(|S| <= Max_Depth)) and
(S = (Rev(?S_Reversed) o ??S) and
(|??S| /= 0 and
??S = (<?Next_Entry> o ?S))))
======================>
(|?S_Reversed| < Max_Depth)


VC: 0_3

((((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(|S| <= Max_Depth)) and
(S = (Rev(?S_Reversed) o ??S) and
```

```
(|??S| /= 0 and
??S = (<?Next_Entry> o ?S))))
======================>
(Rev(?S_Reversed) o ??S) = (Rev((<?Next_Entry> o ?S_Reversed))
o ?S)


VC: 0_4

((((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
(|S| <= Max_Depth)) and
(S = (Rev(?S_Reversed) o ??S) and
(|??S| /= 0 and
??S = (<?Next_Entry> o ?S))))
======================>
(|?S| < |??S|)


Free Variables: Max_Depth:Z, min_int:Z, max_int:Z,
S:String_Theory.Str(Entry), ?P_val:N,
?S:String_Theory.Str(Entry),
?S_Reversed:String_Theory.Str(Entry), Next_Entry:Entry,
S_Reversed:String_Theory.Str(Entry)



VC: 1_1

(((min_int <= 0)and
(0 < max_int) and
(Max_Depth > 0)) and
((|S| <= Max_Depth) and
(S = (Rev(?S_Reversed) o ?S) and
|?S| = 0)))
======================>
?S_Reversed = Rev((Rev(?S_Reversed) o ?S))
```

Appendix E

Examples of VCs from Component-Level Verification

VCs for Circular_Array_Realiz of Queue_Template

```
//
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Circular_Array_Realiz.rb
// on:         Mon Dec 05 12:22:25 EST 2011
//

Free Variables:
Entry, Lower_Bound:Z, Upper_Bound:Z



VC: 0_1:
Requirement for Facility Declaration Rule for
_Contents_Array_Fac_1: Circular_Array_Realiz.rb(5)

Goal:
(Lower_Bound  <= (Upper_Bound  + 1))

Given:

1: (Lower_Bound  <= (Upper_Bound  + 1))



Free Variables:
Q:(Contents:_Contents_Array_Fac_1.Static_Array; Front:Integer;
Length:Integer), Lower_Bound:Z, Upper_Bound:Z,
Last_Char_Num:N, min_int:Z, max_int:Z, Max_Char_Str_Len:N



VC: 1_1:
Correspondence Rule for Queue: Circular_Array_Realiz.rb(12)

Goal:
(|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
<= Max_Length)

Given:
1: (Max_Length  > 0)
2: (min_int  <= 0)
```

```
3: (0  < max_int)
4: (Last_Char_Num  > 0)
5: (0  <= (Max_Length - 1  + 1))
6: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
7: (min_int <= 0) and (0 <= max_int)
8: (0 <= Q.Front) and (Q.Front < Max_Length)
9: (0 <= Q.Length) and (Q.Length <= Max_Length)



Free Variables:
Q:(Contents:_Contents_Array_Fac_1.Static_Array; Front:Integer;
Length:Integer), Conc.Q:Str(Entry), Q.Length:Z, Q.Front:Z,
Q.Contents:Z -> Entry



VC: 2_1:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal:
(0 <= 0) and (0 < Max_Length)

Given:
1: for all i:Z, Entry.Is_Initial(Q.Contents(i))
2: (Max_Length  > 0)


VC: 2_2:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal:
(0 <= 0) and (0 <= Max_Length)

Given:
1: for all i:Z, Entry.Is_Initial(Q.Contents(i))
2: (Max_Length  > 0)


VC: 2_3:
Initialization Rule for Queue: Circular_Array_Realiz.rb(4)

Goal:
Concatenation i:Integer where (0 <= i) and (i <= ((0 + 0) -
1)), <Q.Contents((i mod Max_Length))> = empty_string

Given:
1: for all i:Z, Entry.Is_Initial(Q.Contents(i))
2: (Max_Length  > 0)
3: Conc.Q = Concatenation i:Integer where (0 <= i) and (i <=
((0 + 0) - 1)), <Q.Contents((i mod Max_Length))>
```

```
Free Variables:
Max_Length:Z, Lower_Bound:Z, Upper_Bound:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.Q:Str(Entry),
E:Entry, Q:(Contents:_Contents_Array_Fac_1.Static_Array;
Front:Integer; Length:Integer),
Q':(Contents:_Contents_Array_Fac_1.Static_Array;
Front:Integer; Length:Integer), E':Entry



VC: 3_1:
Requires Clause of Swap_Entry in Procedure Enqueue:
Circular_Array_Realiz.rb(17)

Goal:
(0  <= (Q.Front  + Q.Length) mod Max_Length))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
< Max_Length)


VC: 3_2:
Requires Clause of Swap_Entry in Procedure Enqueue:
Circular_Array_Realiz.rb(17)

Goal:
(Q.Front  + Q.Length) mod Max_Length) <= Max_Length - 1)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
```

```
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
< Max_Length)


VC: 3_3:
Requires Clause of Q.Length + 1 in Procedure Enqueue:
Circular_Array_Realiz.rb(18)

Goal:
(min_int <= (Q.Length + 1)) and ((Q.Length + 1) <= max_int)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
< Max_Length)
13: E' = Q.Contents(((Q.Front  + Q.Length)  mod Max_Length))
14: Q'.Contents = lambda j: Z ({{E if j = (Q.Front  +
Q.Length) mod Max_Length)
Q.Contents(j) otherwise
}})


VC: 3_4:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal:
(0 <= Q.Front) and (Q.Front < Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
```

```
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
< Max_Length)
13: E' = Q.Contents(((Q.Front  + Q.Length)  mod Max_Length))
14: Q'.Contents = lambda j: Z ({{E if j = (Q.Front  +
Q.Length) mod Max_Length)
Q.Contents(j) otherwise
}})
```

```
VC: 3_5:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal:
(0 <= (Q.Length + 1)) and ((Q.Length + 1) <= Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
< Max_Length)
13: E' = Q.Contents(((Q.Front  + Q.Length)  mod Max_Length))
14: Q'.Contents = lambda j: Z ({{E if j = (Q.Front  +
Q.Length) mod Max_Length)
Q.Contents(j) otherwise
}})


VC: 3_6:
Ensures Clause of Enqueue: Circular_Array_Realiz.rb(19)
```

```
Goal:
Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + (Q.Length + 1)) - 1)), <Q'.Contents((i mod
Max_Length))> = (Concatenation i:Integer where (Q.Front <= i)
and (i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>  o <E>)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (|Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
< Max_Length)
13: E' = Q.Contents(((Q.Front  + Q.Length)  mod Max_Length))
14: Q'.Contents = lambda j: Z ({{E if j = (Q.Front  +
Q.Length) mod Max_Length)
Q.Contents(j) otherwise
}})



Free Variables:
Max_Length:Z, Lower_Bound:Z, Upper_Bound:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.Q:Str(Entry),
R:Entry, Q:(Contents:_Contents_Array_Fac_1.Static_Array;
Front:Integer; Length:Integer),
Q':(Contents:_Contents_Array_Fac_1.Static_Array;
Front:Integer; Length:Integer), R':Entry



VC: 4_1:
Requires Clause of Swap_Entry in Procedure Dequeue:
Circular_Array_Realiz.rb(22)

Goal:
(0  <= Q.Front)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
```

```
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0


VC: 4_2:
Requires Clause of Swap_Entry in Procedure Dequeue:
Circular_Array_Realiz.rb(22)

Goal:
(Q.Front  <= Max_Length - 1)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0


VC: 4_3:
Requires Clause of Q.Front + 1 % Max_Length in Procedure
Dequeue: Circular_Array_Realiz.rb(23)

Goal:
Max_Length /= 0

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
```

```
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})
```

VC: 4_4:
Requires Clause of Q.Length - 1 in Procedure Dequeue:
Circular_Array_Realiz.rb(24)

Goal:
(min_int <= (Q.Length - 1)) and ((Q.Length - 1) <= max_int)

Given:

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})
```

VC: 4_5:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal:

```
(0 <= ((Q.Front + 1) mod Max_Length)) and (((Q.Front + 1) mod
Max_Length) < Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})


VC: 4_6:
Convention for Queue generated by intialization rule:
Circular_Array_Realiz.rb(9)

Goal:
(0 <= (Q.Length - 1)) and ((Q.Length - 1) <= Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})
```

```
VC: 4_7:
Ensures Clause of Dequeue: Circular_Array_Realiz.rb(25)

Goal:
Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>
= (<Q.Contents(Q.Front)>  o Concatenation i:Integer where
(((Q.Front + 1) mod Max_Length) <= i) and (i <= ((((Q.Front +
1) mod Max_Length) + (Q.Length - 1)) - 1)), <Q'.Contents((i
mod Max_Length))>)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: Entry.is_initial(R)
13: |Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>|
/= 0
14: Q'.Contents = lambda j: Z ({{R if j = Q.Front
Q.Contents(j) otherwise
}})



Free Variables:
Max_Length:Z, Lower_Bound:Z, Upper_Bound:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.Q:Str(Entry),
Q:(Contents:_Contents_Array_Fac_1.Static_Array; Front:Integer;
Length:Integer), Length:Z



VC: 5_1:
Convention for Queue generated by intialization rule modified
by Variable Declaration rule: Circular_Array_Realiz.rb(9)

Goal:
(0 <= Q.Front) and (Q.Front < Max_Length)

Given:
```

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (min_int <= Length) and (Length <= max_int)
```

```
VC: 5_2:
Convention for Queue generated by intialization rule modified
by Variable Declaration rule: Circular_Array_Realiz.rb(9)

Goal:
(0 <= Q.Length) and (Q.Length <= Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (min_int <= Length) and (Length <= max_int)
```

```
VC: 5_3:
Ensures Clause of Length modified by Variable Declaration
rule: Circular_Array_Realiz.rb(30)

Goal:
Q.Length = |Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>|

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
```

```
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (min_int <= Length) and (Length <= max_int)


VC: 5_4:
Ensures Clause of Length modified by Variable Declaration
rule: Circular_Array_Realiz.rb(30)

Goal:
Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>
= Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
12: (min_int <= Length) and (Length <= max_int)




Free Variables:
Max_Length:Z, Lower_Bound:Z, Upper_Bound:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.Q:Str(Entry),
Q:(Contents:_Contents_Array_Fac_1.Static_Array; Front:Integer;
Length:Integer), Rem_Capacity:Z



VC: 6_1:
Requires Clause of Max_Length - Q.Length in Procedure
Rem_Capacity: Circular_Array_Realiz.rb(33)

Goal:
```

```
(min_int <= (Max_Length - Q.Length)) and ((Max_Length -
Q.Length) <= max_int)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>


VC: 6_2:
Convention for Queue generated by intialization rule modified
by Variable Declaration rule: Circular_Array_Realiz.rb(9)

Goal:
(0 <= Q.Front) and (Q.Front < Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>


VC: 6_3:
Convention for Queue generated by intialization rule modified
by Variable Declaration rule: Circular_Array_Realiz.rb(9)

Goal:
(0 <= Q.Length) and (Q.Length <= Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
```

```
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
```

```
VC: 6_4:
Ensures Clause of Rem_Capacity: Circular_Array_Realiz.rb(34)

Goal:
(Max_Length  - Q.Length) = (Max_Length  - |Concatenation
i:Integer where (Q.Front <= i) and (i <= ((Q.Front + Q.Length)
- 1)), <Q.Contents((i mod Max_Length))>|)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>
```

```
VC: 6_5:
Ensures Clause of Rem_Capacity: Circular_Array_Realiz.rb(34)

Goal:
Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>
= Concatenation i:Integer where (Q.Front <= i) and (i <=
((Q.Front + Q.Length) - 1)), <Q.Contents((i mod Max_Length))>

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
```

```
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
11: Conc.Q = Concatenation i:Integer where (Q.Front <= i) and
(i <= ((Q.Front + Q.Length) - 1)), <Q.Contents((i mod
Max_Length))>



Free Variables:
Max_Length:Z, Lower_Bound:Z, Upper_Bound:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.Q:Str(Entry),
Q:(Contents:_Contents_Array_Fac_1.Static_Array; Front:Integer;
Length:Integer)



VC: 7_1:
Convention for Queue generated by intialization rule modified
by Variable Declaration rule: Circular_Array_Realiz.rb(9)

Goal:
(0 <= 0) and (0 < Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)


VC: 7_2:
Convention for Queue generated by intialization rule modified
by Variable Declaration rule: Circular_Array_Realiz.rb(9)

Goal:
(0 <= 0) and (0 <= Max_Length)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
```

```
VC: 7_3:
Ensures Clause of Clear: Circular_Array_Realiz.rb(38)

Goal:
true

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)


VC: 7_4:
Ensures Clause of Clear: Circular_Array_Realiz.rb(38)

Goal:
Concatenation i:Integer where (0 <= i) and (i <= ((0 + 0) -
1)), <Q.Contents((i mod Max_Length))> = empty_string

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (0  <= (Max_Length - 1  + 1))
5: (min_int <= Max_Length - 1) and (Max_Length - 1 <= max_int)
6: (min_int <= 0) and (0 <= max_int)
7: (Max_Length  > 0)
8: (min_int <= Max_Length) and (Max_Length <= max_int)
9: (0 <= Q.Front) and (Q.Front < Max_Length)
10: (0 <= Q.Length) and (Q.Length <= Max_Length)
```

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Selection_Sort_Realization.rb
// on:         Sat Oct 15 19:28:23 EDT 2011
//

Free Variables:
Max_Length:Z, min_int:Z, max_int:Z, Last_Char_Num:N,
Max_Char_Str_Len:N, Q:Modified_String_Theory.Str(Entry),
Min:Entry, Considered_Entry:Entry,
New_Queue:Modified_String_Theory.Str(Entry),
Considered_Entry':Entry,
New_Queue':Modified_String_Theory.Str(Entry),
Q':Modified_String_Theory.Str(Entry),
Q'':Modified_String_Theory.Str(Entry), Min':Entry,
Min'':Entry, Q''':Modified_String_Theory.Str(Entry)



VC: 0_1:
Requires Clause of Dequeue in Procedure Remove_Min modified by
Variable Declaration rule: Selection_Sort_Realization.rb(18)

Goal:
|Q| /= 0

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0


VC: 0_2:
Base Case of the Invariant of While Statement in Procedure
Remove_Min modified by Variable Declaration rule:
Selection_Sort_Realization.rb(22)

Goal:
Is_Permutation(((empty_string o Q''') o <Min''>), Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
```

```
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
```

```
VC: 0_3:
Base Case of the Invariant of While Statement in Procedure
Remove_Min modified by Variable Declaration rule:
Selection_Sort_Realization.rb(22)

Goal:
Is_Universally_Related(LEQV, <Min''>, empty_string)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
```

```
VC: 0_4:
Requires Clause of Dequeue in Procedure Remove_Min modified by
Variable Declaration rule: Selection_Sort_Realization.rb(26)

Goal:
|Q''| /= 0

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
```

```
VC: 0_5:
```

```
Requires Clause of Enqueue in Procedure Remove_Min , If "if"
condition at Selection_Sort_Realization.rb(28) is true
modified by Variable Declaration rule:
Selection_Sort_Realization.rb(32)

Goal:
(|New_Queue'| < Max_Length)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: LEQV(Considered_Entry', Min')


VC: 0_6:
Inductive Case of Invariant of While Statement in Procedure
Remove_Min , If "if" condition at
Selection_Sort_Realization.rb(28) is true modified by Variable
Declaration rule: Selection_Sort_Realization.rb(22)

Goal:
Is_Permutation((((New_Queue' o <Min'>) o Q') o
<Considered_Entry'>), Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: LEQV(Considered_Entry', Min')
```

```
VC: 0_7:
Inductive Case of Invariant of While Statement in Procedure
Remove_Min , If "if" condition at
Selection_Sort_Realization.rb(28) is true modified by Variable
Declaration rule: Selection_Sort_Realization.rb(22)

Goal:
Is_Universally_Related(LEQV, <Considered_Entry'>, (New_Queue'
o <Min'>))

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: LEQV(Considered_Entry', Min')


VC: 0_8:
Termination of While Statement in Procedure Remove_Min , If
"if" condition at Selection_Sort_Realization.rb(28) is true
modified by Variable Declaration rule:
Selection_Sort_Realization.rb(24)

Goal:
(|Q'| < |Q''|)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: LEQV(Considered_Entry', Min')
```

```
Free Variables:
Max_Length:Z, min_int:Z, max_int:Z, Last_Char_Num:N,
Max_Char_Str_Len:N, Q:Modified_String_Theory.Str(Entry),
Min:Entry, Considered_Entry:Entry,
New_Queue:Modified_String_Theory.Str(Entry),
Considered_Entry':Entry,
New_Queue':Modified_String_Theory.Str(Entry),
Q':Modified_String_Theory.Str(Entry),
Q'':Modified_String_Theory.Str(Entry), Min':Entry,
Min'':Entry, Q''':Modified_String_Theory.Str(Entry)



VC: 1_1:
Requires Clause of Dequeue in Procedure Remove_Min modified by
Variable Declaration rule: Selection_Sort_Realization.rb(18)

Goal:
|Q| /= 0

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0


VC: 1_2:
Base Case of the Invariant of While Statement in Procedure
Remove_Min modified by Variable Declaration rule modified by
Variable Declaration rule: Selection_Sort_Realization.rb(22)

Goal:
Is_Permutation(((empty_string o Q''') o <Min''>), Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
```

```
VC: 1_3:
Base Case of the Invariant of While Statement in Procedure
Remove_Min modified by Variable Declaration rule modified by
Variable Declaration rule: Selection_Sort_Realization.rb(22)

Goal:
Is_Universally_Related(LEQV, <Min''>, empty_string)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')


VC: 1_4:
Requires Clause of Dequeue in Procedure Remove_Min modified by
Variable Declaration rule: Selection_Sort_Realization.rb(26)

Goal:
|Q''| /= 0

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)


VC: 1_5:
Requires Clause of Enqueue in Procedure Remove_Min , If "if"
condition at Selection_Sort_Realization.rb(28) is false
modified by Variable Declaration rule:
Selection_Sort_Realization.rb(32)

Goal:
(|New_Queue'| < Max_Length)
```

```
Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: not(LEQV(Considered_Entry', Min'))


VC: 1_6:
Inductive Case of Invariant of While Statement in Procedure
Remove_Min , If "if" condition at
Selection_Sort_Realization.rb(28) is true modified by Variable
Declaration rule , If "if" condition at
Selection_Sort_Realization.rb(28) is false modified by
Variable Declaration rule: Selection_Sort_Realization.rb(22)

Goal:
Is_Permutation((((New_Queue' o <Considered_Entry'>) o Q') o
<Min'>), Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: not(LEQV(Considered_Entry', Min'))


VC: 1_7:
Inductive Case of Invariant of While Statement in Procedure
Remove_Min , If "if" condition at
Selection_Sort_Realization.rb(28) is true modified by Variable
Declaration rule , If "if" condition at
```

```
Selection_Sort_Realization.rb(28) is false modified by
Variable Declaration rule: Selection_Sort_Realization.rb(22)

Goal:
Is_Universally_Related(LEQV, <Min'>, (New_Queue' o
<Considered_Entry'>))

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: not(LEQV(Considered_Entry', Min'))


VC: 1_8:
Termination of While Statement in Procedure Remove_Min , If
"if" condition at Selection_Sort_Realization.rb(28) is false
modified by Variable Declaration rule:
Selection_Sort_Realization.rb(24)

Goal:
(|Q'| < |Q''|)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q''')
11: Is_Permutation(((New_Queue' o Q'') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: (|Q''| > 0)
14: Q'' = (<Considered_Entry'> o Q')
15: not(LEQV(Considered_Entry', Min'))
```

```
Free Variables:
Max_Length:Z, min_int:Z, max_int:Z, Last_Char_Num:N,
Max_Char_Str_Len:N, Q:Modified_String_Theory.Str(Entry),
Min:Entry, Considered_Entry:Entry,
New_Queue:Modified_String_Theory.Str(Entry), P_val':N,
Q':Modified_String_Theory.Str(Entry), Min':Entry,
New_Queue':Modified_String_Theory.Str(Entry), Min'':Entry,
Q'':Modified_String_Theory.Str(Entry)



VC: 2_1:
Requires Clause of Dequeue in Procedure Remove_Min:
Selection_Sort_Realization.rb(18)

Goal:
|Q| /= 0

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0


VC: 2_2:
Ensures Clause of Remove_Min:
Selection_Sort_Realization.rb(12)

Goal:
Is_Permutation((New_Queue' o <Min'>), Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q'')
11: Is_Permutation(((New_Queue' o Q') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: not((|Q'| > 0))


VC: 2_3:
```

```
Ensures Clause of Remove_Min:
Selection_Sort_Realization.rb(12)

Goal:
Is_Universally_Related(LEQV, <Min'>, New_Queue')

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q'')
11: Is_Permutation(((New_Queue' o Q') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: not((|Q'| > 0))


VC: 2_4:
Ensures Clause of Remove_Min:
Selection_Sort_Realization.rb(12)

Goal:
|New_Queue'| = (|Q| - 1)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: Is_Total_Preordering(LEQV)
7: Entry.is_initial(Min)
8: (|Q| <= Max_Length)
9: |Q| /= 0
10: Q = (<Min''> o Q'')
11: Is_Permutation(((New_Queue' o Q') o <Min'>), Q)
12: Is_Universally_Related(LEQV, <Min'>, New_Queue')
13: not((|Q'| > 0))



Free Variables:
Max_Length:Z, min_int:Z, max_int:Z, Last_Char_Num:N,
Max_Char_Str_Len:N, Q:Modified_String_Theory.Str(Entry),
Sorted_Queue:Modified_String_Theory.Str(Entry),
Lowest_Remaining:Entry, Lowest_Remaining':Entry,
Sorted_Queue':Modified_String_Theory.Str(Entry),
```

```
Q':Modified_String_Theory.Str(Entry),
Q'':Modified_String_Theory.Str(Entry)



VC: 3_1:
Base Case of the Invariant of While Statement in Procedure
Sort modified by Variable Declaration rule:
Selection_Sort_Realization.rb(45)

Goal:
Is_Permutation((Q o empty_string), Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)


VC: 3_2:
Base Case of the Invariant of While Statement in Procedure
Sort modified by Variable Declaration rule:
Selection_Sort_Realization.rb(45)

Goal:
Is_Conformal_With(LEQV, empty_string)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)


VC: 3_3:
Base Case of the Invariant of While Statement in Procedure
Sort modified by Variable Declaration rule:
Selection_Sort_Realization.rb(45)

Goal:
Is_Universally_Related(LEQV, empty_string, Q)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
```

```
VC: 3_4:
Requires Clause of Remove_Min in Procedure Sort modified by
Variable Declaration rule: Selection_Sort_Realization.rb(49)

Goal:
|Q''| /= 0

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q'' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q'')
10: (|Q''| > 0)


VC: 3_5:
Requires Clause of Enqueue in Procedure Sort modified by
Variable Declaration rule: Selection_Sort_Realization.rb(50)

Goal:
(|Sorted_Queue'| < Max_Length)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q'' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q'')
10: (|Q''| > 0)
11: Is_Permutation((Q' o <Lowest_Remaining'>), Q'')
12: Is_Universally_Related(LEQV, <Lowest_Remaining'>, Q')
13: |Q'| = (|Q''| - 1)


VC: 3_6:
Inductive Case of Invariant of While Statement in Procedure
Sort modified by Variable Declaration rule:
Selection_Sort_Realization.rb(45)

Goal:
Is_Permutation((Q' o (Sorted_Queue' o <Lowest_Remaining'>)),
Q)
```

```
Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q'' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q'')
10: (|Q''| > 0)
11: Is_Permutation((Q' o <Lowest_Remaining'>), Q'')
12: Is_Universally_Related(LEQV, <Lowest_Remaining'>, Q')
13: |Q'| = (|Q''| - 1)


VC: 3_7:
Inductive Case of Invariant of While Statement in Procedure
Sort modified by Variable Declaration rule:
Selection_Sort_Realization.rb(45)

Goal:
Is_Conformal_With(LEQV, (Sorted_Queue' o <Lowest_Remaining'>))

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q'' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q'')
10: (|Q''| > 0)
11: Is_Permutation((Q' o <Lowest_Remaining'>), Q'')
12: Is_Universally_Related(LEQV, <Lowest_Remaining'>, Q')
13: |Q'| = (|Q''| - 1)


VC: 3_8:
Inductive Case of Invariant of While Statement in Procedure
Sort modified by Variable Declaration rule:
Selection_Sort_Realization.rb(45)

Goal:
Is_Universally_Related(LEQV, (Sorted_Queue' o
<Lowest_Remaining'>), Q')

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
```

```
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q'' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q'')
10: (|Q''| > 0)
11: Is_Permutation((Q' o <Lowest_Remaining'>), Q'')
12: Is_Universally_Related(LEQV, <Lowest_Remaining'>, Q')
13: |Q'| = (|Q''| - 1)


VC: 3_9:
Termination of While Statement in Procedure Sort modified by
Variable Declaration rule: Selection_Sort_Realization.rb(47)

Goal:
(|Q'| < |Q''|)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q'' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q'')
10: (|Q''| > 0)
11: Is_Permutation((Q' o <Lowest_Remaining'>), Q'')
12: Is_Universally_Related(LEQV, <Lowest_Remaining'>, Q')
13: |Q'| = (|Q''| - 1)



Free Variables:
Max_Length:Z, min_int:Z, max_int:Z, Last_Char_Num:N,
Max_Char_Str_Len:N, Q:Modified_String_Theory.Str(Entry),
Sorted_Queue:Modified_String_Theory.Str(Entry),
Lowest_Remaining:Entry, P_val':N,
Q':Modified_String_Theory.Str(Entry),
Sorted_Queue':Modified_String_Theory.Str(Entry)



VC: 4_1:
Ensures Clause of Sort: Selection_Sort_Realization.rb(54)

Goal:
Is_Conformal_With(LEQV, Sorted_Queue')

Given:
1: (Last_Char_Num > 0)
```

```
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q')
10: not((|Q'| > 0))
```

```
VC: 4_2:
Ensures Clause of Sort: Selection_Sort_Realization.rb(54)

Goal:
Is_Permutation(Q, Sorted_Queue')

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (Max_Length > 0)
5: (min_int <= Max_Length) and (Max_Length <= max_int)
6: (|Q| <= Max_Length)
7: Is_Permutation((Q' o Sorted_Queue'), Q)
8: Is_Conformal_With(LEQV, Sorted_Queue')
9: Is_Universally_Related(LEQV, Sorted_Queue', Q')
10: not((|Q'| > 0))
```

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Sort_Job_Queue.fa
// on:         Fri Nov 11 11:49:10 EST 2011
//

Free Variables:
min_int:Z, max_int:Z, Last_Char_Num:N, Max_Char_Str_Len:N,
S1:(Name:Char_Str; Priority:Integer), S2:(Name:Char_Str;
Priority:Integer)



VC: 0_1:
Ensures Clause of Priority_Order: Sort_Job_Queue.fa(15)

Goal:
(S1.Priority  <= S2.Priority) = Priority_LEQV(S1, S2)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)


VC: 0_2:
Ensures Clause of Priority_Order: Sort_Job_Queue.fa(15)

Goal:
S1 = S1

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)


VC: 0_3:
Ensures Clause of Priority_Order: Sort_Job_Queue.fa(15)

Goal:
S2 = S2

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
```

```
Free Variables:



VC: 1_1:
Requires from Compare: Selection_Sort_Realization.rb(3)

Goal:
true

Given:




Free Variables:



VC: 2_1:
Ensures from QF: Sort_Job_Queue.fa(13)

Goal:
Priority_LEQV(S1, S2) = Priority_LEQV(S1, S2)

Given:
1: Priority_Order = Priority_LEQV(S1, S2)
2: #S1 = S1
3: #S2 = S2


VC: 2_2:
Ensures from QF: Sort_Job_Queue.fa(13)

Goal:
S1 = S1

Given:
1: Priority_Order = Priority_LEQV(S1, S2)
2: #S1 = S1
3: #S2 = S2


VC: 2_3:
Ensures from QF: Sort_Job_Queue.fa(13)

Goal:
S2 = S2

Given:
```

```
1: Priority_Order = Priority_LEQV(S1, S2)
2: #S1 = S1
3: #S2 = S2



Free Variables:
Entry, Max_Length:Z



VC: 3_1:
Requirement for Facility Declaration Rule for QF:
Sorting_Capability.en(5)

Goal:
Is_Total_Preordering(Priority_LEQV)

Given:



VC: 3_2:
Facility Declaration Rule: Queue_Template.co(42)

Goal:
(10  > 0)

Given:

1: true



Free Variables:
min_int:Z, max_int:Z, Last_Char_Num:N, Max_Char_Str_Len:N,
S1:(Name:Char_Str; Priority:Integer), S2:(Name:Char_Str;
Priority:Integer), S3:(Name:Char_Str; Priority:Integer),
Temp:(Name:Char_Str; Priority:Integer),
Q:Modified_String_Theory.Str(Entry), Temp':(Name:Char_Str;
Priority:Integer), Q':Modified_String_Theory.Str(Entry),
Temp'':(Name:Char_Str; Priority:Integer),
Q'':Modified_String_Theory.Str(Entry), Temp''':(Name:Char_Str;
Priority:Integer), Q''':Modified_String_Theory.Str(Entry),
Q'''':Modified_String_Theory.Str(Entry), S3':(Name:Char_Str;
Priority:Integer), Q''''':Modified_String_Theory.Str(Entry),
S2':(Name:Char_Str; Priority:Integer), S1':(Name:Char_Str;
Priority:Integer)



VC: 4_1:
```

```
Requires Clause of Enqueue in Procedure Main modified by
Variable Declaration rule: Sort_Job_Queue.fa(34)

Goal:
(|empty_string|  < 10)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)


VC: 4_2:
Requires Clause of Enqueue in Procedure Main modified by
Variable Declaration rule: Sort_Job_Queue.fa(35)

Goal:
(|(empty_string  o <S1>)|  < 10)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)


VC: 4_3:
Requires Clause of Enqueue in Procedure Main modified by
Variable Declaration rule: Sort_Job_Queue.fa(36)

Goal:
(|(empty_string  o <S1>) o <S2>)|  < 10)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)


VC: 4_4:
Requires Clause of Dequeue in Procedure Main modified by
Variable Declaration rule: Sort_Job_Queue.fa(38)

Goal:
|Q''''| /= 0
```

```
Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)
7: Is_Conformal_With(LEQV, Q'''')
8: Is_Permutation((((empty_string  o <S1>)  o <S2>)  o <S3>),
Q'''')


VC: 4_5:
Requires Clause of Dequeue in Procedure Main modified by
Variable Declaration rule: Sort_Job_Queue.fa(40)

Goal:
|Q'''| /= 0

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)
7: Is_Conformal_With(LEQV, Q'''')
8: Is_Permutation((((empty_string  o <S1>)  o <S2>)  o <S3>),
Q'''')
9: Q'''' = (<Temp'''>  o Q''')


VC: 4_6:
Requires Clause of Dequeue in Procedure Main modified by
Variable Declaration rule: Sort_Job_Queue.fa(42)

Goal:
|Q''| /= 0

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)
7: Is_Conformal_With(LEQV, Q'''')
8: Is_Permutation((((empty_string  o <S1>)  o <S2>)  o <S3>),
Q'''')
9: Q'''' = (<Temp'''>  o Q''')
10: Q''' = (<Temp''>  o Q'')


VC: 4_7:
```

Ensures Clause of Main modified by Variable Declaration rule:
Sort_Job_Queue.fa(25)

Goal:
true

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: Job_Info.is_initial(S1)
5: Job_Info.is_initial(S2)
6: Job_Info.is_initial(S3)
7: Is_Conformal_With(LEQV, Q'''')
8: Is_Permutation((((empty_string o <S1>) o <S2>) o <S3>),
Q'''')
9: Q'''' = (<Temp'''> o Q''')
10: Q''' = (<Temp''> o Q'')
11: Q'' = (<Temp'> o Q')

Appendix F

Benchmark VCs

Benchmark #1: Adding and Multiplying Numbers

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Add_And_Multiply_Realiz.rb
// on:         Fri Oct 21 11:52:13 EDT 2011
//

Free Variables:
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
i:Z, j:Z, Add:Z, zero:Z, P_val:N, j':Z, j'':Z, Add':Z



VC: 0_1:
Requires Clause of Increment in Procedure Add modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(9)

Goal:
(i  + 1) <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: (j  > 0)


VC: 0_2:
Requires Clause of Decrement in Procedure Add modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(10)

Goal:
(min_int  <= (j  - 1))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
```

```
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: (j  > 0)
```

VC: 0_3:
Requires Clause of Add in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(11)

Goal:

```
(min_int  <= (i  + 1) + (j  - 1)))
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: (j  > 0)
```

VC: 0_4:
Requires Clause of Add in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(11)

Goal:

```
(i  + 1) + (j  - 1)) <= max_int)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: (j  > 0)
```

VC: 0_5:
Requires Clause of Increment in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(12)

Goal:

```
(j  - 1) + 1) <= max_int)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
```

```
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: (j  > 0)


VC: 0_6:
Ensures Clause of Add , If "if" condition at
Add_And_Multiply_Realiz.rb(8) is true modified by Variable
Declaration rule: Add_And_Multiply_Realiz.rb(19)

Goal:
(i  + 1) + (j  - 1)) = (i  + j)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: (j  > 0)
10: j' = (j  - 1) + 1)



Free Variables:
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
i:Z, j:Z, Add:Z, zero:Z, P_val:N, j':Z, j'':Z, Add':Z



VC: 1_1:
Requires Clause of Decrement in Procedure Add modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(15)

Goal:
(min_int  <= (i  - 1))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
```

```
9: not((j  > 0))
10: (0  > j)
```

VC: 1_2:
Requires Clause of Increment in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(16)

Goal:
```
(j  + 1) <= max_int)
```

Given:
```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: not((j  > 0))
10: (0  > j)
11: Add' = (i  - 1)
```

VC: 1_3:
Requires Clause of Add in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(17)

Goal:
```
(min_int  <= (i  + (j  + 1)))
```

Given:
```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: not((j  > 0))
10: (0  > j)
11: Add' = (i  - 1)
```

VC: 1_4:
Requires Clause of Add in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(17)

Goal:
```
(i  + (j  + 1)) <= max_int)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: not((j  > 0))
10: (0  > j)
11: Add' = (i  - 1)
```

VC: 1_5:
Requires Clause of Decrement in Procedure Add modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(18)

Goal:

```
(min_int  <= (j  + 1) - 1))
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: not((j  > 0))
10: (0  > j)
11: Add' = (i  - 1)
```

VC: 1_6:
Ensures Clause of Add , If "if" condition at Add_And_Multiply_Realiz.rb(8) is false , If "if" condition at Add_And_Multiply_Realiz.rb(14) is true modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(19)

Goal:

```
(i  + (j  + 1)) = (i  + j)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: not((j  > 0))
10: (0  > j)
```

```
11: Add' = (i  - 1)
12: j' = (j  + 1) - 1)



Free Variables:
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
i:Z, j:Z, Add:Z, zero:Z, P_val:N



VC: 2_1:
Ensures Clause of Add , If "if" condition at
Add_And_Multiply_Realiz.rb(8) is false , If "if" condition at
Add_And_Multiply_Realiz.rb(14) is false modified by Variable
Declaration rule: Add_And_Multiply_Realiz.rb(19)

Goal:
i = (i  + j)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= j) and (j <= max_int)
5: (min_int <= i) and (i <= max_int)
6: (min_int  <= (i  + j))
7: (i  + j) <= max_int)
8: P_val = |j|
9: not((j  > 0))
10: not((0  > j))



Free Variables:
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
I:Z, J:Z, Multiply:Z, nj:Z, zero:Z, J':Z, nj':Z, Multiply':Z



VC: 3_1:
Base Case of the Invariant of While Statement in Procedure
Multiply modified by Variable Declaration rule modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(31)

Goal:
(0  + (I  * J)) = (I  * J)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
```

```
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
```

VC: 3_2:
Base Case of the Invariant of While Statement in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(31)

Goal:
```
(0  + J) = J
```

Given:
```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
```

VC: 3_3:
Requires Clause of Add in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(34)

Goal:
```
(min_int  <= (Multiply'  + I))
```

Given:
```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)
```

```
VC: 3_4:
Requires Clause of Add in Procedure Multiply modified by
Variable Declaration rule modified by Variable Declaration
rule: Add_And_Multiply_Realiz.rb(34)

Goal:
(Multiply'  + I) <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)


VC: 3_5:
Requires Clause of Increment in Procedure Multiply modified by
Variable Declaration rule modified by Variable Declaration
rule: Add_And_Multiply_Realiz.rb(35)

Goal:
(nj'  + 1) <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)


VC: 3_6:
Requires Clause of Decrement in Procedure Multiply modified by
Variable Declaration rule modified by Variable Declaration
rule: Add_And_Multiply_Realiz.rb(36)

Goal:
```

```
(min_int  <= (J'  - 1))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)


VC: 3_7:
Inductive Case of Invariant of While Statement in Procedure
Multiply modified by Variable Declaration rule modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(31)

Goal:
(Multiply'  + I) + (I  * (J'  - 1))) = (I  * J)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)


VC: 3_8:
Inductive Case of Invariant of While Statement in Procedure
Multiply modified by Variable Declaration rule modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(31)

Goal:
(nj'  + 1) + (J'  - 1)) = J

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
```

```
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)
```

VC: 3_9:
Termination of While Statement in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(32)

Goal:

```
(J'  - 1) < J')
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: (J'  > 0)
```

Free Variables:

```
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
I:Z, J:Z, Multiply:Z, nj:Z, zero:Z, P_val':N, J':Z,
Multiply':Z, nj':Z
```

VC: 4_1:
Ensures Clause of Multiply , If "if" condition at Add_And_Multiply_Realiz.rb(28) is true modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(52)

Goal:

```
Multiply' = (I  * J)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: (J  >= 0)
11: (Multiply'  + (I  * J')) = (I  * J)
12: (nj'  + J') = J
13: not((J'  > 0))



Free Variables:
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
I:Z, J:Z, Multiply:Z, nj:Z, zero:Z, J':Z, nj':Z, Multiply':Z



VC: 5_1:
Base Case of the Invariant of While Statement in Procedure
Multiply modified by Variable Declaration rule modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(41)

Goal:
(0  - (I  * J)) = (-(I)  * J)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))


VC: 5_2:
Base Case of the Invariant of While Statement in Procedure
Multiply modified by Variable Declaration rule modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(41)

Goal:
(0  + J) = J

Given:
1: (Last_Char_Num  > 0)
```

```
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
```

VC: 5_3:
Base Case of the Invariant of While Statement in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(41)

Goal:

```
(J  <= 0)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
```

VC: 5_4:
Requires Clause of Add in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(44)

Goal:

```
(min_int  <= (Multiply'  + I))
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
```

```
14: J' /= 0


VC: 5_5:
Requires Clause of Add in Procedure Multiply modified by
Variable Declaration rule modified by Variable Declaration
rule: Add_And_Multiply_Realiz.rb(44)

Goal:
(Multiply'  + I) <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' /= 0


VC: 5_6:
Requires Clause of Decrement in Procedure Multiply modified by
Variable Declaration rule modified by Variable Declaration
rule: Add_And_Multiply_Realiz.rb(45)

Goal:
(min_int  <= (nj'  - 1))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' /= 0


VC: 5_7:
```

Requires Clause of Increment in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(46)

Goal:
(J' + 1) <= max_int)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int <= -(I))
7: (-(I) <= max_int)
8: (min_int <= (I * J))
9: (I * J) <= max_int)
10: not((J >= 0))
11: (Multiply' - (I * J')) = (-(I) * J)
12: (nj' + J') = J
13: (J' <= 0)
14: J' /= 0

VC: 5_8:
Inductive Case of Invariant of While Statement in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(41)

Goal:
(Multiply' + I) - (I * (J' + 1))) = (-(I) * J)

Given:
1: (Last_Char_Num > 0)
2: (min_int <= 0)
3: (0 < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int <= -(I))
7: (-(I) <= max_int)
8: (min_int <= (I * J))
9: (I * J) <= max_int)
10: not((J >= 0))
11: (Multiply' - (I * J')) = (-(I) * J)
12: (nj' + J') = J
13: (J' <= 0)
14: J' /= 0

VC: 5_9:
Inductive Case of Invariant of While Statement in Procedure Multiply modified by Variable Declaration rule modified by Variable Declaration rule: Add_And_Multiply_Realiz.rb(41)

```
Goal:
(nj'  - 1) + (J'  + 1)) = J

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' /= 0


VC: 5_10:
Inductive Case of Invariant of While Statement in Procedure
Multiply modified by Variable Declaration rule modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(41)

Goal:
(J'  + 1) <= 0)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' /= 0


VC: 5_11:
Termination of While Statement in Procedure Multiply modified
by Variable Declaration rule modified by Variable Declaration
rule: Add_And_Multiply_Realiz.rb(42)

Goal:
(-((J'  + 1))  < -)

Given:
```

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' /= 0



Free Variables:
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Last_Char_Num:N,
I:Z, J:Z, Multiply:Z, nj:Z, zero:Z, P_val':N, J':Z,
Multiply':Z, nj':Z



VC: 6_1:
Requires Clause of Negate in Procedure Multiply modified by
Variable Declaration rule: Add_And_Multiply_Realiz.rb(48)

Goal:
(min_int <= -) and (- <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' = 0


VC: 6_2:
Ensures Clause of Multiply , If "if" condition at
Add_And_Multiply_Realiz.rb(28) is false modified by Variable
Declaration rule: Add_And_Multiply_Realiz.rb(52)
```

```
Goal:
- = (I  * J)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (min_int <= J) and (J <= max_int)
5: (min_int <= I) and (I <= max_int)
6: (min_int  <= -(I))
7: (-(I)  <= max_int)
8: (min_int  <= (I  * J))
9: (I  * J) <= max_int)
10: not((J  >= 0))
11: (Multiply'  - (I  * J')) = (-(I)  * J)
12: (nj'  + J') = J
13: (J'  <= 0)
14: J' = 0
```

Benchmark #2:Binary Search in an Array

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  MyBinarySearchRealiz.rb
// on:         Fri Nov 11 10:27:28 EST 2011
//

Free Variables:
Lower_Bound:Z, Upper_Bound:Z, Max_Char_Str_Len:N, min_int:Z,
max_int:Z, Last_Char_Num:N, x:Entry, y:Entry



VC: 0_1:
Ensures Clause of Are_Equal: MyBinarySearchRealiz.rb(8)

Goal:
1: LEQ(x, y) and LEQ(y, x) = x = y

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Total(LEQ)
8: Is_Transitive(LEQ)
```

```
9: Is_Symmetric(LEQ)


VC: 0_2:
Ensures Clause of Are_Equal: MyBinarySearchRealiz.rb(8)

Goal:
x = x

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Total(LEQ)
8: Is_Transitive(LEQ)
9: Is_Symmetric(LEQ)


VC: 0_3:
Ensures Clause of Are_Equal: MyBinarySearchRealiz.rb(8)

Goal:
y = y

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Total(LEQ)
8: Is_Transitive(LEQ)
9: Is_Symmetric(LEQ)



Free Variables:
Lower_Bound:Z, Upper_Bound:Z, Max_Char_Str_Len:N, min_int:Z,
max_int:Z, Last_Char_Num:N, key:Entry, A:Z -> Entry,
Is_Present:Boolean.B, low:Z, mid:Z, high:Z, midVal:Entry,
lowVal:Entry, highVal:Entry, A':Z -> Entry, midVal':Entry,
A'':Z -> Entry, midVal'':Entry, mid':Z, A''':Z -> Entry,
Is_Present':Boolean.B, low':Z, high':Z



VC: 1_1:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)
```

```
Goal:
false = (Exists_Between(key, A, Lower_Bound, (Lower_Bound  -
1))  or Exists_Between(key, A, (Upper_Bound  + 1),
Upper_Bound))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 1_2:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
(Lower_Bound  <= Lower_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 1_3:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
(Upper_Bound  <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 1_4:
```

```
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
A = A

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 1_5:
Requires Clause of high - low in Procedure Is_Present:
MyBinarySearchRealiz.rb(33)

Goal:
(min_int <= (high' - low')) and ((high' - low') <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')


VC: 1_6:
Requires Clause of Divide in Procedure Is_Present:
MyBinarySearchRealiz.rb(34)

Goal:
If (2 <= 0) then ((((2 * (max_int + 1)) < (high' - low')) and
((high' - low') < (2 * (min_int - 1)))))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
```

```
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')


VC: 1_7:
Requires Clause of low + mid in Procedure Is_Present:
MyBinarySearchRealiz.rb(35)

Goal:
(min_int <= (low' + mid')) and ((low' + mid') <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)


VC: 1_8:
Requires Clause of Swap_Entry in Procedure Is_Present:
MyBinarySearchRealiz.rb(36)

Goal:
(Lower_Bound  <= (low'  + mid'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
```

```
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low'))))|  < |2|)


VC: 1_9:
Requires Clause of Swap_Entry in Procedure Is_Present:
MyBinarySearchRealiz.rb(36)

Goal:
(low'  + mid') <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low'))))|  < |2|)


VC: 1_10:
Requires Clause of high + 1 in Procedure Is_Present:
MyBinarySearchRealiz.rb(39)

Goal:
(min_int <= (high' + 1)) and ((high' + 1) <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
```

```
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key


VC: 1_11:
Requires Clause of Swap_Entry in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is true:
MyBinarySearchRealiz.rb(47)

Goal:
(Lower_Bound  <= (low'  + mid'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key


VC: 1_12:
Requires Clause of Swap_Entry in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is true:
MyBinarySearchRealiz.rb(47)
```

```
Goal:
(low'  + mid') <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key


VC: 1_13:
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is true: MyBinarySearchRealiz.rb(30)

Goal:
true = (Exists_Between(key, lambda j: Z ({{key if j = (low'  +
mid')
A''(j) otherwise
}}), Lower_Bound, ((high'  + 1)  - 1))  or Exists_Between(key,
lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}}), (high'  + 1), Upper_Bound))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
```

```
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low'))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key
18: midVal' = A''((low'  + mid'))
19: A' = lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}})


VC: 1_14:
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is true: MyBinarySearchRealiz.rb(30)

Goal:
(Lower_Bound  <= (high'  + 1))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low'))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key
18: midVal' = A''((low'  + mid'))
19: A' = lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}})


VC: 1_15:
```

```
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is true: MyBinarySearchRealiz.rb(30)

Goal:
(high'  <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key
18: midVal' = A''((low'  + mid'))
19: A' = lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}})


VC: 1_16:
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is true: MyBinarySearchRealiz.rb(30)

Goal:
lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}}) = A

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
```

```
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key
18: midVal' = A''((low'  + mid'))
19: A' = lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}})
```

VC: 1_17:
Termination of While Statement in Procedure Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37) is true:
MyBinarySearchRealiz.rb(31)

Goal:

```
(high'  - (high'  + 1)) < (high'  - low'))
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) = key
18: midVal' = A''((low'  + mid'))
19: A' = lambda j: Z ({{key if j = (low'  + mid')
A''(j) otherwise
}})
```

```
Free Variables:
Lower_Bound:Z, Upper_Bound:Z, Max_Char_Str_Len:N, min_int:Z,
max_int:Z, Last_Char_Num:N, key:Entry, A:Z -> Entry,
Is_Present:Boolean.B, low:Z, mid:Z, high:Z, midVal:Entry,
lowVal:Entry, highVal:Entry, A':Z -> Entry, midVal':Entry,
A'':Z -> Entry, midVal'':Entry, mid':Z, A''':Z -> Entry,
Is_Present':Boolean.B, low':Z, high':Z


VC: 2_1:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
false = (Exists_Between(key, A, Lower_Bound, (Lower_Bound  -
1))  or Exists_Between(key, A, (Upper_Bound  + 1),
Upper_Bound))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 2_2:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
(Lower_Bound  <= Lower_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 2_3:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)
```

```
Goal:
(Upper_Bound  <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 2_4:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
A = A

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 2_5:
Requires Clause of high - low in Procedure Is_Present:
MyBinarySearchRealiz.rb(33)

Goal:
(min_int <= (high' - low')) and ((high' - low') <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
```

```
12: A''' = A
13: (low'  <= high')


VC: 2_6:
Requires Clause of Divide in Procedure Is_Present:
MyBinarySearchRealiz.rb(34)

Goal:
If (2 <= 0) then ((((2 * (max_int + 1)) < (high' - low')) and
((high' - low') < (2 * (min_int - 1)))))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')


VC: 2_7:
Requires Clause of low + mid in Procedure Is_Present:
MyBinarySearchRealiz.rb(35)

Goal:
(min_int <= (low' + mid')) and ((low' + mid') <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
```

```
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)


VC: 2_8:
Requires Clause of Swap_Entry in Procedure Is_Present:
MyBinarySearchRealiz.rb(36)

Goal:
(Lower_Bound  <= (low'  + mid'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)


VC: 2_9:
Requires Clause of Swap_Entry in Procedure Is_Present:
MyBinarySearchRealiz.rb(36)

Goal:
(low'  + mid') <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
```

```
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)


VC: 2_10:
Requires Clause of mid + 1 in Procedure Is_Present:
MyBinarySearchRealiz.rb(42)

Goal:
(min_int <= ((low' + mid') + 1)) and (((low' + mid') + 1) <=
max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)


VC: 2_11:
Requires Clause of Swap_Entry in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is false , If
"if" condition at MyBinarySearchRealiz.rb(41) is true:
MyBinarySearchRealiz.rb(47)

Goal:
(Lower_Bound  <= (low'  + mid'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
```

```
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
```

VC: 2_12:
Requires Clause of Swap_Entry in Procedure Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37) is false , If "if" condition at MyBinarySearchRealiz.rb(41) is true:
MyBinarySearchRealiz.rb(47)

Goal:
(low' + mid') <= Upper_Bound)

Given:
```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
```

VC: 2_13:
Inductive Case of Invariant of While Statement in Procedure Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)

```
is false , If "if" condition at MyBinarySearchRealiz.rb(41) is
true: MyBinarySearchRealiz.rb(30)

Goal:
(Exists_Between(key, A, Lower_Bound, (low'  - 1))  or
Exists_Between(key, A, (high'  + 1), Upper_Bound)) =
(Exists_Between(key, lambda j: Z ({{A'''((low'  + mid')) if j
= (low'  + mid')
A''(j) otherwise
}}), Lower_Bound, (((low'  + mid')  + 1)  - 1))  or
Exists_Between(key, lambda j: Z ({{A'''((low'  + mid')) if j =
(low'  + mid')
A''(j) otherwise
}}), (high'  + 1), Upper_Bound))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})


VC: 2_14:
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is false , If "if" condition at MyBinarySearchRealiz.rb(41) is
true: MyBinarySearchRealiz.rb(30)

Goal:
(Lower_Bound  <= (low'  + mid') + 1))
```

```
Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})


VC: 2_15:
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is false , If "if" condition at MyBinarySearchRealiz.rb(41) is
true: MyBinarySearchRealiz.rb(30)

Goal:
(high'  <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
```

```
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})
```

VC: 2_16:
Inductive Case of Invariant of While Statement in Procedure Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37) is false , If "if" condition at MyBinarySearchRealiz.rb(41) is true: MyBinarySearchRealiz.rb(30)

Goal:

```
lambda j: Z ({{A'''((low'  + mid')) if j = (low'  + mid')
A''(j) otherwise
}}) = A
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})
```

```
VC: 2_17:
Termination of While Statement in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is false , If
"if" condition at MyBinarySearchRealiz.rb(41) is true:
MyBinarySearchRealiz.rb(31)

Goal:
(high'  - (low'  + mid') + 1)) < (high'  - low'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: LEQ(A'''((low'  + mid')), key)
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})




Free Variables:
Lower_Bound:Z, Upper_Bound:Z, Max_Char_Str_Len:N, min_int:Z,
max_int:Z, Last_Char_Num:N, key:Entry, A:Z -> Entry,
Is_Present:Boolean.B, low:Z, mid:Z, high:Z, midVal:Entry,
lowVal:Entry, highVal:Entry, A':Z -> Entry, midVal':Entry,
A'':Z -> Entry, midVal'':Entry, mid':Z, A''':Z -> Entry,
Is_Present':Boolean.B, low':Z, high':Z



VC: 3_1:
```

```
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
false = (Exists_Between(key, A, Lower_Bound, (Lower_Bound  -
1))  or Exists_Between(key, A, (Upper_Bound  + 1),
Upper_Bound))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 3_2:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
(Lower_Bound  <= Lower_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 3_3:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
(Upper_Bound  <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
```

```
VC: 3_4:
Base Case of the Invariant of While Statement in Procedure
Is_Present: MyBinarySearchRealiz.rb(30)

Goal:
A = A

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)


VC: 3_5:
Requires Clause of high - low in Procedure Is_Present:
MyBinarySearchRealiz.rb(33)

Goal:
(min_int <= (high' - low')) and ((high' - low') <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')


VC: 3_6:
Requires Clause of Divide in Procedure Is_Present:
MyBinarySearchRealiz.rb(34)

Goal:
If (2 <= 0) then ((((2 * (max_int + 1)) < (high' - low')) and
((high' - low') < (2 * (min_int - 1)))))

Given:
1: (Last_Char_Num  > 0)
```

```
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')


VC: 3_7:
Requires Clause of low + mid in Procedure Is_Present:
MyBinarySearchRealiz.rb(35)

Goal:
(min_int <= (low' + mid')) and ((low' + mid') <= max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)


VC: 3_8:
Requires Clause of Swap_Entry in Procedure Is_Present:
MyBinarySearchRealiz.rb(36)

Goal:
(Lower_Bound  <= (low'  + mid'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
```

```
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low'))))|  < |2|)


VC: 3_9:
Requires Clause of Swap_Entry in Procedure Is_Present:
MyBinarySearchRealiz.rb(36)

Goal:
(low'  + mid') <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low'))))|  < |2|)


VC: 3_10:
Requires Clause of mid - 1 in Procedure Is_Present:
MyBinarySearchRealiz.rb(44)

Goal:
(min_int <= ((low' + mid') - 1)) and (((low' + mid') - 1) <=
max_int)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
```

```
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))


VC: 3_11:
Requires Clause of Swap_Entry in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is false , If
"if" condition at MyBinarySearchRealiz.rb(41) is false:
MyBinarySearchRealiz.rb(47)

Goal:
(Lower_Bound  <= (low'  + mid'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))
```

```
VC: 3_12:
Requires Clause of Swap_Entry in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is false , If
"if" condition at MyBinarySearchRealiz.rb(41) is false:
MyBinarySearchRealiz.rb(47)

Goal:
(low'  + mid') <= Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))


VC: 3_13:
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is false , If "if" condition at MyBinarySearchRealiz.rb(41) is
false: MyBinarySearchRealiz.rb(30)

Goal:
(Exists_Between(key, A, Lower_Bound, (low'  - 1))  or
Exists_Between(key, A, (high'  + 1), Upper_Bound)) =
(Exists_Between(key, lambda j: Z ({{A'''((low'  + mid')) if j
= (low'  + mid')
A''(j) otherwise
}}), Lower_Bound, (low'  - 1))  or Exists_Between(key, lambda
j: Z ({{A'''((low'  + mid')) if j = (low'  + mid')
A''(j) otherwise
}}), (((low'  + mid')  - 1)  + 1), Upper_Bound))

Given:
```

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})
```

VC: 3_14:
Inductive Case of Invariant of While Statement in Procedure Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37) is false , If "if" condition at MyBinarySearchRealiz.rb(41) is false: MyBinarySearchRealiz.rb(30)

Goal:
(Lower_Bound <= low')

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
```

```
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})
```

VC: 3_15:
Inductive Case of Invariant of While Statement in Procedure Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37) is false , If "if" condition at MyBinarySearchRealiz.rb(41) is false: MyBinarySearchRealiz.rb(30)

Goal:

```
(low'  + mid') - 1) <= Upper_Bound)
```

Given:

```
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})
```

VC: 3_16:

```
Inductive Case of Invariant of While Statement in Procedure
Is_Present , If "if" condition at MyBinarySearchRealiz.rb(37)
is false , If "if" condition at MyBinarySearchRealiz.rb(41) is
false: MyBinarySearchRealiz.rb(30)

Goal:
lambda j: Z ({{A'''((low'  + mid')) if j = (low'  + mid')
A''(j) otherwise
}}) = A

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})


VC: 3_17:
Termination of While Statement in Procedure Is_Present , If
"if" condition at MyBinarySearchRealiz.rb(37) is false , If
"if" condition at MyBinarySearchRealiz.rb(41) is false:
MyBinarySearchRealiz.rb(31)

Goal:
(low'  + mid') - 1) - low') < (high'  - low'))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
```

```
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A''', Lower_Bound, (low'
- 1))  or Exists_Between(key, A''', (high'  + 1),
Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A''' = A
13: (low'  <= high')
14: (|(2  * (high'  - low'))|  <= |(high'  - low')|)
15: (|(high'  - low') - (2  * (high'  - low')))|  < |2|)
16: A'' = lambda j: Z ({{midVal'' if j = (low'  + mid')
A'''(j) otherwise
}})
17: A'''((low'  + mid')) /= key
18: not(LEQ(A'''((low'  + mid')), key))
19: midVal' = A''((low'  + mid'))
20: A' = lambda j: Z ({{A'''((low'  + mid')) if j = (low'  +
mid')
A''(j) otherwise
}})
```

```
Free Variables:
Lower_Bound:Z, Upper_Bound:Z, Max_Char_Str_Len:N, min_int:Z,
max_int:Z, Last_Char_Num:N, key:Entry, A:Z -> Entry,
Is_Present:Boolean.B, low:Z, mid:Z, high:Z, midVal:Entry,
lowVal:Entry, highVal:Entry, P_val':N, A':Z -> Entry,
Is_Present':Boolean.B, low':Z, high':Z
```

```
VC: 4_1:
Ensures Clause of Is_Present: MyBinarySearchRealiz.rb(48)

Goal:
(Exists_Between(key, A, Lower_Bound, (low'  - 1))  or
Exists_Between(key, A, (high'  + 1), Upper_Bound)) =
Exists_Between(key, A, Lower_Bound, Upper_Bound)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
```

```
9: Is_Present' = (Exists_Between(key, A', Lower_Bound, (low'
- 1))  or Exists_Between(key, A', (high'  + 1), Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A' = A
13: not((low'  <= high'))
```

```
VC: 4_2:
Ensures Clause of Is_Present: MyBinarySearchRealiz.rb(48)

Goal:
key = key

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A', Lower_Bound, (low'
- 1))  or Exists_Between(key, A', (high'  + 1), Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A' = A
13: not((low'  <= high'))
```

```
VC: 4_3:
Ensures Clause of Is_Present: MyBinarySearchRealiz.rb(48)

Goal:
A = A

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Lower_Bound  <= (Upper_Bound  + 1))
5: (min_int <= Upper_Bound) and (Upper_Bound <= max_int)
6: (min_int <= Lower_Bound) and (Lower_Bound <= max_int)
7: Is_Ordered(A, Lower_Bound, Upper_Bound)
8: (Upper_Bound  + 1) <= max_int)
9: Is_Present' = (Exists_Between(key, A', Lower_Bound, (low'
- 1))  or Exists_Between(key, A', (high'  + 1), Upper_Bound))
10: (Lower_Bound  <= low')
11: (high'  <= Upper_Bound)
12: A' = A
13: not((low'  <= high'))
```

Benchmark #4:Layered implementation of a Map ADT

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Search_Store_Realiz.rb
// on:         Mon Dec 05 07:40:37 EST 2011
//

Free Variables:
Entry, Max_Length:Z



VC: 0_1:
Requirement for Facility Declaration Rule for PQ_Fac:
Search_Store_Realiz.rb(10)

Goal:
(Max_Length  > 0)

Given:

1: (Max_Length  > 0)




Free Variables:
S:(Contents:PQ_Fac.P_Queue), Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N



VC: 1_1:
Correspondence Rule for Store: Search_Store_Realiz.rb(20)

Goal:
(Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))  <=
Max_Capacity)

Given:
1: (Max_Capacity  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: (Last_Char_Num  > 0)
5: (Max_Capacity  > 0)
6: (|S.Contents|  <= Max_Capacity)
7: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
8: Is_Duplicate_Free(S.Contents)
```

```
Free Variables:
S:(Contents:PQ_Fac.P_Queue), Conc.S:Key -> Boolean.B,
S.Contents:Modified_String_Theory.Str(Entry)



VC: 2_1:
Convention for Store generated by intialization rule:
Search_Store_Realiz.rb(18)

Goal:
Is_Duplicate_Free(empty_string)

Given:

1: (Max_Capacity  > 0)


VC: 2_2:
Initialization Rule for Store: Search_Store_Realiz.rb(15)

Goal:
Key_Ct(lambda k2: Key (Is_Substring(<k2>, empty_string))) = 0

Given:
1: (Max_Capacity  > 0)
2: Conc.S = lambda k2: Key (Is_Substring(<k2>, empty_string))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
Boolean.B, k:Key, S:(Contents:PQ_Fac.P_Queue), t:Key, t':Key,
S':(Contents:PQ_Fac.P_Queue)



VC: 3_1:
Requires Clause of Enqueue in Procedure Add:
Search_Store_Realiz.rb(25)

Goal:
(|S.Contents|  < Max_Capacity)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
```

```
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
< Max_Capacity)
12: not(Conc.S(k))


VC: 3_2:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(25)

Goal:
Is_Duplicate_Free((S.Contents  o <k>))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
< Max_Capacity)
12: not(Conc.S(k))


VC: 3_3:
Ensures Clause of Add: Search_Store_Realiz.rb(26)

Goal:
Conc.S'(k)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
< Max_Capacity)
12: not(Conc.S(k))
13: Conc.S' = lambda k2: Key (Is_Substring(<k2>, (S.Contents
o <k>)))
```

```
VC: 3_4:
Ensures Clause of Add: Search_Store_Realiz.rb(26)

Goal:
for all k1:Key, If k1 /= k then (Conc.S'(k1) = Conc.S(k1))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
< Max_Capacity)
12: not(Conc.S(k))
13: Conc.S' = lambda k2: Key (Is_Substring(<k2>, (S.Contents
o <k>)))


VC: 3_5:
Ensures Clause of Add: Search_Store_Realiz.rb(26)

Goal:
k = k

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
< Max_Capacity)
12: not(Conc.S(k))
13: Conc.S' = lambda k2: Key (Is_Substring(<k2>, (S.Contents
o <k>)))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
```

```
Boolean.B, k:Key, S:(Contents:PQ_Fac.P_Queue), b:Boolean.B,
k':Key, S':(Contents:PQ_Fac.P_Queue),
S'':(Contents:PQ_Fac.P_Queue), b':Boolean.B



VC: 4_1:
Requires Clause of Dequeue in Procedure Remove:
Search_Store_Realiz.rb(31)

Goal:
(|S''.Contents|  > 0)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Conc.S(k)
12: Is_Permutation(S''.Contents, S.Contents)
13: (Is_Substring(<k>, S.Contents)  iff b' = true)
14: b' = true
15: Is_Prefix(<k>, S''.Contents)


VC: 4_2:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(31)

Goal:
Is_Duplicate_Free(S'.Contents)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Conc.S(k)
12: Is_Permutation(S''.Contents, S.Contents)
13: (Is_Substring(<k>, S.Contents)  iff b' = true)
14: b' = true
15: Is_Prefix(<k>, S''.Contents)
16: S''.Contents = (<k'>  o S'.Contents)
```

```
VC: 4_3:
Ensures Clause of Remove: Search_Store_Realiz.rb(32)

Goal:
not(Conc.S'(k'))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Conc.S(k)
12: Is_Permutation(S''.Contents, S.Contents)
13: (Is_Substring(<k>, S.Contents)  iff b' = true)
14: b' = true
15: Is_Prefix(<k>, S''.Contents)
16: S''.Contents = (<k'>  o S'.Contents)
17: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))


VC: 4_4:
Ensures Clause of Remove: Search_Store_Realiz.rb(32)

Goal:
for all k1:Key, If k1 /= k' then (Conc.S'(k1) = Conc.S(k1))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Conc.S(k)
12: Is_Permutation(S''.Contents, S.Contents)
13: (Is_Substring(<k>, S.Contents)  iff b' = true)
14: b' = true
15: Is_Prefix(<k>, S''.Contents)
16: S''.Contents = (<k'>  o S'.Contents)
17: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))
```

```
VC: 4_5:
Ensures Clause of Remove: Search_Store_Realiz.rb(32)

Goal:
k = k'

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Conc.S(k)
12: Is_Permutation(S''.Contents, S.Contents)
13: (Is_Substring(<k>, S.Contents)  iff b' = true)
14: b' = true
15: Is_Prefix(<k>, S''.Contents)
16: S''.Contents = (<k'>  o S'.Contents)
17: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
Boolean.B, k:Key, S:(Contents:PQ_Fac.P_Queue), k':Key,
S':(Contents:PQ_Fac.P_Queue)



VC: 5_1:
Requires Clause of Dequeue in Procedure Remove_Any:
Search_Store_Realiz.rb(35)

Goal:
(|S.Contents|  > 0)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
```

```
11: Key.is_initial(k)
12: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
> 0)


VC: 5_2:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(35)

Goal:
Is_Duplicate_Free(S'.Contents)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Key.is_initial(k)
12: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
> 0)
13: S.Contents = (<k'>  o S'.Contents)


VC: 5_3:
: Search_Store_Template.co(63)

Goal:
Conc.S(k')

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Key.is_initial(k)
12: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
> 0)
13: S.Contents = (<k'>  o S'.Contents)
14: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))


VC: 5_4:
Ensures Clause of Remove_Any: Search_Store_Realiz.rb(36)
```

```
Goal:
not(Conc.S'(k'))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Key.is_initial(k)
12: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
> 0)
13: S.Contents = (<k'>  o S'.Contents)
14: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))


VC: 5_5:
Ensures Clause of Remove_Any: Search_Store_Realiz.rb(36)

Goal:
for all k1:Key, If k1 /= k' then (Conc.S'(k1) = Conc.S(k1))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Key.is_initial(k)
12: (Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents)))
> 0)
13: S.Contents = (<k'>  o S'.Contents)
14: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
Boolean.B, k:Key, S:(Contents:PQ_Fac.P_Queue),
Is_Present:Boolean.B, b:Boolean.B,
S':(Contents:PQ_Fac.P_Queue), b':Boolean.B
```

```
VC: 6_1:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(41)

Goal:
Is_Duplicate_Free(S'.Contents)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Is_Permutation(S'.Contents, S.Contents)
12: (Is_Substring(<k>, S.Contents)  iff b' = true)
13: b' = true
14: Is_Prefix(<k>, S'.Contents)


VC: 6_2:
Ensures Clause of Is_Present: Search_Store_Realiz.rb(42)

Goal:
b' = Conc.S'(k)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Is_Permutation(S'.Contents, S.Contents)
12: (Is_Substring(<k>, S.Contents)  iff b' = true)
13: b' = true
14: Is_Prefix(<k>, S'.Contents)
15: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))


VC: 6_3:
Ensures Clause of Is_Present: Search_Store_Realiz.rb(42)

Goal:
```

```
k = k

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Is_Permutation(S'.Contents, S.Contents)
12: (Is_Substring(<k>, S.Contents)  iff b' = true)
13: b' = true
14: Is_Prefix(<k>, S'.Contents)
15: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))


VC: 6_4:
Ensures Clause of Is_Present: Search_Store_Realiz.rb(42)

Goal:
lambda k2: Key (Is_Substring(<k2>, S.Contents)) = lambda k2:
Key (Is_Substring(<k2>, S'.Contents))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))
11: Is_Permutation(S'.Contents, S.Contents)
12: (Is_Substring(<k>, S.Contents)  iff b' = true)
13: b' = true
14: Is_Prefix(<k>, S'.Contents)
15: Conc.S' = lambda k2: Key (Is_Substring(<k2>, S'.Contents))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
Boolean.B, S:(Contents:PQ_Fac.P_Queue), Key_Count:Z
```

```
VC: 7_1:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(45)

Goal:
Is_Duplicate_Free(S.Contents)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))


VC: 7_2:
Ensures Clause of Key_Count: Search_Store_Realiz.rb(46)

Goal:
|S.Contents| = Key_Ct(lambda k2: Key (Is_Substring(<k2>,
S.Contents)))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))


VC: 7_3:
Ensures Clause of Key_Count: Search_Store_Realiz.rb(46)

Goal:
lambda k2: Key (Is_Substring(<k2>, S.Contents)) = lambda k2:
Key (Is_Substring(<k2>, S.Contents))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
```

```
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
Boolean.B, S:(Contents:PQ_Fac.P_Queue), Rem_Capacity:Z



VC: 8_1:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(49)

Goal:
Is_Duplicate_Free(S.Contents)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))


VC: 8_2:
Ensures Clause of Rem_Capacity: Search_Store_Realiz.rb(50)

Goal:
(Max_Capacity  - |S.Contents|) = (Max_Capacity  -
Key_Ct(lambda k2: Key (Is_Substring(<k2>, S.Contents))))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))


VC: 8_3:
```

```
Ensures Clause of Rem_Capacity: Search_Store_Realiz.rb(50)

Goal:
lambda k2: Key (Is_Substring(<k2>, S.Contents)) = lambda k2:
Key (Is_Substring(<k2>, S.Contents))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
10: Conc.S = lambda k2: Key (Is_Substring(<k2>, S.Contents))



Free Variables:
Max_Capacity:Z, Key_Ct:N, Max_Length:Z, Last_Char_Num:N,
min_int:Z, max_int:Z, Max_Char_Str_Len:N, Conc.S:Key ->
Boolean.B, S:(Contents:PQ_Fac.P_Queue),
S':(Contents:PQ_Fac.P_Queue)



VC: 9_1:
Convention for Search_Store_Realiz: Search_Store_Realiz.rb(53)

Goal:
Is_Duplicate_Free(empty_string)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)


VC: 9_2:
Ensures Clause of Clear: Search_Store_Realiz.rb(54)

Goal:
true

Given:
```

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)


VC: 9_3:
Ensures Clause of Clear: Search_Store_Realiz.rb(54)

Goal:
Key_Ct(lambda k2: Key (Is_Substring(<k2>, empty_string))) = 0

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (Max_Capacity  > 0)
5: (|S.Contents|  <= Max_Capacity)
6: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
7: (Max_Capacity  > 0)
8: (min_int <= Max_Capacity) and (Max_Capacity <= max_int)
9: Is_Duplicate_Free(S.Contents)
```

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Queue_Location_Linking_Realiz.rb
// on:         Mon Dec 05 17:39:20 EST 2011
//

Free Variables:
Info



VC: 0_1:
Requirement for Facility Declaration Rule for Entry_Ptr_Fac:
Queue_Location_Linking_Realiz.rb(15)

Goal:
true

Given:




Free Variables:
Q:(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position),
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z



VC: 1_1:
Correspondence Rule for Queue:
Queue_Location_Linking_Realiz.rb(26)

Goal:
true

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)



Free Variables:
```

```
Q:(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position),
Conc.Q:Str(Entry), Content:Z -> Info, Ref:Z -> Z, Q.Back:Z,
Q.Front:Z



VC: 2_1:
Convention for Queue generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Void, Void, Ref)

Given:



VC: 2_2:
Convention for Queue generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Ref(Void) = Void

Given:



VC: 2_3:
Convention for Queue generated by intialization rule generated
by intialization rule generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
(Void = Void  iff Void = Void)

Given:

1: true


VC: 2_4:
Initialization Rule for Queue:
Queue_Location_Linking_Realiz.rb(18)

Goal:
Str_Info(Void, Content, Ref) = empty_string

Given:

1: Conc.Q = Str_Info(Void, Content, Ref)
```

```
Free Variables:
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z, Conc.Q:Str(Entry), R:Entry,
Q:(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position),
Content:Z -> Info, Ref:Z -> Z, Temp:Z,
Q':(Front:Entry_Ptr_Fac.Position;
Back:Entry_Ptr_Fac.Position), R':Entry



VC: 3_1:
Requires Clause of Swap_Info in Procedure Dequeue:
Queue_Location_Linking_Realiz.rb(31)

Goal:
Q.Front /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string


VC: 3_2:
Requires Clause of Follow_Link in Procedure Dequeue:
Queue_Location_Linking_Realiz.rb(32)

Goal:
Q.Front /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})


VC: 3_3:
```

```
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Void, Void, Ref)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) = Void


VC: 3_4:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Ref(Void) = Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) = Void


VC: 3_5:
Convention for Queue generated by intialization rule generated
by intialization rule generated by intialization rule , If
"if" condition at Queue_Location_Linking_Realiz.rb(33) is
true: Queue_Location_Linking_Realiz.rb(24)

Goal:
```

```
(Void = Void  iff Void = Void)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) = Void


VC: 3_6:
Ensures Clause of Dequeue , If "if" condition at
Queue_Location_Linking_Realiz.rb(33) is true:
Queue_Location_Linking_Realiz.rb(35)

Goal:
Str_Info(Q.Front, Content, Ref) = (<Content(Q.Front)>  o
Str_Info(Void, lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}}), Ref))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) = Void



Free Variables:
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z, Conc.Q:Str(Entry), R:Entry,
Q:(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position),
Content:Z -> Info, Ref:Z -> Z, Temp:Z,
Q':(Front:Entry_Ptr_Fac.Position;
Back:Entry_Ptr_Fac.Position), R':Entry
```

```
VC: 4_1:
Requires Clause of Swap_Info in Procedure Dequeue:
Queue_Location_Linking_Realiz.rb(31)

Goal:
Q.Front /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string


VC: 4_2:
Requires Clause of Follow_Link in Procedure Dequeue:
Queue_Location_Linking_Realiz.rb(32)

Goal:
Q.Front /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})


VC: 4_3:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true , If
"if" condition at Queue_Location_Linking_Realiz.rb(33) is
false: Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Ref(Q.Front), Q.Back, Ref)

Given:
```

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) /= Void
```

VC: 4_4:
Convention for Queue generated by intialization rule , If "if" condition at Queue_Location_Linking_Realiz.rb(33) is false: Queue_Location_Linking_Realiz.rb(24)

Goal:

```
Ref(Q.Back) = Void
```

Given:

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) /= Void
```

VC: 4_5:
Convention for Queue generated by intialization rule generated by intialization rule generated by intialization rule , If "if" condition at Queue_Location_Linking_Realiz.rb(33) is false: Queue_Location_Linking_Realiz.rb(24)

Goal:

```
(Q.Back = Void  iff Ref(Q.Front) = Void)
```

Given:

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
```

```
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) /= Void
```

```
VC: 4_6:
Ensures Clause of Dequeue , If "if" condition at
Queue_Location_Linking_Realiz.rb(33) is false:
Queue_Location_Linking_Realiz.rb(35)

Goal:
Str_Info(Q.Front, Content, Ref) = (<Content(Q.Front)>  o
Str_Info(Ref(Q.Front), lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}}), Ref))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Entry.is_initial(R)
9: Str_Info(Q.Front, Content, Ref) /= empty_string
10: Content' = lambda L: Z ({{R if L = Q.Front
Content(L) otherwise
}})
11: Ref(Q.Front) /= Void
```

```
Free Variables:
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z, Conc.Q:Str(Entry), E:Entry,
Q:(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position),
Content:Z -> Info, Ref:Z -> Z, Temp:Z,
Q':(Front:Entry_Ptr_Fac.Position;
Back:Entry_Ptr_Fac.Position), E':Entry, Temp':Z
```

```
VC: 5_1:
Requires Clause of Swap_Info in Procedure Enqueue:
Queue_Location_Linking_Realiz.rb(42)

Goal:
```

```
Temp' /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp' /= Void


VC: 5_2:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true , If
"if" condition at Queue_Location_Linking_Realiz.rb(33) is
false , If "if" condition at
Queue_Location_Linking_Realiz.rb(43) is true:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Temp', Temp', Ref)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp' /= Void
9: E' = Content(Temp')
10: Content' = lambda L: Z ({{E if L = Temp'
Content(L) otherwise
}})
11: Q.Front = Void


VC: 5_3:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(43) is true:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Ref(Temp') = Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
```

```
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp' /= Void
9: E' = Content(Temp')
10: Content' = lambda L: Z ({{E if L = Temp'
Content(L) otherwise
}})
11: Q.Front = Void
```

```
VC: 5_4:
Convention for Queue generated by intialization rule generated
by intialization rule generated by intialization rule , If
"if" condition at Queue_Location_Linking_Realiz.rb(43) is
true: Queue_Location_Linking_Realiz.rb(24)

Goal:
(Temp' = Void  iff Temp' = Void)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp' /= Void
9: E' = Content(Temp')
10: Content' = lambda L: Z ({{E if L = Temp'
Content(L) otherwise
}})
11: Q.Front = Void
```

```
VC: 5_5:
Ensures Clause of Enqueue , If "if" condition at
Queue_Location_Linking_Realiz.rb(43) is true:
Queue_Location_Linking_Realiz.rb(48)

Goal:
Str_Info(Temp', lambda L: Z ({{E if L = Temp'
Content(L) otherwise
}}), Ref) = (Str_Info(Q.Front, Content, Ref)  o <E>)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp' /= Void
```

```
9: E' = Content(Temp')
10: Content' = lambda L: Z ({{E if L = Temp'
Content(L) otherwise
}})
11: Q.Front = Void



Free Variables:
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z, Conc.Q:Str(Entry), E:Entry,
Q:(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position),
Content:Z -> Info, Ref:Z -> Z, Temp:Z, Temp':Z, E':Entry,
Temp'':Z



VC: 6_1:
Requires Clause of Swap_Info in Procedure Enqueue:
Queue_Location_Linking_Realiz.rb(42)

Goal:
Temp'' /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp'' /= Void


VC: 6_2:
Requires Clause of Redirect_Link in Procedure Enqueue:
Queue_Location_Linking_Realiz.rb(47)

Goal:
Q.Back /= Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp'' /= Void
9: E' = Content(Temp'')
10: Content' = lambda L: Z ({{E if L = Temp''
```

```
Content(L) otherwise
}})
11: Q.Front /= Void


VC: 6_3:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true , If
"if" condition at Queue_Location_Linking_Realiz.rb(33) is
false , If "if" condition at
Queue_Location_Linking_Realiz.rb(43) is true , If "if"
condition at Queue_Location_Linking_Realiz.rb(43) is false:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Q.Front, Q.Back, lambda L: Z ({{Temp'' if L =
Q.Back
Ref(L) otherwise
}}))

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp'' /= Void
9: E' = Content(Temp'')
10: Content' = lambda L: Z ({{E if L = Temp''
Content(L) otherwise
}})
11: Q.Front /= Void
12: Ref' = lambda L: Z ({{Temp'' if L = Q.Back
Ref(L) otherwise
}})
13: Temp' = Ref(Q.Back)


VC: 6_4:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(43) is false:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Ref'(Q.Back) = Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
```

```
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp'' /= Void
9: E' = Content(Temp'')
10: Content' = lambda L: Z ({{E if L = Temp''
Content(L) otherwise
}})
11: Q.Front /= Void
12: Ref' = lambda L: Z ({{Temp'' if L = Q.Back
Ref(L) otherwise
}})
13: Temp' = Ref(Q.Back)
```

VC: 6_5:
Convention for Queue generated by intialization rule generated by intialization rule generated by intialization rule , If "if" condition at Queue_Location_Linking_Realiz.rb(43) is false: Queue_Location_Linking_Realiz.rb(24)

Goal:

```
(Q.Back = Void  iff Q.Front = Void)
```

Given:

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp'' /= Void
9: E' = Content(Temp'')
10: Content' = lambda L: Z ({{E if L = Temp''
Content(L) otherwise
}})
11: Q.Front /= Void
12: Ref' = lambda L: Z ({{Temp'' if L = Q.Back
Ref(L) otherwise
}})
13: Temp' = Ref(Q.Back)
```

VC: 6_6:
Ensures Clause of Enqueue , If "if" condition at Queue_Location_Linking_Realiz.rb(43) is false: Queue_Location_Linking_Realiz.rb(48)

Goal:

```
Str_Info(Q.Front, lambda L: Z ({{E if L = Temp''
Content(L) otherwise
}}), lambda L: Z ({{Temp'' if L = Q.Back
Ref(L) otherwise
}})) = (Str_Info(Q.Front, Content, Ref)  o <E>)
```

```
Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)
8: Temp'' /= Void
9: E' = Content(Temp'')
10: Content' = lambda L: Z ({{E if L = Temp''
Content(L) otherwise
}})
11: Q.Front /= Void
12: Ref' = lambda L: Z ({{Temp'' if L = Q.Back
Ref(L) otherwise
}})
13: Temp' = Ref(Q.Back)



Free Variables:
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z, Conc.Q:Str(Entry), Q:(Front:Entry_Ptr_Fac.Position;
Back:Entry_Ptr_Fac.Position), Content:Z -> Info, Ref:Z -> Z,
Is_Empty:Boolean.B, Temp:Z



VC: 7_1:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true , If
"if" condition at Queue_Location_Linking_Realiz.rb(33) is
false , If "if" condition at
Queue_Location_Linking_Realiz.rb(43) is true , If "if"
condition at Queue_Location_Linking_Realiz.rb(43) is false:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Q.Front, Q.Back, Ref)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)


VC: 7_2:
```

```
Convention for Queue generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Ref(Q.Back) = Void

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)


VC: 7_3:
Convention for Queue generated by intialization rule generated
by intialization rule generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
(Q.Back = Void  iff Q.Front = Void)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)


VC: 7_4:
Ensures Clause of Is_Empty:
Queue_Location_Linking_Realiz.rb(55)

Goal:
Q.Front = Void = Str_Info(Q.Front, Content, Ref) =
empty_string

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)


VC: 7_5:
```

```
Ensures Clause of Is_Empty:
Queue_Location_Linking_Realiz.rb(55)

Goal:
Str_Info(Q.Front, Content, Ref) = Str_Info(Q.Front, Content,
Ref)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)
7: Conc.Q = Str_Info(Q.Front, Content, Ref)



Free Variables:
Void:Z, Max_Char_Str_Len:N, Last_Char_Num:N, min_int:Z,
max_int:Z, Conc.Q:Str(Entry), Q:(Front:Entry_Ptr_Fac.Position;
Back:Entry_Ptr_Fac.Position), Content:Z -> Info, Ref:Z -> Z,
Q':(Front:Entry_Ptr_Fac.Position; Back:Entry_Ptr_Fac.Position)



VC: 8_1:
Convention for Queue generated by intialization rule , If "if"
condition at Queue_Location_Linking_Realiz.rb(33) is true , If
"if" condition at Queue_Location_Linking_Realiz.rb(33) is
false , If "if" condition at
Queue_Location_Linking_Realiz.rb(43) is true , If "if"
condition at Queue_Location_Linking_Realiz.rb(43) is false:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Is_Reachable(Void, Void, Ref)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)


VC: 8_2:
Convention for Queue generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
Ref(Void) = Void
```

```
Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)


VC: 8_3:
Convention for Queue generated by intialization rule generated
by intialization rule generated by intialization rule:
Queue_Location_Linking_Realiz.rb(24)

Goal:
(Void = Void  iff Void = Void)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)


VC: 8_4:
Ensures Clause of Clear: Queue_Location_Linking_Realiz.rb(60)

Goal:
true

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
6: (Q.Back = Void  iff Q.Front = Void)


VC: 8_5:
Ensures Clause of Clear: Queue_Location_Linking_Realiz.rb(60)

Goal:
Str_Info(Void, Content, Ref) = empty_string

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: Is_Reachable(Q.Front, Q.Back, Ref)
5: Ref(Q.Back) = Void
```

```
6: (Q.Back = Void  iff Q.Front = Void)
```

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  UnboundedQueue_List_Realiz.rb
// on:         Fri Nov 11 09:40:00 EST 2011
//

Free Variables:
Entry



VC: 0_1:
Requirement for Facility Declaration Rule for List_Fac:
UnboundedQueue_List_Realiz.rb(4)

Goal:
true

Given:




Free Variables:
Q:List_Fac.List, Max_Char_Str_Len:N, min_int:Z, max_int:Z,
Last_Char_Num:N



VC: 1_1:
Correspondence Rule for Queue:
UnboundedQueue_List_Realiz.rb(10)

Goal:
true

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)



Free Variables:
Q:List_Fac.List, Conc.Q:Str(Entry)
```

```
VC: 2_1:
Initialization Rule for Queue:
UnboundedQueue_List_Realiz.rb(7)

Goal:
(Q.Prec  o Q.Rem) = empty_string

Given:
1: Q.Prec = empty_string
2: Q.Rem = empty_string
3: Conc.Q = (Q.Prec  o Q.Rem)



Free Variables:
Max_Char_Str_Len:N, min_int:Z, max_int:Z, Last_Char_Num:N,
Conc.Q:Str(Entry), R:Entry, Q:List_Fac.List, R':Entry,
Q':List_Fac.List, Q'':List_Fac.List



VC: 3_1:
Requires Clause of Remove in Procedure Dequeue:
UnboundedQueue_List_Realiz.rb(14)

Goal:
(Q.Prec  o Q.Rem) /= empty_string

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Entry.is_initial(R)
5: (Q.Prec  o Q.Rem) /= empty_string
6: Conc.Q = (Q.Prec  o Q.Rem)


VC: 3_2:
Ensures Clause of Dequeue: UnboundedQueue_List_Realiz.rb(15)

Goal:
(Q.Prec  o Q.Rem) = (<R'>  o (empty_string  o Q'.Rem))

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Entry.is_initial(R)
5: (Q.Prec  o Q.Rem) /= empty_string
6: Conc.Q = (Q.Prec  o Q.Rem)
7: (Q.Prec  o Q.Rem) = (<R'>  o Q'.Rem)
```

```
Free Variables:
Max_Char_Str_Len:N, min_int:Z, max_int:Z, Last_Char_Num:N,
Conc.Q:Str(Entry), E:Entry, Q:List_Fac.List, E':Entry,
Q':List_Fac.List



VC: 4_1:
Ensures Clause of Enqueue: UnboundedQueue_List_Realiz.rb(20)

Goal:
(Q.Prec  o Q.Rem) o (<E>  o empty_string)) = (Q.Prec  o Q.Rem)
o <E>)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Conc.Q = (Q.Prec  o Q.Rem)



Free Variables:
Max_Char_Str_Len:N, min_int:Z, max_int:Z, Last_Char_Num:N,
Conc.Q:Str(Entry), Q:List_Fac.List, Is_Empty:Boolean.B



VC: 5_1:
Ensures Clause of Is_Empty: UnboundedQueue_List_Realiz.rb(24)

Goal:
1: Q.Prec = empty_string and Q.Prec = empty_string = (Q.Prec
o Q.Rem) = empty_string

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
4: Conc.Q = (Q.Prec  o Q.Rem)


VC: 5_2:
Ensures Clause of Is_Empty: UnboundedQueue_List_Realiz.rb(24)

Goal:
(Q.Prec  o Q.Rem) = (Q.Prec  o Q.Rem)

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
```

```
3: (0  < max_int)
4: Conc.Q = (Q.Prec  o Q.Rem)



Free Variables:
Max_Char_Str_Len:N, min_int:Z, max_int:Z, Last_Char_Num:N,
Conc.Q:Str(Entry), Q:List_Fac.List, Q':List_Fac.List



VC: 6_1:
Ensures Clause of Clear: UnboundedQueue_List_Realiz.rb(28)

Goal:
true

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)


VC: 6_2:
Ensures Clause of Clear: UnboundedQueue_List_Realiz.rb(28)

Goal:
empty_string = empty_string

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)


VC: 6_3:
Ensures Clause of Clear: UnboundedQueue_List_Realiz.rb(28)

Goal:
empty_string = empty_string

Given:
1: (Last_Char_Num  > 0)
2: (min_int  <= 0)
3: (0  < max_int)
```

Auxiliary Code Rotate VCs

VC Number	With Auxiliary Code	Without Auxiliary Code
1 **Requires Clause of Dequeue in Procedure Rotate:** **Obvious_Rotate_Realiz.rb**	Goal: |Q| /= 0 Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0 • Proved in 453 milliseconds. • Overall, 1 proofs were directly considered and 0 useful backtracks were performed.	Goal: |Q| /= 0 Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0

2 **Requires Clause of Enqueue in Procedure Rotate:** **Obvious_Rotate_Realiz.rb(17)**	Goal: (|Q'| < Max_Length) Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0 8: Q = (<TE'> o Q') • Proved in 171 milliseconds. • Overall, 1 proofs were directly considered and 0 useful backtracks were performed.	Goal: (|Q''| < Max_Length) Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0 8: Q = (<TE'> o Q'')

3 **Ensures Clause of Rotate: Rotating_Capability.en(5)**	Goal: Q = (<TE'> o Q') Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0 8: Q = (<TE'> o Q') • Proved in 63 milliseconds. • Overall, 1 proofs were directly considered and 0 useful backtracks were performed.	Goal: there exists E:Entry, there exists R:Str(Entry), (Q = (<E> o R) and (Q'' o <TE'>) = (R o <E>)) Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0 8: Q = (<TE'> o Q'') 9: Q' = (Q'' o <TE'>)

4 **Ensures Clause of Rotate:** **Rotating_Capability.en(5)**	Goal: (Q' ○ <TE'>) = (Q' ○ <TE'>) Given: 1: (min_int <= 0) 2: (0 < max_int) 3: (Last_Char_Num > 0) 4: (Max_Length > 0) 5: (min_int <= Max_Length) and (Max_Length <= max_int) 6: (|Q| <= Max_Length) 7: |Q| /= 0 8: Q = (<TE'> ○ Q') • Proved in 62 milliseconds. • Overall, 1 proofs were directly considered and 0 useful backtracks were performed.

Table 4: VC Comparison when Using Auxiliary Code

Appendix G

Recursive Educational Example Verification Conditions

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Recursive_Append_Realiz.rb
// on:         Sat Jul 02 08:19:38 EDT 2011
//

Free Variables:
Max_Length:Z, min_int:Z, max_int:Z,
P:Modified_String_Theory.Str(Entry),
Q:Modified_String_Theory.Str(Entry), P_val:N,
P':Modified_String_Theory.Str(Entry),
Q':Modified_String_Theory.Str(Entry), E':Entry, E:Entry



VC: 0_1:
Requires Clause of Dequeue in Procedure Append:
Recursive_Append_Realiz.rb(9)

Goal:
|Q| /= 0

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0


VC: 0_2:
Requires Clause of Enqueue in Procedure Append:
Recursive_Append_Realiz.rb(10)

Goal:
(|P| < Max_Length)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
```

```
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
12: Q = (<E'> o Q')
```

```
VC: 0_3:
Show Termination of Recursive Call:
Recursive_Append_Realiz.rb(5)

Goal:
(|Q'| < |Q|)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
12: Q = (<E'> o Q')
```

```
VC: 0_4:
Requires Clause of Append in Procedure Append:
Recursive_Append_Realiz.rb(11)

Goal:
(|(P o <E'>)| + |Q'|)<= Max_Length)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
12: Q = (<E'> o Q')
```

```
VC: 0_5:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is true: Append_Capability.en(4)

Goal:
(P o <E'>)o Q') = (P o Q)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
12: Q = (<E'> o Q')


VC: 0_6:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is true: Append_Capability.en(4)

Goal:
empty_string = empty_string

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0
12: Q = (<E'> o Q')



Free Variables:
Max_Length:Z, min_int:Z, max_int:Z,
P:Modified_String_Theory.Str(Entry),
Q:Modified_String_Theory.Str(Entry), P_val:N, E:Entry
```

```
VC: 1_1:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is false:
Append_Capability.en(4)

Goal:
P = (P o Q)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| = 0


VC: 1_2:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is false:
Append_Capability.en(4)

Goal:
Q = empty_string

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| = 0
```

VCs for Recursive Append with wrong decreasing clause:

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Recursive_Append_Realiz.rb
// on:         Sat Jul 09 08:20:12 EDT 2011
//

Free Variables:
Max_Length:Z, min_int:Z, max_int:Z,
P:Modified_String_Theory.Str(Entry),
Q:Modified_String_Theory.Str(Entry), P_val:N,
P':Modified_String_Theory.Str(Entry),
Q':Modified_String_Theory.Str(Entry), E':Entry, E:Entry



VC: 0_1:
Requires Clause of Dequeue in Procedure Append:
Recursive_Append_Realiz.rb(9)

Goal:
|Q| /= 0

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0


VC: 0_2:
Requires Clause of Enqueue in Procedure Append:
Recursive_Append_Realiz.rb(10)

Goal:
(|P| < Max_Length)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
```

```
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0
12: Q = (<E'> o Q')


VC: 0_3:
Show Termination of Recursive Call:
Recursive_Append_Realiz.rb(5)

Goal:
(|(P o <E'>)| < |P|)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0
12: Q = (<E'> o Q')


VC: 0_4:
Requires Clause of Append in Procedure Append:
Recursive_Append_Realiz.rb(11)

Goal:
(|(P o <E'>)| + |Q'|)<= Max_Length)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0
12: Q = (<E'> o Q')


VC: 0_5:
```

```
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is true: Append_Capability.en(4)

Goal:
(P o <E'>)o Q') = (P o Q)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0
12: Q = (<E'> o Q')


VC: 0_6:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is true: Append_Capability.en(4)

Goal:
empty_string = empty_string

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| /= 0
12: Q = (<E'> o Q')



Free Variables:
Max_Length:Z, min_int:Z, max_int:Z,
P:Modified_String_Theory.Str(Entry),
Q:Modified_String_Theory.Str(Entry), P_val:N, E:Entry



VC: 1_1:
```

```
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is false:
Append_Capability.en(4)

Goal:
P = (P o Q)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| = 0


VC: 1_2:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(8) is false:
Append_Capability.en(4)

Goal:
Q = empty_string

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |P|
11: |Q| = 0
```

VCs for Recursive Append with incorrect implementation :

```
//
// Generated by the RESOLVE Verifier, December 2011 version
// from file:  Recursive_Append_Realiz.rb
// on:         Sat Jul 09 08:50:00 EDT 2011
//

Free Variables:
Max_Length:Z, min_int:Z, max_int:Z,
P:Modified_String_Theory.Str(Entry),
Q:Modified_String_Theory.Str(Entry), P_val:N,
P':Modified_String_Theory.Str(Entry),
Q':Modified_String_Theory.Str(Entry)



VC: 0_1:
Show Termination of Recursive Call:
Recursive_Append_Realiz.rb(5)

Goal:
(|Q| < |Q|)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0


VC: 0_2:
Requires Clause of Append in Procedure Append:
Recursive_Append_Realiz.rb(8)

Goal:
(|P| + |Q|)<= Max_Length)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
```

```
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0


VC: 0_3:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(7) is true: Append_Capability.en(4)

Goal:
(P o Q) = (P o Q)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0


VC: 0_4:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(7) is true: Append_Capability.en(4)

Goal:
empty_string = empty_string

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| /= 0



Free Variables:
```

```
Max_Length:Z, min_int:Z, max_int:Z,
P:Modified_String_Theory.Str(Entry),
Q:Modified_String_Theory.Str(Entry), P_val:N



VC: 1_1:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(7) is false:
Append_Capability.en(4)

Goal:
P = (P o Q)

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| = 0


VC: 1_2:
Ensures Clause of Append , If "if" condition at
Recursive_Append_Realiz.rb(7) is false:
Append_Capability.en(4)

Goal:
Q = empty_string

Given:
1: (min_int <= 0)
2: (0 < max_int)
3: (min_int <= 0)
4: (0 < max_int)
5: (Max_Length > 0)
6: (min_int <= Max_Length) and (Max_Length <= max_int)
7: (|Q| <= Max_Length)
8: (|P| <= Max_Length)
9: (|P| + |Q|)<= Max_Length)
10: P_val = |Q|
11: |Q| = 0
```

REFERENCES

[1] J. C. King, "A Program Verifier," Carnegie-Mellon University, USA, PhD Thesis 1969.

[2] J. C. King, "Symbolic Execution and Program Testing.," *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[3] C.A. Hoare, "An Axiomatic basis for computer proramming," *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 1969.

[4] Tony Hoare, "The verifying compiler: A grand challenge for computing research," *Journal of the ACM*, vol. 50, no. 1, pp. 63-69, January 2003.

[5] J. Bloch. (2006) [Online]. http://googleresearch.blogspot.com/2006/06/extra-extra-readall-about-it-nearly.html

[6] (2011, Ocbober) VSTTE ’10 competition. [Online]. http://www.macs.hw.ac.uk/vstte10/Competition.html

[7] V. Klebanov et al., "The 1st Verified Software Competition: Experience Report.," in *FM 2011: Formal Methods - 17th International Symposium on Formal Methods*, vol. 6664, Limerick, Ireland, June 20-24, 2011, pp. 154-168.

[8] K.R.M Leino, G. Nelson, and J.B. Saxe, "ESC/Java User's Manual," *Technical Note*, no. 002, 2000.

[9] B. Hackett and R. Rugina, "Region-Based Shape Analysis with Tracked Locations," in *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '05)*, Long Beach, CA, January, 2005.

[10] J. Krone, "The Role of Verification in Software Reusability," The Ohio State University, PhD. Thesis 1988.

[11] W. Heym, "Computer Program Verification: Improvements for Human Reasoning," The Ohio State University, PhD. Thesis 1995.

[12] B W Weide et al., "Incremental Benchmarks for Software Verification Tools and Techniques," in *VSTTE 2008 (Verified Software: Theories, Tools, and Experiments)*, Toronto, Canada, 2008, pp. 84-98.

[13] G.T. Leavens et al., "Roadmap for Enhanced Languages and Methods to Aid Verification," in *Proceedings of the 5th international Conference on Generative Programming and Component Engineering*, Portland, Oregon, USA, October 22 - 26, 2006, pp. 221-236.

[14] H. Harton, J. Krone, and M. Sitaraman, "Formal Program Verification," *Wiley Encyclopedia of Electrical and Electronics Engineering, Software Engineering Volume*, 2007.

[15] J. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification," in *19th International Conference on Computer Aided Verification*, Berlin, Germany, July 2007, pp. 173-177.

[16] C. Marché, C. Paulin-Mohring, and X. Urbain, "The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML," *Journal of Logic and Algebraid Programming*, pp. 89-106, 2004.

[17] D. Detlefs, G. Nelson, and J.B. Saxe, "Simplify: A Theorem Prover for Program Checking," *J. ACM*, vol. 52, no. 3, pp. 365-473, 2005.

[18] S. Ranise and D. Deharbe. The haRVey decision procedure. [Online]. http://www.loria.fr/~ranise/haRVey/

[19] C. March'e and C. Paulin-Mohring, "Reasoning about Java Programs with Aliasing and Frame," in *18th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Compute rScience*, vol. 3603, 2005, pp. 179-194.

[20] C.B. Jones, *Systematic Software Development using VDM*, 2nd ed.: Prentice Hall International, 1990.

[21] A.A. Koptelov and A.K. Petrenko, "VDM vs. Programming Language Extensions or their Integration," in *Proceedings of the First International Overture Workshop*, Newcastle, July 2005.

[22] S. Owre, J.M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, June 15-18, 1992, pp. 748-752.

[23] S. Owre, J. Rusby, N. Shanka, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107-125, February 1995.

[24] W.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*.: The MIT Press, January 2000.

[25] W. Visser, K. Havelund, G. Brat, and S. Park, "Model Checking Programs," in *IEEE International Conference on Automated Software Engineering*, September 2000, pp. 3-12.

[26] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani, "Automatic Predicate Abstraction of C Programs," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, Utah, United States, 2001, pp. 203-213.

[27] The Isabelle Theorem Proving Environment. Developed by Larry Paulson at Cambridge University and Tobias Nipkow at TU Munich. [Online]. http://www.cl.cam.ac.uk/Research/HVG/Isabelle

[28] D. Leinenbach, W. Paul, and E. Petrova, "Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness," in *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*, Sept. 2005, pp. 2-11, 7-9.

[29] P. Müller and A. Poetzsch-Heffter, "Modular Specification and Verification Techniques for Object-Oriented Software Components," in *Foundations of Component-Based System*, G.T., Sitaraman, M. Leavens, Ed. New York, NY, USA: Cambridge University Press, 2000, pp. 137-159.

[30] G.T. Leavens, A.L. Baker, and C. Ruby, "Preliminary Design of JML: A Behavioral Interface Specification Language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1-38, March 2006.

[31] L. Burdy et al., "An Overview of JML Tools and Applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212-232, June 2005.

[32] E. Poll, J. Kiniry, and D. Cok. Introduction to JML. [Online]. http://secure.ucd.ie/products/open-source/ESCJava2/ESCTools/papers/CASSIS2004.pdf

[33] L. Burdy, A. Requet, and J. Lanet, "Java applet correctness: A developer-oriented approach," *FME 2003, LNCS*, vol. 2805, pp. 422-439, 2003.

[34] W. Ahrendt et al., "The KeY tool.," *Software and System Modeling 4*, pp. 32-54, 2005.

[35] J.V. Guttag and J.J. Horning, *Larch: Languages and Tools for Formal Specification*.: Springer-Verlag New York, Inc., 1993.

[36] K. Kuncak and M. Rinard, "An Overview of the Jahob Analysis System: Project Goals and Current Status," in *Proceedings 20th IEEE International Parallel & Distributed Processing*, Rhodes Island, Greece, 2006, p. 285.

[37] T. Weis, Symbolic Shape Analysis. Master's thesis, 2004.

[38] T. Weis, V. Kuncak, P. Lam, A. Podeleski, and M. Rinard, "Field Constraint Analysis," in *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[39] Murali Sitaraman and Bruce W Weide, "Component-based software using RESOLVE," *Software Engineering Notes*, vol. 19, pp. 21-67, 1994.

[40] G. Kulczycki, M. Sitaraman, W.F. Ogden, and B.W. Weide, "Clean Semantics for Calls with Repeated Arguments," Department of Computer Science, Clemson University, Clemson, SC, Technical Report RSRG-05-01 March 2005.

[41] M. Sitaraman et al., "Reasoning about Software-Component Behavior," in *Proceedings of the 6th International Conference on Software Reuse*, 2000, pp. 266-283.

[42] J Kirschenbaum et al., "Verifying component-based software: deep mathematics or simple bookkeeping?," in *Formal Foundations of Reuse and Domain Engineering (Proc. 11th Intl. Conf. on Software Reuse),* 2009, pp. 31-40.

[43] J He, C. A. R. Hoare, and J. W. Sanders, "Data Refinement Refined," in *Proceedings of the European Symposium on Programming*, London, UK, 1986, pp. 187-196.

[44] M Sitaraman, B. W. Weide, and W. F. Ogden, "On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations," *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 157-170, March 1977.

[45] G. Kulczycki, "Direct Reasoning," Clemson University, Clemson, SC, Ph. D. Dissertation 2004.

[46] C. Cook, H. Harton, H. Smith, and M. Sitaraman, "Modular Verification of Generic Compoents Using a Web-Integrated Environment," Clemson University, Clemson, SC, Technical Report RSRG-11-03, School of Computing September 2011.

[47] G Kulczycki et al., "The Location Linking Concept: A Basis for Verification of Code Using Pointers," in *Proceedings of VSTTE 2012 (Verified Software: Theories, Tools, and Experiments)*, Philadelphia, USA, 2012.

[48] C. Cook, "A Web-Integrated Environment for Component-Based Software Reasoning," Clemson University, 2011.

[49] H. Harton, "Use of Unprovable Verification Conditions for Debugging," in *Proceedings of the RESOLVE Workshop 2007*, pp. 17-19.

[50] H. Keown, "Automation of Verification Condition Generation for a Verifying Compiler," in *Proceedings of the RESOLVE Workshop 2006*, 2006, pp. 14-18.

[51] G. Kulczycki, M. Sitaraman, H. Keown, and B. Weide, "Abstracting Pointers for a Verifying Compiler," in *Proceedings 31st Annual Software Engineering Workshop*, Baltimore, MD, March, 2007, pp. 204–213.

[52] G. Kulczycki, M. Sitaraman, B. W. Weide, and A Rountev, "A Specification-Based Approach to Reasoning about Pointers," *ACM Software Engineering Notes*, vol. 31, no. 2, pp. 55-62, September, 2005.