SPECIFICATION AND MECHANICAL VERIFICATION OF PERFORMANCE PROFILES OF SOFTWARE COMPONENTS

A Thesis

presented in partial fulfillment of requirements

for the degree of Doctor of Philosophy

in the Department of Engineering Science-emphasis Computer Science

The University of Mississippi

By

NIGHAT YASMIN

May 2015

ABSTRACT

Software performance predictability is vital to a system design and unpredictable performance is a leading cause of software failure. The emphasis of this dissertation is on verification that component-based software performs as specified. Performance profiles (specifications) depend on functional specifications and are necessary for all components for modular verification. Modular verification process is scalable because it uses profiles as contracts and allows verification of a single component in isolation with the assumption that any underlying component would have already been verified or will be verified to meet its specifications independently.

This dissertation presents an integration of performance specification (profiles) with functional specifications within a single language. It contains a mechanizable and modular proof system to verify the performance bounds of reusable software components built reusing other components. The proof system forms the basis for a prototype verification condition (VC) generator. Experimentation with the VC generator illustrates that software component performance can be formally specified and verified. This dissertation discusses only duration (timing) aspect of performance, but the results can be extended to include space constraints.

DEDICATION

To my parents Wajid Husain and Shamim Akhtar, my husband Abdul Aziz Khan, my children Nayel, Haaris, and Kirin and my siblings for being there for me, and believing I could finish. I could not have done it without all of you!!!

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor, Dr. C. Cunningham because without his support I would never have been able to finish this work.

I would like to express special thanks to Dr. M. Sitaraman form Clemson University, SC, who suggested this topic and supervised the research and the writing of the dissertation. His guidance and encouragement are greatly appreciated. I would like to thank Dr. W. F. Ogden and Dr. J. Krone in working with each iteration of the proof rules and members of RESOLVE Software Research Group (RSRG) for their valuable comments and suggestions. Also, the technical support provided by Dr. H. K. Harton, Dr. H. Smith, and Yu-Shan Sun is gratefully acknowledged.

I would also like to thank my dissertation committee, Dr. T. Ammeter, Dr. Y. Chen, and Dr. Wilkins for providing me with valuable feedback and for serving on my committee.

This research was supported in part through NSF research grant CCF-1161916. This support is gratefully acknowledged.

On a personal note, I would like to acknowledge my husband Abdul Khan and my children Nayel, Haaris, and Kirin for their continuing support, without which this dissertation would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Meaning
#	Incoming value
#E	Incoming value of a variable E
< E >	A string containing entry E
:=	Assignment
:=:	Swap operator
Λ	Empty string
$\|\alpha\|$	Length of string α
o	Concatenation
Π	Mathematical definition of concatenation
$\mathscr{C}$	Context
x ⇝ y	Replace every x with y
\	Separator between the context and the assertive code

RESOLVE KEYWORDS AND COMMON TERMS

Note: In the following table, the keywords are shown in boldface.

Keyword	Meaning
alters	A parameter mode to indicate its value is unspecified at the end of an operation
changing	List of variables that are affected during the loop execution
clears	A parameter mode to indicate the parameter is initialized
Cnts_Dur	Sum of the durations to finalize every entry
Concept	Functional specification of a component
decreasing	Termination progress metric clause for loops and recursion
defines	Definitions are deferred
definitions	Actual definition
DeString	The entry in a string containing a single entry
Dur_Call(n)	Duration overheads for an operation call with n parameters
elapsed_time	an invariant for a loop that specifies how much time has elapsed since the beginning of the loop
Enhancement	An extension to a Concept
ensures	Post-condition to an operation
Entry	A generic type

F_Dur	The duration to finalize any Entry
F_IV_Dur	The duration to finalize an initial valued Entry
finalization	Destructor
I_Dur	The duration to create an initial valued Entry
initialization	Constructor
maintaining	A loop invariant assertion about loop variables
Operation	Keyword to indicate specification
preserves	A parameter mode to indicate it will not be touched
Procedure	Keyword to indicate implementation
Prt_Btwn	A math definition to specify the substring between two given indices
Profile	Performance specification of a concept or enhancement
Realization	Implementation
replaces	Under this parameter mode, the incoming value of a parameter is of no importance
requires	Pre-condition to an operation
restores	This parameter mode signifies that incoming and outgoing values are same
short_for	Used in profile header
Sqnt_Dur_Exp	Subsequent duration expression
Str	A string structure that is simpler than finite sequences (because

	no positions are involved)
updates	A parameter mode to indicate that a parameter is changed as specified
VC	Verification condition
with_profile	A keyword added to a realization header

LIST OF ABBREVIATIONS

Abbreviation	Meaning
CBSE	Component Based Software Engineering
CPI	Cycles per instruction
DES	Design and evaluation system
HRT	Hard Real Time systems/application
IC	Instruction count
JOP	Java optimized processor
LHS	Left hand side
LQN	Layered queuing network
PASD	Performance aware software development
RT	Real-time systems/application
SPE	Software performance engineering
SRT	Soft real-time systems/application

CHAPTER 1.

INTRODUCTION

1.1. Introduction

Software performance predictability is vital to system design. Software performance prediction can be defined as an estimate of the resources used during software execution [Cormen90, Woodside02]. In order to predict the performance of a software system, it is important to understand the bottlenecks of the system, the memory usage and the response time required to execute its building blocks.

Unpredictable performance is a leading cause of software failure [Smith02]. In one instance, Smith notes that after spending $20 million, a project was abandoned because it could not be tuned to satisfy the performance goals. In another example, NASA delayed the launch of a satellite for at least eight months because the software and hardware had unacceptable execution times, among other problems.

The goal of this dissertation is to improve software performance (execution time) predictability techniques by integrating functionality and performance specifications, developing and modifying proof rules for verification conditions and automatically generating verification conditions.

1.2. Current State of Software Performance Prediction

More than half a century ago, with very limited memory computers, performance was a decisive factor in software design. With present-day multiprocessor computers, the software performance predictability problem has received little attention. It is assumed that the hardware has become so cheap and fast that execution time should be of no concern. It is also assumed that the design and management of a performance model is very complicated and that software should be partially complete before its performance can be predicted. Additionally, there is also the concern that attention to performance prediction might also cause delivery to be delayed. So there are few formal or well-understood techniques for software performance prediction.

Software performance can be predicted through measurement and/or analytical analysis. Most of the existing research is either based on static analysis of loose bounds [Bertolino03, Bertolino04] or a collection of run-time experimental and measurement data [Denning81, Konkin98, Yacoub02, Wu03, Wu04, Schupp04]. In static analysis, the performance of software is predicted by examining the source, byte, or binary code [Heckmann04, Schoeberl10] without actually executing it. In this technique, software developers must know the duration bounds to execute every statement and predict the overall performance of software by combining the duration bounds of the included statements. In measurement-based analysis, the data are collected for workloads and corresponding execution paths, execution platform, resource usage, and process overhead, using external or internal measurements [Konkin98]. Anderson predicts the performance of a system by simulation and measurement [Anderson84], whereas Siddiqui uses a performance-aware software development technique to find the performance profile of a software system [Siddiqui02].

Generally, software design and performance prediction methods are developed independently [Williams95]. However, in order to improve the software development process and explore various design options, it is important to analyze performance during the development process. This can be made possible if functional behavior and performance prediction methods are integrated into the software design. That is, software specification and analysis should be capable of representing resource usage, such as for time and memory.

1.3. Component-Based Software

Modern software development processes use pre-existing software components to achieve the desired results, so the focus of this dissertation is on prediction of performance for component-based software. In component-based software engineering (CBSE), large and complex software systems can be developed using independently-developed generic and reusable components that have been previously verified to be correct. A software component, the basic unit of reuse and deployment, is designed as a black box. That is, it has a visible interface and a hidden implementation. The focal point of a component design is the information hiding principle [ParnasD72] and reuse.

The interface of a component may have multiple implementations and might possibly use other components. Fig 1.1 shows a typical component-based system. In the figure, interfaces are shown as circles and implementations as rectangles. A green dashed arrow terminating at the interface indicates that the implementation satisfies that interface. The uses relationship is shown by a purple solid arrow starting from an implementation and terminating at the functional specification. The figure shows the interfaces of functional specification for three components: The interface that is being implemented and the interfaces that are used.

Component-based software development has several advantages. Its development time and cost can be minimized because components can be reused and components can be developed at the same time. It is flexible because a component can be replaced by another component of the same functionality, possibly with a different performance behavior.

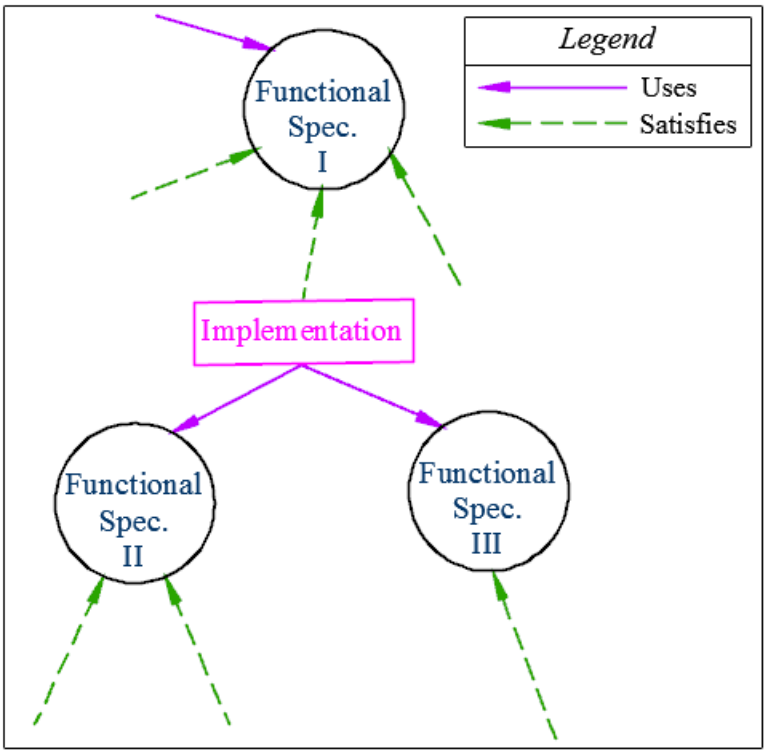

Fig 1.1: A typical component-based system

1.3.1. Specification

A specification (an interface with a description of behavior) is a contract between clients and providers to specify the properties and capabilities of a system [Clarke96, Szyperski98]. The specification must provide all the information that is necessary to use a component properly, both to the implementers and the users [ParnasN72].

Specification may be informal or formal. Informal specification techniques attempt to specify the system using natural language, thereby introducing several problems such as ambiguities, noise, and over-specification [Meyer85]. These problems can be eliminated using formal specification. Formal specifications use mathematically based languages, techniques, and tools for specifying a software system [Clarke96, Hatcliff12].

A functional specification of a component describes its behavior, whereas, a performance specification is a non-behavioral specification of a component. It is a performance contract between users and implementation(s) of the component. It represents the expected performance behavior of a (class of) component's implementation(s) without disclosing the internal details of the implementation(s) [Krone06].

1.3.2. Compositional analysis and scalability

Component-based software systems must be designed to support modular analysis. In this analysis, it is possible to establish properties of the target system formally without a need to inspect the internal details of any of its constituent components [Hooman91, Stata95, Szyperski98, Kiczales05, Land09].

Scalability is achieved from the ability to perform modular analysis. The term "scalable" refers to a system or process that is "capable of being easily expanded or upgraded on demand" [Merriam-Webster dictionary]. In software engineering, scalability of a system is its capacity to either handle larger inputs or its ability to expand easily to a bigger system. In this dissertation, software is designed to be scalable (and supports modular analysis) if its reasoning is localized irrespective of the size of the input data and the number and size of its constituent components [SitaramanM01].

To better illustrate the scalability issue, consider Fig 1.2. The figure shows only one implementation (*C0_IMP_1*) of the target system (*C0*). The three subsystems *C1*, *C2*, and *C3* have multiple implementations. For example, *C0* may be implemented by *C0_IMP_1*; and *C1_IMP_2* may be one of the implementations of *C1*. *C0_IMP_1* uses specifications *C1* and *C2*; and *C1_IMP_2* uses *C3*.

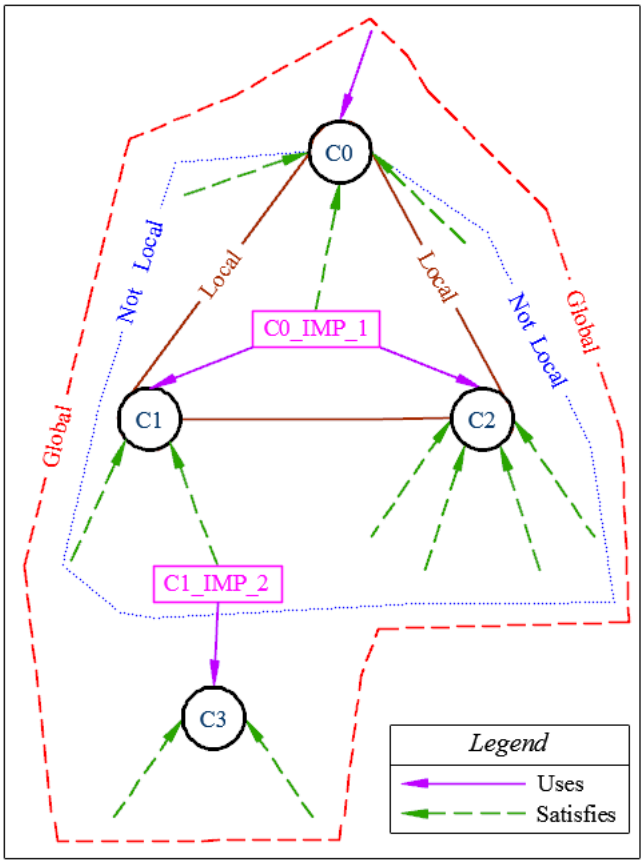

Fig 1.2: Scalability and localized reasoning

Software reasoning can become tractable only by using localized reasoning. In Fig 1.2, the loop labeled as *Local* represents localized reasoning [Kiczales05]. That is, it is possible to formally reason about an implementation (*C0_IMP_1*) of a component using only its functional specification (*C0*) and the functional specification of the components being used (*C1* and *C2*).

On the other hand, for the loop labeled as *Not Local*, knowledge of additional implementations is necessary. Finally, the loop labeled as *Global* represents an exhaustive approach. In this case, the reasoning about an implementation of a component is not possible without reasoning about the implementation of the entire subsystem. That is, to formally reason about *C0_IMP_1* it may be necessary to reason about *C0*, *C1*, *C2*, and *C3*, and their implementations. The objective of this research is to enable local reasoning of performance.

1.4. An Integrated Language

In this research, for the purpose of representation, RESOLVE, an integrated specification and programming language is used [Sitaraman94, Sitaraman11]. In RESOLVE, a functionality specification of a component is termed a concept and its implementation as a realization. A realization may claim to satisfy a concept. Fig 1.3 represents the components shown in Fig 1.1 using concept and realization naming.

1.5. The Need for Performance Profiles

Generally, multiple clients of a component may require different performance criteria. So the performance behavior of a component cannot be categorized as one size fits all, that is, it may not satisfy the requirement of every client. Hence, the same functionality specification can be implemented with multiple realizations with different performance behaviors [SitaramanM01]. In RESOLVE, a performance specification of a component is termed as a profile. A performance profile of a component, in general, depends on the functional specification. A realization may claim to satisfy a corresponding performance profile. Multiple realizations may satisfy the same profile and the same implementation may have multiple profiles. Whenever there are

multiple components providing different performance trade-offs for the same functional specification, then there must be a mechanism for a client to choose the best component(s) to match the performance requirements of their problem.

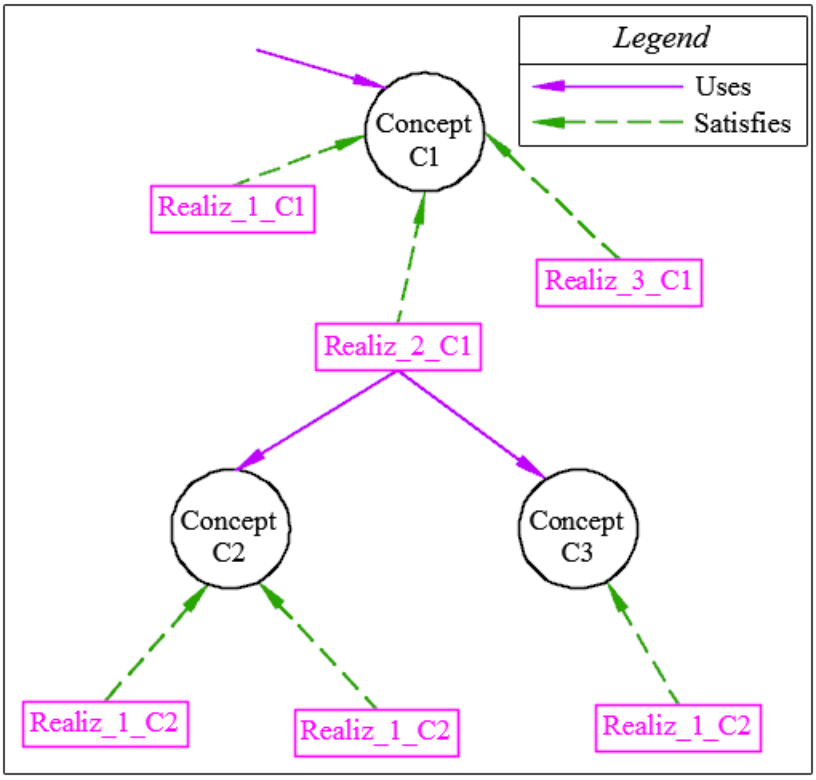

Fig 1.3: A component-based system with RESOLVE terminology

In Fig 1.4, performance profiles are added to the components shown in Fig 1.3. In the figure, an arrow starting from a realization and terminating at a profile represents that the realization satisfies the profile. A profile can be expressed only within the context of a concept and the figure shows these dependency relationships.

Performance profile specifications are necessary for localized, modular verification of performance of the software developed from components. If the performance profiles of software components are known then the performance profile of the target system can be analyzed. For

example in Fig 1.4, the profile, *Profile_2_C1*, for one of the realizations of the target system can be predicted using modular performance reasoning knowing the performance profiles of its constituent components (*Concept 2* and *Concept 3*). Alternatively, knowing the performance profile of the target system (*Profile_2_C1 for Concept 1)* will help in the selection of the appropriate realizations of lower-level components (for *Concept 2,* realization with *Profile_1_C2* or *Profile_2_C2*).

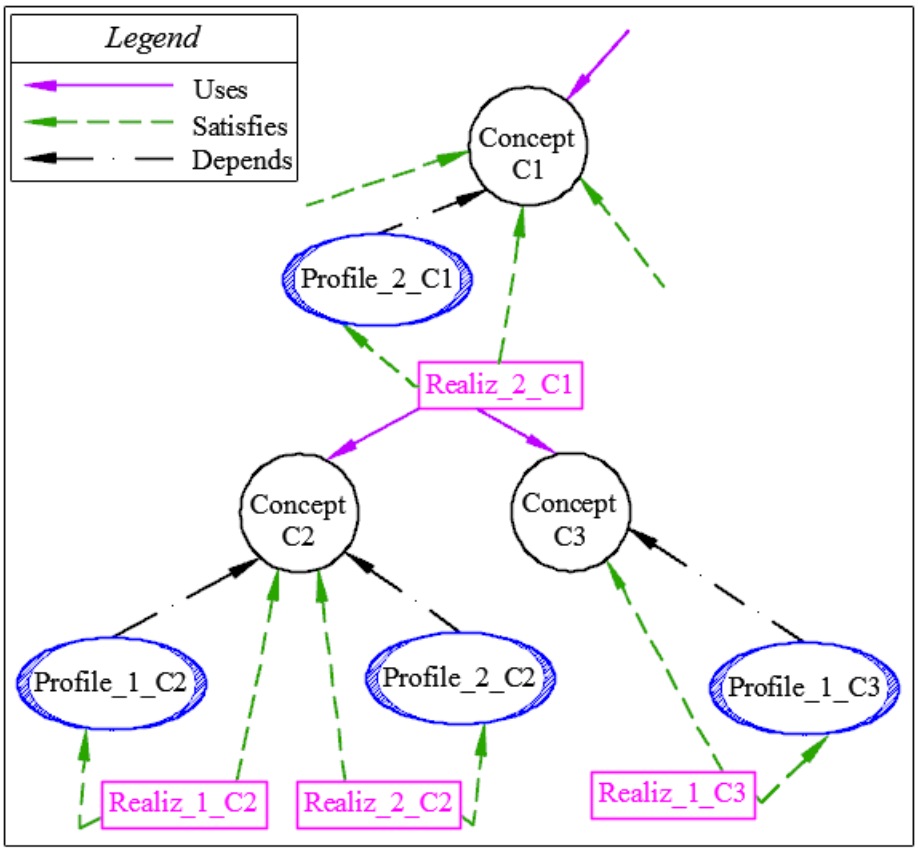

Fig 1.4: A component-based software system with performance profiles

1.5.1. Modular analysis and performance profiles

Typically, modular reasoning focuses on functional specification. The goal of this dissertation is to propose and support the thesis that modular performance reasoning should be an inte-

gral part of the software design and development process. This will enable a system's developer to:

- build software from components of known performance;
- predict the performance of components using modular reasoning; and
- repeat the process to scale up to larger component-based systems.

1.6. Verification in General

The importance of program verification can be traced back to 1947 [Dijkstra90, Giannakopoulou01, Ireland04]. Verification is important, because in the US, for example, every year, 20–60 billion dollars are lost due to avoidable software errors [Hoare05]. Verification is defined as the mathematical proof that the program satisfies its formal specifications [Hoare69, Giordano80, Jackson06]. A proof is a demonstration that one formal statement follows from another. In other words, verification is a process to determine whether a program carries out its intended function (described in its abstract specification) or not.

For example, consider a Hoare tuple ***{P} S {Q}*** where ***P*** and ***Q*** are the assertions before and after the statement ***S***. The assertions are defined as the logical formulas about participating variables. The triple is correct if we can show that if ***P*** were true before the execution of ***S*** then ***Q*** is true after the execution of ***S*** [Hoare69] (and nothing goes wrong during the execution of statements ***S***).

1.6.1. Mechanical verification

Program verification is a tedious process. Verification of a computer program can be done by hand or software can be developed to carry out the job. If code verification is performed by hand then the possibility of errors is higher, compared to a mechanized method.

The accuracy and efficiency of a verification process can be increased and maintenance processes can be simplified by using a mechanical verification system. A mechanical functional verification system is based on formal specifications, implementation annotated with suitable assertions, appropriate theorems from mathematics, and inference rules [Harton11]. A mechanical performance verification system also includes performance profiles.

1.7. Research Outline

In the field of performance prediction of software, most of the existing research is either based on the collection of run-time experimental data and measurement or loose static analysis of theoretical bounds. The challenge and novelty of the current research is in enabling specification and static analysis of bounds at various levels of precision. To ensure that the results are scalable, this research focuses on the modular performance analysis of components and component-based software.

1.7.1. Thesis objectives

This dissertation will defend the following thesis: *It is possible to develop an integrated language and automatically generate verification conditions for performance correctness of components and component-based software systems*. The present work focuses only on duration analysis of performance. The research roadmap has three milestones.

(i) Integration of performance specification mechanisms with functional specifications in a single language to specify performance profiles of components.

(ii) Development of a formal proof system to verify in a modular fashion the performance of reusable software components built reusing other components.

(iii) Design and development of a prototype mechanical performance verification condition (VC) generation system.

1.7.2. Approach

This research is based on RESOLVE language. RESOLVE is an imperative, specification-implementation language combine [Sitaraman11]. The ultimate goal of RESOLVE is to increase software quality and reduce its development cost. The language can be used to specify and implement component-based software systems. It also provides the capabilities to make modular reasoning possible. Currently, formal functional specifications and realizations are integrated into the RESOLVE language design. Functional verification is a work in progress. For details refer to Chapter 3.

1.7.3. Research deliverables

Based on the three milestones of the thesis, this research leads to three additions to the RESOLVE programming language and its supporting systems. All three additions are of significant importance because currently no language or system provides these facilities. The conditions are listed here.

(i) Extensions to the RESOLVE language: adding performance specification (specifically, time bounds) and performance verification capabilities. This is the topic of Chapter 3.

(ii) Mechanizable proof rules for performance VC generation of a component-based software system. This is the topic of Chapter 4.

(iii) A prototype of a mechanical performance VC generation system. Performance specification constructs developed in Chapter 3 and proof rules developed in Chapter 4 have been incorporated into RESOLVE compiler. Chapter 5 presents results from experimental evaluation of the prototype system.

The prototype system has been used to generate the VCs for a selected class of reusable components in a modular fashion. In order to verify a performance profile of a realization, Fig 1.5, the VCs generation process will use (i) the realization, (ii) the concept it is implementing, (iii) the concepts it is using, and (iv) the corresponding profiles.

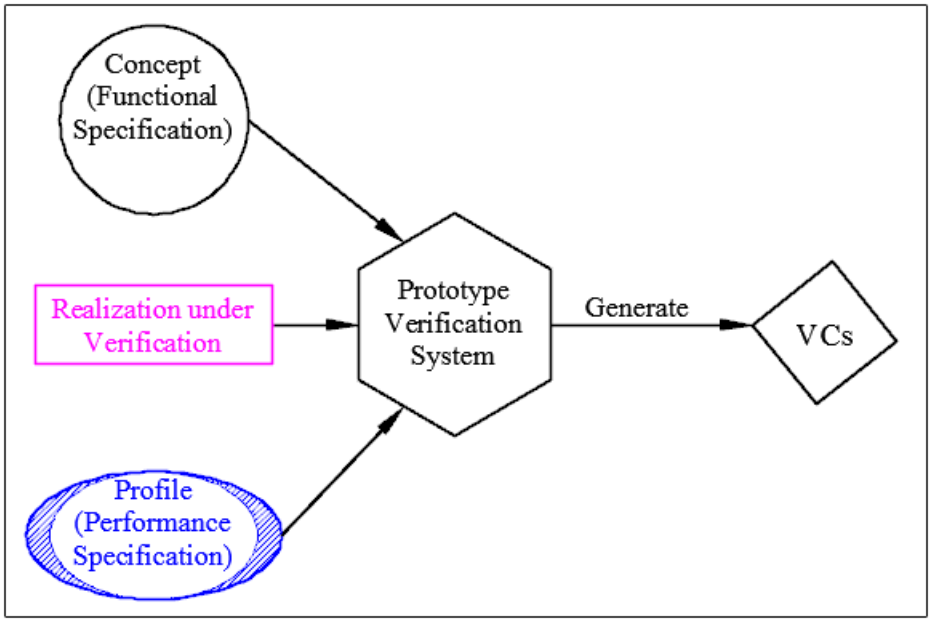

Fig 1.5: A prototype of mechanical performance VCs generation system

1.7.4. Prior and Current research

(i) The existing RESOLVE language and system offers functional specification, implementation, and mechanical functional verification capabilities. This research has extended the RESOLVE language by adding performance specification capabilities

(ii) The existing formal proof system for RESOLVE is intended for functional verification. Since functional verification is a prerequisite for performance verification, this research has led to a performance verification proof system of rules based on functional verification rules.

(iii) The existing RESOLVE implementation has an initial, mechanical VC generator for functional correctness for RESOLVE. This research has incorporated performance verification capabilities developed in steps (i) and (ii).

1.7.5. Experimental Evaluation

The products of the dissertation are evaluated through the use of a push-button interface of a prototype verification system. The interface provides options to choose a concept and enhancement and their associated implementations and performance profiles. The evaluation involves experimentation with specifications (functional and performance) and realizations for a class of generic software components. For a subset of these components, the research evaluates the complexity of specification with respect to the tightness of proven bounds and correctness of performance VCs, using of the prototype VC generator system as a principal method for evaluation.

1.8. Organization

The rest of this dissertation is organized into the following chapters. Chapter 2 provides a brief introduction of the RESOLVE language and its features for functionality specification and verification. Chapter 3 discusses performance specifications. Chapter 4 discusses mechanizable proof rules for modular performance analysis. Chapter 5 presents results from experimental evaluation. Chapter 6 provides a detailed description of related research. Chapter 7 presents future directions and conclusions.

CHAPTER 2.

FUNCTIONAL BEHAVIOR SPECIFICATION AND VERIFICATION

2.1. Introduction

This research is based on RESOLVE, a modular specification-implementation language combined [Sitaraman94]. The ultimate goal of RESOLVE is to increase software quality and reduce its development cost. The language can be used to specify and implement component-based software systems. It also enables modular reasoning. Prior to this research, formal functional specifications and realizations have been integrated into the RESOLVE language design and implementation. The goal of this research is to integrate performance specification features. This chapter discusses the basics of the RESOLVE language that are necessary to understand the functional and performance specification and verification.

RESOLVE includes a behavioral interface specification language (BISL) for functionality specification. There are several such languages in the literature [Harton11]; and in principle, any of them can be extended with the ideas presented in this dissertation for performance specification and verification.

2.2. Functionality Specification: Concepts

In RESOLVE terminology, a functional specification of a component is known as a **Concept**. A concept can be considered a contract of functionality between clients and developers. Fig 2.1 shows the functional specification of a bounded queue concept, named *Preemptable_Queue_Template*. The concept is parameterized by *Entry*, a generic type and *Max_Length*, a bound indicating the maximum number of elements allowed in a queue. In order to use *Preemptable_Queue_Template* in Fig 2.1, a client must create its instance by identifying *Entry* type (for example, integers, trees, etc.) and a functional expression for *Max_Length* that will be evaluated. The **requires** clause specifies that the *Max_Length* must be a non-zero positive integer.

Following the modularity criteria, the mathematical theories that are employed are kept independent and are imported using the **uses** clause [Sitaraman11, Smith08]. *Preemptable_Queue_Template* uses the *String_Theory*; and *String_Theory* is a math unit that precisely defines string theoretic terms. In RESOLVE, strings are assumed to be finite sequences. In the specification, Λ denotes the empty string and $|S|$ denotes the length of a string S. The notation $\alpha \, o \, \beta$ represents the concatenation of two strings α and β (that is, β is appended to α). The notation $< >$ represents the conversion of an entry to a string; hence, the notation $<E>$ represents the string containing the single entry E.

```
Concept Preemptable_Queue_Template(type Entry; evaluates Max_Length: Integer);
    uses String_Theory, Integer_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;
    end;

    Operation Enqueue(alters E: Entry; updates 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 Q = Prt_Btwn(0, |#Q| - 1, #Q) o <#E> and
                E = DeString(Prt_Btwn(|#Q| - 1, |#Q|, #Q));

    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;
```

Fig 2.1: A functional specification of the Preemptable_Queue_Template

The abstract data type (ADT) provided by this concept is called *P_Queue* and it is mathematically modeled as a string of entries. The **exemplar** clause is used to introduce the name of a sample variable; using an example *P_Queue* variable Q, the specification tells a client what is true about every *P_Queue* (preemptable queue) variable. The **constraint** clause indicates that the length of Q should never exceed *Max_Length*. The **ensures** clause of the **initialization** (constructor) specifies that every new object of *P_Queue* type is an empty string. For the better understanding and possibility of multiple implementations, a concept should be composed of minimum number of basic operations; Fig 2.1 shows seven basic or primary operations.

The entries are entered at the tail end of a queue, one at a time, through the call to the *Enqueue* operation. This operation takes two parameters: an entry *E*, and a *P_Queue* object *Q* with the parameter modifier mode of **alters** and **updates**, respectively. A parameter modifier mode indicates how, at the end of the operation, the variable is changed. These modes make specification readily comprehensible. The **alters** mode indicates that the operation may leave an arbitrary value in the parameter. The **updates** mode indicates that the parameter may be changed at the end of the operation. The **requires** clause shows that the length of incoming *Q* should be less than *Max_Length*. The **ensures** clause of the operation guarantees that, after the call to the operation, *Q* is updated by concatenating the incoming values of *Q* and *E,* denoted by #*Q* and #*E*, respectively. Fig 2.2 shows the effect of the *Enqueue* operation. In the figure, a queue is assumed to be a string of geometrical shapes. The left and right columns of the figure shows the input and output parameters to and from the *Enqueue* operation, respectively. Since, the input parameter *E* is passed through the **alters** mode, its outgoing value is unspecified as "?" shown in right column of Fig 2.2.

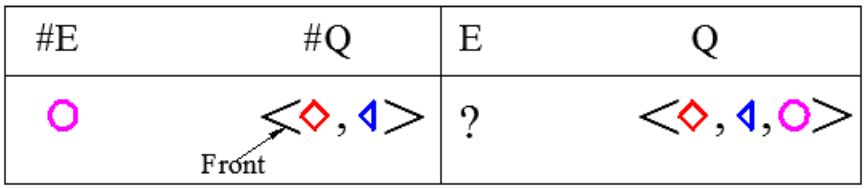

Fig 2.2: Effect of a call to the Enqueue operation

Traditionally, the entries are entered at the tail end of a queue. However, this discussion is based on a modified queue called as *preemptable queue*. Here, a client is allowed to preempt the queue order and insert an entry in front of the first entry in the queue through the call to the *Inject* operation, Fig 2.1. As a primary operation, this function can be implemented to be a constant time operation. Fig 2.3 shows the effect of a call to the *Inject* operation.

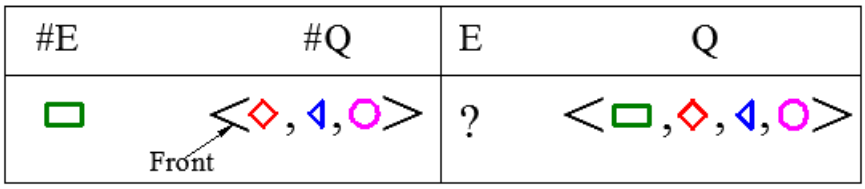

Fig 2.3: Effect of a call to the Inject operation

The entries from the front of a queue can be removed, one at a time, through calls to the *Dequeue* operation, Fig 2.1. This operation also takes two parameters: an entry *R*, and a *P_Queue* object *Q* with the parameter modifier mode of **replaces** and **updates**, respectively. The **replaces** mode for the input parameter *R*, indicates that the incoming value of *R* is of no importance to the client and it is replaced by the value specified in the **ensures** clause. The **requires** clause shows that *Q* must contain at least one entry. The **ensures** clause of the operation affirms that, after the call to the operation, *Q* is updated. Also, it is clear from the **ensures** clause that the value of the incoming *Q* is mathematically the same as concatenating the outgoing values of *R* and *Q*. Fig 2.4 shows the effect of a call to the *Dequeue* operation.

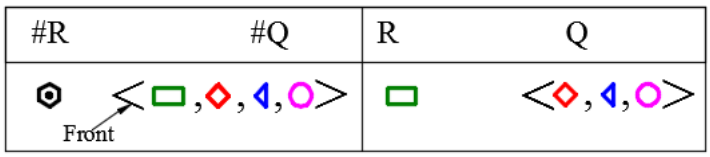

Fig 2.4: Effect of a call to the Dequeue operation

To look at the most recently enqueued entry efficiently, the concept includes a *Swap_Last_Entry* operation in Fig 2.1. As a primary operation, this function can be implemented to be a constant time operation. In the **ensures** clause, the terms *Prt_Btwn* and *DeString* comes from *String_Theory*. The term *Prt_Btwn* is a notation that is used to return the substring between two given indices. It takes starting and ending indices and a source string; and returns the sub-string between the two indices. This definition assumes that a string begins at index *0*. Fig 2.5 shows two substrings *Prt_Btwn(Prt_Btwn(0, |#Q|-1, #Q)* and *Prt_Btwn(|#Q|-1, |#Q|, |#Q|)*. Note that *0* is before the beginning and *|#Q|-1* is after the end of first substring. The term *DeString* has the opposite effect of < > operator, and it is the entry in a string containing a single entry. It is clear from the **ensures** clause that mathematically the value of the outgoing *Q* is the same as the incoming *Q* except using the incoming value of *E* as the last entry; and that the outgoing value of *E* is the last entry of the incoming *Q*. Fig 2.5 shows the effect of a call to the *Swap_Last_Entry* operation and the various terms used in the **ensures** clause.

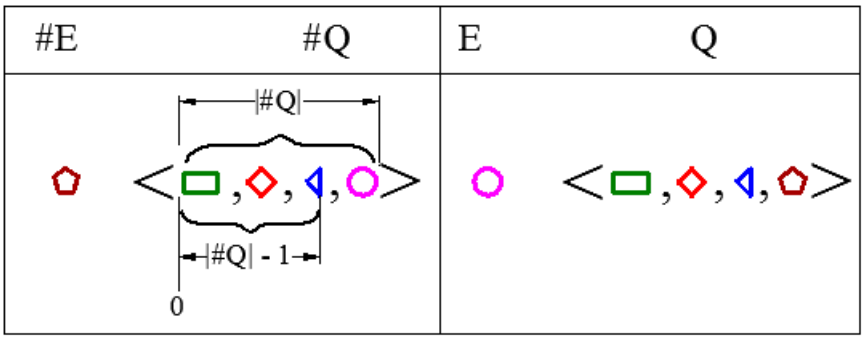

Fig 2.5: Effect of a call to the Swap_Last_Entry operation

The *Length* operation (Fig 2.1) returns the number of entries in a queue. The input parameter *Q* is in **restores** mode. The **restores** mode indicates that the parameter's value may change during the operation execution; however, it is restored to the incoming value at the end of the operation. (Not shown in the current example, RESOLVE also provides the **preserves** modifier mode to guarantees that the parameter will not be changed.) The *Rem_Capacity* operation (Fig 2.1) is used to count how many more entries can be entered. A queue object may be emptied (and reused) by calling the *Clear* operation (Fig 2.1). The input parameter *Q* is in **clears** mode and it indicates that the operation will initialize the parameter. The default parameter mode is **alters**, that is, if no mode is specified then **alters** mode is used.

2.3. Client Usage: Facilities

In RESOLVE terminology, unlike generic concepts and realizations that need to be instantiated, a facility is a module that is ready to be used. A facility may be constructed from scratch or by instantiating pre-existing concepts and realizations. This section contains examples of both. A facility starts with the keyword **Facility**. If FF is a facility constructed from scratch then FF is written as: ***Facility*** *FF*. However, if FF is a facility for a concept C and R_C is the realization of C then FF is written as: ***Facility*** *FF* ***is*** *C* ***realized by*** R_C.

Fig 2.6a shows the facility *Example_Fac*, a custom facility built from scratch that bundles necessary specifications and code. However, generally, facilities are the instantiations of the previously developed concepts and realizations. Here, *Preemptable_Queue_Template* is instantiated (*Integer_Queue_Fac*) for Integer type Entry and *Max_length* of 10. The instantiation supplies the appropriate parameters to the concept as well as picks a particular implementation of the concept. Implementations (or realizations) are discussed in Section 2.4. From the Queue facility declaration, users can declare variables of type *P_Queue* and use them as any other object. *Example_Fac* contains a local operation and a procedure to implement that operation. The keyword **Operation** is used to indicate its functional specification and **Procedure** is used to indicate the executable code (implementation). Operation *Append_to* takes two parameters of *P_Queue* type: *P* and *Q* with the parameter modifier modes of **updates** and **clears**, respectively. The operation **requires** that the sum of the lengths of *P* and *Q* should be less than 10 (*Max_Length*) and it ensures that the outgoing *P* is the concatenation of incoming *P* and *Q*.

The loop design assertions capture the loop design decisions. These assertions are necessary to make the manual reasoning easy and mechanical reasoning possible and are demanded by almost all automated verification systems [Klebanov11]. These assertions are provided by the implementer (procedure writer). The **maintaining** clause is an invariant assertion about loop variables; it must be true before the first iteration, before and after every iteration, and after the last iteration of the loop. Hence, the invariant is $P \ o \ Q = \#P \ o \ \#Q$ (Fig 2.6a). Since, elements are dequeued from *Q* and enqueued to *P*, therefore, the **maintaining** clause indicates that after each iteration of the loop, the concatenation of the current values of the *P* and *Q* is the same as concatenating the original values of *P* and *Q*. The **decreasing** clause is the progress metric; it is a natural number (an ordinal, in general) that must decrease after each iteration for the loop to ter-

minate. Hence, the invariant is based on |*Q*| (Fig 2.6a). Since, elements are dequeued from *Q*, therefore, its length is decreasing.

```
Facility Example_Fac;
        uses String_Theory;

        Facility Integer_Queue_Fac is
                        Preemptable_Queue_Template (Integer, 10)
                realized by Clean_Circular_Array_Realiz;

        Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
                requires |P| + |Q| <= 10;
                ensures P = #P o #Q;

        Procedure
                Var Next: Integer;
                Var Len: Integer;

                Len := Length(Q);
                While (Len /= 0)
                        maintaining (P o Q = #P o #Q) and Len = |Q|;
                        decreasing |Q|;
                do
                        Dequeue(Next, Q);
                        Enqueue(Next, P);
                        Decrement(Len);
                end;
        end Append_to;
end Example_Fac;
```

Fig 2.6a: Facility declaration using a concept

Fig 2.6b shows the relationship between a **Facility** (*Example_Fac*), a **Concept** (*Preemptable_Queue_Template*), and the **Concept**'s implementations. The figure shows that the **Concept** has two implementations named as *Clean_Circular_Array_Realiz* and

Clean_Linear_Array_Realiz; these realizations are discussed in detail in next section. In Fig 2.6b, the loop labeled as *Local* represents modular verification. That is, it is possible to formally reason about the *Example_Fac* facility using only the functional specification of the component being used (*Preemptable_Queue_Template*). Hence, the localized reasoning process is independent of the **Concept**'s realizations.

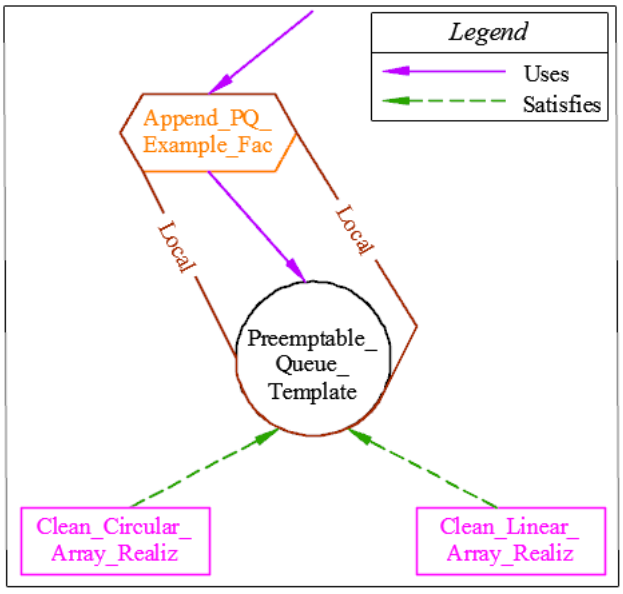

Fig 2.6b: A relationship between a Facility, Concept and the Concept's implementations

2.4. Implementations of Specifications: Realizations

In RESOLVE terminology, an implementation is called a **Realization**. It is used to describe the data structures and algorithms for how the functionality of a module is achieved, that is, how it is implemented. The realization of functionality is correct if it satisfies the corresponding concept. Thus, for a given concept, many different realizations would be possible; the conceptual view is shown in Fig 2.7a. This section discusses two realizations: A *Clean_Circular_Array_Realiz* realization using a "circular array" shown in Fig 2.7b; and a

Clean_Linear_Array_Realiz realization using a linear array shown in Fig 2.11. In both realizations, every entry in the array that is not part of the conceptual queue is restricted to being the initial value of the Entry; hence "Clean" in the name of the realizations.

A realization starts with the key word **Realization** (Fig 2.8a and Fig 2.12a). In these realization the keyword **for** (followed by the name of a concept) is used to indicate that the code is the implementation for the *Preemptable_Queue_Template* concept. A clean version of the realization maintains initial values for the part of the array that is not a part of the conceptual queue, using a predicate *Array_is_Clean*.

Realizations are annotated with assertions to capture the design decisions, and they are used by the verifier to generate VCs. Two basic assertions concern type representation: *conventions* and *correspondence*. The *conventions* clause, also known as *representation invariant*, contains constraints on the realization representation which make it possible to interpret specification representation (for example, string in specification and records in realization) and to keep the realizations of the operations internally consistent. This invariant must be verified to be true at the end of initialization and at the end of every operation in the concept. However, it is assumed to be true at the beginning of finalization and at the beginning of every operation in the concept. The conventions clause is written before the correspondence clause because the correspondence needs to be interpreted only for the representations satisfying the conventions. The *conventions* for the two realizations are discussed in their respective sections. The *correspondence* clause, also known as *abstraction function* or *abstraction relation*, relates the specification representation (conceptual view) to the realization representation (realized view); it is necessary to reason that the realization captures the specification. The *correspondence* assertions for the two realizations are discussed in their respective sections.

A concept realization contains a **procedure** for each operation of the concept; the procedures are written in terms of realization's representation. The implementations of the procedure begin with the assumption that the pre-condition of the operation is true. The goal of each procedure is to prove the post-conditions of the operation. In these realizations, the symbols ':=' and ':=:' represent assignment and swap operations, respectively.

2.4.1. Swap operation

A distinguishing feature of RESOLVE language is that it has clean semantics, which makes it easier to specify and verify software components [Kulczycki04, Kirschenbaum09]. One of the novel ideas of RESOLVE language to support clean semantics is the use of a swap operation (instead of a copy operation as a basic data movement operator) to change values of objects [Harms91]. In a programming environment, generally, object values are changed by copying data or by copying references. Both of these techniques have associated problems: (i) For large objects, copying data is expensive in terms of execution time and memory usage, and (ii) copying references leads to aliasing, in turn making it difficult to reason about correctness. Swapping can be implemented efficiently at the same time without introducing aliasing. In RESOLVE, the symbol ':=:' is used to represent swap operations. The swap operation is discussed here because it takes a constant time to swap the values of two objects, an important criterion for the performance discussion.

2.4.2. Clean circular array realization

Fragments of the *Clean_Circular_Array_Realiz* (CCAR) realization are shown in Fig 2.8, Fig 2.9a, and Fig 2.10. The realization fragment shown in Fig 2.8a begins with the *Array_is_Clean* definition. Using this definition, the *conventions* asserts that any entry which is

not part of the conceptual queue is an initial value entry. In this realization, a *P_Queue* object is represented as a record of an array (*Content*), an index (*PreFront*), and a length (*Length*). The *PreFront* index is the location (in the content array) just in front of the queue contents. The *Length* field is the number of elements in the queue.

Fig 2.7a: Conceptual view

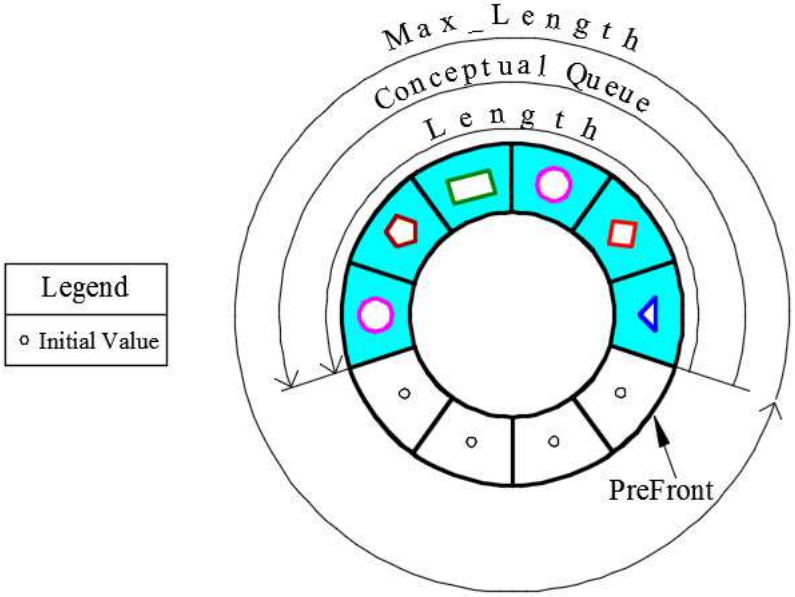

Fig 2.7b: Representation view: explanation of Conventions and Correspondence Clauses

The *conventions* clause for the realizations provides the following invariants, as shown in Fig 2.7b and Fig2.8. (i) *Q.PreFront* is non-negative number that is less the *Max_Length*; hence, *Q.PreFront* is a valid index in the array. (ii) *Q.Length* is non-negative number that can be as big as *Max_Length*. (iii) The entries that are not part of the conceptual array are the initial valued entries. The *correspondence* clause (Fig 2.7b and Fig 2.8) for the realizations indicates that

Conc.Q is equal to the concatenation of every entry from the front of the queue to the end of the queue.

The *Enqueue* procedure (Fig 2.8) increments the length of the queue and adds the new entry at the tail of the queue. The *Inject* procedure (Fig 2.8) increments the length of the queue, adds the new entry at the front end of the queue, and updates the *PreFront* based on its current location.

```
Realization Clean_Circular_Array_Realiz for Preemptable_Queue_Template;

    Definition Array_is_Clean(Q: P_Queue): B =
        For all i: Integer,
                if (((Q.PreFront + Q.Length) mod Max_Length) < i and
                   i <=  ((Q.PreFront + 1) mod Max_Length))
                then Entry.Is_Initial(Q.Contents(i));

    Type P_Queue = Record
        Contents: Array 0..Max_Length - 1 of Entry;
        PreFront, Length: Integer;
    end;

    convention
        0 <= Q.PreFront < Max_Length and
            0 <= Q.Length <= Max_Length and Array_is_Clean(Q);

    correspondence
        Conc.Q = (Concatenation i: Integer
            where (Q.PreFront + 1) mod Max_Length <= i <=
                        (Q.PreFront + Q.Length) mod Max_Length,
                                <Q.Contents(i mod Max_Length)>);

    Procedure Enqueue(clears E: Entry; updates Q: P_Queue);
        Q.Length := Q.Length + 1;
        Q.Contents[(Q.PreFront + Q.Length) mod Max_Length] :=: E;
    end Enqueue;

    Procedure Inject(clears E: Entry; updates Q: P_Queue);
        Q.Length := Q.Length + 1;
        Q.Contents[Q.PreFront] :=: E;

        if (Q.PreFront = 0) then
            Q.PreFront := Max_Length;
        else
            Q.PreFront := Q.PreFront - 1;
        end;
    end Inject;
```

Fig 2.8: Clean_Circular_Array_Realiz -- a realization for Preemptable_Queue_Template

The *Dequeue* procedure shown in Fig 2.9a is graphically represented in Fig 2.9b, Fig 2.9c, and Fig 2.9d. In order to understand the *Dequeue* procedure (Fig 2.9a), consider two

key points. (i) The input parameter *R* is specified using the **replaces** mode; hence, the incoming value of the parameter *R* can be any value. (ii) In the clean array realization, any entry that is not part of the array that corresponds to the conceptual queue's value, must be an initial valued *Entry*. Therefore, a local variable *cleanEntry* is declared and the incoming value of *R* is swapped with the *cleanEntry* (Fig 2.9c) before swapping with the front *Entry* of the queue (Fig 2.9d). The code for *Swap_Last_Entry*, *Length*, and *Rem_Capacity* procedures are straightforward.

```
Procedure Dequeue(replaces R: Entry; updates Q: P_Queue);
    Var cleanEntry: Entry;

    cleanEntry :=: R;
    Q.PreFront := (Q.PreFront + 1) mod Max_Length;
    Q.Contents[Q.PreFront] :=: R;
    Q.Length := Q.Length - 1;
end Dequeue;

Procedure Swap_Last_Entry(updates E: Entry; updates Q: P_Queue);
    Q.Contents[(Q.PreFront + Q.Length) mod Max_Length] :=: E;
end Swap_Last_Entry;

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

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

Fig 2.9a: Clean_Circular_Array_Realiz (continued)

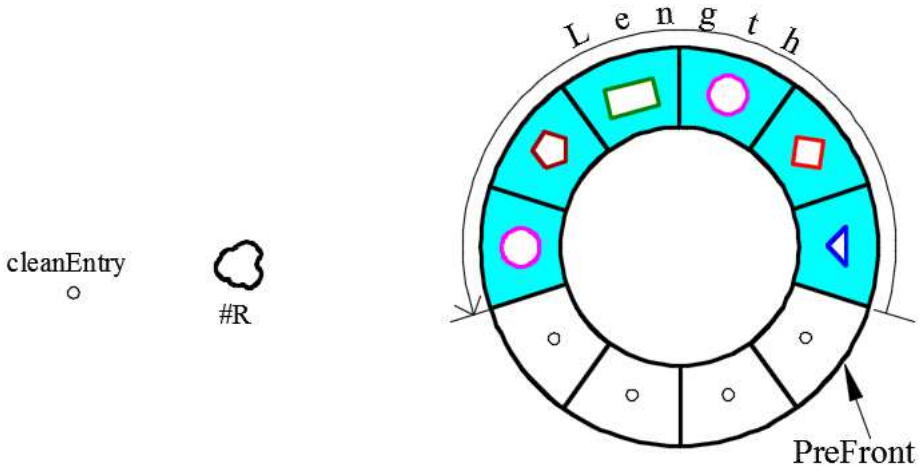

Fig 2.9b: Dequeue – The inputs and the local variable

Fig 2.9c: Dequeue – The incoming E and local variable swapped

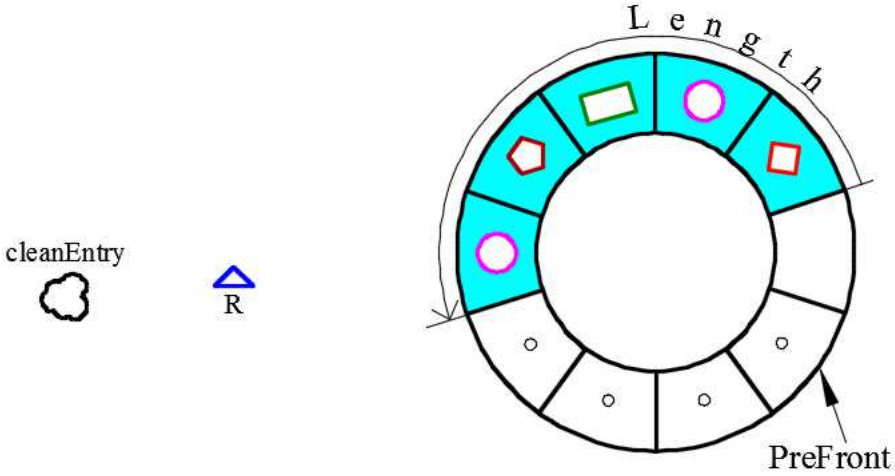

Fig 2.9d: Dequeue – The outputs and the finalized local variable

The *Clear* procedure (Fig 2.10) calls a locally declared operation *Clear_Entry*. The **clears** mode for the *Entry* type variable in the *Clear_Entry* operation assures that the outgoing value of *E* is the initial value of the *Entry* type. The **maintaining** clause captures the fact that any element which is not part of the conceptual queue is the initial valued entry.

```
    Operation Clear_Entry(clears E: Entry);
    Procedure
        Var Temp: Entry;
        E :=: Temp;
    end Clear_Entry;

    Procedure Clear(clears Q: P_Queue);
        Var Len: Integer;
        Len := Length(Q);
        While (Len /= 0)
            maintaining Array_is_Clean(Q) and Len = |Q|;
            changing Q, Len;
            decreasing |Q|;
        do
            Clear_Entry(Q.Contents[(Q.PreFront + Q.Length) mod Max_Length]);
            Q.Length := Q.Length - 1;
            Decrement(Len);
        end;
        Q.PreFront := 0;
    end Clear;

end Clean_Circular_Array_Realiz;
```

Fig 2.10: Clean_Circular_Array_Realiz (continued)

2.4.3. Clean linear array realization

Fragments of the *Clean_Linear_Array_Realiz* (CLAR) realization are shown in Fig 2.12, Fig 2.13a, Fig 2.14a, and Fig 2.15. In this realization, a *P_Queue* object is represented as a record of an array (*Content*) and length (*Length*). The *Length* field is the number of elements in the queue.

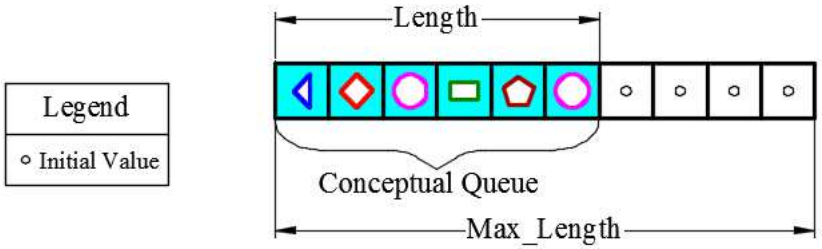

Fig 2.11: Explanation of Conventions and Correspondence Clauses for the CLAR

```
Realization Clean_Linear_Array_Realiz for Preemptable_Queue_Template;

    Definition Array_is_Clean(Q: P_Queue): B =
        For all i: Integer, if Q.Length + 1 = i <= Max_Length
                                then Entry.Is_Initial(Q.Contents(i));

    Type P_Queue = Record
        Contents: Array 1..Max_Length of Entry;
            Length: Integer;
        end;
        convention
            0 <= Q.Length <= Max_Length and Array_is_Clean(Q);
        correspondence
            Conc.Q = (Concatenation i: Integer
                        where 0 + 1 <= i <= Q.Length, <Q.Contents(i)>);
    end;
```

Fig 2.12a: Clean_Linear_Array_ Realiz -- a realization for Preemptable_Queue_Template

The *Inject* procedure shown in Fig 2.13a is graphically represented in Fig 2.13b, Fig 2.13c, Fig 2.13d, and Fig 2.13e. This procedure can be described in two major steps. (i) Before the loop: the incoming value is added at the tail end of the queue, Fig 2.13c. (ii) The loop: using swap operation the entry is moved towards the front of the queue and the location of the entry (*E_Index*) is updated (Fig 2.13a, Fig 2.13d, and Fig 2.13e).

```
Procedure Enqueue(alters E: Entry; updates Q: P_Queue);
        Q.Length := Q.Length + 1;
        Q.Contents[Q.Length] :=: E;
    end Enqueue;

    Procedure Inject(alters E: Entry; updates Q: P_Queue);
        Var E_Index: Integer;

        Q.Length := Q.Length + 1;
        Q.Contents[Q.Length] :=: E;
        E_Index := Q.Length;
        While (E_Index > 1)
            changing Q.Contents, E_Index;
            decreasing E_Index;
            maintaining
                PROD_{0+1 <= i, i <= Q.Length} < Q.Contents (i) >  =  PROD_{0+1 <= i, i < E_Index} < #Q.Contents (i) > o < # E > o
                                                                      PROD_{E_Index+1 <= i, i <= Q.Length} < #Q.Contents (i ) >;

        do
            Q.Contents[E_Index] :=: Q.Contents[E_Index - 1];
            E_Index := E_Index - 1;
        end;
    end Inject;
```

Fig 2.13a: Clean_Linear_Array_Realiz (Continued)

The **changing** (documents the variables that are affected during the loop execution) and the **decreasing** clauses (of the *Inject* procedure) are clear from the loop body, however, the **maintaining** clause is explained here.

- The term Π represents the mathematical definition of concatenation. In earlier sections, it is referred as **Concatenation**.
- The term <#Q.Contents(i)> represents the ith entry of the incoming Q converted to a string.
- The term $\left(\prod_{0+1 <= i}^{i < E\_Index} < \#Q.Contents(i) > \right)$ represents the concatenation of the ith strings where $0 + 1 <= i < E\_Index$.
- It is clear from Fig 2.13d, that the outgoing Q can be divided into three parts: (i) front of the queue before *E-Index*; (ii) the *E-Index*; and (iii) from after the *E-Index* to the end of the queue.
- Using the above bullets, the invariant states that the outgoing Q can be obtained by concatenating the string for the three parts of the queue.

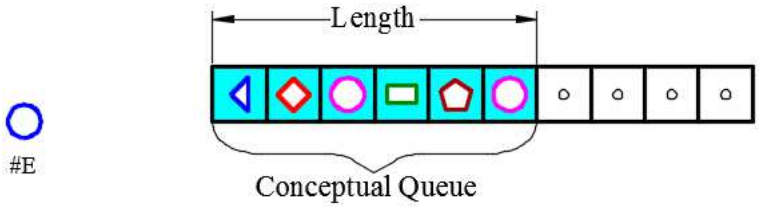

Fig 2.13b: Inject – Inputs

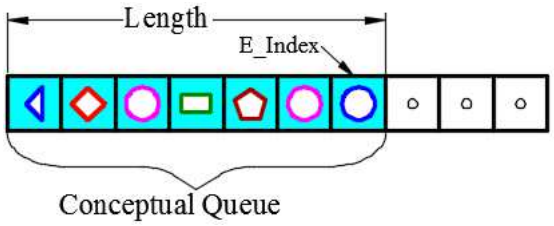

Fig 2.13c: Inject – Before the while loop

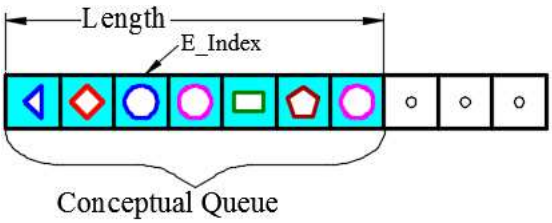

Fig 2.13d: Inject – During the while loop

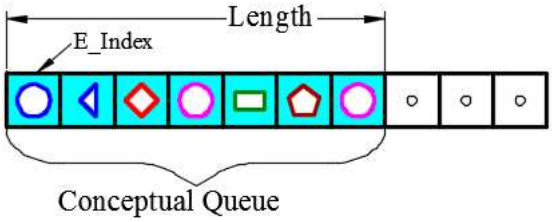

Fig 2.13e: Inject – After the while loop and before exiting the procedure

The remaining procedures are straightforward for the realization; and are shown in Fig 2.14 and Fig 2.15.

```
Procedure Dequeue(replaces R: Entry; updates Q: P_Queue);
    Var cleanEntry: Entry;
    Var R_Index: Integer;

    R_Index := 1;
    While (R_Index <= Q.Length)
        changing Q.Contents, R_Index;
        decreasing |Q.Length - R_Index|;
        maintaining Q.Length and
            ∏ (0+1 <= i, i <= Q.Length) < Q.Contents (i) > = ∏ (0+2 <= i, i <= R_Index) < #Q.Contents (i) > o
                                                              < #Q.Contents (1) > o
                                                              ∏ (R_Index+1 <= i, i <= Q.Length) < #Q.Contents (i ) >;
    do
        Q.Contents[R_Index] :=: Q.Contents[R_Index+1];
        R_Index := R_Index + 1;
    end;

    R :=: cleanEntry;
    Q.Contents[Q.Length] :=: R;
    Q.Length := Q.Length - 1;
end Dequeue;

Procedure Swap_Last_Entry(updates E: Entry; updates Q: P_Queue);
    Q.Contents[Q.Length] :=: E;
end Swap_Last_Entry;

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

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

Fig 2.14: Clean_Linear_Array_Realiz (Continued)

```
    Operation Clear_Entry(clears E: Entry);
        Var Temp: Entry;
        E :=: Temp;
    end Clear_Entry;

    Procedure Clear(clears Q: P_Queue);
        Var Len: Integer;

        Len := Q.Length;
        While (Len /= 0)
            maintaining Array_is_Clean(Q) and Len = |Q|;
            changing Q, Len;
            decreasing |Q|;
        do
            Clear_Entry(Q.Contents[Q.Length]);
            Q.Length := Q.Length - 1;
            Decrement(Len);
        end;
    end Clear;

end Clean_Linear_Array_Realiz;
```

Fig 2.15: Clean_Linear_Array_Realiz (Continued)

2.5. Enhancement

Even a well-designed concept cannot support every feature needed by every user. However, the concept can be extended to provide the desired functionality. In RESOLVE terminology, an extension to a *Concept*, to add a new feature or operation is known as an *Enhancement*. The operations of a concept are the primary operations and the operations of enhancement are the secondary operations. Implementations of enhancements are written using the primary operations. For a demonstration, consider a simple example *Append_Capability*, an enhancement to a generic concept *Preemptable_Queue_Template* to append one queue to another queue. Fig 2.16 shows the relationship between the enhancement, concept, and their realizations.

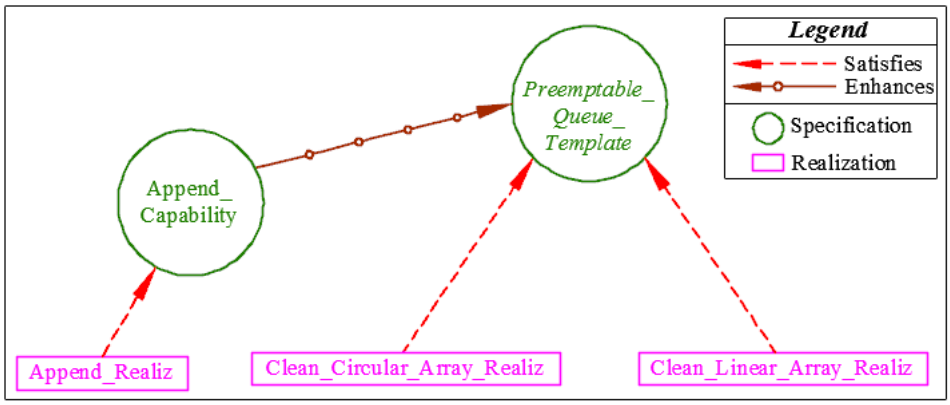

Fig 2.16: The relationship of enhancement, concept, and realizations

2.5.1. Functional specification

Fig 2.17 shows the functional specification of the *Append_Capability* enhancement for the *Preemptable_Queue_Template* concept. The functional specification of an enhancement begins with the keyword **Enhancement**. *Append_Capability* contains the functional specification of an operation *Append_to*. This operation takes two P_Queue parameters P in the **updates** mode and Q in the **clears** mode. The **requires** clause states that the summation of the lengths of incoming queues cannot exceed the *Max_Length*. The **ensures** clause states that the resulting P_Queue object P is equal to the concatenation of the incoming queues.

2.5.2. Realization

Fig 2.18 shows a realization of *Append_Capability* enhancement and is called *Append_Realiz_1*. In this example, the procedure is written using the basic operations of the *Preemptable_Queue_Template* so that the code could be reused with any implementation of the concept. Since procedure *Append_to* calls *Dequeue* and *Enqueue* operations, therefore, it is the responsibility (or goal) of *Append_to* procedure to satisfy the pre-conditions of the called opera-

tion just before their calls. However, *Append_to* assumes that the post-conditions of the called *operation* are true after the return from the call, once the code has been verified.

```
Enhancement Append_Capability for Preemptable_Queue_Template;

    Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
        requires |P| + |Q| <= Max_Length;
        ensures P = #P o #Q;
end Append_Capability;
```

Fig 2.17: Specification of Append_Capability enhancement

```
Realization Append_Realiz_1 for Append_Capability
                            of Preemptable_Queue_Template;

    Procedure Append_to(updates P: P_Queue; clears Q: P_Queue);
        Var Next: Entry;
        Var Len: Integer;

        Len := Length(Q);
        While (Len /= 0)
            maintaining (P o Q = #P o #Q) and Entry.Is_Initial(Next) and Len = |Q|;
            decreasing |Q|;
        do
            Dequeue(Next, Q);
            Enqueue(Next, P);
            Decrement(Len);
        end;
    end Append_to;
end Append_Realiz_1;
```

Fig 2.18: Realization of the Append_Capability enhancement

2.6. Functionality Verification Condition Generation

A verification condition (VC) of a statement is defined as a formula that is derived using the statement, its pre- and post-conditions, and an appropriate proof rule [Berg82]. The proof

rules for functionality are shown in [Harton11]. The VCs generation and formal verification for functional correctness for RESOLVE software are discussed in [Harton11, Smith13]. Formal verification provides proof that a realization is correct with respect to its specification on its valid inputs [Harton11]. This section explains the relationship between a functional specification, realization, and the verification condition (VC).

RESOLVE is a specification-implementation language. Its web integrated development environment (IDE) provides component-based development and modular verification capabilities [Cook12]. For its users, a push-button verifier for functional behavior [Sitaraman11] is available through a web IDE. The web IDE is shown in Fig 2.19; the *VCs* button (shown on the upper left corner) is used to generate VCs and the *Verify* button is used to verify the generated VCs.

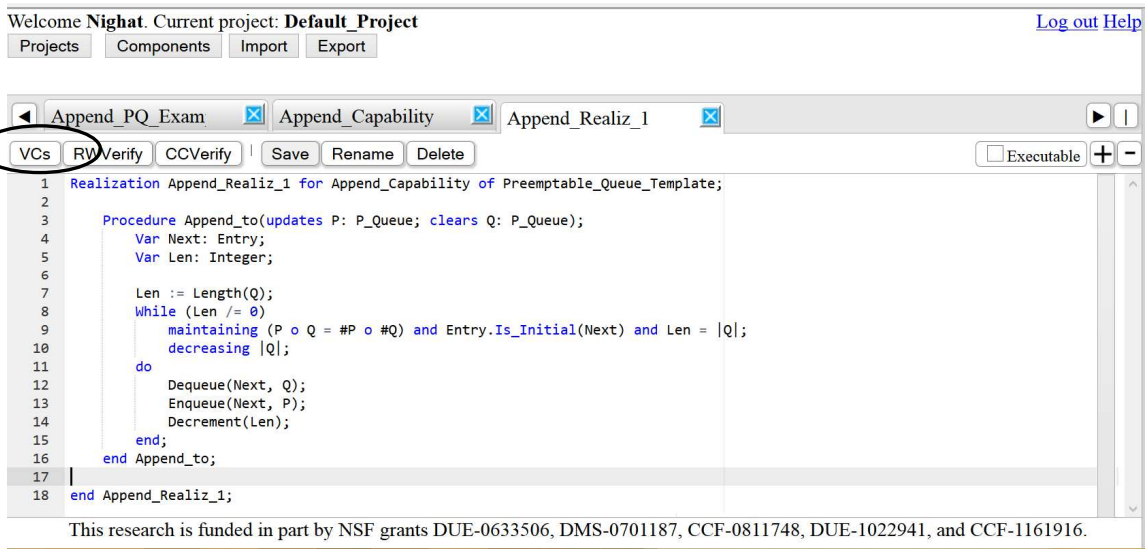

Fig 2.19: RESOLVE web IDE

In RESOLVE, modular verification of a program can be divided into two steps [Kirschenbaum09, Sitaraman11]: (i) automatic VC generation and (ii) VC proof. Fig 2.21 shows the automatically generated VC [Harton11] for the *Append_to* operation shown in Fig 2.18 re-

produced as Fig 2.20 (the line numbers are added to the code). The VC generator outputs the VCs in *VC:Path_number*, *location in realization*, *Goal(s)* and *Given(s)* as shown in Fig 2.21. The complete list of VCs is shown in Appendix C.

- *VC:Path_number*: The term *VC:0_4* means that the fourth verification condition on path number '*0*'. If a realization includes If/Else or loops then there is more than one possible paths.
- *Location in realization*: The term "*Requires Clause of Dequeue in Procedure Append_to: Append_Realiz.rb(12)*" in *VC:0_4* indicates that the VC is generated for the **requires** clause of the *Dequeue* operation; and the *Dequeue* operation is called at line 12 of the code, Fig 2.20.
- *Goal(s)*: The term "*Goal(s)*" means that what needs to be proved.
- *Given(s)*: The term "*Given(s)*" means the assumptions that can be used to prove the goal.

```
1   Realization Append_Realiz_1 for Append_Capability
                                of Preemptable_Queue_Template;
2
3       Procedure Append_to(updates P: P_Queue; clears Q: P_Queue);
4           Var Next: Entry;
5           Var Len: Integer;
6
7           Len := Length(Q);
8           While (Len /= 0)
9               maintaining (P o Q = #P o #Q) and Entry.Is_Initial(Next) and Len = |Q|;
10              decreasing |Q|;
11          do
12              Dequeue(Next, Q);
13              Enqueue(Next, P);
14              Decrement(Len);
15          end;
16      end Append_to;
17
18  end Append_Realiz_1;
```

Fig 2.20: Realization of the Append_to enhancement

```
VCs for Append_Realiz_1.rb generated Tue Mar 03 16:31:51 EST 2015

============================ VC(s): ============================

VC 0_4
Requires Clause of Dequeue in Procedure Append_to: Append_Realiz_1.rb(12)

Goal(s):
(|Q"| /= 0)

Given(s):
1. (Len" /= 0)
2. (P_val' = |Q"|)
3. (Len" = |Q"|)
4. Entry.Is_Initial(Next"')
5. ((P" o Q") = (P o Q))
6. Entry.Is_Initial(Next)
7. (|Q| <= Max_Length)
8. (|P| <= Max_Length)
9. ((|P| + |Q|) <= Max_Length)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)
```

Fig 2.21: Assertive code generation process for the operation Append_to

CHAPTER 3.

PERFORMANCE SPECIFICATION

3.1. Introduction

While correctness is a fundamental concern in any realization, good performance is an essential property for a useful realization. A language for software engineering, in addition to being influenced by functionality, also must include syntax and notation for expressing performance estimates of realization. A performance specification is a non-behavioral specification of a component. It is a performance contract between the user code and implementation(s) of the component. It represents the expected performance behavior of a component's implementation without disclosing the internal details of the implementation. Therefore, extension of the RESOLVE language with performance specification features, specifically duration bounds—the first research deliverable—is the focus of this chapter.

In order to choose the best realization that fits a desired application, estimates of both execution time duration and memory capacity requirement are needed. The focus of this research is on performance contracts with the emphasis on duration specification.

The remainder of this chapter is organized into three sections. Section 3.2 discusses the performance of an example facility discussed in Chapter 2; section 3.3 is the description of the performance specification notations in RESOVE; and section 3.4 explains the modification to the realization to incorporate performance specifications.

3.2. A Simple Performance Estimate Example

For a simple example to illustrate the use of performance estimates, consider the *Append_to* procedure from Fig. 2.6a (reproduced as Fig 3.1). The performance of this code depends on the length of incoming value of Q (the queue to be appended). Based on a loose-bound analysis, its running time (**duration**) is given as $C + D*|\#Q|$; where C and D are implementation dependent constants. This is a linear time *O(n)* procedure where $n = |\#Q|$. In the realizations (Fig 3.1), to enable automated analysis of correctness of the duration bounds, the **elapsed_time** assertion is added to document loops as discussed in Section 2.3. The **elapsed_time** clause for a loop is a running time (duration) expression. It is set to zero before the first iteration; and the duration of the iteration is added to it after each iteration.

However, the running time not only depends on the size of the input, it also depends on time to execute the code statements; the time to execute an operation can be subdivided into operation call, parameter passing, number of parameters, and the operation body [Cormen90]. Hence, for tight analysis, time to create and destroy local variables, time to call operations, time for testing loop conditions before every iteration, time to execute the loop body, etc. must be included in the running time estimates. The tight running time for the *Append_to* procedure can be written as function of these times. Thus, the term *C* in the duration clause (Fig 3.1) can represent the summation of time to create and destroy local variables *Next* and *Len*; time to call and execute the *Length* operation; and assigning the length to the *Len*. The term *C* also included the time to execute the loop condition before exiting the loop. The term *D* in the duration clause (Fig 3.1) can represent the summation of time to call and execute the *Dequeue*, *Enqueue*, and *Decrement* operation; and assigning the length to the *Len*. The term *D* also included the time to execute the

loop condition before every iteration of the loop. The tight runtime estimate is the focus of this chapter.

```
Facility Example_Fac;
    …
    Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
        requires |P| + |Q| <= Max_Length;
        ensures P = #P o #Q;
        duration C + D * |#Q|;

    Procedure
        Var Next: Integer;
        Var Len: Integer;

        Len := Length(Q);
        While (Len /= 0)
            maintaining (P o Q = #P o #Q) and Len = |Q|;
            decreasing |Q|;
            elapsed_time D * (|#Q| - |Q|);
        do
            Dequeue(Next, Q);
            Enqueue(Next, P);
            Decrement(Len);
        end;
    end Append_to;
end Example_Fac;
```

Fig 3.1: Facility declaration (loose-bound analysis)

3.3. Performance specification of a concept

This section describes the performance specification syntactically defined as a *profile*. A profile is a performance contract, and in general, includes time and space constraints, though the focus of this dissertation is only on the former. The performance profile includes duration for the

initialization and finalization of objects of the provided type; it also shows the duration for operations to manipulate those objects.

3.3.1. Performance profile of Clean_Circular_Array_Realiz

A performance profile, *PQ_CT_Profile* (*CT* being the shorthand for *Constant Time*) for the *Clean_Circular_Array_Realiz* is demonstrated in Fig 3.2a and Fig 3.2b. This profile is suitable for realizations in which most routine Queue operations take a constant time. In particular, it is suitable for a realization that uses a circular array and that gets rid of unused entries in the array immediately; specifically, for those implementations where every entry in the array that is not part of the conceptual queue is restricted to being only the initial value of the Entry.

A performance profile starts with the key word **Profile** followed by its name (*PQ_CT_Profile*), which is a shorthand for a more descriptive profile name, given following the keyword **short_for**; the profile is described in the context of the corresponding concept. A profile (like a concept) may contain both explicit definitions of terms and terms whose definitions are deferred using the keyword **defines**. Generally, implementation-dependent constants are named in the profile and the actual definitions are deferred to the corresponding realizations.

In performance profile *PQ_CT_Profile*, the duration expression to initialize a *P_Queue* variable has two parts (Fig 3.2a): (i) Part 1 is *CI1*, an implementation-dependent constant. (ii) Part 2 is the time to initialize the entries in the array. In the space-conscious implementation, the array contains only the initial valued entries. Therefore, the duration for initialization of this part is captured by the terms *(CI2+* **I_Dur** *(Entry)* * *Max_Length*; where, *CI2* is an implementation-dependent constant. In this expression, the term **I_Dur** represents the duration to create an initial valued *Entry*.

```
Profile PQ_CT_Profile short_for
            Space_Conservative_Mostly_Constant_Time_Profile for
                                        Preemptable_Queue_Template;

    uses  Duration_Basics_Theory;
    uses Integer_Template;
    Defines CI1, CI2, CF1, CF2, CEn, CIn, CDq, CSle, CRc,
                                                    CLe, CCl1, CCl2: RPos;

    Definition Cnts_Dur(S: Str(Entry): RPos = Sigma (x: Entry,
                                        Occurs_Ct(x, S) * F_Dur(Entry, x));

    Type Family P_Queue is modeled by Str(Entry);
    initialization
        duration  CI1 + (CI2 + I_Dur(Entry)) * Max_Length;

    finalization
        duration  CF1 + Cnts_Dur(#Q) + (CF2 + F_IV_Dur(Entry)) *
                                                    (Max_Length - |#Q|);
    end;

    Operation Enqueue(clears E: Entry; updates Q: P_Queue);
        duration  CEn;

    Operation Inject(clears E: Entry; updates Q: P_Queue);
        duration  CIn;
```

Fig 3.2a: PQ_CT_Profile - Performance profile of Clean_Circular_Array_Realiz

In the performance profile *PQ_CT_Profile*, the duration expression to finalize a *P_Queue* variable (*Q*) has two parts (Fig 3.2a). (i) Part 1 is the time to finalize the entries in the array that are part of the conceptual queue (Fig 2.7b). This part is given as the summation of *CF1* (implementation-dependent constant) and *Cnts_Dur(#Q)*. The term *Cnts_Dur(#Q)* is the sum of the durations to finalize every entry in the conceptual queue. (ii) Part 2 is the time to finalize the entries in the array that are not part of the conceptual queue (Fig 2.7b). Therefore, the duration for

finalization of this part is captured by the terms *(CF2* + **F_IV_Dur***(Entry))* * *(Max_Length - |#Q|)*; where, *CF2* is an implementation-dependent constant. In this expression, the term **F_IV_Dur** represents the duration to finalize an initial valued *Entry*.

In the performance profile *PQ_CT_Profile* shown in Fig 3.2a, the durations for *Enqueue* and *Inject* operations are implementation-dependent constants, *CEn* and *CIn*, respectively. Note that both the *Enqueue* and *Inject* operations are passed *E* (an Entry type parameter) with the **clears** mode; it is used to capture the fact that outgoing value of the Entry *E* is initial valued. This additional clause is used in performance analysis of user code to provide tighter estimates as will be discussed later.

The duration expression for the *Dequeue* operation in the profile of space-efficient implementation Fig 3.2b has two parts. (i) Part 1 is the summation of the implementation-dependent constant (*CDq)* and the time to initialize the local Entry *cleanEntry*, (**I_Dur***(Entry)*). (ii) Part 2 is the time to finalize the incoming value of the Entry *R* (**F_Dur***(Entry, #R)*); In this expression, the term **F_Dur** represents the duration to finalize *#R* of type *Entry*. In analyzing the duration for the *Dequeue* operation, note that its duration is directly proportional to the Part 2. If the incoming value is the initial valued Entry then substitute **F_IV_Dur***(Entry)* for **F_Dur***(#R)* in the duration expression; and the expression is reduced to: *(CDq* + **I_Dur***(Entry)* + **F_IV_Dur***(Entry)*.

The duration expression for the *Clear* operation follows the same rationale as that for the finalization operation.

```
        Operation Dequeue(replaces R: Entry; updates Q: P_Queue);
                duration  CDq + I_Dur(Entry) + F_Dur(Entry, #R);

        Operation Swap_Last_Entry(updates E: Entry; updates Q: P_Queue);
                 duration  CSle;

        Operation Length(restores Q: P_Queue): Integer;
                duration  CLe;

        Operation Rem_Capacity(restores Q: P_Queue): Integer;
                duration  CRc;

        Operation Clear(clears Q: P_Queue);
                duration CCl1 + Cnts_Dur( #Q ) +  (CCl2 + I_Dur(Entry)) * |#Q|;

end PQ_CT_Profile;
```

Fig 3.2b: PQ_CT_Profile (Continued)

3.3.2. Performance profile of Clean_Linear_Array_Realiz

A performance profile, *PQ_LT_Profile*, for the *Clean_Linear_Array_Realiz* is demonstrated in Fig 3.3a and Fig 3.3b. This profile is suitable for a realization that uses a linear array and that gets rid of unused entries in the array immediately; specifically, every entry in the array that is not part of the conceptual queue is restricted to being only the initial value of the Entry. The major differences in *PQ_LT_Profile* and *PQ_CT_Profile* profiles is in the duration expressions of the *Inject* and *Dequeue* operation. Unlike the previous profile, the duration of these operations depends on the length of queue; hence, the name linear.

```
Profile PQ_LT_Profile short_for
                Space_Conservative_Mostly_Linear_Time_Profile for
                                                Preemptable_Queue_Template;

        …

        Operation Inject(clears E: Entry; updates Q: P_Queue);
                duration CIn1 + (CIn2 * |#Q|) + F_Dur(Entry, #E);

        Operation Dequeue(replaces R: Entry; updates Q: P_Queue);
                duration  CDq1 + I_Dur(Entry) + (CDq2 * |#Q|) + F_Dur(Entry, #R);

        …

end PQ_LT_Profile;
```

Fig 3.3: PQ_LT_Profile - Performance profile of Clean_Linear_Array_Realiz

3.3.3. Concept realization revisited

Fig 3.4 shows the modified version of *Clean_Circular_Array_Realize* shown in Fig 2.8a, Fig 2.9a, and Fig 2.10. The term ***with_profile*** *PQ_CT_Profile* is added to the file header. The keyword **with_profile** is used to indicate that VC generator will not only generate the VC for the functional specification but it will generate VCs for the performance, too. The realization also shows the **definitions** of the constants defined in the *PQ_CT_Profile* profile. The ***elapsed_time*** clause is also added to the loop invariant for the **clears** code.

```
Realization Clean_Circular_Array_Realiz with_profile
                    PQ_CT_Profile for Preemptable_Queue_Template;
    ….
    Definition CDq: RPos =
        ((2*Dur_Swap) + (2*Dur_Assgn) + (Dur_Add) + (Dur_Sub));
    …..

    Procedure Clear(clears Q: P_Queue);
        While (Len /= 0)
        …
            elapsed_time ((CCl2 + I_Dur(Entry) + F_Dur(Entry)) *
                                                (#Q.Length - Q.Length));
        do
            Clear_Entry(Q.Contents[(Q.PreFront +
                                            Q.Length) mod Max_Length]);
            Q.Length := Q.Length - 1;
        end;
         Q.PreFront := 0;
    end Clear;
    …
end Clean_Circular_Array_Realiz;
```

Fig 3.4: Clean_Circular_Array_Realiz modified for the performance profile

3.4. Performance profile of an enhancement

Fig 3.5 shows the modifications to Fig 2.18, whereby performance profiles are added to the relationship. As noted earlier, a performance profile of an enhancement depends upon the performance profile of the parent concept.

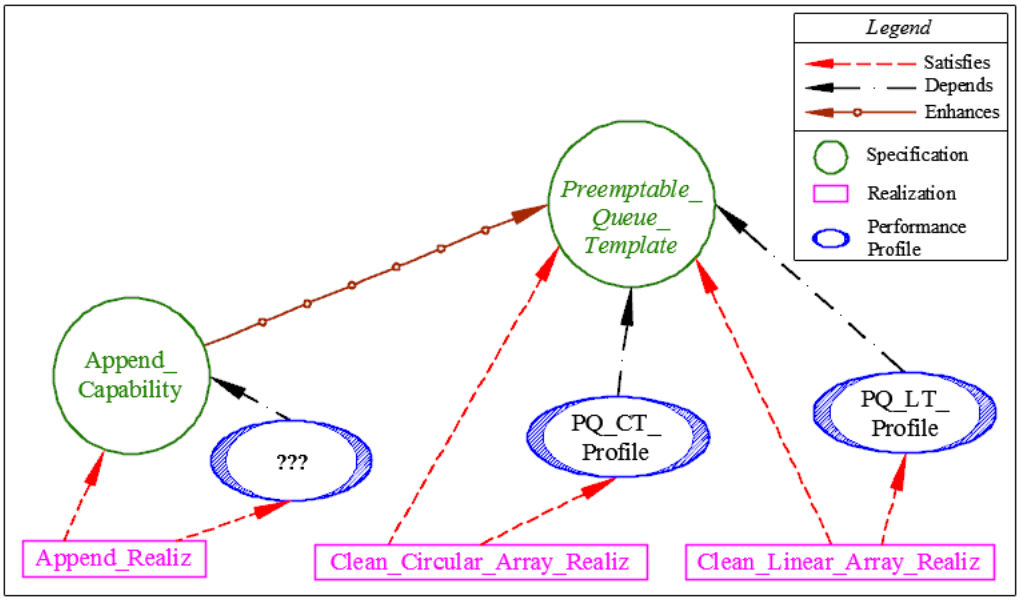

Fig 3.5: The relationship of concept, enhancement, realizations, and performance profiles

A performance profile for *Append_Realiz* realization of the *Append_Capability* enhancement is called *Apnd_Profile* for a class of implementations. *Append_Capability* is an enhancement for the *Preemptable_Queue_Template*; and *PQ_CT_Profile* is one of the performance profiles of the *Preemptable_Queue_Template*. Therefore, the terms *Preemptable_Queue_Template* **with_profile** *PQ_CT_Profile* is added to *Apnd_Profile*. Here the keyword **with_profile** is used to indicate the selection of the performance profile of the parent concept. Fig 3.6 show the performance profile *Apnd_Profile*. The duration expression of the *Append_to* operation depends on the length of the array.

```
Profile Apnd_Profile short_for Append_Profile for Append_Capability for
                        Preemptable_Queue_Template with_profile PQ_CT_Profile;

        Defines C1: RPos;
        Defines C2: RPos;

        Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
                duration C1 + I_Dur(Entry) + F_IV_Dur(Entry) +
                                        |#Q| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
end Apnd_Profile;
```

Fig 3.6: Performance profile of the realization shown in Fig 2.18

3.4.1. Realization revisited

Since the performance profile is written for a realization, therefore, it is important to relate profile and the realization. The realization also shows the **definitions** of the deferred constants in the profile.

The **elapsed_time** duration expression for the loop of the *Append_to* operation using the *Clean_Circular_Array_Realiz* realization is developed as follow.

(i) Specifying the durations.

- The duration to create an initial valued local variable *x* and *n* is **I_Dur***(Entry)* and **I_Dur***(Integer)*, respectively.
- The duration to call and execute the *Length* operation and assigning the return value to *n*; *Dur_Call(1)* + *Cle* + *Dur_Assgn*.
- The duration to call and execute the *Greater* (*n* > *0*) operation; *Dur_Call(2)* + *ITP_Gr*.
- The duration of the loop is calculated as follow:

a. Duration of the *Dequeue* operation: *Dur_Call(2)* + *CDq* + **I_Dur***(Entry)* + **F_Dur***(Entry, #R)*. However, **F_Dur***(Entry, #R)* will reduce to **F_IV_Dur***(Entry)* because the entry is cleared by Dequeue.

b. Duration of the *Enqueue* operation: *Dur_Call(2)* + *CEn*.

c. Duration of the *Length* and assignment operations; *Dur_Call(1)* + *Cle* + Dur_Assgn.

d. The duration to call and execute the *Greater* (*Length(Q)* > *0*) operation; *Dur_Call(2)* + *ITP_Gr*.

(ii) Add the durations for the loop. The loop is executed |#Q| times.

|*#Q*|* *((Dur_Call(2)* + *CDq* + **I_Dur***(Entry)* + **F_IV_Dur***(Entry))* +

(Dur_Call(2) + *CEn)* + *(Dur_Call(1)* + *Cle* + Dur_Assgn*)* + *Dur_Call(2)* + *ITP_Gr)*

(iii) Rearrange the terms from the previous step.

|*#Q*|* *((3*Dur_Call(2)* + *Dur_Call(1)* + **I_Dur***(Entry)* + **F_IV_Dur***(Entry))* +

+ *CDq* + *CEn* + *Cle* + Dur_Assgn + *ITP_Gr)*

(iv) Simplify from the previous step, to create the duration expression shown in Fig 3.7.

|#Q| * (*3*Dur_Call(2)* + *Dur_Call(1)* + **I_Dur***(Entry)* + **F_IV_Dur***(Entry))* +

C2).

```
Realization Append_Realiz with_profile Apnd_Profile for
                                Append_Capability of Preemptable_Queue_Template;
    uses ITP;
    uses Integer_Template;

    Definition C1: RPos = Dur_Assgn + CLe + ITP_AreNotEq +
                I_Dur(Integer) + 2*F_IV_Dur(Integer) + Dur_Call(1) + Dur_Call(2);
    Definition C2: RPos = CDq + CEn + ITP_Decr + ITP_AreNotEq +
                                                  Dur_Call(1) + 3*Dur_Call(2);

    Procedure Append_to(updates P: P_Queue; clears Q: P_Queue);
        Var Next: Entry;
        Var Len: Integer;

        Len := Length(Q);
        While (Len /= 0)
            maintaining (P o Q = #P o #Q) and
                                                  Entry.Is_Initial(Next) and Len = |Q|;
            decreasing |Q|;
            elapsed_time (|#Q| - |Q|) * (C2 + I_Dur(Entry) + F_Dur(Entry, Next));
        do
            Dequeue(Next, Q);
            Enqueue(Next, P);
            Decrement(Len);
        end;
    end Append_to;

end Append_Realiz;
```

Fig 3.7: Realization of the Append_Capability enhancement

3.5. Facility revisited

This section will first describe the idea of a facility augmented with a performance profile. If FF is a facility for a concept C, R_C is the realization of C, P_C is a performance profile of R_C, E is an enhancement for the concept C, R_E is the realization of E, and P_E is the performance

profile of R_E then FF is written as shown in Fig. 3-8a. Fig. 3-8b shows the syntax for the *Integer_Queue_Fac* created in Chapter 2.

Facility FF **is** C(…) **realized by** R_C [**with_profile** P_C (…)]
enhanced by E **realized by** R_E [**with_profile** P_E (…)]
end FF;

Fig 3.ba: Facility syntax augmented with performance profile

Facility Integer_Queue_Fac **is** Preemptable_Queue_Template (Integer, 10)
realized by Clean_Circular_Array_Realiz
with_profile PQ_CT_Profile
enhanced by Append_Capability
realized by Append_Realiz
with_profile Apnd_Profile;

….

end Intger_Queue_Fac;

Fig 3.8b: Example Use of Profiles

CHAPTER 4.

THE PROOF SYSTEM

4.1. Introduction

A proof system is an integral part of a verifying compiler, because it is the basis for the verification condition (VC) generator. In [Harton11], Harton established mechanizable proof rules to generate VCs using functional specifications and realizations in RESOLVE. The specification-based proof rules lead to modular and scalable verification. In addition, the same rules are used to generate the VC for the code containing built-in objects (e.g., Integer objects) and user-defined objects (e.g., queues). Using the same basic approach, this chapter augments the existing proof rules for functional correctness with the duration analysis. Since duration assertions and verification depend on values of objects from the functional correctness analysis, the system for duration analysis must be layered over a system for functional correctness. It cannot be independent. This chapter also illustrates the working of the rules using implementation of the modified rules to generate duration related VCs. Hence, mechanizable proof rules for performance VC generation of a component-based software system—the second research deliverable—is the focus of this chapter.

A proof system consists of inference rules (or proof rules) for each language construct. In the proof rules, the typical Hoare triple *{P} S {Q}* is written as ***Assume*** *P; S;* ***Confirm*** *Q*, where *P* and *Q* are assertions and *S* is a sequence of statements.

In [Harton11], proof rules used in functionality behavior have been designed such that the VC generator starts at the end of the program and backs over the code replacing the program statement using the proof rules. The proof rules are written in the vertical style as $\frac{Premises}{Conclusion}$. In this notation, to prove *Conclusion*, it is both necessary and sufficient to prove *Premises*. In the remainder of this chapter, each subsection begins with the discussion of a proof rule followed by an example to illustrate the generated VC.

4.2. Context Specification Enrichment Rules

In the rules discussed in this chapter, $\mathscr{C}$ stands for *Context*. The *Context* is the supporting information that is needed to reason about the correctness of the code. For example, the context contains concepts including type information and specification of all operations within scope, and mathematical theories including definitions and results.

4.2.1. Concept Declaration Rule

The concept declaration rule is used to add the concept to the context. The rule is shown in Fig 4.1 and the VCs generated by this rule are explained later. In the premises of the rule, { CT } needs to be added to $\mathscr{C}$, that is, update the context. This is accomplished in the rule by replacing (using the notation $\rightsquigarrow$) $\mathscr{C}$ with $\mathscr{C} \cup$ { CT }. This chapter will use *Preemptable_Queue_Template* to explain the proof rules and the resulting automatically generated VCs.

The functional specification of the concept is shown in Fig 4.2. This figure is a reproduction of Fig 2.1 and the details can be found in Chapter 2.

$\mathcal{C} \rightsquigarrow \mathcal{C} \cup \{ \text{CT} \};$

$\mathcal{C} \setminus \text{CT};$

where CT is
Concept Concept_Template(**…**);
…
end Concept_Template;

Fig 4.1: Concept declaration rule

```
Concept Preemptable_Queue_Template(type Entry; evaluates Max_Length: Integer);
    uses String_Theory, Integer_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;
    end;

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

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

Fig 4.2: A functional specification of Preemptable_Queue_Template concept

4.2.2. Concept Profile Declaration Rule

The concept profile declaration rule is used to add the concept's profile to the context. The rule is shown in Fig 4.3 and the VCs generated by this rule are explained later.

$\mathcal{C} \rightsquigarrow \mathcal{C} \cup \{ \text{CT} \} \cup \{ \text{CPT} \};$

$\mathcal{C} \cup \{ \text{CT} \} \setminus \text{CPT};$

where CPT is
Profile PT **short_for** Profile_Template **for** Concept_Template;
…
end PT;

Fig 4.3: Profile declaration rule

This chapter will use *PQ_CT_Profile*, a performance profile for *Preemptable_Queue_Template* to explain the proof rules and the resulting automatically generated VCs. *PQ_CT_Profile* is shown in Fig 4.4. This figure is the combination of Fig 3.2a and Fig 3.2b, which are described in detail in Chapter 3.

```
Profile PQ_CT_Profile short_for
                        Space_Conservative_Mostly_Constant_Time_Profile for
                                                    Preemptable_Queue_Template;
    uses  Duration_Basics_Theory, Integer_Template;
    Defines CI1, CI2, CF1, CF2, CEn, CIn,CDq, CSle, CRc, CLe,
                                                          CCl1, CCl2: RPos;
    Definition Cnts_Dur(S: Str(Entry): RPos = Sigma (x: Entry,
                                          Occurs_Ct(x, S) * F_Dur(Entry, x));

    Type Family P_Queue is modeled by Str(Entry);
        initialization
            duration  CI1 + (CI2 + I_Dur(Entry)) * Max_Length;

        finalization
            duration  CF1 + Cnts_Dur(#Q) + (CF2 + F_IV_Dur(Entry)) *
                                                    (Max_Length - |#Q|);
    end;

    Operation Enqueue(clears E: Entry; updates Q: P_Queue);
        duration  CEn;

    Operation Inject(clears E: Entry; updates Q: P_Queue);
        duration  CIn;

    Operation Dequeue(replaces R: Entry; updates Q: P_Queue);
        duration  CDq + I_Dur(Entry) + F_Dur(Entry, #R);

    Operation Length(restores Q: P_Queue): Integer;
       duration  CLe

    Operation Clear(clears Q: P_Queue);
        duration  CCl1 + Cnts_Dur( #Q ) +  (CCl2 + I_Dur(Entry)) *
                                                    (Max_Length - |#Q|);
        …

end PQ  CT  Profile;
```

Fig 4.4: PQ_CT_Profile – An example performance profile

4.2.3. Enhancement Declaration Rule

The enhancement declaration rule is used to add the enhancement to the context. The rule is shown in Fig 4.5 and the VCs generated by this rule are explained later.

$$\frac{\mathscr{C} \rightsquigarrow \mathscr{C} \cup \{ \text{CT} \} \cup \{ \text{CPT} \} \cup \{ \text{CT\_Enh} \};}{\mathscr{C} \cup \{ \text{CT} \} \cup \{ \text{CPT} \} \setminus \text{CT\_Enh};}$$

where CT_Enh is
Enhancement CT_Enhancement **for** CT;
Operation P (**updates** t: T);
requires P_Pre;
ensures P_Post;
duration Dur_Bd_Exp;
end CT_Enhancement;

Fig 4.5: Enhancement declaration rule

This chapter uses several enhancements. Fig 4.6 shows a typical enhancement. The detailed description of functionality specification enhancements can be found in Chapter 2. The *Append_Some* operation is discussed in procedure declaration rule (Section 4.3).

Enhancement Append_Some_Capability **for** Preemptable_Queue_Template;

Operation Append_Some(**updates** P: P_Queue; **updates** Q: P_Queue);
ensures P o Q = #P o #Q;

end Append_Some_Capability;

Fig 4.6: Specification of Append_Some_Capability Enhancement

4.2.4. Enhancement Profile Declaration Rule

The enhancement profile declaration rule is used to add the enhancement's profile to the context. The rule is shown in Fig 4.7.

$$\frac{\mathcal{C} \rightsquigarrow \mathcal{C} \cup \{ \text{CT} \} \cup \{ \text{CPT} \} \cup \{ \text{CT\_Enh} \} \cup \{ \text{CT\_EnhP} \};}{\mathcal{C} \cup \{ \text{CT} \} \cup \{ \text{CPT} \} \cup \{ \text{CT\_Enh} \} \setminus \text{CT\_EnhP};}$$

where CT_EnhP is
Profile CT_EP **short_for** CT_Enhancement_Profile **for**
CT_Enhancment **for** CT **with_profile** CPT;
Operation P (**…**);
duration …
end CT_EP;

Fig 4.7: Enhancement profile declaration rule

Fig 4.8 shows a performance profile for the enhancement shown in Fig 4.6. The detailed description of a performance profile for an enhancement can be found in Chapter 3.

Profile ApndS_Profile_1 **short_for** Append_Some_Profile **for**
Append_Some_Capability **for** Preemptable_Queue_Template
with_profile PQ_CT_Profile;
Defines C: RPos;
Operation Append_Some(**updates** P: P_Queue; **updates** Q: P_Queue);
duration C;

end ApndS_Profile_1;

Fig 4.8: A performance profile for Append_Some_Capability

4.2.5. Enhancement Realization Rule

The enhancement realization declaration rule needs to verify that each procedure (code) in the realization is correct with respect to both its functional and performance specification. The rule for verifying each procedure's correctness is presented as a procedure declaration rule in the next subsection.

4.3. Procedure Declaration Rule

The procedure declaration rule is used by VC generator system to ensure the correctness of a procedure declaration with respect to its specification. A simplified version of the rule is shown in Fig 4.11. This version of the rule is for an operation with a parameter in the **updates** mode. In the rule, \ represent a separator between the context and the assertive code; and it is read as "assertive code supported by the context".

Below the line, in the *Conclusion* of the rule, the terms *P_Spec* and *P_Code* represent the specification and implementation of the operation *P*, respectively; and together they constitute the procedure declaration. *Code* represents the code that follows that procedure declaration and calls operation *P*. Before *Code*, *Pre_w_Dur* (not to be confused with *P_Pre*) may be assumed and after *Code*, *Post_w_Dur* needs to be confirmed for it to be correct. Here, *Pre_w_Dur* and *Post_w_Dur* are assumed to include both functionality and duration assertions.

To establish the *Conclusion* below the line, first the *Context* is updated with *P_Spec* (only) above the line. If the procedure is for an enhancement operation, *P_Spec* is essentially a combination of the functional specification in the enhancement and the duration specification in a corresponding profile, and the context would have been already enriched with those specifications.

Based on the updated context, which includes *P_Spec*, two items needs to be proved. The first item is showing that *P_Code* is correct with respect to *P_Spec*. The second item is showing the correctness of *Code* using the specification of operation *P*. This separation supports modularity and scalability. The verification process can focus on verifying a single component with the assumption that any supporting components have already been verified or is verified separately.

$\mathscr{C} \cup \{$ P_Spec $\} \setminus$ **Assume** P_Pre $\wedge$ T.*Constraint*(t) $\wedge$ **Cum_Dur** = 0.0;
Remember;
P_Code;
Confirm P_Post $\wedge$ **Cum_Dur** $\leq$ Dur_Bd_Exp;

$\mathscr{C} \cup \{$ P_Spec $\} \setminus$ **Assume** Pre_w_Dur; Code; **Confirm** Post_w_Dur;

$\mathscr{C} \setminus$ **Assume** Pre_w_Dur;
P_Spec;
P_Code;

Code;
Confirm Post_w_Dur;

where P_Spec is **Operation** P (**updates** t: T) ;
requires P_Pre;
ensures P_Post;
duration Dur_Bd_Exp;

and P_Code is an implementation of P_Spec;

Fig 4.11: A simplified version of procedure declaration rule

The term **Cum_Dur** is a verifier-introduced variable and it is used to represent the cumulative duration. Initially, **Cum_Dur** is zero. Since every statement requires some time to execute; hence, one or more terms are added to it after the execution of each statement. This term is updated automatically by the verifier. The term Dur_Bd_Exp represents the duration's upper bound

expression. This term is provided by the programmer in the duration specification and is written in terms of the program variables.

We are now ready to describe the mechanics of the first part above the line that is necessary to establish the correctness of *P_Code*. This verification involves creation of an appropriate Hoare triple. Before *P_Code*, we may assume the pre-conditions of the operation (*P_Pre*), since it is the calling code's responsibility. We also assume any constraints on the parameter. The rule also assumes that **Cum_Dur** is zero before the procedure call; therefore, the term **Cum_Dur** = *0.0* is added to the assumptions before *P_Code*. This assumption supports compositional analysis by localizing the proof correctness only to this procedure. The **Remember** statement is a reminder to the verifier that at current location, the value of a variable preceded by # is the same as its value without the # prefix, that is, *#x* and *x* are the same at the beginning of the procedure, where *x* is a parameter. Following *P_Code*, we need to confirm the post-condition (*P_Post*) of the operation. For duration verification, the rule also confirms that **Cum_Dur** is less than or equal to the duration's upper bound provided by the performance profile of the operation (Dur_Bd_Exp).

This chapter uses sample enhancement operations of objects to explain the proof rules and the resulting automatically generated VCs. Since enhancements depend on the parent concept, therefore, the functional specifications and the performance profiles for the parent concept are shown in Fig 4.2 and Fig 4.4, respectively.

To understand the essence of the procedure declaration rule, consider Fig 4.12a that shows the functionality specification for the *Append_Some* operation. The specification contains the signature for the operation *Append_Some* and its **ensures** clause only states that the resulting queues concatenated is equal to concatenation of the incoming queues, meaning any form among

several output queues are acceptable. Fig 4.12b shows its performance profile where the duration expression is an implementation-dependent constant. The typing *C*: RPos means that *C* is a non-negative real number; **Defines** means that its definition is provided somewhere else. Fig 4.12c shows one possible realization that satisfies the functional specification (Fig 4.12a) and performance profile (Fig 4.12b). The realization does nothing and it is an acceptable realization (no code) of *Append_Some* operation. The realization also provides the actual definition of the constant *C*; here *C* is set to zero (Real_0 means 0.0) because operation call rule (discussed later) will account for the overheads of operation call time and the number and type of parameters.

```
Operation Append_Some (updates P: P_Queue; updates Q: P_Queue);
    ensures P o Q = #P o #Q;
```

Fig 4.12a: Specification of Append_Some operation

```
Defines C: RPos;
Operation Append_Some (updates P: P_Queue; updates Q: P_Queue);
    duration C;
```

Fig 4.12b: Performance profile for Append_Some operation

```
Definition C: RPos = Real_0;
Procedure Append_Some (updates P: P_Queue; updates Q: P_Queue);
end Append_Some;
```

Fig 4.12c: A realization of Append_Some operation to illustrate procedure declaration rule

The automated system has mechanically generated two VCs for the current example and are shown in Fig 4.13. The naming *VC 0_1* means that it is the first verification condition on path number '*0*'. The first VC concerns functionality, that is, it corresponds to ensuring the post-

condition of the *Append_Some* operation. The term on the right hand side of the *Goal* (*P o Q*) is obtained after applying the **Remember** statement on the right side of the **ensures** clause (*#P o #Q*) retrieved from the functional specification shown in Fig 4.12a.

The *Goal* needs to be established using the *Givens*. Given #1 represents the assumption that **Cum_Dur** is zero before the procedure call; Givens #2 and #3 arise from the constraints on the type of the parameters. Given #4 represents the constraints from *Character_Template*, concept that is included in the context by default; Givens #5 and #6 represent the constraints from *Integer_Template*, another concept included in the context; Given #7 comes from the **requires** clause of *Preemptable_Queue_Template* concept parameter. It is obvious that several of the givens are not necessary to establish the goal. A version of the VC generator for functional correctness that eliminates unnecessary givens is in progress. Henceforth, we omit Givens that are clearly irrelevant.

The second VC shown in Fig 4.13 is related to the duration bound. The term on the right hand side (RHS) of the *Goal* inequality is the duration clause retrieved from the profile shown in Fig 4.12b. The left had side (LHS) of the *Goal* is derived mechanically for the code shown in Fig 4.12c. The LHS involves the term **Cum_Dur** allowing the *Goal* inequality to be proved. While at this time the RESOLVE prover is not able to prove the result automatically, it is a straightforward task for a prover equipped to handle real numbers. Noting that **Cum_Dur** (a verification variable) is given to be 0.0, the *Goal* can be established with an automated prover.

```
VCs for Append_Some_Realiz_1.rb generated Thu Mar 19 02:26:24 EDT 2015
========================= VC(s): ========================

VC 0_1
Ensures Clause of Append_Some: Append_Some_Realiz_1.rb(4)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. (Cum_Dur = 0.0)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. (Last_Char_Num > 0)
5. (0 < max_int)
6. (min_int <= 0)
7. (Max_Length > 0)

VC 0_2
Duration Clause of Append_Some: Append_Some_Realiz_1.rb(4)

Goal(s):
(Cum_Dur <= C)

Given(s):
1. (Cum_Dur = 0.0)
```

Fig 4.13: VC to illustrate the application of procedural declaration rule

4.4. A Simple Function Declaration Rule

A function declaration rule is shown in Fig 4.14. This version of the rule is for an operation with one parameter in **evaluates** mode. The operation returns a value of type T2. Below the line, in the *Conclusion* of the rule, the terms *F_Spec* and *F_Code* represent the functionality specification and implementation of the operation *F,* respectively; and together they constitute

the function declaration. In the functionality specification of the operation, the **ensures** clause is required to be given in the form *F = (F_Post)*, where F is the name of the operation and *F_Post* is the specification of what the function is expected to produce.

$\mathcal{C} \cup$ { F_Spec } \ **Assume** F_Pre $\wedge$ T1.*Constraint*(e) $\wedge$ **Cum_Dur** = 0.0;
Remember;
Var F: T2;
F_Code;
Confirm F = F_Post $\wedge$ **Cum_Dur** + T1.**F_Dur**(e) $\leq$ Dur_Bd_Exp;

$\mathcal{C} \cup$ { F_Spec } \ **Assume** Pre_w_Dur; Code; **Confirm** Post_w_Dur;

$\mathcal{C}$ \ **Assume** Pre_w_Dur;
F_Spec;
F_Code;

Code;
Confirm Post_w_Dur;

where F_Spec is **Operation** F (**evaluates** e: T1): T2 ;
requires F_Pre;
ensures F = (F_Post);
duration Dur_Bd_Exp;

and F_Code is an implementation of F_Spec;

Fig 4.14: A simplified version of function declaration rule

As in the procedure declaration rule, the first part above the line is necessary to establish the correctness of *F_Code* with respect to *F_Spec*. In this assertive code, a local variable *F* of type *T2* is explicitly included because a variable is used to store the function's return value. The initialization duration for this local variable is accounted for in the variable declaration rule (discussed later). No finalization duration is added because *F* is used to hold the return value of the

function and is returned to the caller. Since, parameter *e* of type *T1* is passed with **evaluates** mode, the variable used to store the evaluated value of the expression *e* is needed to be finalized (or destroyed) at the end of the function. Hence, the term *T1*.**F_Dur***(e)* is added at the end of *P_Code*. The VCs generated by this rule is explained in the context of the function assignment rule later.

4.5. Variable Declaration Rule

The variable declaration rule is shown in Fig 4.15. Below the line, the *Conclusion* part of the rule shows that a variable *s* of type *T* is created. The newly generated variable has an initial value according to the specification of the concept that defines its type. This fact is captured above the line by the conjunct *T*.**Is_initial**(s); this predicate represents that the variable *s* of type *T* is initial valued variable. Recall that *T*.**I_Dur** is the (maximum) duration to create a variable of type *T* with the assumption that the newly created variable is initial valued. The term *T*.**F_Dur***(s)* represents the duration to finalize a variable *s* of type *T2*; and it depends on the value of *s* at the end of the operation. So *T*.**I_Dur** and *T*.**F_Dur***(s)* need to be added to **Cum_Dur**, that is, to the time taken by the code. This is accomplished in the rule by replacing (using the notation ⇝) **Cum_Dur** with **Cum_Dur** + *T*.**I_Dur** + *T*.**F_Dur**(s).

To understand the essence of this rule, consider Fig 4.12a that shows functionality specification of *Append_Some* operation. Fig 4.16a shows its performance profile. The duration bound includes an implementation dependent constant and time to initialize and finalize an Entry and a P_Queue object.

$\mathscr{C}$ \	**Assume** Pre_w_Dur ∧ T.**Is_Initial**(s); Code; **Confirm** Post_w_Dur [**Cum_Dur** ⇝ **Cum_Dur** +T.**I_Dur**+T.**F_Dur**(s)];
$\mathscr{C}$ \	**Assume** Pre_w_Dur; **Var** s: T; Code; **Confirm** Post_w_Dur;

Fig 4.15: Variable declaration rule

```
Defines C: RPos;

Operation Append_Some(updates P: P_Queue; updates Q: P_Queue);
    duration C + I_Dur(Entry) + I_Dur(P_Queue) +
                                F_IV_Dur(Entry) + F_IV_Dur(P_Queue);
```

Fig 4.16a: A performance profile for Append_Some operation

Fig 4.16b shows one possible realization. For demonstration purposes, three local variables *Next* of *Entry* type (generic type), *T* of *P_Queue* type (user defined type), *I* of *Integer* type (built-in type) are declared. Note that the duration to finalize the local variables are **F_IV_Dur**(Entry*)* for *Next* and **F_IV_Dur**(P_Queue) for *Temp_Q*, because the local variable are never used, therefore, their values are not changed from their initial values. Since, the duration to create and finalize an (initial valued) integer are constants, the terms **I_Dur**(Integer) and **F_IV_Dur**(Integer) are absorbed in an implementation constant, *C*, whose definition is given in the realization, satisfying the information hiding principle.

The complete listing of VCs is given in Appendix C. The second VC is related to the duration bound and is shown in Fig 4.17. The term on the right hand side (RHS) of the *Goal* inequality (Fig 4.17) is the duration clause retrieved from the profile shown in Fig 4.16b; and the

term on the left had side of the *Goal* is derived mechanically for the code shown in Fig 4.16b and involves the term **Cum_Dur**, **F_Dur**(Entry, Next), **F_Dur**(P_Queue, temp_Q), and **F_Dur**(Integer, I). Using the *Given(s)* #1, #2, and #3, the finalization expressions can be written as: **F_IV_Dur**(Entry), **F_IV_Dur**(P_Queue), and **F_IV_Dur**(Integer). Replacing the constant on the right side of the *Goal* leads to the satisfaction of the goal statement.

```
Definition C: RPos = I_Dur(Integer) + F_IV_Dur(Integer);

Procedure Append_Some(updates P: P_Queue; updates Q: P_Queue);
    Var Next: Entry;
    Var Temp_Q: P_Queue;
    Var I: Integer;
end Append_Some;
```

Fig 4.16b: A realization of Append_Some operation to illustrate variable declaration rule

```
VC 0_2
Duration Clause of Append_Some: Append_Some_Realiz_2.rb(4)

Goal(s):
(((((Cum_Dur + I_Dur(Entry)) + I_Dur(P_Queue)) + I_Dur(Integer)) +
    ((F_Dur(Entry, Next) + F_Dur(P_Queue, temp_Q)) + F_Dur(Integer, I))) <=
                ((((C + I_Dur(Entry)) + I_Dur(P_Queue)) + F_IV_Dur(Entry)) +
                                                        F_IV_Dur(P_Queue)))

Given(s):
1. (I = 0)
2. (temp_Q = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
```

Fig 4.17: VC to illustrate the application of variable declaration rule

4.6. Assume Rule

The assume rule is a basic rule that processes **Assume** statements generated by other proof rules. The rule is shown in Fig 4.18. Below the line, the *Conclusion* part of the rule shows that a predicate *P* is assumed. The rule for removing the **Assume** clause has the effect of making the resulting assertion an implication as shown above the line. The VCs generated by this rule is explained in the context of the swap rule later.

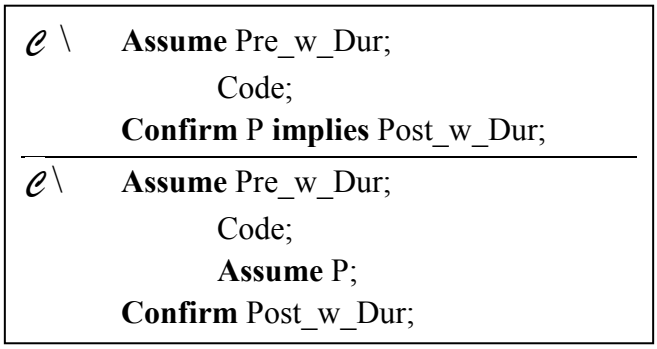

Fig 4.18: Assume rule

4.7. Swap Rule

The swap statement rule is shown in Fig 4.19. The swap operator is represented as :=: and is used to swap the values of two variables. The term $X \rightsquigarrow Y$, $Y \rightsquigarrow X$, means that simultaneously, every *X* is replaced with *Y* and every *Y* is replaced with *X*. To account for the duration of the swap statement *Dur_Swap*, is added to **Cum_Dur** (by replacing **Cum_Dur** with **Cum_Dur** + *Dur_Swap*) above the line.

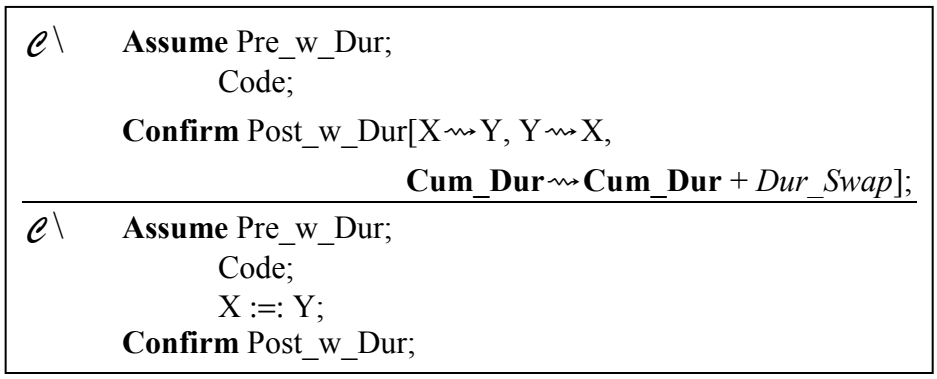

Fig 4.19: Swap rule

Fig 4.12a shows functional specification of *Append_Some* operation. Fig 4.20a shows yet another performance profile. The duration bound is an implementation-dependent constant, *C*. In Fig 4.20b shows one possible realization that satisfies the functional specification (Fig 4.12a) and performance profile (Fig 4.20a). To illustrate the rule, in the realization, the two incoming queues are swapped twice; hence, C = *2* Dur_Swap*.

```
Defines C: RPos;

Operation Append_Some(updates P: P_Queue; updates Q: P_Queue);
        duration C;
```

Fig 4.20a: A performance profile for Append_Some operation

The second VC generated is related to the duration bound and is shown in Fig 4.20c. The complete listing is given in Appendix C. The remainder of this section discusses the VC generation process by the application of the proof rules.

```
Definition C: RPos = 2*Dur_Swap;

Procedure Append_Some(updates P: P_Queue; updates Q: P_Queue);
        P :=: Q;
        Q :=: P;
end Append_Some;
```

Fig 4.20b: A realization of Append_Some operation to illustrate swap Rule

```
VC 0_2
Duration Clause of Append_Some: Append_Some_Realiz_3.rb(4)

Goal(s):
(((Cum_Dur + Dur_Swap) + Dur_Swap) <= C)

Given(s):
1. (Cum_Dur = 0.0)
```

Fig 4.20c: VC to illustrate the application of swap rule

Fig 4.21a show the application of procedure declaration rule to the realization shown in Fig 4.20b. The free variables are not shown in Fig 4.21a. In this figure, the first **Assume** statement includes the **Constraints** (min_int <= 0) and (0 < max_int) from the Integer_Template; (Last_Char_Num > 0) from the Character_Template; (|P| <= Max_Length) and (|Q| <= Max_Length) on the parameters to Append_Some from Preemptable_Queue_Template; and (Max_ Length > 0 from the **requires** clause from Preemptable_Queue_Template.

The body of the procedure is inserted into the assertive code after the pre-condition is assumed and before the post-condition needs to be confirmed. The first part of the **Confirm** clause is created from the **ensures** clause of functional specification (Fig 4.12a) and the second part is created from the **duration** clause of performance profile (Fig 4.20a). Furthermore, there are no

givens that can be used to prove the goal. However, givens are generated as the rest of the assertive code is processed.

```
Assume (Max_Length > 0);
Assume (((min_int <= 0) and (0 < max_int)) and (Last_Char_Num > 0));
Assume ((|P| <= Max_Length) and (|Q| <= Max_Length));
Assume Cum_Dur = 0.0;
Remember;
P :=: Q;
Q :=: P;
Confirm ((P o Q) = (#P o #Q) and (Cum_Dur <= C));
```

Fig 4.21a: Procedure declaration rule applied

Fig 4.21b show the application of swap rule to the assertive code shown in Fig 4.21a. The rule is applied to the swap statement, just before the **Confirm** clause at the end. Notice, the change in the second part of the **Confirm** clause; Dur_Swap is added to **Cum_Dur**.

```
Assume (Max_Length > 0);
Assume (((min_int <= 0) and (0 < max_int)) and (Last_Char_Num > 0));
Assume ((|P| <= Max_Length) and (|Q| <= Max_Length));
Assume Cum_Dur = 0.0;
Remember;
P :=: Q;
Confirm ((Q o P) = (#P o #Q) and ((Cum_Dur + Dur_Swap) <= C));
```

Fig 4.21b: Swap rule applied

Fig 4.21c show the application of swap rule to the assertive code shown in Fig 4.21b. Notice the change in the second part of the **Confirm** clause; *Dur_Swap* is added to **Cum_Dur**.

Assume (Max_Length > 0);
Assume (((min_int <= 0) and (0 < max_int)) and (Last_Char_Num > 0));
Assume ((|P| <= Max_Length) and (|Q| <= Max_Length));
Assume Cum_Dur = 0.0;
Remember;
Confirm ((*P o Q*) = (#P o #Q) and
(((Cum_Dur + *Dur_Swap*) + Dur_Swap) <= C))

Fig 4.21c: Swap rule applied

Fig 4.21d show the application of remember rule to the assertive code shown in Fig 4.21c. Since *#P* and *#Q* denote the remembered values of the parameters at this state (initial state), they are the same as *P* and *Q*, respectively, so they are replaced by *P* and *Q*, respectively, in the **Confirm** clause.

Assume (Max_Length > 0);
Assume (((min_int <= 0) and (0 < max_int)) and (Last_Char_Num > 0));
Assume ((|P| <= Max_Length) and (|Q| <= Max_Length));
Assume Cum_Dur = 0.0;
Confirm ((P o Q) = (*P o Q*) and
(((Cum_Dur + Dur_Swap) + Dur_Swap) <= C))

Fig 4.21d: Remember rule applied

Fig 4.21e show the application of **Assume** rule to the assertive code shown in Fig 4.21d. Fig 4.21f show the application of **Assume** rule three times to the assertive code shown in Fig 4.21e. Every conjunct between **Confirm** and last implies is converted into *Givens*; and every conjunct after the last implies is converted into *Goals* of VC. Fig 4.21g shows the second VC.

Assume (Max_Length > 0);
Assume (((min_int <= 0) and (0 < max_int)) and (Last_Char_Num > 0));
Assume ((|P| <= Max_Length) and (|Q| <= Max_Length));
Confirm (*Cum_Dur = 0.0* ***implies***
((P o Q) = (P o Q) **and** (((Cum_Dur + Dur_Swap) + Dur_Swap) <= C)));

Fig 4.21e: Assume rule applied

Confirm ((Max_Length > 0) **implies**
((((min_int <= 0) and (0 < max_int)) and (Last_Char_Num > 0)) **implies**
(((|P| <= Max_Length) and (|Q| <= Max_Length)) **implies**
(Cum_Dur = 0.0 **implies**
((P o Q) = (P o Q) **and**
(((Cum_Dur + Dur_Swap) + Dur_Swap) <= C))))));

Fig 4.21f: Assume rule applied

VC 0_2
Duration Clause of Append_Some: Append_Some_Realiz_3.rb(4)

Goal(s):
(((Cum_Dur + Dur_Swap) + Dur_Swap) <= C)

Given(s):
1. (Cum_Dur = 0.0)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. (Last_Char_Num > 0)
5. (0 < max_int)
6. (min_int <= 0)
7. (Max_Length > 0)

Fig 4.21g: The second VC

4.8. A Simple Operation Call Rule

A simplified version of the operation call rule is shown in Fig 4.22. This version of the rule is for an operation with one parameter in the **updates** mode. In the *Conclusion* of the rule, an operation P is called with the parameter *a*.

$\mathcal{C} \cup$ { P_Spec } \
Assume Pre_w_Dur;
Code;
Confirm P_Pre[t⇝a];
Assume P_Post[#t⇝a, t⇝ a'] ;
Confirm Post_w_Dur[a⇝a',
Cum_Dur ⇝**Cum_Dur** + *Dur_Call*(1) + Dur_Bd_Exp[#t⇝a, t⇝ a']];

$\mathcal{C} \cup$ { P_Spec } \ **Assume** Pre_w_Dur;
Code;
P(a);
Confirm Post_w_Dur;

where P_Spec is **Operation** P (**updates** t: T) ;
requires P_Pre;
ensures P_Post;
duration Dur_Bd_Exp;

and P_Code is an implementation of P_Spec
and where a' is a shorthand for the more formal name Next_Prime_Name(Post, a)

Fig 4.22: Operation call rule

In this rule, *a* and *a'* represent the values of the actual parameter *a*, before and after the call whereas #*t* and *t* represent the same for the formal parameter *t*. Since, the operation's parameter is in **updates** mode, therefore, pre-conditions of the operation *P* is updated so that any instance of the formal parameter *t* is replaced with the actual parameter, *a*; and in post-conditions,

the old and new values of the formal parameter are appropriately replaced with actual the parameter.

Fig 4.23a shows functionality specification of *Append_One* operation. The specification contains the signature for the operation *Append_One*. This operation takes two P_Queue parameters *P* and *Q* in the **updates** mode. The **requires** clause states that the incoming P_Queue object *Q* is not empty and *P* is not full. The **ensures** clause states that the resulting queues concatenated is equal to the concatenation of the incoming queues (same as *Append_Some*), except now the specification additionally ensures that the length of the outgoing *P* is incremented by one. This has the effect of ensuring exactly one entry is moved from *Q* to *P*.

```
Operation Append_One(updates P: P_Queue; updates Q: P_Queue);
        requires |Q| /= 0 and |P| < Max_Length;
        ensures P o Q = #P o #Q and |P| = |#P| + 1;
```

Fig. 4.23a: Functionality specification of Append_One operation

Fig 4.23b shows a performance profile for *Append_One*; the duration expression is summation of an implementation dependent constant and duration to create and destroy (finalize) two initial valued entries. Fig 4.23c shows one possible realization that satisfies the functionality specification (Fig 4.23a) and the performance profile (Fig 4.23b).

```
Defines C: RPos;

Operation Append_One(updates P: P_Queue; updates Q: P_Queue);
    duration C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry);
```

Fig. 4.23b: A performance profile for Append_One operation

```
Definition C: RPos = CDq + CEn + 2*Dur_Call(2);

Procedure Append_One(updates P: P_Queue; updates Q: P_Queue);
    Var Next: Entry;
    Dequeue(Next, Q);
    Enqueue(Next, P);
end Append_One;
```

Fig. 4.23c: A realization of Append_One operation to illustrate the operation call rule

The five generated VCs for the current example are listed in Appendix C. The fifth VC is related to the duration bound and is shown in Fig 4.23d. In Fig 4.23d, the term on the right hand side of the *Goal* inequality represents the duration clause retrieved from the profile shown in Fig 4.23b. The left hand side of the *Goal* is derived mechanically for the code shown in Fig 4.23c. The term **I_Dur***(Entry)* represents the duration to create the local variable *Next*; *CDq* + **I_Dur***(Entry)* + *F_Dur(Entry, Next)* + *Dur_Call(2)* is the duration for the *Dequeue* operation; *CEn* + *Dur_Call(2)* is the duration for the *Enqueue* operation; Before exiting the procedure, the local variable, *Next*, is finalized, so **F_Dur***(Entry, Next')* is added to the *Goal*. *Next* and *Next'* represent the value of a variable (*Next*) before and after the operation call, respectively. Using *Givens* #1 and #4, **F_Dur***(Entry, Next)* and **F_Dur***(Entry, Next')* can be replaced by 2***F_IV_Dur***(Entry)*. Rearranging the terms on the LHS and replacing *C* with its value from Fig 4.23c will help prove the *Goal* shown in Fig 4.23b.

VC 0_5
Duration Clause of Append_One: Append_One_Realiz_1.rb(5)

Goal(s):
(((((Cum_Dur + I_Dur(Entry)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next))
+ Dur_Call(2))) + (CEn + Dur_Call(2))) + F_Dur(Entry, Next')) <=
((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)

Fig. 4.23d: VC to illustrate the application of the operation call rule

4.9. Function Assignment Rule

A simplified version of the function assignment rule is shown in Fig 4.24. This version of the rule is for an operation with one parameter in the **evaluates** mode. The assignment operator is represented as := and is used to assign the return value of a function to a variable.

Exp can be a simple or a nested expression. Above the line, the conjunct *Pre_Aggr*(exp) is added to the **Confirm** clause, to show that pre-conditions of the nested expression are true. For example, *((X / Y) * Z)* would need to show the pre-conditions of division and multiplication are both true. The term *Math(exp)* represents the mathematical expression that corresponds to the programming expression *exp*. The conjunct *Post_w_Dur[result ⇝F_Post[e ⇝Math(exp)]* means replacing result with the effect of calling the function.

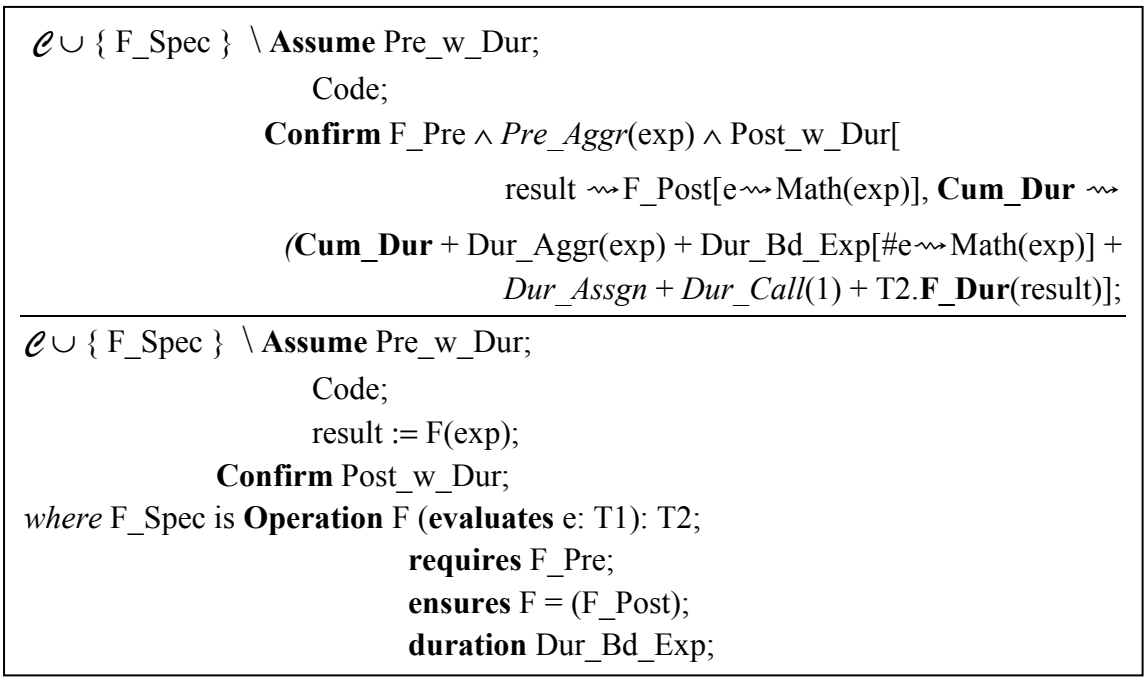

Fig 4.24: Function assignment rule

Function call assignment involves several costs and they are added to **Cum_Dur** above the line. These include the (i) duration to execute the pre-conditions of the nested expression, Dur_Aggr(exp); (ii) the duration's upper bound expression with the formal parameter replaced by mathematical expression; (iii) duration to assign, *Dur_Assgn*; (iv) term *Dur_Call*(1) will account for the overhead of operation call time, depending on the number of parameters; and (v) the term T2.**F_Dur**(result) represents the duration to finalize the incoming value of *result*.

Fig 4.12a shows functionality specification of *Append_Some* operation. Fig 4.25a shows its performance profile and Fig 4.25b shows one possible realization.

The five generated VCs for the current example are listed in Appendix C. The fifth VC is related to the duration bound and is shown in Fig 4.25d.

```
Defines C: RPos;

Operation Append_Some(updates P: P_Queue; updates Q: P_Queue);
    duration C;
```

Fig. 4.25a: Performance profile for Append_Some operation

```
Definition F_Dur_Integer: Rpos = F_Dur(Integer, max_int);
Definition C: RPos = I_Dur(Integer) + F_IV_Dur(Integer) +
                                    CLe + Dur_Call(1) + Dur_Assgn + F_Dur_Integer;

Procedure Append_Some(updates P: P_Queue; updates Q: P_Queue);
    Var Len: Integer;
    Len := Length(P);
end Append_Some;
```

Fig. 4.25b: A realization of Append_Some operation to illustrate function assignment rule

```
VC 0_5
Duration Clause of Append_One: Append_One_Realiz_1.rb(5)

Goal(s):
  (((((Cum_Dur + I_Dur(Entry)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next))
                + Dur_Call(2))) + (CEn + Dur_Call(2))) + F_Dur(Entry, Next')) <=
                                    ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next'">))
3. (Q = (<Next'"> o Q'))
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
```

Fig. 4.25d: VC to illustrate the application of function assignment rule

4.10. If-Then-Else Rule

The if-then-else rule is shown in Fig 4.26. Before discussing the duration section of the rule, let us consider the execution of the rule: (i) first *BF(Exp)* (Boolean function) of the rule is executed; and (ii) then the proper branch is followed. Based on the return value of *BF(Exp)*, the assertive code generated by the if-then-else rule is divided in two sections. One section will process the *If* code and the other section will process the *Else* code.

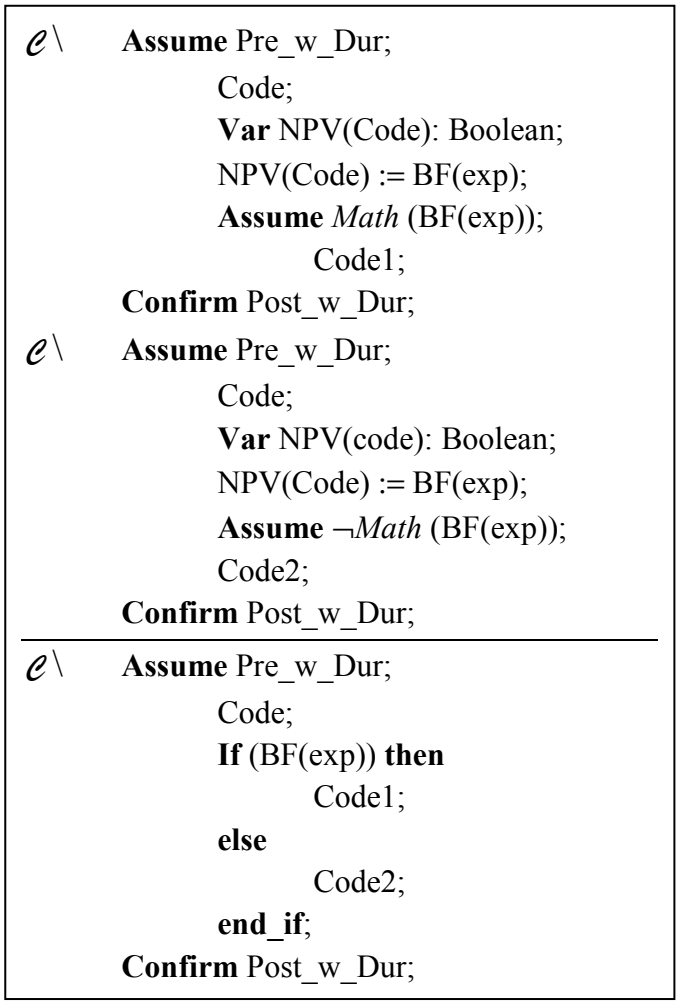

Fig 4.26: If-then-else rule

Now, we discuss the duration aspects of the rule. Above the line, the term *NPV* represents "Next Programing Variable"; and NPV(Code) will produce a new Boolean type variable that is not already in code and it is used to store the return value of *BF(Exp)* (Boolean function). *BF(Exp)* is executed irrespective of its outcome (true or false). Therefore, its duration should be added to the duration section of the rule above the line. However, using function assignment rule, the duration of *BF(Exp)* is added implicitly and not explicitly! Finally, the duration of the Code1 or Code2 is added during their execution.

Fig 4.27a reproduces the functionality specification of *Append_to* operation from Fig 2.17. Fig 4.27b shows its performance profile; the duration expression is summation of an implementation dependent constant and duration to create and destroy two initial valued entries. Fig 4.27c shows one possible realization.

The six generated VCs for the current example are listed in Appendix C. Since, the if-then-else rule has created two paths, two of the six VCs (VC 0_4 and VC 1_2) are related to the duration bound and are shown in Fig 4.27d and Fig 4.27e, respectively. Both of them are provable.

```
Operation Append_to(updates P: P_Queue; updates Q: P_Queue);
        requires |P| < Max_Length;
        ensures P o Q = #P o #Q;
```

Fig 4.27a: Functionality specification of Append_to operation

```
Defines C: RPos;

Operation Append_to(updates P: P_Queue; updates Q: P_Queue);
        duration C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry) ;
```

Fig 4.27b: Performance profile for Append_to operation

```
Definition F_Dur_Integer: Rpos = F_Dur(Integer, max_int);
Definition C: RPos = CLe + CDq + CEn + Dur_Assgn + ITP_AreNotEq +
                                                Dur_Call(1) + 3*Dur_Call(2) +
                        I_Dur(Integer) + F_IV_Dur(Integer) + F_Dur_Integer;

Procedure Append_to(updates P: P_Queue; updates Q: P_Queue);
    Var Next: Entry;
    Var Len: Integer;
    Len := Length(Q);
    if (Len /= 0) then
        Dequeue(Next, Q);
        Enqueue(Next, P);
    end;
end Append_to;
```

Fig 4.27c: A realization of Append_to operation to illustrate If-Then-Else rule

```
VC 0_4
Duration Clause of Append_to: Append_to_Realiz_1.rb(12)

Goal(s):
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) +
                    Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) +
                    F_Dur(Entry, Next)) + Dur_Call(2))) + (CEn + Dur_Call(2))) +
                        (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next') +
                                                        F_Dur(Integer, |Q|))) <=
                                    ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. (|Q| /= 0)
5. (Len = 0)
6. Entry.Is_Initial(Next)
7. (Cum_Dur = 0.0)
```

Fig 4.27d: VC #1 to illustrate duration correctness using if-then-else rule

```
VC 1_2
Duration Clause of Append_to: Append_to_Realiz_1.rb(12)

Goal(s):
(((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) +
          Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) +
                                    (F_Dur(Entry, Next) + F_Dur(Integer, |Q|))) <=
                                ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. (|Q| = 0)
2. (Len = 0)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
```

Fig 4.27e: VC #2 to illustrate duration correctness using if-then-else rule

4.11. While Loop Rule

A mechanizable proof rule for establishing functional correctness of code with while statements annotated with suitable assertions is given in [Harton11] and reproduced in Fig 4.28a. It is necessary to understand this rule, to understand the rule that handles duration bounds. Let us consider the execution of the rule. Below the line: (i) the *BE* (Boolean expression) of the rule is executed; and (ii) if it is true then the loop is executed and if it is false then the loop is not executed. Hence, above the line, the loop is reduced to an *If-Then-Else* rule.

In Fig 4.28a, the first **Confirm** is used for the automation of the rule. The **change** clause provides list of the variables that are modified by the loop; any variable not listed in this list is assumed to not be affected by the loop rule. *NQV* (Next Quotation Mark) variable is used to generate new names for the variables listed in the **changing** clause; for example, every *x* is replaced by *x'*. The rules assumes that the *Inv* is true before the loop and it must be true after every iteration of the loop; this fact is captured by the **Assume** clause before the beginning of the *if* statement and **Confirm** clause before exiting the loop, respectively. *P_Exp* is the ordinal valued progress metric expression; it is used to show the termination of the loop. The assertion **Confirm true** at the end has no effect, except that it makes the assertive code fit the syntax of the Hoare triple.

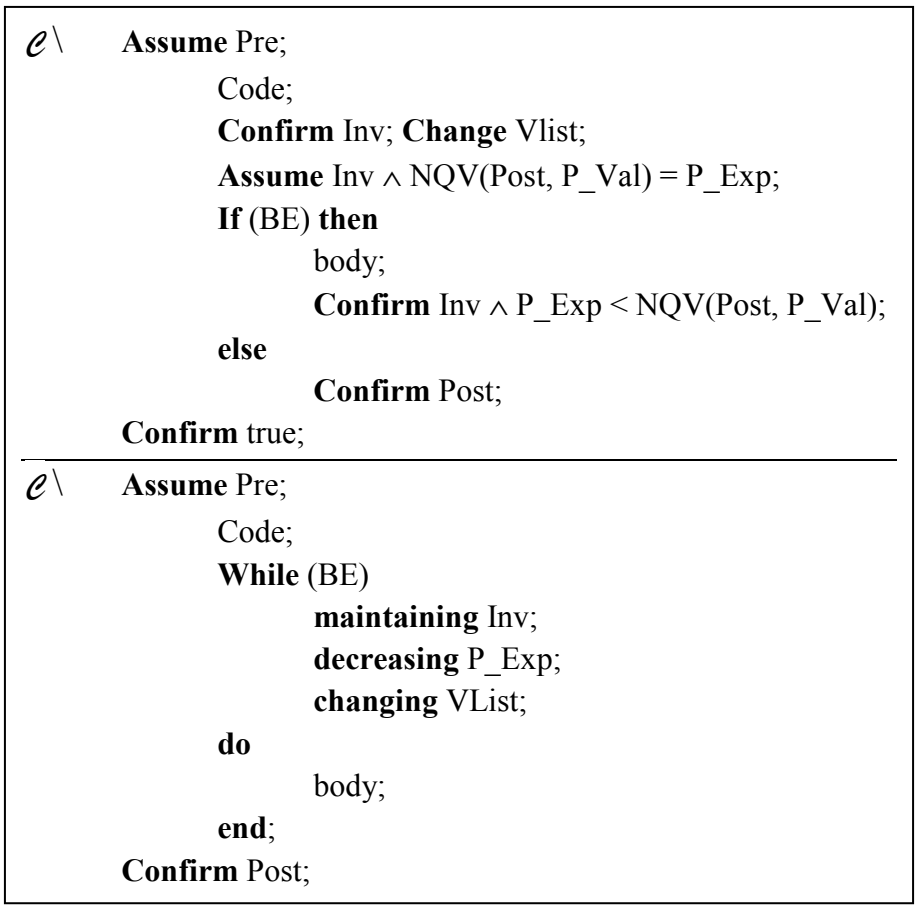

Fig 4.28a: While loop rule for functional specification reproduced from [Harton 2011]

Now consider the duration aspects of the rule. The while loop rule used in this study is based on the while loop rule shown in Fig 4.28a; and the rule with the addition of the performance profile is shown in Fig 4.28b. The **elapsed_time** clause for a loop is a running time (duration) expression. Therefore, it must be zero before the beginning of the loop. This obligation is captured by adding the *El_Dur_Exp = 0.0* to the first **Confirm** above the line. NQV(Post_w_Dur, **Cum_Dur**) is used to create a new verification variable **Cum_Dur'** for **Cum_Dur**. Inside the loop body, **Cum_Dur'** cannot exceed *El_Dur_Exp*, therefore, NQV(Post_w_Dur, **Cum_Dur**) ≤ El_Dur_Exp needs to be confirmed inside the *if* statement. At

the loop termination, cumulative duration is updated by the adding the **elapsed_time** duration expression.

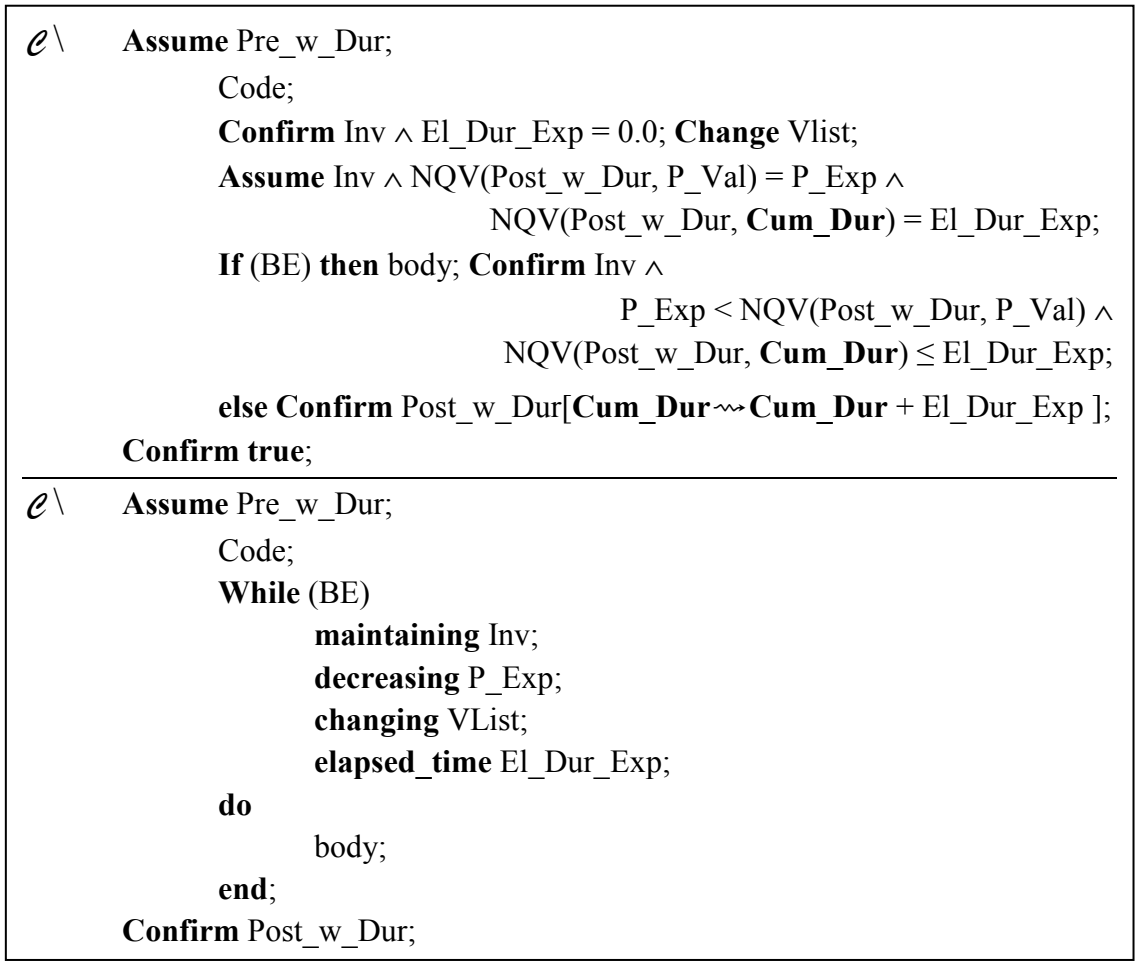

Fig 4.28b: While loop rule for functional specification and performance profile

Fig 4.29a shows functionality specification of *Append_to* operation reproduced from Fig 2.17; and Fig 4.29b shows its performance profile.

```
Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
        requires |P| + |Q| <= Max_Length;
        ensures P = #P o #Q;
```

Fig 4.29a: Functionality specification of Append_to operation

```
Defines C1: RPos;
Defines C2: RPos;

Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
    duration C1 + I_Dur(Entry) + F_IV_Dur(Entry) +
                        |#Q| * (C2 + I_Dur(Entry) +
```

Fig 4.29b: A performance profile for Append_to operation

Fig 4.29c shows one possible realization. The loop annotations are developed as below.

- Loop invariant or **maintaining** clause:
  - P o Q = #P o #Q: Entries are dequeued from *Q* and enqueued in P.
  - **Is_Initial**(Next): Any entry of the array that is not part of the conceptual array is initialed value, therefore, after the call to Enqueue operation, Next entry is clean.
  - Len = |Q|: Before the loop, Len := |Q|; Entries are removed from *Q* and *Len* is decremented, therefore, the two terms are the same.
- Decreasing clause:
  - |Q|: Length of queue is decreasing, therefore, the loop will terminate.

- Elapsed_time invariant:
  - Dequeue, Enqueue, and Decrement operations are called in the loop body and the conjunct C2 + **I_Dur**(Entry) + **F_Dur**(Entry, Next) is the summation of their duration. Before the first iteration, |#Q| is equal to |Q|, therefore, elapsed time is zero before the first iteration of the loop.

```
Definition C1: RPos = Dur_Assgn + CLe + ITP_AreNotEq + I_Dur(Integer) +
                              2*F_IV_Dur(Integer) + Dur_Call(1) + Dur_Call(2);
          Definition C2: RPos = CDq + CEn + ITP_Decr + ITP_AreNotEq +
                                                    Dur_Call(1) + 3*Dur_Call(2);

Procedure Append_to(updates P: P_Queue; clears Q: P_Queue);
        Var Next: Entry;
        Var Len: Integer;

        Len := Length(Q);
        While (Len /= 0)
                maintaining (P o Q = #P o #Q) and
                                          Entry.Is_Initial(Next) and Len = |Q|;
                decreasing |Q|;
                elapsed_time (|#Q| - |Q|) *
                                    (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
        do
                Dequeue(Next, Q);
                Enqueue(Next, P);
                Decrement(Len);
        end;
end Append_to;
```

Fig 4.29c: A realization of Append_to operation to illustrate while loop rule

The automated system generates nineteen VCs for the current example and they are listed in Appendix C. Since, the while loop created two paths, therefore, two of the nineteen VCs

(VC 0_4 and VC 1_4) are shown in Fig 4.30a and Fig 4.30b, respectively. The *Goal* of these two figures corresponds to **Confirm** … *El_Dur_Exp = 0.0* statement of the rule. Both these VCs are provable. Irrespective of the Q size, *(|Q| - |Q|)* will always be zero; it means the left hand side of the *Goal* will reduce to zero, thus proving the *Goal*.

VC 0_4
Base Case of Elapsed Time Duration of While Statement:
Append_Realiz_2.rb(18)

Goal(s):
((((|Q| - |Q|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)

Fig 4.30a: Duration VC for El_Dur_Exp = 0.0 for path-1 for while loop rule

VC 1_4
Base Case of Elapsed Time Duration of While Statement:
Append_Realiz_2.rb(18)

Goal(s):
((((|Q| - |Q|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)

Fig 4.30b: Duration VC for El_Dur_Exp = 0.0 for path-2 for while loop rule rule

VC 0_12 shown in Fig 4.30c corresponds to the duration of the termination of the loop. In this VC, the *Given* #6 is created by the application of *NQV(Post,* **Cum_Dur***) = El_Dur_Exp*. The Goal of the VC is created by the application of *NQV(Post,* **Cum_Dur***) ≤ El_Dur_Exp*; furthermore, **Cum_Dur** is updated to account the duration of the loop body.

VC 0_12
Termination of While Statement: Append_Realiz_2.rb(16)

Goal(s):
((((((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) +
Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) +
F_Dur(Entry, Next'")) + Dur_Call(2))) + (CEn + Dur_Call(2))) +
(ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <=
((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P" o <Next">))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) +
Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) +
F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next'")
10. ((P" o Q") = (P o Q))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)

Fig 4.30c: Duration VC for if part for path-1 of while loop rule

VC 1_7 shown in Fig 4.30d corresponds to the duration of the else part of the loop. In this VC, the right hand side of the *Goal* is retrieved from Fig 4.29b; and the left hand side is mechanically created by the application of variable declaration, function declaration, and function assignment rules. Both VC 0_12 and VC 1_7 are provable. The VC proofs are discussed in Chapter 5.

VC 1_7
Duration Clause of Append_to: Append_Realiz_2.rb(8)

Goal(s):
(((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) +
Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) +
((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) +
(F_Dur(Entry, Next') + F_Dur(Integer, Len'))) <=
(((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) +
(|Q| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (Len' = 0)
2. (Cum_Dur' = ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q'|)
4. (Len' = |Q'|)
5. Entry.Is_Initial(Next')
6. ((P' o Q') = (P o Q))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)

Fig 4.30d: Duration VC for else part for path-2 of while loop rule

4.12. General Rules

This subsection includes more general proof rules for procedure and function declarations, function assignment, and operation call to illustrate how different parameter modes are handled.

4.12.1. Procedure Declaration Rule

Before discussing the generalized version of the procedure declaration rule, consider a general functional specification of an operation *P* presented as *P_Spec* in Fig 4.31a. The operation *P* is not a function operation, so it does not return any value. It uses seven parameters with the parameter modes available in RESOLVE. Since the pre-condition of an operation (*P_Pre*) depends on the incoming values of the parameters, therefore, it is not necessary to show # sign in front of the parameters; also, the parameter *v* is not listed in *P_Pre* because the incoming value is of no importance for **replaces** mode parameter. In the post condition (*P_Post*), *#t* and *t* are included because *t* is passed with **updates** mode and both may appear in it; only *#u*, *#y*, and *#z* are included because *u*, *y*, and *z* are passed with **evaluates, alters** and **clears** modes, respectively, because only their inputs are relevant; only *w* and *x* are included because these parameters are passed with **restores** and **preserves** modes, respectively, and their input and output values are the same.

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

Fig 4.31a: Functional specification of an operation P

The performance description of the operation *P* is presented as *P_Perf* in Fig 4.31b. The **duration** clause indicates the time bound to execute the operation and is written in terms of the operation parameters.

```
Operation P( updates t: T1; evaluates u: T2; replaces v: T3;
                    restores w: T4; preserves x: T5; alters y: T6; clears z: T7);
        duration Dur_Bd_Exp /_ #t, t, #u, v, w, x, #y, #z _\;
```

Fig 4.31b: Performance profile of the operation P

The generalized version of the procedure declaration rule is shown in Fig 4.31c. This version of the rule is more involved because it describes the behavior of the seven parameter modes available in RESOLVE. In contrast to the previous version of the rule, here both above and below the line, *Pre* and *Post* include only the functionality assertions and the duration assertions are given explicitly. In the conjunct **Cum_Dur** + *Sqnt_Dur_Exp* ≤ *Code_Dur_Bd_Exp*, the term *Sqnt_Dur_Exp* represents the subsequent duration expression; it is the duration that the program will need to execute if started at the current statement. This expression is updated automatically by the verifier. The rule adds *P_Spec* and *P_Perf* to $\mathscr{C}$ above the line for both parts of the proof above the line. For the first part, before *P_Code*, constraints on all parameters, except for parameter *v* of type *T3* which is passed in **replaces** mode, are assumed because the input value of *v* is not relevant.

After *P_Code*, a variety of assertions in addition to *P_Post* from the functional specification need to be confirmed. The conjuncts *w* = *#w* and *T7*.**Is_Initial**(z) are added because *w* is passed with the **restores** mode and *z* of type *T7* is passed with the **clears** mode. The duration clause is given as: **Cum_Dur** + *T2*.**F_Dur**(u) ≤ *Dur_Bd_Exp*. This is because the parameter *u* of

type *T2* is passed by the **evaluates** mode and the temporary variable that stores the evaluated value of the caller-supplied expression argument will need to be finalized (or destroyed) at the end of the procedure. In the current version of the rule, *T2*.**F_Dur**(u) is added to **Cum_Dur**; because of where this term is added (after *P_Code*), this term is the duration to finalize the value *u* happens to have at the end of the code.

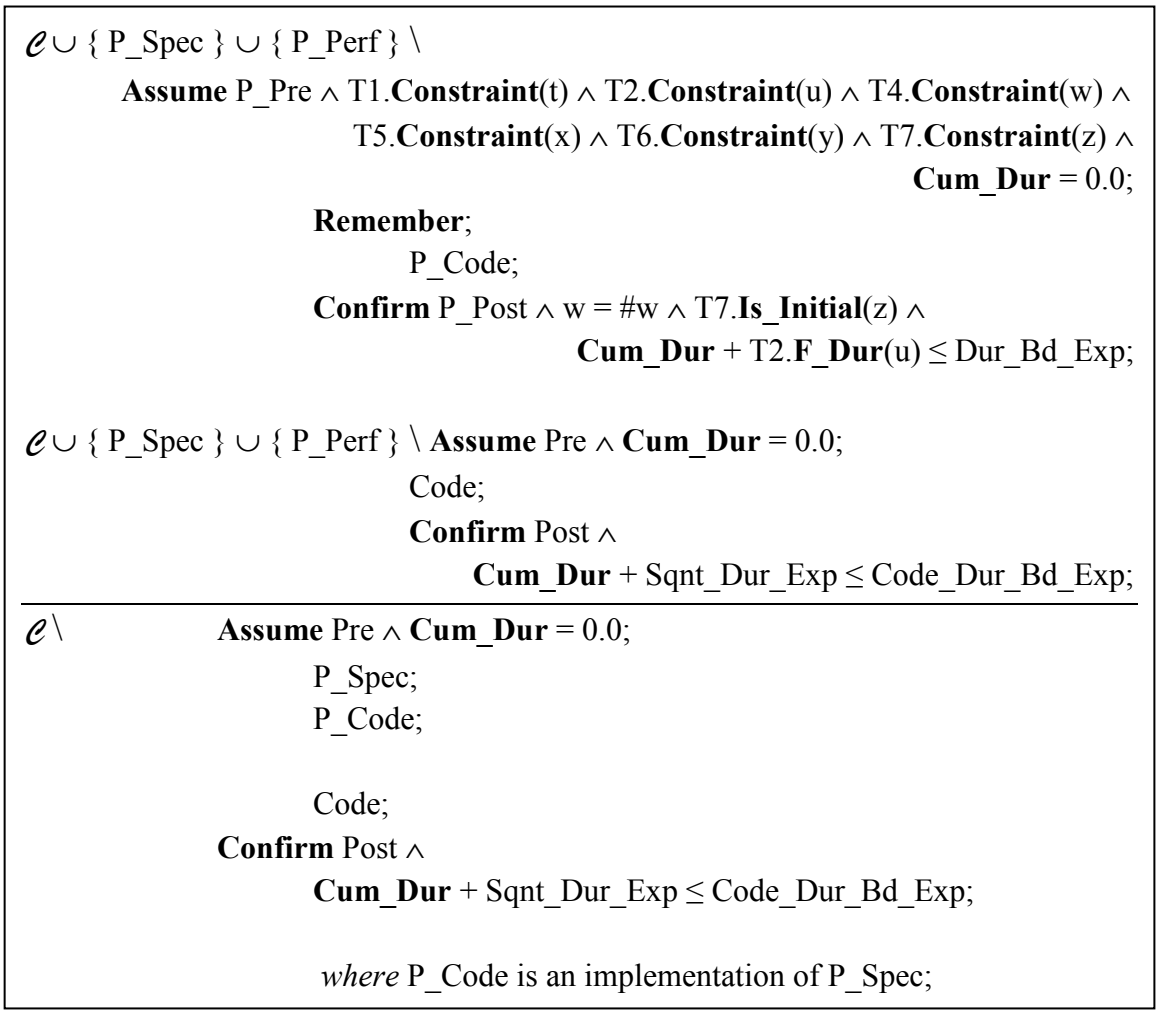

Fig 4.31c: Generalized version of procedure declaration rule

4.12.2. Operation Call Rule

The generalized version of the operation call rule is shown in Fig 4.32. This version of the rule is more involved because it describes the behavior of the seven parameter modes available in RESOLVE. To understand this rule, refer to a general functional specification of an operation *P* shown in Fig 4.31a and its performance description shown in Fig 4.31b.

In the replacement expression for **Cum_Dur**, the term *Dur_Call*(7) represents the duration to call an operation with seven parameters. In the current version of the rule, *T3*.**F_Dur**(#v) is the duration to finalize the incoming value of *v*; because *v* is passed with replaces mode and the arbitrary incoming value must be finalized.

$\mathcal{C} \cup$ { P_Spec } \
Assume Pre_w_Dur;
Code;
Var NPV(Code) := exp;
Confirm P_Pre[t⇝a, u⇝Math(exp), w⇝c, x⇝d, y⇝e, z⇝f];
Assume P_Post([#t⇝a, t⇝ a', #u⇝Math(exp), v⇝b', w⇝c, x⇝d, #y⇝e, #z⇝f]) ∧ T7.**Is_Initial**(f') ;
Confirm Post_w_Dur[a⇝a', b⇝b', e⇝e', f⇝f', **Cum_Dur** ⇝**Cum_Dur** + *Dur_Call*(7) + *T3*.**F_Dur**(#v) + Dur_Bd_Exp[#t⇝a, t⇝ a', #u⇝Math(exp), v⇝b, w⇝c, x⇝d, #y⇝e, #z⇝f]];

$\mathcal{C} \cup$ { P_Spec } \ **Assume** Pre_w_Dur;
Code;
P(a, exp, b, c, d, e, f);
Confirm Post_w_Dur;

Fig 4.32: Generalized version of operation call rule

4.12.3. Function Declaration Rule

The generalized version of the function declaration rule is shown in Fig 4.33c, is based on the context in Fig 4.33a and Fig 4.33b. Function operations in RESOLVE cannot modify their parameters, so only **restores**, **preserves**, and **evaluates** modes are allowed.

```
Operation F (evaluates u: T1; restores w: T2; preserves x: T3): T4;
    requires F_Pre /_ u, w, x _\;
    ensures F = F_Post( #u, w, x);
```

Fig 4.33a: Functional specification of an operation F

```
Operation F (evaluates u: T1; restores w: T2; preserves x: T3): T4;
    duration Dur_F_Exp /_ #u, x, w _\;
```

Fig 4.33b**: Performance specification of the operation F**

$\mathcal{C} \cup \{ F\_Spec \} \cup \{ F\_Perf \} \setminus$

Assume F_Pre $\wedge$ T1.*Constraint*(u) $\wedge$ T2. *Constraint* (w) $\wedge$
T3. *Constraint* (x) $\wedge$ **Cum_Dur** = 0.0;

Remember;
Var F: T4;
F_Code;

Confirm F = F_Post $\wedge$ w = #w $\wedge$
Cum_Dur + T1.**F_Dur**(u) $\leq$ Dur_Bd_Exp;

$\mathcal{C} \cup \{ F\_Spec \} \cup \{ F\_Perf \} \setminus$ **Assume** Pre $\wedge$ **Cum_Dur** = 0.0;
Code;
Confirm Post $\wedge$
Cum_Dur + Sqnt_Dur_Exp $\leq$ Code_Dur_Bd_Exp;

$\mathcal{C} \setminus$ **Assume** Pre $\wedge$ **Cum_Dur** = 0.0;
F_Spec;
F_Code;
Code;

Confirm Post $\wedge$
Cum_Dur + Sqnt_Dur_Exp $\leq$ Code_Dur_Bd_Exp;

where F_Code is an implementation of F_Spec;

Fig 4.33c: Generalized version of function declaration rule

4.12.4. Function Assignment Rule

The generalized version of function assignment rule is shown in Fig 4.34. To understand this rule, refer to a general functional specification of an operation *P* shown in Fig 4.33a and its performance description shown in Fig 4.33b. This version of the rule describes the behavior of the three parameter's modes allowed for function operations in RESOLVE.

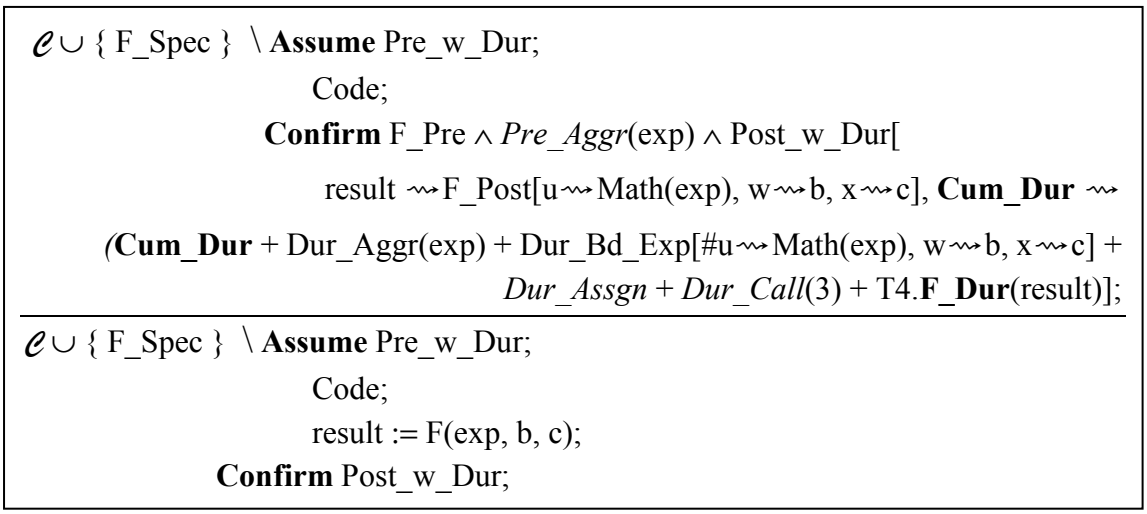

Fig 4.34: Generalized version of function assignment rule

4.13. Soundness and Relative Completeness

Soundness and completeness are two important factors of a mechanical verification system. Soundness is defined as follows: If a verification system proves that the code is correct then it is valid code. On the other hand, completeness is defined as if the code is valid then the verification system will prove that it is correct. Using normal and three stuck states (manifestly wrong, vacuously correct, and bottom), Harton and Krone describe in detail program semantics, soundness and completeness of the verification system [Harton11, Krone 88].

A normal state is defined as a state from which execution can move to another state; whereas a stuck state is defined as a state that program cannot come out. Manifestly Wrong (**MW**) state means that the code is incorrect; that is, if the pre-condition of a called operation is violated in a code or if one of the invariants, such as the loop invariant is wrong, then the calling code will result in **MW** state (that is, it is invalid). Vacuously Correct (**VC**) state means that the code is correct by false assumptions; that is, if its pre-condition is violated or if the post-condition of a

called operation is not satisfied, then the calling code will result in **VC** state (that is, it is trivially valid). The bottom state is entered when the code does not terminate.

The above four states are based on functionality specifications. For duration correctness, we need to introduce two more stuck states. Reaching Manifestly Wrong Performance (**MWP**) state means that a procedure does not satisfy its duration bounds or that one of the internal duration assertions, such as elapsed time assertion is wrong. Reaching this state essentially means that the program is wrong because of performance violation. A program may reach Vacuously Correct Performance (**VCP**) state, if a called operation does not satisfy its duration bounds, that is, the calling code is trivially correct because of a performance violation elsewhere.

A formal performance semantic analysis of the rules (including both time and space bounds) is beyond the scope of this dissertation and is left for future work.

CHAPTER 5.

RESULTS FROM EXPERIMENTATION

5.1. Introduction

The case studies in this chapter are used to illustrate the usage of functional specifications, performance profiles, and realizations for generating performance verification conditions. The same rules are employed for both typically "built-in" types such as *Integer* defined in *Integer_Template* concept and programmer created concepts, such as *Stack_Template* and *Preembtable_Queue_Template*. The overall goal of this chapter is the evaluation of the proposed performance verification system using a prototype—the third research deliverable.

Each case study is further subdivided into several sub-cases (or enhancements). As we progress from the simplest examples to more complex ones, we illustrate different aspects of software verification, including variable declaration, operation calls, function assignment statements, nested if-else branching, and recursive and iterative procedures. These cases do not cover all programming features; however, these examples highlight many points made throughout the dissertation and the proof rules discussed in Chapter 4.

Generation of functional correctness VCs corresponding to the examples in this section are discussed in [Harton 2013]; most of those VCs are dispatched automatically by the minimal-

ist rewriting prover developed in [Smith 2013]. But automatically generated duration-related VCs are not yet proved automatically. So this section includes manual proofs of them.

5.2. Integer_Template Enhancements

Integer_Template is one of the built-in concepts in RESOLVE, and it describes the data type Integer and operations. This section will discuss VC generation and proof of the VCs of two enhancements for *Integer_Template*: Halving_Capability and Int_Do_Nothing_Capability. In order to understand an enhancement, it is important to understand the parent component. Fig 5.1a shows the functional specification and Fig 5.1b shows a performance profile (ITP) of the parent component. These figures only show the operations used in the current examples. The complete listing is shown in Appendix B.

```
Concept Integer_Template;
    uses Integer_Theory;

    Definition min_int: Z;
    Definition 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;
    end;

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

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

Fig. 5.1a: A functional specification of Integer_Template

```
Profile ITP short_for Int_Tmplt_Prfl for Integer_Template;
    uses Duration_Basics_Theory;

    Defines ITP_Incr: RPos;
    Defines ITP_Decr: RPos;
    Defines ITP_Gr: RPos;
    Defines ITP_GrOrEq: RPos;
    Defines ITP_Div: RPos;

    Type Family Integer is modeled by Z;
        initialization
            duration ITP_Init;
        finalization
            duration ITP_Flz;
    end;

    Operation Increment(updates i: Integer);
        duration ITP_Incr;

    Operation Decrement(updates i: Integer);
        duration ITP_Decr;

    Operation Greater(evaluates i, j: Integer): Boolean;
        duration ITP_Gr;

    Operation Div(evaluates i, j: Integer): Integer;
        duration ITP_Div;
…
end ITP;
```

Fig. 5.1b: ITP – A performance profile for Integer_Template

5.2.1. Halve

***Problem Description*:** *Develop a function to divide an Integer by 2 and generate VCs to prove its correctness*.

Fig 5.2a shows a possible functionality specification of an enhancement *Halving_Capability* satisfying the halving problem. The specification contains an operation *Halve*. This operation takes one Integer parameter *I* in the **evaluates** mode. The **ensures** clause states that the resulting integer (Halve) is equal to the half of the incoming integer (#I). Fig 5.2b shows its performance profile. Fig 5.2c shows one possible realization.

```
Enhancement Halving_Capability for Integer_Template;
    uses Integer_Theory;

    Operation Halve(evaluates I: Integer): Integer;
        ensures Halve = #I/2;

end Halving_Capability;
```

Fig. 5.2a: Functionality specification of Halving_Capability

```
Profile HP short_for Halving_Profile for Halving_Capability for
                                        Integer_Template with_profile ITP;

    Defines C: RPos;
    Operation Halve(evaluates I: Integer): Integer;
        duration C;
end HP;
```

Fig. 5.2b: Performance profile of Halve operation

```
Realization Halving_Realiz with_profile HP for
                                    Halving_Capability of Integer_Template;

    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C: RPos = I_Dur(Integer) + 2*ITP_Incr + Dur_Assgn +
              ITP_Div + 2*Dur_Call(1) + Dur_Call(2) + 4*F_Dur_Integer;

    Procedure Halve(evaluates I: Integer): Integer;
        Var J: Integer;

        Increment(J);
        Increment(J);
        Halve := Div(I, J);

    end Halve;
end Halving_Realiz;
```

Fig. 5.2c: Realization of the of Halve operation

VC 0_5 is for the duration bound is shown in Fig 5.3. The complete listing of the assertions is shown in Appendix C. The term in the right hand side (RHS) of the *Goal* inequality represents the duration clause retrieved from the profile shown in Fig 5.2b. The left had side (LHS) of the *Goal* is derived mechanically for the code shown in Fig 5.2c. The *Goal* of the VC shown in Fig 5.3 can be proved as shown in Table 5.1.

VC 0_5
Duration Clause of Halve: Halving_Realiz.rb(5)

Goal(s):
((((((((((Cum_Dur + I_Dur(Integer)) + (ITP_Incr + Dur_Call(1))) + (ITP_Incr + Dur_Call(1))) + (ITP_Div + Dur_Call(2))) + F_Dur(Integer, I)) + F_Dur(Integer, J')) + Dur_Assgn) + F_Dur(Integer, Halve)) + F_Dur(Integer, J')) <= C)

Given(s):
1. (J' = (J'' + 1))
2. (J'' = (J + 1))
3. (J = 0)
4. (Cum_Dur = 0.0)

Fig 5.3: Verification condition of duration correctness for Halving_Ver2 operation

Table 5.1: Proof of duration VC 0_5 for Halve operation

LHS	((((((((((Cum_Dur + I_Dur(Integer)) + (ITP_Incr + Dur_Call(1))) + (ITP_Incr +Dur_Call(1))) + (ITP_Div + Dur_Call(2))) + F_Dur(Integer, I)) + F_Dur(Integer, J')) + Dur_Assgn) + F_Dur(Integer, Halve)) + F_Dur(Integer, J'))	Remove extra paran-thesis and rearrange terms
	Cum_Dur + I_Dur(Integer) + 2*ITP_Incr + 2*Dur_Call(1) + ITP_Div + Dur_Call(2) + F_Dur(Integer, I) + 2*F_Dur(Integer, J') + Dur_Assgn + F_Dur(Integer, Halve)	Apply Given #1 and Apply Given #4
	0.0 + I_Dur(Integer) + 2*ITP_Incr + 2*Dur_Call(1) + ITP_Div + Dur_Call(2) + F_Dur_Integer + 2* F_Dur_Integer + Dur_Assgn + F_Dur_Integer	Simplify
	I_Dur(Integer) + 2*ITP_Incr + 2*Dur_Call(1) + ITP_Div + Dur_Call(2) + Dur_Assgn + 4* F_Dur_Integer	Apply definition from Fig 5.2c
	C	= RHS
	Hence proved.	

5.2.2. Do nothing

***Problem Description*:** *Develop a procedure that restores an Integer and generate VCs to prove its correctness. Use the procedure to illustrate nested if-then construct verification.*

Fig 5.4a shows a possible functionality specification of an enhancement *Do_Nothing_Capability* satisfying the above problem. The specification contains an operation *Do_Nothing*. This operation takes one Integer parameter *I* in the **restores** mode. The **requires** clause states that the incremented integer can at the most be as large as the maximum integer. Fig 5.4b shows its performance profile. Notice that, there is no explicit **ensures** clause, because the parameter is in the **restores** mode which implies that it ensures I = #I.

```
Enhancement Int_Do_Nothing_Capability for Integer_Template;

    Operation Do_Nothing(restores I: Integer);
        requires (I + 1 <= max_int);

end Int_Do_Nothing_Capability;
```

Fig. 5.4a: Functionality specification of Int_Do_Nothing_Capability

```
Profile IDNP_2 short_for Int_Do_Nothing_Profile for
          Int_Do_Nothing_Capability for Integer_Template with_profile ITP;

    Defines C: RPos;
    Operation Do_Nothing(restores I: Integer);
        duration C;
end IDNP_2;
```

Fig. 5.4b: Performance profile of Do_Nothing operation

The ideal realization code for *Do_Nothing* would indeed be no code at all. However, for the purposes of illustration, consider a non-trivial realization code for the *Do_Nothing* operation given in Fig 5.4c; this realization satisfies its functionality specification (Fig 5.4a) and performance profile (Fig 5.4b). The code contains two nested *if* statements, therefore, there are three execution paths (I = 0; I /= 0 and I > J; and I /= 0 and I < J).

```
Realization Int_Do_Nothing_Realiz_2 with_profile IDNP_2
                    for Int_Do_Nothing_Capability of Integer_Template;

    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C: RPos = I_Dur(Integer) + ITP_Zero + Dur_Swap +
                        ITP_Gr + ITP_Incr + ITP_Decr + 3*Dur_Call(1) +
                                        Dur_Call(2) + F_IV_Dur(Integer);

    Procedure Do_Nothing(restores I: Integer);
        Var J: Integer;

        If (Is_Zero(I)) then
            I :=: J;
        else
            If (Greater(I, J)) then
                Increment(I);
                Decrement(I);
            end;
        end;
    end Do_Nothing;
end Int_Do_Nothing_Realiz_2;
```

Fig. 5.4c: Realization of the of Do_Nothing operation

Since, the if-statements created three paths, therefore, three of the eight VCs (VC 0_2, VC 1_4, and VC 2_2) are related to the duration bound of the three paths and are shown in Fig 5.5a, Fig 5.5b, and Fig 5.5c, respectively. The complete listing is available in Appendix C.

The *Goal* of the VC 1_4 shown in Fig 5.5b can be proved as shown in Table 5.2. Similarly, the other two VCs can be proved.

```
VC 0_2
Duration Clause of Do_Nothing: Int_Do_Nothing_Realiz_2.rb(5)

Goal(s):
(((((Cum_Dur + I_Dur(Integer)) + Dur_Swap) +
                (ITP_Zero + Dur_Call(1))) + F_Dur(Integer, I)) <= C)

Given(s):
1. (I = 0)
2. (J = 0)
3. (Cum_Dur = 0.0)
```

Fig 5.5a: Verification condition, 0_2, of duration correctness for Do_Nothing operation

```
VC 1_4
Duration Clause of Do_Nothing: Int_Do_Nothing_Realiz_2.rb(5)

Goal(s):
(((((((Cum_Dur + I_Dur(Integer)) + (ITP_Incr + Dur_Call(1))) +
              (ITP_Decr + Dur_Call(1))) + (ITP_Gr + Dur_Call(2))) +
                (ITP_Zero + Dur_Call(1))) + F_Dur(Integer, J)) <= C)

Given(s):
1. (I' = (I" - 1))
2. (I" = (I + 1))
3. (I > J)
4. (I /= 0)
5. (J = 0)
6. (Cum_Dur = 0.0)
```

Fig 5.5b: Verification condition, 0_4, of duration correctness for Do_Nothing operation

VC 2_2
Duration Clause of Do_Nothing: Int_Do_Nothing_Realiz_2.rb(5)

Goal(s):
(((((Cum_Dur + I_Dur(Integer)) + (ITP_Gr + Dur_Call(2))) + (ITP_Zero + Dur_Call(1))) + F_Dur(Integer, J)) <= C)

Given(s):
1. not((I > J))
2. (I /= 0)
3. (J = 0)
4. (Cum_Dur = 0.0)

Fig 5.5c: Verification condition, 2_2, of duration correctness for Do_Nothing operation

Table 5.2: Proof of duration VC 1_4 for Do_Nothing operation

LHS	(((((((Cum_Dur + I_Dur(Integer)) + (ITP_Incr + Dur_Call(1))) + (ITP_Decr + Dur_Call(1))) + (ITP_Gr + Dur_Call(2))) + (ITP_Zero + Dur_Call(1))) + F_Dur(Integer, J))	Remove extra paranthesis and rearrange terms
	Cum_Dur + I_Dur(Integer) + ITP_Incr + 3*Dur_Call(1) + ITP_Decr + ITP_Gr + Dur_Call(2) + ITP_Zero + F_Dur(Integer, J)	*Apply Given #5 and* Apply Given #6
	0.0 + I_Dur(Integer) + ITP_Incr + 3*Dur_Call(1) + ITP_Decr + ITP_Gr + Dur_Call(2) + ITP_Zero + F_IV_Dur(Integer)	Rearrange terms
	I_Dur(Integer) + ITP_Zero + ITP_Gr + ITP_Incr + ITP_Decr + 3*Dur_Call(1) + Dur_Call(2) + F_IV_Dur(Integer)	*Apply definition from Fig 5.4c*
	< C	*< RHS*
	Hence proved.	

5.3. Stack_Template Enhancements

Stack_Template is an example of a user-defined object-based concept in RESOLVE. This section will discuss VC generation and proof of the VCs of an enhancement (Flipping_Capability) for *Stack_Template*. The parent (*Stack_Template*) component on which the enhancement is based is a bounded stack and its formal functional specification is named as *Stack_Template*. Fig 5.6a shows the functional specification and Fig 5.6b shows a performance profile *SSC* for the space-conscious class of stack implementations.

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

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

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

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

    Operation Depth(restores S: Stack): Integer;
            ensures Depth = (|S|);
        …
end Stack_Template;
```

Fig. 5.6a: A functional specification of Stack_Template

```
Profile SSC short_for Space_Conscious for Stack_Template;
    uses Real_Number_Theory, Duration_Basics_Theory;

    Defines SSCI1: RPos;
    Defines SSCI2: RPos;
    Defines SSCF1: RPos;
    Defines SSCF2: RPos;
    Defines SSCPu: RPos;
    Defines SSCPo: RPos;
    Defines SSCDp: RPos;

    Definition Cnts_Dur(s : Str(Entry)) : RPos;
    Type Family Stack is modeled by Str( Entry );
        initialization
            duration SSCI1 + (SSCI2 + I_Dur(Entry)) * Max_Depth;

        finalization
            duration  SSCF1 + Cnts_Dur( #S) +
                  ( SSCF2 + F_IV_Dur(Entry) ) * ( Max_Depth - |#S| );
    end;

    Operation Push( clears E: Entry; updates S: Stack );
        duration  SSCPu;

    Operation Pop( replaces E: Entry; updates S: Stack );
        duration  ( SSCPo + I_Dur(Entry) ) + F_Dur(Entry, #E);

    Operation Depth( restores S: Stack ): Integer;
        duration  SSCDp;
…
end SSC;
```

Fig. 5.6b: SSC – A performance profile for Stack_Template

5.3.1. Flip

Problem Description**:** *Using a stack concept that is generic and parameterized by type of entries, generate VC for an operation that reverses a given stack.*

Fig 5.7a shows a possible functionality specification of an enhancement *Flipping_Capability* satisfying the above. *Flip* operation takes one stack parameter S in the **updates** mode. The **ensures** clause states that the resulting stack object S is equal to the reverse of the incoming Stack. Fig 5.7b shows performance profile for *Flip* operation. Fig 5.7c shows one possible realization that satisfy the functionality specification (Fig 5.7a) and performance profile (Fig 5.7b).

```
Enhancement Flipping_Capability for Stack_Template;

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

end Flipping_Capability;
```

Fig. 5.7a: Functionality specification of Flipping_Capability

```
Profile SSCFP short_for Space_Concious_Stack_Flip_Profile for
                        Flipping_Capability for Stack_Template with_profile SSC;
    Defines C1: RPos;
    Defines C2: RPos;

    Operation Flip(updates S: Stack);
        duration   C1 + I_Dur(Entry) + F_IV_Dur(Entry) + I_Dur(Stack) +
                F_IV_Dur(Stack) + |#S| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
end SSCFP;
```

Fig. 5.7b: Performance profile of Flip operation

```
Realization Obvious_Flip_Realiz with_profile SSCFP for
                                        Flipping_Capability of Stack_Template;

    uses ITP;
    uses Integer_Template;

    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C1: RPos = I_Dur(Integer) + Dur_Assgn + SSCDp +
                                    Dur_Swap + ITP_AreNotEq + Dur_Call(1) +
                                Dur_Call(2) + F_IV_Dur(Integer) + F_Dur_Integer;
    Definition C2: RPos = SSCPo + SSCPu + ITP_Decr +
                                ITP_AreNotEq + Dur_Call(1) + 3*Dur_Call(2);

    Procedure Flip(updates S: Stack);
        Var Next: Entry;
        Var S_Flipped: Stack;
        Var Dep: Integer;

        Dep := Depth(S);
        While (Dep /= 0)
            maintaining #S = Reverse(S_Flipped) o S and
                                            Entry.Is_Initial(Next) and Dep = |S|;
            decreasing |S|;
            elapsed_time  |S_Flipped| *
                                        (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
        do
            Pop(Next, S);
            Push(Next, S_Flipped);
            Decrement(Dep);
        end;
        S_Flipped :=: S;
    end Flip;
end Obvious_Flip_Realiz;
```

Fig. 5.7c: Realization of the of Flip operation

In the current example, the *Flip* operation is implemented using an iterative algorithm. The conjunct *#S = Reverse(S_Flipped) o S* in the **maintaining** clause of the loop invariants is developed as follow: (i) entries are removed from the incoming *S* and are entered in *S_Flipped*; and (ii) the entries in *S_Flipped* are in reverse order of *#S*. The **elapsed_time** clause is developed as follows: (i) the loop body calls *Dequeue*, *Enqueue*, and *Decrement* operations; therefore, the conjunct C2 + **I_Dur**(Entry) + **F_IV_Dur**(Entry) is the summation of their duration; and (ii) the loop is executed |*S_Flipped*| time. Before the first iteration, |*S_Flipped*| is zero, therefore, elapsed time is zero before the first iteration of the loop.

The complete listing of VCs is shown in Appendix C. Since, the while loop rule created two paths, therefore, two of the eighteen VCs (VC 0_4 and VC 1_4) are related to the duration bound for the elapsed time in the beginning of the while loop and are shown in Fig 5.8a and Fig 5.8b, respectively. The *Goal* of the VC 0_4 shown in Fig 5.8c can be proved as shown in Table 5.3.

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Obvious_Flip_Realiz.rb(20)

Goal(s):
((|S_Flipped| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)

Fig 5.8a: Verification condition, 0_4, of duration correctness for Flip operation

Table 5.3: Proof of duration VC 0_4 for Flip operation

LHS	((|S_Flipped| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))	Remove extra paran-thesis
	|S_Flipped| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Apply Given #2
	|Empty_String| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Using String_Theory |Empty_String| = 0
	(0) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Simplify
	0.0	= RHS
	Hence proved.	

VC 1_4
Base Case of Elapsed Time Duration of While Statement: Obvi-ous_Flip_Realiz.rb(20)

Goal(s):
((|S_Flipped| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)

Fig 5.8b: Verification condition, 1_4, of duration correctness for Flip operation

Two of the eighteen VCs (VC 0_12 and VC 1_6) are related to the duration bound for the while loop termination and are shown in Fig 5.8c and Fig 5.8d, respectively. The *Goal* of the VC 0_12 shown in Fig 5.8c can be proved as shown in Table 5.4.

```
VC 0_12
Termination of While Statement: Obvious_Flip_Realiz.rb(19)

Goal(s):
((((((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) +
   (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) + (((SSCPo +
                I_Dur(Entry)) + F_Dur(Entry, Next'")) + Dur_Call(2))) + (SSCPu +
 Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2)))
                     <= (|S_Flipped'| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (Dep' = (Dep" - 1))
2. Entry.Is_Initial(Next')
3. (S_Flipped' = (<Next"> o S_Flipped"))
4. (S" = (<Next"> o S'))
5. (Dep" /= 0)
6. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) +
                   (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) =
                         (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |S"|)
8. (Dep" = |S"|)
9. Entry.Is_Initial(Next'")
10. (S = (Reverse(S_Flipped") o S"))
11. (Dep = 0)
12. (S_Flipped = Empty_String)
13. Entry.Is_Initial(Next)
14. (Cum_Dur = 0.0)
```

Fig 5.8c: Verification condition, 0_12 , of duration correctness for Flip operation

Table 5.4: Proof of duration VC 0_12 for Flip operation

LHS	((((((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) + (((SSCPo + I_Dur(Entry)) + F_Dur(Entry, Next'")) + Dur_Call(2))) + (SSCPu + Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2)))	Remove extra paran-thesis
	Cum_Dur' + I_Dur(Entry) + I_Dur(Stack) + I_Dur(Integer) + SSCDp + Dur_Call(1) + Dur_Assgn + F_Dur(Integer, Dep) + SSCPo + I_Dur(Entry) + F_Dur(Entry, Next'") + Dur_Call(2) + SSCPu + Dur_Call(2) + ITP_Decr + Dur_Call(1) + ITP_AreNotEq + Dur_Call(2)	Apply Given #6
	|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)) + SSCPo + I_Dur(Entry) + F_Dur(Entry, Next'") + Dur_Call(2) + SSCPu + Dur_Call(2) + ITP_Decr + Dur_Call(1) + ITP_AreNotEq + Dur_Call(2)	Apply Given #9 and combine terms
	|S_Flipped"| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry)) + SSCPo + I_Dur(Entry) + F_IV_Dur(Entry) + SSCPu + ITP_Decr + Dur_Call(1) + ITP_AreNotEq + 3*Dur_Call(2)	Rearrange
	|S_Flipped"| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry)) + SSCPo + SSCPu + ITP_Decr + ITP_AreNotEq + Dur_Call(1) + 3*Dur_Call(2) + I_Dur(Entry) + F_IV_Dur(Entry)	Using definition of C2 from Fig 5.7c
	|S_Flipped"| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry)) + C2 + I_Dur(Entry) + F_IV_Dur(Entry)	Simplify
	(|S_Flipped"| + 1) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Using |<α>| = 1 and Apply Given #3
	|S_Flipped'| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	= RHS
	Hence proved.	

```
VC 1_6
Duration Clause of Flip: Obvious_Flip_Realiz.rb(10)

Goal(s):
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) +
                    (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) +
          (ITP_AreNotEq + Dur_Call(2))) + (|S_Flipped'| * ((C2 + I_Dur(Entry)) +
                    F_IV_Dur(Entry)))) + Dur_Swap) + ((F_Dur(Entry, Next') +
                                        F_Dur(Stack, S')) + F_Dur(Integer, Dep'))) <=
                    (((((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) + I_Dur(Stack)) +
          F_IV_Dur(Stack)) + (|S| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (Dep' = 0)
2. (Cum_Dur' = (|S_Flipped'| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |S'|)
4. (Dep' = |S'|)
5. Entry.Is_Initial(Next')
6. (S = (Reverse(S_Flipped') o S'))
7. (Dep = 0)
8. (S_Flipped = Empty_String)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
```

Fig 5.8d: Verification, 1_6, condition of duration correctness for Flip operation

5.4. Preemptable_Queue_Template Enhancements

Preemptable_Queue_Template is another example of a user-defined object-based concept. This section will discuss VC generation and proof of the VCs of three enhancements Appending_Capability, Rotating_Capabillity, and Copying_Capabillity. The parent component's functional specification and performance profile are discussed in great detail in Chapter 2 and Chapter 3, respectively.

5.4.1. Append

***Problem Description*:** *Using preemptable queue that is generic and parameterized by type of entries in a queue, generate VC for an operation that merges two queues recursively.*

Fig 5.9a shows a possible functionality specification of an enhancement *Appending_Capability* (reproduced from Fig 2.17) satisfying the append problem. Fig 5.9b shows its performance profile.

```
Enhancement Append_Capability for Preemptable_Queue_Template;

    Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
        requires |P| + |Q| <= Max_Length;
        ensures P = #P o #Q;

end Append_Capability;
```

Fig. 5.9a: Functionality specification of Append_Capability

```
Profile Apnd_Profile_3 short_for Append_Profile for Append_Capability
            for Preemptable_Queue_Template with_profile PQ_CT_Profile;

    uses Integer_Theory;
    Defines C: RPos;
    Operation Append_to(updates P: P_Queue; clears Q: P_Queue);
        duration |#Q| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry));
end Apnd_Profile_3;
```

Fig. 5.9b: Performance profile of Append_to operation

Fig 5.9c shows one possible realization. The performance profile and the realization shown in this chapter are different then corresponding files (Fig 4.21b and Fig 4.21c), because in the current example, the *Append_to* operation is implemented using recursion. Since elements

are removed from Q using Dequeue operation, therefore, its length is decremented in each iteration. Hence, the **decreasing** clause can be proved and recursion can be shown to terminate.

```
Realization Append_Realiz_3 with_profile Apnd_Profile_3 for
                        Append_Capability of Preemptable_Queue_Template;

    uses ITP;
    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C: RPos = CLe + Dur_Assgn + ITP_AreNotEq + CDq +
                                        CEn + Dur_Call(1) + 4*Dur_Call(2) +
                        I_Dur(Integer) + F_IV_Dur(Integer) + F_Dur_Integer;

    Procedure Append_to(updates P: P_Queue; clears Q: P_Queue);
        decreasing |Q|;

        Var Next: Entry;
        Var Len: Integer;

        Len := Length(Q);
        if (Len /= 0) then
            Dequeue(Next, Q);
            Enqueue(Next, P);
            Append_to(P, Q);
        end;
    end Append_to;
end Append_Realiz_3;
```

Fig. 5.9c: A recursive realization of the of Append_to operation

The if-else pair in the realization created two paths, therefore, two of the ten VCs (VC 0_7 and VC 1_3) are related to the duration bound of the two paths and are shown in Fig 5.10a and Fig 5.10b, respectively. The complete listing is shown in Appendix C. The *Goal* of VC 0_7 shown in Fig 5.10a can be proved as shown in Table 5.5. Similarly, the *Goal* of VC 1_3 shown in Fig 5.10b can be proved.

VC 0_7
Duration Clause of Append_to: Append_Realiz_3.rb(8)

Goal(s):

```
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) +
                    Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) +
                    F_Dur(Entry, Next)) + Dur_Call(2))) + (CEn + Dur_Call(2))) +
  ((|Q"| * ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))) + Dur_Call(2))) +
                    (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next') +
                                                    F_Dur(Integer, |Q|))) <=
                    (|Q| * ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))))
```

Given(s):

1. (Q' = Empty_String)
2. (P' = (P" o Q"))
3. Entry.Is_Initial(Next')
4. (P" = (P o <Next">))
5. (Q = (<Next"> o Q"))
6. (|Q| /= 0)
7. (P_val = |Q|)
8. (Len = 0)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)

Fig 5.10a: Verification condition, 0_7, of duration correctness for Append operation

Table 5.5: Proof of duration VC 0_7 for Append operation

LHS	((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next)) + Dur_Call(2))) + (CEn + Dur_Call(2))) + ((|Q''| * ((C + (2*I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))) + Dur_Call(2))) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next') + F_Dur(Integer, |Q|)))	Remove extra paren-thesis
	Cum_Dur + I_Dur(Entry) + I_Dur(Integer) + CLe + Dur_Call(1) + Dur_Assgn + F_Dur(Integer, Len) + CDq + I_Dur(Entry) + F_Dur(Entry, Next) + Dur_Call(2) + CEn + Dur_Call(2) + |Q''| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry)) + Dur_Call(2) + ITP_AreNotEq + Dur_Call(2) + F_Dur(Entry, Next') + F_Dur(Integer, |Q|)	Apply Given #10 and rearrange and combine terms
	0.0 + I_Dur(Integer) + CLe + Dur_Call(1) + Dur_Assgn + F_Dur(Integer, Len) + CDq + 2*I_Dur(Entry) + F_Dur(Entry, Next) + CEn + 4*Dur_Call(2) + |Q''| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry)) + ITP_AreNotEq + F_Dur(Entry, Next') + F_Dur(Integer, |Q|)	Apply Given #3 and #9; #8 and #6 and definition from Fig 5.9c,
	0.0 + I_Dur(Integer) + CLe + Dur_Call(1) + Dur_Assgn + F_IV_Dur(Integer) + CDq + 2*I_Dur(Entry) + F_IV_Dur(Entry) + CEn + 4*Dur_Call(2) + |Q''| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry)) + ITP_AreNotEq + F_IV_Dur(Entry) + F_Dur_Integer	Simplify and Rearrange
	CLe + Dur_Assgn + ITP_AreNotEq + CDq + CEn + Dur_Call(1) + 4*Dur_Call(2) + I_Dur(Integer) + F_IV_Dur(Entry) + F_Dur_Integer + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry) + |Q''| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry))	Apply definition from Fig 5.2c
	C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry) + |Q''| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry))	Simplify
	(|Q''| + 1) * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry))	Using |<α>| = 1 and applying Given #5
	|Q'| * (C + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry))	= RHS
	Hence proved.	

VC 1_3
Duration Clause of Append_to: Append_Realiz_3.rb(8)

Goal(s):

(((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next) + F_Dur(Integer, |Q|))) <= (|Q| * ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))))

Given(s):

1. (|Q| = 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)

Fig 5.10b: Verification condition of duration correctness for Append operation

Table 5.6: Proof of duration VC 1_3 for Append operation

LHS	(((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next) + F_Dur(Integer, |Q|)))	Remove extra parenthesis
	Cum_Dur + I_Dur(Entry) + I_Dur(Integer) + CLe + Dur_Call(1) + Dur_Assgn + F_Dur(Integer, Len) + ITP_AreNotEq + Dur_Call(2) + F_Dur(Entry, Next) + F_Dur(Integer, |Q|)	Apply Given #3, #4 and #5; and definition from Fig 5.9c,
	0.0 + I_Dur(Entry) + I_Dur(Integer) + CLe + Dur_Call(1) + Dur_Assgn + F_IV_Dur(Integer) + ITP_AreNotEq + Dur_Call(2) + F_IV_Dur(Entry) + F_Dur_Integer	Simplify and Rearrange
	(CLe + Dur_Call(1) + Dur_Assgn + ITP_AreNotEq + Dur_Call(2) + Dur_Call(1) + I_Dur(Integer) + F_IV_Dur(Integer) + F_Dur_Integer)) + I_Dur(Entry) + F_IV_Dur(Entry)	< RHS
	Hence proved.	

5.4.2. Rotate

***Problem Description*:** *Using preemptable queue that is generic and parameterized by type of entries in a queue, generate VC for an operation that moves n entries from the front of a queue to the back of a second queue.* That is, the resulting queue (Q) is obtained by moving *n* elements to its back.

Fig 5.11a shows a possible functionality specification of an enhancement *Rotating_Capability* satisfying the **Rotate** problem. The specification contains an operation *Rotate*. This operation takes one P_Queue parameter *Q* in the **updates** mode and an Integer *n* in the **evaluates** mode. The *Rotate* operation is written using *Prt_Btwn* definition from String_Theory. Fig 5.11b shows two substrings *Prt_Btwn(0, n, #Q)* and *Prt_Btwn(n, |#Q|, #Q)*. Note that *0* is before the beginning and *n* is after the end of first substring. Since *n* entries are removed from the front of a queue and entered at its back; therefore, the **requires** clause states that minimum value of *n* is *0* and maximum value is the length of the source string. The **ensures** clause states that the resulting object *Q* is equal to the concatenation of two substrings of the incoming queue. The second substring is the string of n entries from the front of the incoming queue and the first substring is the string of the remaining entries.

```
Enhancement Rotating_Capability for Preemptable_Queue_Template;

    Operation Rotate (updates Q: P_Queue; evaluates n: Integer);
        requires 0 <= n <= |Q|;
        ensures Q = Prt_Btwn(n, |#Q|, #Q) o Prt_Btwn(0, n, #Q);

end Rotating_Capability;
```

Fig. 5.11a: Functionality specification of Rotating_Capability

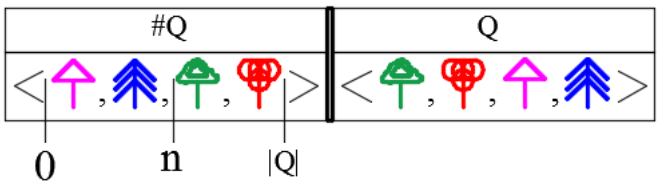

Fig. 5.11b: Rotating a queue

Fig 5.11c shows performance profile for the *Rotate* operation. Fig 5.11d shows one possible realization. In the current example, the *Rotate* operation is implemented using an iterative algorithm. The conjunct #Q = Prt_Btwn(#n - n, |#Q|, #Q) o Prt_Btwn(0, #n - n, #Q) in the **maintaining** clause of the loop invariants is developed as follows: (i), entries are removed from incoming Q and are entered in Q. Hence, part of the **maintaining** clause is #Q = Prt_Btwn(#n - n, |#Q|, #Q) o Prt_Btwn(0, #n - n, #Q). The loop is executed (#n - n) time. Before the first iteration, #n is equal to n, therefore, elapsed time is zero before the first iteration of the loop.

```
Profile Rot_Prfile_2 short_for Rotate_Profile for Rotating_Capability for
                        Preemptable_Queue_Template with_profile PQ_CT_Profile;

    Defines C1: RPos;
    Defines C2: RPos;
    Operation Rotate (updates Q: P_Queue; evaluates n: Integer);
        duration C1 + I_Dur(Entry) + F_IV_Dur(Entry) +
                                    #n * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
end Rot_Prfile_2;
```

Fig. 5.11c: Performance profile of Rotate operation

```
Realization Rotate_Realiz_2 with_profile Rot_Prfile_2 for Rotating_Capability
                                                    of Preemptable_Queue_Template;

    uses ITP, Integer_Theory;
    Definition C1: RPos = ITP_AreNotEq + Dur_Call(2);
    Definition C2: RPos = CDq + CEn + ITP_Decr + ITP_AreNotEq +
                                                    Dur_Call(1) + 3*Dur_Call(2);

    Procedure Rotate (updates Q: P_Queue; evaluates n: Integer);
        Var Next: Entry;

        While (n /= 0)
            maintaining  Entry.Is_Initial(Next) and |#Q| = |Q| and
                        #Q = Prt_Btwn(#n - n, |#Q|, #Q) o Prt_Btwn(0, #n - n, #Q);
            decreasing n;
            elapsed_time (#n - n) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
        do
            Dequeue(Next, Q);
            Enqueue(Next, Q);
            Decrement(n);
        end;
    end Rotate;
end Rotate_Realiz_2;
```

Fig. 5.11d: Realization of the of Rotate operation

Since, the while loop rule created two paths, therefore, two of the eighteen VCs (VC 0_4 and VC 1_4) are related to the duration bound for the elapsed time at the beginning of the while loop; these two VCs are identical; and VC 0_4 is shown in Fig 5.12a. Two of the eighteen VCs (VC 0_12 and VC 1_6) are related to duration bound for the while loop termination and are shown in Fig 5.12b and Fig 5.12c, respectively. The complete listing is shown in Appendix C. Proofs are not shown.

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Rotate_Realiz_2.rb(14)

Goal(s):
(((n - n) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)

Fig 5.12a: Verification condition of duration correctness for Rotate operation

VC 0_12
Termination of While Statement: Rotate_Realiz_2.rb(13)

Goal(s):
((((((Cum_Dur' + I_Dur(Entry)) + (((CDq + I_Dur(Entry)) +
F_Dur(Entry, Next'")) + Dur_Call(2))) + (CEn + Dur_Call(2))) + (ITP_Decr +
Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <=
((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (Q'" = (<Next"> o Q"))
5. (n" /= 0)
6. ((Cum_Dur' + I_Dur(Entry)) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. (|Q| = |Q'"|)
9. Entry.Is_Initial(Next'")
10. (Q = (Prt_Btwn((n - n"), |Q|, Q) o Prt_Btwn(0, (n - n"), Q)))
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)

Fig 5.12b: Verification condition of duration correctness for Rotate operation

VC 1_6
Duration Clause of Rotate: Rotate_Realiz_2.rb(8)

Goal(s):
(((((Cum_Dur + I_Dur(Entry)) + (ITP_AreNotEq + Dur_Call(2))) + ((n - n') *
((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + F_Dur(Entry, Next')) <=
(((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) +
(n * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (n' = 0)
2. (Cum_Dur' = ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n')
4. (|Q| = |Q'|)
5. Entry.Is_Initial(Next')
6. (Q = (Prt_Btwn((n - n'), |Q|, Q) o Prt_Btwn(0, (n - n'), Q)))
7. Entry.Is_Initial(Next)
8. (Cum_Dur = 0.0)

Fig 5.12c: Verification condition of duration correctness for Rotate operation

5.4.3. Copy

Problem Description**:** *Using preemptable queue that is generic and parameterized by type of entries in a queue, generate VC for an operation that create one copy of the given Queue.*

Fig 5.13a shows a possible functionality specification of an enhancement *Copying_Capability* satisfying the *Copy* problem. The specification contains an operation *Copy*. Fig 5.13b shows a performance profile for the *Copy* operation. The performance profile is parameterized by definition of a function to copy an entry; and its implementation is not known yet. This parameterization is necessary. Because, now any implementation of this component can be used. *Cnts_Copy_Dur* is used to find the duration to copy entries; this conjunct first count the number of occurrences of an Entry *x* in the string S using *Occurs_Ct(x, S)* expression and then

multiply with the duration to copy *x*. It repeats the process to account the duration for every Entry in S.

```
Enhancement Copying_Capability for Preemptable_Queue_Template;
    Operation Copy_PQ(restores Q: P_Queue;
                                        replaces Q_Copy: P_Queue);
        ensures Q_Copy = Q;
end Copying_Capability;
```

Fig. 5.19a: Functionality specification of Copying_Capability

```
Profile CP_Profile_2 ( Definition Clr_Dur_Fn (X: Entry): RPos;
Definition Copy_Dur_Fn (X: Entry): RPos; )
                    for Copying_Capability of Preemptable_Queue_Template;

    Defines C1: RPos;
    Defines C2: RPos;
    Defines C3: RPos;
    Definition Cnts_Copy_Dur(S: Str(Entry): RPos = Sigma (x: Entry,
                                        Occurs_Ct(x, S) * Copy_Dur_Fn(x));

    Operation Copy_PQ(restores Q: P_Queue; replaces Q_Copy: P_Queue);
        duration C1 + Cnts_Dur ( #Q_Copy ) + (C2 + I_Dur ) * |Q_Copy| +
                                        Cnts_Copy_Dur( #Q )  +
                    (|#Q|) * (C3 + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry)) +
                                        2*I_Dur(Entry) + 2*F_IV_Dur(Entry);
end CP_Profile_2;
```

Fig. 5.10c: Performance profile of Copy operation

Fig 5.13c and Fig 5.13d shows one possible realization that satisfies the functionality specification (Fig 5.13a) and performance profile (Fig 5.13b). The key feature of this realization is the client-provided operation *Copy_Entry*; a specification of an operation to copy one entry. Since the type Entry is supplied as a parameter to the concept (*Preembtable_Queue_Template*)

and it is not possible to copy an arbitrary type. Therefore, *Copy_Entry* operation must be supplied as a parameter to the realization because it is necessary in order to write a realization. Hence, enhancement realization is parameterized by client-provided operation *Copy_Entry*. This example illustrates the general need for the language to be able to pass operation on generic Entry type as parameters to realization of concepts and enhancements to concepts.

In the current example, the *Copy* operation is implemented using an iterative algorithm. Since parameter *Q_Copy* is passed with replace mode, therefore, its incoming value is of no importance. Moreover, the current implementation of the preemptable queue is based on clean array; therefore, it is important to clean any entry that is not part of the conceptual Q; hence the call to the *Clear* operation, *Clears*(Q).

```
Realization Copy_Realiz_2  (
    Definition C: RPos;
    operation Copy_Entry(replaces Copy: Entry; restores Orig: Entry);
        ensures Copy = Orig;
        duration C + I_Dur(Entry) + F_Dur(Entry, #Copy) + Copy_Dur_Fn(Orig);
) with_profile CP_Profile_2 for Copying_Capability of Preemptable_Queue_Template;

    uses ITP, Integer_Theory;
    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C1: RPos = CLe + Dur_Assgn + ITP_AreNotEq + Dur_Call(1) +
                                  CCl1 * F_Dur_Integer + Dur_Call(2) + I_Dur(Integer) +
                                                F_IV_Dur(Integer) + F_Dur_Integer;
    Definition C2: RPos = CCl2;
    Definition C3: RPos = C + CDq + 2*CEn + ITP_Decr + ITP_AreNotEq + Cp +
                                                    Dur_Call(1) + 5*Dur_Call(2);
```

Fig. 5.13c: Realization of the of Copy_PQ operation

```
    Procedure Copy_PQ(restores Q: P_Queue; replaces Q_Copy: P_Queue);
        Var copy, Next: Entry;
        Var Len: Integer;

        Clear(Q_Copy);
        Len := Length(Q);
        While (Len /= 0)
            maintaining Q_Copy = Prt_Btwn(0, |#Q| - Len, #Q) and
                Q = Prt_Btwn(|#Q| - Len, |#Q|, #Q)) o Prt_Btwn(0, (|#Q| - Len, #Q)) and
                                    Entry.Is_Initial(Next) and Entry.Is_Initial(copy) and
                                                                    Len = |Q| - |Q_Copy|;
            decreasing |Q| - |Q_Copy|;
            elapsed_time |Q_Copy| *
                                (C3 + 2*I_Dur(Entry) + 2*F_IV_Dur(Entry)) +
                                 Cnts_Copy_Dur(Prt_Btwn(0, |#Q| - Len, #Q));
        do
            Dequeue(Next, Q);
            Copy_Entry(copy, Next);
            Enqueue(copy, Q_Copy);
            Enqueue(Next, Q);
            Decrement(Len);
        end;
    end Copy_PQ;
end Copy_Realiz_2;
```

Fig. 5.13d: Realization of the of Copy_PQ operation (continued)

In Fig 5.13d the loop annotations are developed as shown below.

- Loop invariant or **maintaining** clause:
    - Q_Copy = Prt_Btwn(0, |#Q| - Len, #Q): Entries are dequeued from *Q* and enqueued in *Q_Copy*; here |*#Q*| - *Len* represents length of *Q_Copy*.

- Q = Prt_Btwn(|#Q| - Len, |#Q|, #Q)) o Prt_Btwn(0, (|#Q| - Len, #Q): Since the entries are dequeued from *Q* and after the copy operation enqueued in Q; and |*#Q*| - *Len* represent an index in incoming queue. Therfore, after an iteration of loop, the outgoing Q can be visualized as concatenation of two substrings.
- **Is_Initial**(Next) and Entry.**Is_Initial**(copy): Any entry of the array that is not part of the conceptual array is initialed value, therefore, after the call to *Enqueue* operations, both *Next* and *copy* entry is clean.
- Len = |Q| - |Q_Copy|: Before the loop, *Len* := |*Q*|; Entries are dequeued from *Q* and enqueued in Q_Copy. Therfore, difference in lengths for the two queues is equal to *Len*, and *Len* is decremented, therefore, the two terms are same.

- Decreasing clause:
  - |Q| - |Q_Copy|: Initially, |*Q_Copy*| is zero. As the copy operation proceeds, |*Q_Copy*| increases. Hence, the decreasing clause of the loop.
- Elapsed_time invariant:
  - |Q_Copy| * (C3 + **I_Dur**(Entry) + 2***F_IV_Dur**(Entry)): In the loop body, the *Enqueue* operation is called twice and *Dequeue* is called once; this process has been repeated in |*Q_Copy*|; hence the term in elapsed time.
  - Cnts_Copy_Dur(Prt_Btwn(0, |#Q| - Len, #Q)): This conjunct counts the duration to copy the entries in the substring represented by the two indices.
  - Dequeue, Enqueue, and Decrement operations are called in the loop body and the conjunct C2 + **I_Dur**(Entry) + **F_Dur**(Entry, Next) is the summation of their duration. Before the first iteration, | Q_Copy | is equal to zero, therefore, elapsed time is zero before the first iteration of the loop.

5.5. VC Generation of parameters to realization

Currently, the mechanical VC generator cannot generate the VC for the Copy_PQ operation shown in Fig 5.13c and Fig 5.13d. The instrumentation to account for the duration clause of the *Copy_Entry* (operation is not in place. For a similar reason, the VC for the facility shown in Fig 3.1 cannot be generated. The necessary fixes are straightforward.

CHAPTER 6.

RELATED WORK

6.1. Introduction

Work related to the topic of this research is presented in this chapter in three sub-sections: component-based software, software performance, and verification.

6.2. Component-based software

In component-based software (CBS), a software component is the basic unit for reuse [Booch87, Petriu02, Kim04]. A component is a collection of the minimum number of closely related primitive operations with the complete interface [Booch87]. Usually, a component is released as a black-box. [ParnasN72, ParnasD72, ParnasM85, Liskkov93]. The components so created can interact with each other without additional information. Thus, a large and complex software system can be developed using independently developed generic and reusable components [Bertolino03, Kim04, Petriu02, Pour98, Wu03, Wu04]. This approach has several advantages: (i) modular analysis: the system can be analyzed without re-analyzing its constituent components; (ii) scalable: larger systems can be developed using components or smaller systems; (iii) verifiable: verify the target system using the specifications for individual components; (iv) flexible: replace existing components with new components of the same functionality.

Selection of proper components is one of the major issues in the component-based software design. Nevertheless, whenever there are multiple components with the same interface, it may be difficult for the client to choose the best component to design a software system [Wu03, Wu04]. To select the right combination of components, a client needs additional information related to each component, such as its performance specification.

Software performance specification is a challenging problem [Smith90]. In CBS, software performance is emerging as a major issue in the selection of appropriate components; the performance of the final system can be improved by predicting and improving the performance of its components. A functional specification of a software component can be implemented in multiple ways, and different implementations provide important space/time trade-offs to clients. Since clients should not be burdened with the internal details of the implementations, developers must supply abstract performance specifications for their implementations so that clients can choose a component that best fits their performance needs.

6.3. Software Performance

Performance is an essential quality attribute of every software system and recent studies have shown the importance of integrating performance as a quantitative validation tool in the software development process [Balsamo04, Bertolino03, Edwards94, Siddiqui02]. The performance of software can be defined as a prediction of the resources used during the execution of the software [Cormen90, Gerlich01, Smith02, Woodside02]. Generally, memory usage and the execution time are the two major resources under consideration. However, it is the execution time that the research community has considered most at the present time.

6.3.1. Importance of software performance

In the early era of computing, software performance was a major part of software design because of the limited memory space and long execution time. However, in the early seventies, the software community started developing software with a "fix-it-later" approach [Smith90]. In this approach, software correctness has the highest priority [Balsamo04, Bertolino04, Cortellessa05, Pour98] and software performance has the lowest priority among the list of important factors affecting the software. This approach is also known as a reactive performance management; that is, design, develop, and deliver the software and, if necessary, add performance features later. There are several factors behind this style.

- The design and management of the performance model is very complicated and time consuming. Thus, software development would need extra time to complete and software delivery would be delayed.
- Software should be partially complete before creating the performance model.
- Performance is not a major issue for most software.
- Hardware is becoming cheaper and faster so there is no need to be concerned about execution time performance.

A fix-it-later approach in a software development process leads to ignoring the bottlenecks of the system. In some cases, this ignorance may result in system failure. Boehm discusses the importance of identifying and resolving the risk factors involved in a software system during the early stages of the development cycle [Boehm91]. Extra costs, delays in project delivery, execution time, and memory usage are some of the risk factors. A poor performance or performance failure of software may have adverse effects on a business.

In order to create software satisfying their performance goals, performance characteristics should be managed at every stage of the development process [Smith02], that is, a proactive approach is needed. Smith provides several examples emphasizing the fact that managing software performance will lead to stable and efficient software, and argues for a proactive approach instead of reactive approach. Schupp et al., discuss performance-related issues in CBS composition process using a library of components [Schupp04]. This research uses a Fast Fourier Transform Library as an experimental study to show the importance of performance analysis.

6.3.2. Software performance analysis

Fig 6.1 shows a classical method of software development. It shows that the program implementation depends on its requirements and functional and performance analysis (arrow from *Implementation* to *Requirements and Functional and Performance Analysis*). Both functional and performance analysis can be carried out after the requirements stage (arrows from *Requirements to Functional and Performance Analysis*). However, due to incomplete specification and implementation, the system is iterative. That is, at different stages of program development, the process of implementation and functional and performance analysis is repeated.

Various researchers [Williams95, Smith03] have provided guidelines that can be used to develop software systems achieving their performance goals. Identification of the risk factors is on the top of the list. Based on software requirements, alternative designs must be compared before committing to a particular design. The designer must establish well-defined performance goals. Software performance must be evaluated at different stages of the development cycle, and if necessary, changes should be made to the software to meet the desired goals.

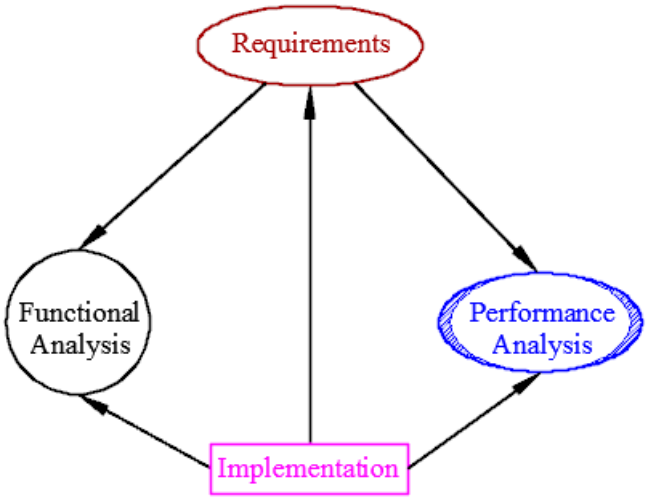

Fig 6.1: Classical method of software performance analysis

A decision regarding software performance analysis can be made as early as during the requirement analysis stage [Graham73, Pooley00, Petriu02] or as late as the implementation stage using best- and worst-case techniques. However, if the performance analysis is carried out in an early stage, it will have several deficiencies. Smith and Woodside discuss the issues related to performance validation in the early stages of the development cycle [Smith99]. Before the implementation, software information is sketchy. For example, data types are not clear, algorithms and other components to be used in the final product are not selected, and the exact resource usage is unknown.

6.3.3. Performance prediction strategies

Software performance can be predicted using either a qualitative process (excellent/good/poor, fast/slow) or a quantitative process (measuring response or execution time, memory usage, etc.) [Abdullatif09, Petriu02, Dugan04]. There are several techniques to predict software performance. These techniques can be divided into two major groups: non-real time systems and real time systems.

6.3.3.1. Non-real time systems

This section describes the performance prediction techniques shown in Fig 6.2 used for non-real time systems.

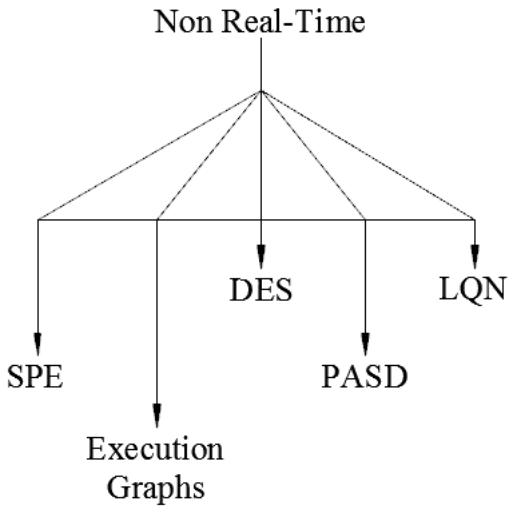

Fig 6.2: Non real-time system performance analysis techniques

Software performance engineering (SPE) is a software-oriented quantitative technique to design and evaluate performance of software at every stage of the development [Smith02]. Fig 6.3 shows a bottom-up approach. In this approach, the initial development is an iterative process. After the initial development, running time analyses are performed.

- If the worst case results are acceptable, then the development is moved to the next stage.
- If the best case results are unacceptable, then the process is repeated with alternative algorithms.
- However, if the worst case results are unacceptable and the best case results are acceptable, then before proceeding to the next development stage, bottlenecks are identified. These bottlenecks are carefully studied and precise data are collected in current and subsequent development stages.

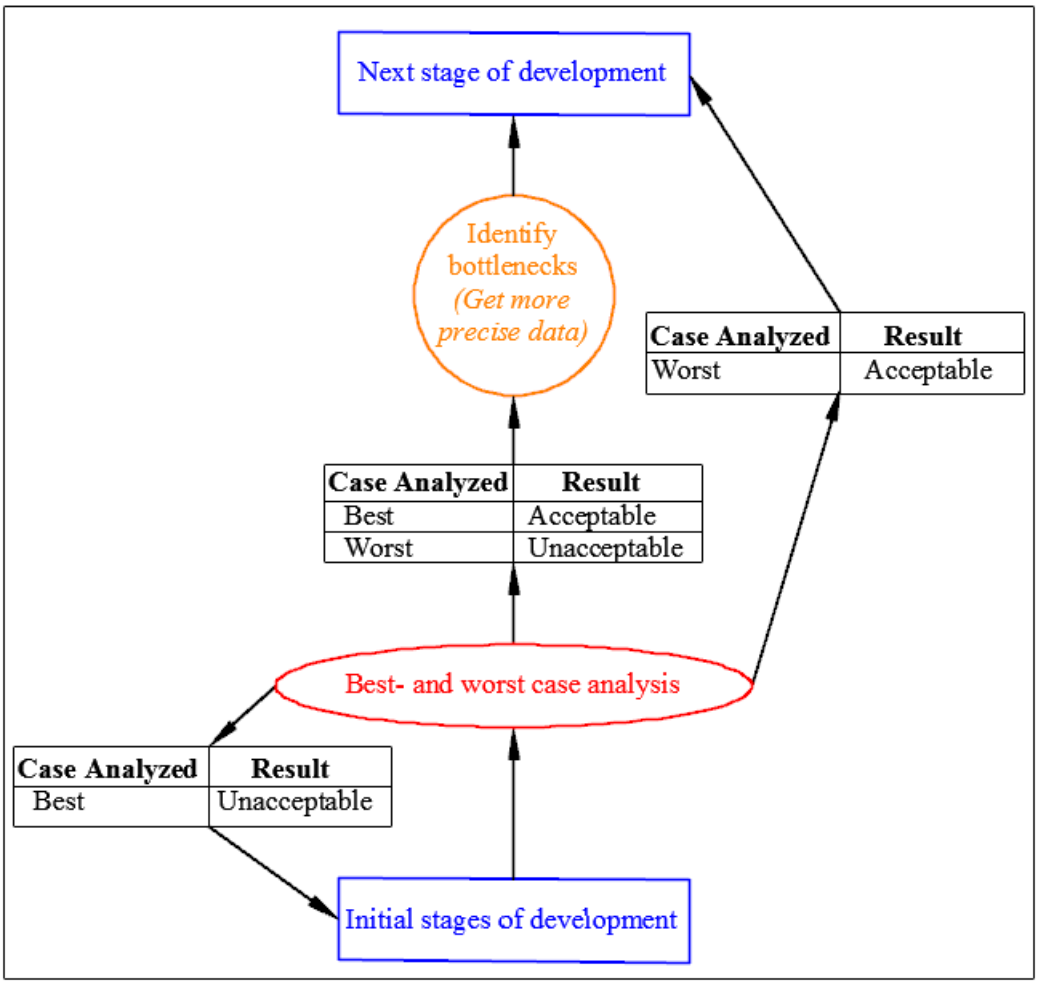

Fig 6.3: Software performance (bottom-up approach)

The CBSE performance study carried out by [Bertolino03, Bertolino04] assumes that performance profiles of basic components are available and validated. It proposes an iterative process of how components of known performance profile can be used to develop a predictable software system. The method is applied in two steps: (i) Search for the components of the desired performance. (ii) If at the end of the first step there is more than one component available with the same functionality, then software is developed using the best performing components from the list of components selected in step (i). [Bertolino03] finds the performance profile using best-case analysis; assuming that only the application under consideration is being executed, that

is, there is no other program in the execution environment. On the other hand, [Bertolino04] finds the performance profile using worst-case analysis: assuming that N applications similar to the application under consideration are in the execution environment and are therefore competing for the same resources.

Jasmine et al. provide an experimental study to identify the performance bottlenecks in a software development process using reusable components [Jasmine07]. This study assumes that performance profiles of basic components are available and uses execution graph technique to predict software system performance. In the execution graph technique, software components and inter-component communication are represented by vertices and edges of an undirected graph, respectively. The edges are labeled with the number of times a particular edge is used in the software execution. The performance evaluation model uses only those edges that are affecting performance. A software engineer must focus on the part of the code executed most frequently, yet critical code executed rarely should not be ignored.

Graham suggests that to circumvent the cost of redesign and reimplementation of a software system, automated techniques should be used for predicting and validating performance before the implementation [Graham73]. He describes an integrated Design and Evaluation System called DES. The distinguishing features of this system are the use of a single language (for specification, implementation, and performance evaluation) and a single database. The database stores the information such as the execution time, memory usage, size of input data, and probability of the selection of execution paths (DES uses a directed graph to document the execution paths) about every procedure and data component used by the system. The database also stores information about the hardware used by the system. The performance evaluation system com-

bines the graphs of the participating components and predicts the maximum, minimum, and average execution time and memory usage of the desired system.

Siddiqui uses a performance-aware software development (PASD) approach to find the performance profile of a software system [Siddiqui02]. The technique has three major elements: (i) functional specification; (ii) resource demand budget which is based on the requirement and is prepared by the software developers; and (iii) a model responsible for the platform behavior and for evaluating and adjusting the resource budget. PASD follows a divide-and-conquer style. The given problem is divided into smaller sub-problems; resources are allocated to the sub-problems using intuition, prediction, and sensitivity analysis; and the model is executed. The model is validated if the software achieves its goal within the allocated resources.

Wu et al. uses a layered queuing network (LQN) model to predict the software performance of component-based software [Wu03]. This system is based on a queuing network (QN) model. In this methodology, software components are represented as QN models using functional and non-functional characteristics of the components; these models are the sub-models for LQN. The sub-models have "entries", and the entries are used to connect one or more sub-models.

6.3.3.2. Real time systems

In real-time systems (RT), performance or execution time prediction is a challenging problem. In these systems, functional correctness and performance profiles are inter-related; that is, not only is logical correctness important, but also the execution time is important [Cottet02, Stankovic88]. The major steps in an RT system design are requirement analysis followed by the functional and timing (performance) analysis; these analyses lead to specific hardware and soft-

ware development. An RT system classification shown in Fig 6.4 is the topic of discussion for this section. Table 6.1 shows the research techniques discussed in this section.

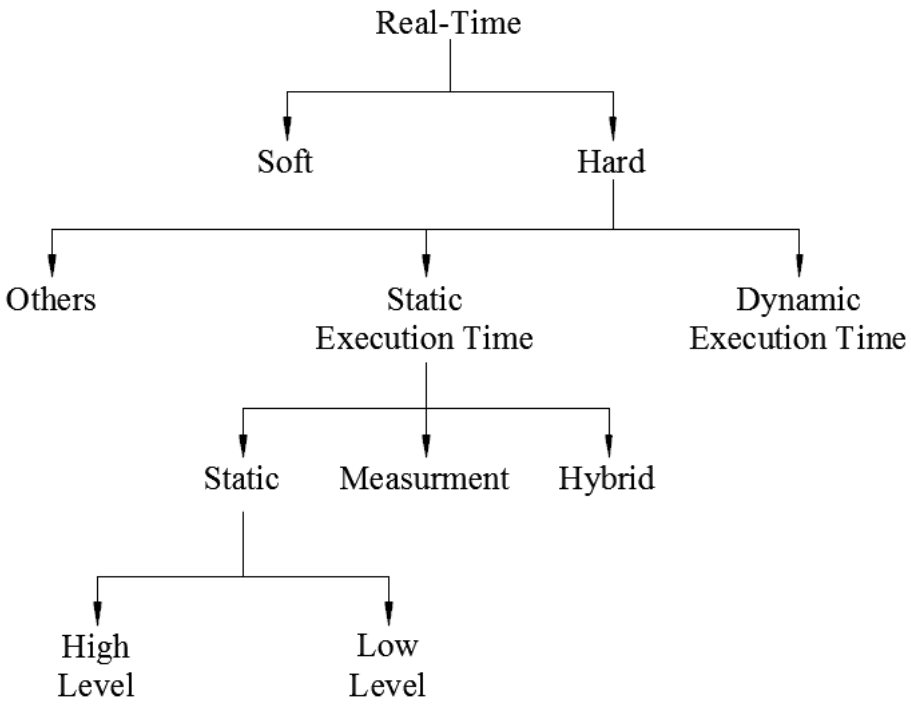

Fig 6.4: Classification of real-time systems

Table 6.1: Hard real-time: Static execution time

Timing Analysis	***Example Systems***
Low Level Static	Schoeberl10, Petters04, Heckmann04
High Level Static	Wilhelm08
Measurement	Wilhelm08, Ravindar11G, Ravindar12
Hybrid	Kirner04, Lokuciejewski11, Petters07, Tetzlaff13
Others	Gustafsson02

RT systems can be divided into two groups: soft (SRT) and hard real-time (HRT) systems [Stankovic88]. In an SRT system, missing the deadline of one or more task (degraded performance) is acceptable. On the other hand, in an HRT system, missing a deadline (system failed) is not acceptable. In an HRT system, the execution time of each task must be less than the worst-case execution time (WCET) for the task. WCET analysis is used to calculate the maximum execution time for a given code on a specific hardware [Kirner05, Puschner99]. HRT systems are further subdivided into static and dynamic systems. In a static HRT system, the program developer must know the execution time of every task before developing the system. In dynamic HRT systems, execution times of tasks can vary as needed.

The issues and problems in estimating WCET are clear from [Wilhelm08]. WCET analysis may be static, measurement-based, or a hybrid [Lokuciejewski11]. In static timing analysis, the possible control paths of a task code and abstract hardware is used to estimate the WCET. In measurement-based timing analysis, the task code is run on a given hardware for possible inputs to the system and execution time is measured. The hybrid approach is the combination of the first two techniques. The WCET analysis is high level if the execution time is estimated for the flow paths and low level if it is estimated for machine instructions [Schoeberl10].

Schoeberl presents a design for Java optimized processor (JOP) [Schoeberl06]. This processor design is based on the idea that, instead of providing all bytecode instructions, provide only a reduced set of instructions. This processor is useful to provide an accurate WCET analysis. Using that Java processor and cache method, Schoeber describes a static technique to estimate WECT [Schoeberl10]. Since the analysis depends on the byte codes, it is not suitable for modular reasoning. In [Petters04] Petters proposes a technique to estimate WCET for path-based low level analysis. Due to the limitations of measurement-based WCET analysis, Heckmann's

static HRT analysis approach relies on binaries to estimate WECT [Heckmann04], but it sacrifices abstraction and modular composition in the process.

Wilhelm discusses several approaches for estimating WCET for hard real-time systems in presence of caches, pipelines, branches, etc. [Wilhelm08]. Wilhelm reports several problems in estimating WCET and proposes solutions, including correct and complete loop bounds and flow annotation, integrated timing analysis and compiler design, and extension of timing analysis to component-based real-time systems. In this survey, the timing analysis is defined as the process of estimating WCET and is either based on the static or measurement based method.

Using several benchmarks and assuming programs as single blocks, Ravindar proposed a measurement based technique for WCET estimation [Ravindar11G]. This technique can be divided into three steps: (i) for various test cases, measure the execution time (ET) and instruction count (IC); (ii) calculate the cycles per instruction (CPI) from the information in step (i); and (iii) calculate WCET using worst case IC and worst case CPI.

The focus of Ravindar's research [Ravindar11] is the accuracy of measurement-based techniques for WCET estimation; that is, the measurement process should not be affecting the measurements. Based on the solution offered in the research, a program is divided into phases such that the variation in the measuring characteristic (CPI, cache miss/hit, etc.) of a phase is minimum as compared across the phases. Finally, the WCET for the program is calculated from the measured WCET of each phase. The research in [Ravindar12] is the extension to the research in [Ravindar11]. In [Ravindar12], the phases with higher variation in CPI are divided into sub-phases and, using Chebbyshev's inequality, their WCET is estimated. Finally, the WCET for the program is calculated from the measured/estimated WCET of each phase.

[Kirner04, Lokuciejewski11] provide a hybrid approach for WCET analysis. Based on the importance of WCET analysis for a RT, using L4 microkernel API, Petters shows that the measurement-based analysis and static analysis can be used as complementary techniques instead of as rivals for the estimation of WCET [Petters07]. Using machine learning techniques and static WCET analysis, Tetzlaff estimates the execution time of functions [Tetzlaff13].

Gustafsson's research emphasized the WCET prediction for the RT systems using object-oriented programming languages (Java, Ada, Smalltalk, and C++). The author first provides a list of problems such as memory allocation and deallocation (garbage collector), dynamic binding, recursion, and modern hardware that can interrupt the execution of RT system. The author then provides the solution to the problems - either do not allow dynamic memory allocation, deallocation, or recursion or allow them with restriction [Gustafsson02].

Source code-based analysis approaches, such as ours and others, do not account for compiler and hardware-related issues at this time. The performance (duration) analysis of this dissertation is based on a non real-time analysis.

6.4. Verification

Verification process is designed to determine whether a program will actually behave as promised by its abstract specification. However, the verification process will not be able to detect specification errors and compiler or hardware bugs. Formal verification of software is based on a set of theorems. A piece of software is formally verified when the theorem prover proves all of the theorems [Popovic02]. A program is correct (or erroneous) if it can ever reach an assert statement whose condition evaluates to be true (or false) [Barnett03]. The complexity involved in the verification of nontrivial program can be resolved with abstractions.

In [Hall90], Hall published an article listing seven myths followed by seven facts about the nature and application of formal methods. Although formal methods can guarantee perfect software by proving that the programs are correct, the technique is generally used only in safety-critical software. Moreover, the formal methods take a longer time because they involve intricate mathematical details. Finally, the author elucidates that formal methods are successfully used in industrial projects. Based on the observation of industrial projects, Bowen published an article that listed and dispelled seven more myths about the nature and application of formal methods [Bowen95].

Verification proof generator is a complicated process. It involves theorem proving techniques. Hence, if automatic methods were available, then program verification would be adopted by the software community at large. The accuracy of software design can be increased and maintenance processes can be simplified by using mechanical verification tools [Barnett03].

6.4.1. Automated verification of functional behavior

Mackenzie has written an outstanding historical and social overview of automated theorem proving [Mackenzie95]. Popovic, Kovacevic, and Velikic demonstrate a formal software verification technique based on automated theorem proving and reverse engineering using finite state machines [Popovic02]. Model checking and theorem proving are the two commonly used paradigms for program verification [Clarke96].

Using SPARK and Z programming languages, King describes the extensive use of formal methods in safety and security critical applications [King00]. Using formal proof rules and a combination of automated and interactive techniques, King generated the VC [King00]. In this system, VCs could be generated by two methods - either VC generated automatically using proof rules, or if the first method failed then VC are generated interactively using a proof checker.

King, also shows that the verification technique are more cost effective in finding faults in the system than in testing the system.

There is considerable work in the literature on automated verification of functional correctness summarized in [Klebanov11]. Boogie 2 is an intermediate programing language; it is used to convert a program written in source language into code that is understood by a theorem prover [Leino10T]. Dafny is an imperative programing language and automatic program verifier for functional correctness [Leino10L] and its verifier is based on Boogie.

As noted earlier, Harton has developed an automated system to generate and prove the functionality VC [Harton11]. There, several other systems for functionality verification are discussed compared with the RESOLVE system. The proof system and VC generator presented in this dissertation are based on the work of Harton.

6.4.2. Formal verification of performance bounds

Using ghost variables for time and space, Hehner's research provide techniques to specify and prove the time and space requirements for a real time system [Hehner99]. The ghost variables are the variables that are (a) not part of the implementation, (b) do not need any space, and (c) do not take any time to execute. Hehner's research is based on three major steps: (i) specify the quantities and represent each quantity by a variable; (ii) create a Boolean expression based on the variables of the quantities selected in step (i); and (iii) prove the Boolean expression developed in step (ii). This research is explained with two simple programs: $y' = 2^x$ and tower of Hanoi. This research discusses the recursion, but does not mention if/then/else and loops. Furthermore, this research does not discusses the scaling-up, and the prover is not automated.

Refinement calculus (also known as top-down design) is defined as the process of converting a specification into an executable program using transformation rules [Morgan88]. Hayes

provides a comprehensive refinement calculus to convert an RT specification (functional and execution time) into a machine-independent RT program [Hayes01, Hayes07]. The resultant program must pass the timing analysis, if not then it is rejected. Lermer's technique of timing analysis can be divided into two steps [Lermer05]: (i) using refinement calculus, conversion of abstract specifications into a program in a high level programming language (HLPL). (ii) separation of functional and performance behavior. Step (ii) is further subdivided in to two parts: (iia) conversion of HLPL program into machine-independent code annotated with timing requirements to find WCET and BCET and (iib) machine-independent code to check that the compiled code satisfies the timing requirements.

In [Shaw89], Shaw outlines proof rules for duration analysis for a "hard" real time system. Shaw's research is based on two major ideas: (i) each process has its own dedicated processor and each processor has its own clock, that is, no sharing; (ii) the extension of the Hoare logic (*{P} S {Q}*) to include time of execution of each statement to *{P} <S; rt := rt + t(S)> {Q}*. In this expression, *rt* is the real time before the execution of statement *S* and *t(S)* is the time to execute *S*. The research also explains methods to find the upper and lower time bounds for the statements. Knowing the execution time of statements, the research provides schemas to calculate the time to run the program. Shaw assumes that *S* can be an elementary statement or a control structure (if/then/else or loop); and the loop is unrolled for calculating the upper and lower bounds. Since, the Shaw system does not uses specification, its drawback is that it cannot be used for modular reasoning.

The EVES (Euclid-based Verification and Evaluation System) was developed to produce formally verifiable code written in the Euclid programming language [Craigen85]. Kaivola and Narasimhan use a combination of theorem proving and binary decision diagrams to verify multiplication supported by Pentium 4 hardware [Kaivola02].

Woodside uses a specification language to specify and validate the performance of a system [Woodside02]. Here, functional specification is annotated with non-functional properties including the execution time of the operations. Using a video server example, it shows how an annotated specification language can be used to specify and predict the performance of a system.

In [Richter03], Richter et al. show that the hardware simulation can be supplemented by formal analysis for performance verification.

Hooman's work on specification and modular verification [Hooman91] is among the most comprehensive theoretical foundations for compositional verification of real-time systems. However, these ideas have not been mechanized and they do not concern object-based software.

A system for approximate resource verification of logic programs is the topic of [Lopez-Garcia12]. The system infers the intervals of resource usage for input sizes and compares them against specifications.

Aspinall, Beringer, and others [Aspinall07, Beringer04] propose a system for analyzing resource consumptions for functional languages that like our works combines functional and performance verification. Their work uses Grail (Guaranteed Resource Aware Intermediate Language) and a notion of resource algebras, and involves verification of recursive algorithms. Using program logic and set of rules the authors verify the the heap consumption in Grail's bytecode logic. A VC generator based on their system, including relevant proof rules is discussed in [Sevcik07]. This work differs from this dissertation in two ways. It considers both time and

space, and addresses complexities that arise in lower-level programs at the level of heap locations. On the other hand, unlike this dissertation, their work (presented in the context of functional languages) does not explicitly address generic data abstractions or performance profile-based modular verification.

For object-oriented programs, Leavens and Haddad describe a formal technique to predict the WCET; this technique can be used to overcome the complexities involved in subtyping and modular reasoning [Leavens06, Haddad13].

The work in this dissertation represents one of the first efforts in mechanical duration analysis of object-based programs, designed using data abstraction principles. Although the generated VCs have not yet been proven for duration bounds, the system is able to dispatch functional correctness VCs for numerous benchmarks [Sitaraman11]. A push-button verifier for functional behavior [Sitaraman11] is available through a web IDE [Cook12]. VC generation and verification for functional correctness are discussed in detail in [Harton11, Smith13]. Performanc VCs are not automatically discharged at this time because provers do not currently handle duration-related VCs involve multiple theories (including real numbers).

One of the fundamental differences between this research and earlier efforts is that this research connects performance analysis to abstract values of variables. For example, if the duration of an operation depends on the front entry of a queue or the ordering of entries in the queue, then the timing analysis is not possible, unless the queue's value is calculated from an analysis for functional correctness. While the duration analysis in this paper is done at a certain level of precision, the analysis can be weakened to big-O level expressions or made more precise.

CHAPTER 7.

CONCLUSIONS AND FUTURE DIRECTIONS

7.1. Introduction

Performance prediction of software is a complex problem and this research has taken an important first step by developing a modular performance verification system. In this analysis, it is possible to establish properties of the target system formally without a need to inspect the internal details of any of its constituent components. The generated verification conditions (VC) are based both on the functionality and performance specification of software components. Performance profiles (specification) are layered on top of functional specification. The modularity of the approach makes it scalable. If a performance profile changes (perhaps because a previous implementation is unplugged and a new one is inserted), then only the performance-related VCs will need to be regenerated and verified. This chapter summarizes the main contributions of this dissertation and sets forth directions for future research.

7.2. Scope of this study

- **Language design:** To facilitate component-based modular reasoning of performance, this dissertation has introduced a language for performance specification, including performance profiles for components and elapsed time loop annotations. Using several illustra-

tive examples, this research has shown that it is not only possible to write performance profiles, but also use them in developing predictable software, thus achieving the first objective of the thesis.

- **Proof system:** A mechanizable proof system is essential to develop a practical verification system for performance. This research has focused on establishing modular proof rules for duration correctness. This goal has been achieved by extending rules for functional correctness and thus achieving the second objective of the thesis.
- **Experimentation:** This research has led to a prototype VC generator for performance correctness and experimentation. The same system is used for generating VCs for both built-in and user-defined data types. Experimentation and analysis of a variety of examples fulfill the third objective of the thesis.

7.3. Future Directions

- **Performance VCs Prover:** This research has focused on generating VCs for performance correctness. Ultimately, the VCs need to be proved. Proving VCs involving real numbers is therefore the first challenge.
- **Performance VCs for Concept Realizations:** This research has only led to generation of VCs for performance correctness for realizations of enhancements to a concept. However, this research can be extended to generate performance VCs for concept realizations and user facilities. This should not be difficult once full functionality VC generation is in place.
- **Supplemental Model:** When concepts such as *Preemptable_Queue_Template* are implemented with time-efficient realizations, the entries in the array that are not part of the

conceptual queue can contain arbitrary values. A supplemental model will need to be used to remember these arbitrary values in order to analyze the duration to finalize them. So performance profiles need to support intermediate models to provide tighter bounds [Sitaraman94].

- **Memory Usage:** This research has focused on duration prediction. In order to prove that software runs within time and space constraints, memory usage must be taken into account. In this research, the RESOLVE language is extended to account for memory usage; however, the proof rules must be extended and implemented to include memory usage, too.
- **Tight Bounds:** Performance prediction complexity and tightness of the bounds are correlated, so experimentation and evaluation will be necessary to understand what levels of tightness are necessary in practice.
- **Compiler Optimization and Hardware Performance:** Naturally optimizations performed by compilers and hardware aspects (e.g., caching) affect performance, so they must be included in the performance prediction process.
- **Web IDE:** For RESOLVE users, a push-button verifier for functional behavior is available through a web integrated development environment (IDE). However, performance specification and analysis capabilities need to be added to facilitate research and education.
- **Pedagogical Issues:** Ideally, formal software performance analysis must be taught to undergraduate computer science students. Students need to be taught classical big-O analysis and software engineering principles, along with performance specification and verification principles in a suitable fashion. In the survey carried out in the United States

schools, Dugan pointed out serious shortcomings in teaching software performance engineering and has suggested an outline for an undergraduate level course [Dugan04]. To teach performance reasoning, a web-integrated interface is useful.

BIBLIOGRAPHY

Abdullatif09: A. A. Abdullatif and R. Pooley, "From UML to EQN: Studying System Performance from an Early Stage of Systems Life Cycle," in 25th UK Performance Engineering Workshop, Leeds, 2009, pp. 111-122.

Anderson84: G. E. Anderson, "The coordinated use of five performance evaluation methodologies". Communications of the ACM, Volume 27, Number 2, February 1984, pp. 119-125.

Aspinall07: D. Aspinall, L. Beringer, M. Hofmann, H. W. Loidl, A. Momigliano, "A program logic for resources", Theoretical Computer Science, Volume 389, Issue 3, December, 2007, pp. 411-445.

Balsamo04: S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey", IEEE transactions on Software Engineering, Volume 30, Issue 5, May 2004, pp. 295-310.

Barnett03: M. Barnett, R. DeLine, M. Fahndrich, K. Rustan M. Leino and W. Schulte, "Verification of object-oriented programs with invariants", in Journal of Object Technology, Volume 3, Number 6, June 2004, Special issue: ECOOP 2003 workshop on FTfJP, pp27-56. http://www.jot.fm/issues/issue_2004_06/article2.

Berg82: H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher, "Formal Methods of program Verification and Specification", Prentice-Hall, Second edition, 1982, ISBN 0-13-328807-2.

Beringer04: L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, "Automatic Certification of Heap Consumption", 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04), 2004, pp. 347-362.

Bertolino03: A. Bertolino and R. Mirandola, "Towards component-based software performance engineering", Proceedings of the 6th ICSE workshop on component-based software engineering: Automated reasoning and prediction, Portland, Oregon, May 3-4, 2003.

Bertolino04: A. Bertolino and R. Mirandola, "Software performance engineering of component-based systems", ACM SIGSOFT Software Engineering Notes, Proceedings of the fourth international workshop on Software and performance, Volume 29, Issue 1, January 2004.

Boehm91: B. Boehm, "Software risk management: principles and practices", IEEE Software, Volume 8, Number 1, 1991, pp. 32-41.

Booch87: G. Booch, "Software components with Ada", The Benjamin/Cummings Publishing Company, 1987, Second Edition, ISBN 0-8053-0610-2.

Bowen95: J. P. Bowen and M. G. Hinchey, "Seven more myths of formal methods", IEEE software, Volume 5, Number 2, July 1995, pp. 34-41.

Clarke96: E. M. Clarke, J. M. Wing, et al, "Formal Methods: State of the art and future direction", ACM Computing Surveys, Volume 28, Number 4, December 1996, pp. 626-643.

Cook12: C. T., Cook, H.K, Harton, H., Smith, and M., Sitaraman, "Specification Engineering and Modular Verification Using a Web-Integrated Verifying Compiler". In Proceedings of ICSE 2012, June 2-9, Zurich, Switzerland.

Cormen90: T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to algorithms", Published by The MIT Press and McGraw-Hill Book Company, Fourth edition, 1990.

Cortellessa05: V. Cortellessa, "How far are from the definition of a common software performance ontology?", WOSP'05 Palma de Mallorca, July 2005.

Cottet02: F. Cottet, J. Delacroix, C. Kaiser, and Z. mammeri, "Scheduling in real-time systems", John Wiley & Sons, LTD, 2002, ISBN 0-470-84766-2.

Craigen85: P. Craigen and M. Saaltink, "An EVES update", ACM SIGSOFT Software Engineering Notes, Volume 10, Number 4, August 1985, pp. 33-34.

Denning81: P. J. Denning, "Performance analysis: experimental computer science as its best", Communications of the ACM, Volume 24, Issue 11, November 1981, pp. 725-727.

Dijkstra90: E. W. Dijkstra, "Formal development of programs and proofs", Edited by E. W. Dijkstra, Addison-Wesley Publishing Company, 1990, ISBN 0-201-17237-2.

Dolbec95: J. Dolbec and T. Shepard, "A component-based software reliability model", Proceedings of the 1995 conference of the Center for Advanced Studies on Collaborative research, November 1995.

Dugan04: R. F. Dugan Jr., "Performance lies my professor told me: The case for teaching software performance engineering to undergraduates", In Workshop on Software and Performance archive Proceedings of the 4th international workshop on Software and performance, Redwood Shores, California, 2004, pp. 37-48.

Edwards94: S.H. Edwards, W.D. Heym, T.J. Long, M. Sitaraman, and B. W. Weide, "Part II: Specifying components in RESOLVE", ACM SIGSOFT Software Engineering Notes, Volume 19, No. 4, October 1994, pp. 29-39.

Floyd67: R. W. Floyd, "Assigning meanings to programs", Proceedings of symposia in Applied Mathematics 19, American Mathematical Society, 1967, pp. 19-32.

Gerlich01: R. Gerlich, "Performance and robustness engineering and the role of automated software development", in Performance Engineering: State of the art and current trends, number 2047 in Lecture Notes in Computer Science Springer-Verlag, 2001, pp 20-39.

Giannakopoulou01: D. Giannakopoulou and J. Penix, "Component Verification and Certification in NASA Missions", 4th ICSE Workshop on Component-Based Software Engineering Component Certification and System Prediction, May 14-15, Toronto, Canada, 2001.

Giordano80: J. V. Giordano, "Some verification problems in pascal-like languages", ACM SIGSOFT, Software Engineering Notes, Volume 5, Number 1, January 1980, pp. 18-27.

Graham73: R. M. Graham, G. J. Clancy Jr., and D. B. DeVaney, "A software design and evaluation system", Communications of the ACM, Volume 16, Number 2, February 1973, pp. 110-116.

Gustafsson02: J. Gustafsson, "Worst case execution time analysis of object-oriented programs". In *Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02)*. IEEE Computer Society: Los Alamitos, CA, USA, 2002.

Haddad13: G. Haddad and G. Leavens, "Specifying subtypes in Safety Critical Java programs", Concurrency and Computation: Practice and Experience, 2013, Volume 25, Issue 16, pp. 2290-2306 Published in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2930.

Hall90: A. Hall, "Seven myths of formal methods", IEEE software, Volume 5, Number 2, September 1990, pp. 11-19.

Harms91: D. E. Harms and B. W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components", IEEE Transactions on Software Engineering, Volume 17, Number 5, May 1991, pp. 424-435.

Harton11: H.K.Harton, "Mechanical and Modular Verification Condition Generation for Object-Based Software". Ph. D. Dissertation, Clemson University, 2011, 305 pages.

Hatcliff12: J. Hatcliff, G. T. G. Leavens, R. Leino, P. Muller and M. Parkinson, "Behavioral interface specification languages", ACM Communications Survey, Volume 44, Number 3, Article 16, June 2012, pp. 16:1-16:58.

Hayes01: I. J. Hayes and M. A. Utting, "Sequential real-time refinement calculus", Acta Informatica, 2001, Volume 37, Issue 6, pp. 385–448.

Hayes07: Ian J. Hayes, "Procedure and parameters in the real-time program refinement calculus", Science of Computer Programming, Februry 2005, Volume 64, Issue 3, pp. 286-311.

Heckmann04: R. Heckmann and C. Ferdinand, "Worst-Case Execution Time Prediction by Static Program Analysis." AbsInt Angewandte Informatik GmbH, 2004, Saarbrucken, Germany, http://www.absint.com/aiT_WCET.pdf.

Hehner99: E. C. R., Hehner, "Formalization of Time and Space", Formal Aspects of Computing, 1999, Vol. 10, pp.290-306.

Hoare69: C. A. R. Hoare, "An axiomatic basis for computer programming", Communication of the ACM, October 1969, Volume 12, Number 10, pp 576-583.

Hoare05: T. Hoare and J. Misra, "Verified software: theories, tools, experiments, Vision of a Grand Challenge project", The IFIP working conference on verified software: Theories, tools, experiments. October 2005. http://vstte.inf.ethz.ch/pdfs/vstte-hoare-misra.pdf.

Hooman91: J. Hooman, "Specification and Compositional Verification of Real-Time Systems", in Specification and Compositional Verification of Real-Time Systems, number 558 in Lecture Notes in Computer Science Springer-Verlag, 1991, ISBN 3-540-54947-1.

Ireland04: A. Ireland, "A Practical Perspective on the Verifying Compiler Proposal", Grand Challenges in Computing Research Conference, BCS, IEE and UKCRC. Available from School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0025, 2004. http://www.macs.hw.ac.uk/~air/clamspark/publications/tr25.pdf

Jackson06: M. Jackson, "What can we expect from program verification?", IEEE Computer Society, October 2006, pp. 65-71.

Jasmine07: K. S. Jasmine and R. Vasantha, "Identification of software performance bottleneck components in reuse based software products with the application of acquaintanceship graphs", International conference on software engineering advances (ICESA 2007), IEEE Computer Society, 2007.

Kaivola02: R. Kaivola, and N. Narasimhan, "Formal verification of the Pentium 4 floating point multiplier" in Proceedings of Design, Automation and Test in Europe Conference and Exhibition, Paris, France, 2002, pp. 20-27.

Kiczales05: G. Kiczales and Mira Mezini, "Aspect-oriented programming and modular reasoning", ICSE '05: International Conference on Software Engineering, Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, May 15-21, 2005, pp. 49-58.

Kim04: S. D. Kim and S. H. Chang, "A systematic method to identify software components", Software Engineering Conference, 2004. 11th Asia-Pacific 30 Nov. - 3 Dec. 2004, pp. 538-545.

King00: S. King, J. Hammond, R. Chapman, and A. Pryor, "Is proof more cost-effective than testing?", IEEE Transactions on Software Engineering, Volume 26, Issue 8, August 2000, pp. 675-686.

Kirner05: R. Kirner and P. Puschner, "Classification of WCET analysis techniques", Object-oriented real time distributed computing, IEEE, May 2005, ISBN 0-7695-2356-0, pp. 190-199.

Kirner04: R. Kirner, P. Puschner, and I. Wenzel, "Measurement-based worst-case execution time analysis using automatic test-data generation", In Proceedings of the 4th Euromicro International Workshop on WCET Analysis , June 2004, pp. 67–70.

Kirschenbaum09: J. Kirschenbaum, B. Adcock, D. Bronish, H. Smith, H. Harton, M. Sitaraman, and B. W. Weide, "Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping?", Formal foundations of reuse and domain engineering, LNCS 5791, Proceedings of 11th international conference on software reuse, ICSR 2009, Falls Church, Virginia, pp. 31-40.

Klebanov11: V. Klebanov, et al, "The 1st Verified Software Competition: Experience Report". In FM 2011: Formal Methods - 17th International Symposium on Formal Methods, volume 6664 of LNCS, pages 154-168. Springer, 2011

Konkin98: D. P. Konkin, G. M. Oster, and R. B. Bunt, "Exploiting software interface for performance measurement", Workshop on Software and Performance, WOSP98, Sante Fe, New Mexico, October 12-16, 1998, pp. 208-218.

Krone06: J. Krone, W. F. Ogden, and M. Sitaraman, "Performance analysis based upon complete Profiles", Fifth International Workshop on Specification and verification of Component-Based Systems (SAVCBS 2006), Portland, Oregon, USA, November 2006.

Kulczycki04: G. Kulczycki, "Direct Reasoning", Ph. D. Dissertation, Clemson University, January 2004.

Land09: R. Land, D. Sundmark, F. Luders, I. Krasterva, and A. Causevic, "Reuse with software components – A survey of industrial state of practice", Formal foundations of reuse and domain engineering, LNCS 5791, Proceedings of 11th international conference on software reuse, ICSR 2009, Falls Church, Virginia, pp. 150-159.

Leavens: Leavens, G. T., JML Reference Manual, http://www.eecs.ucf.edu/~leavens/JML.

Leavens06: G.T. Leavens, D.A. Naumann, "Behavioral subtyping, specification inheritance, and modular reasoning", Technical Report 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 2006.

Leino10L: K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness", In LPAR-16, Springer, 2010, Volume 6355 of LNCS, pp. 348-370.

Leino10T: K. R. M. Leino and P. Rummer, "A polymorphic intermediate verification language: Design and logical encoding", In TACAS-16, Springer, 2010, Volume 6015 of LNCS, pp. 312-327.

Leonard09: Leonard, D., Hallstrom, J., and Sitaraman, M., "Injecting rapid feedback and collaborative reasoning in teaching specifications". In ACM SIGCSE Bulletin, ACM, 2009, Vol. 41, pp. 524-528.

Lermer05: K. Lermer, C. J. Fidge, and I. J. Hayes, "A theory for execution-time derivation in real-time programs", Theoretical Computer Science, November 2005, Volume 346, Issue 1, pp. 3-27.

Liskkov93: B. Liskkov, "A History of CLU", ACM SIGPLAN Notices, Volume 28, Number 3, March 1993, pp. 133-147.

Lokuciejewski11: P. Lokuciejewski, P. Marwedel, "Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems", Springer Netherlands, ISBN 978-90-481-9928-0.

Lopez-Garcia12: P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo, "Interval-Based Resource Usage Verification: Formalization and Prototype", Foundational and Practical Aspects of Resource Analysis, Lecture Notes in Computer Science, Volume 7177, 2012, pp 54-71.

Mackenzie95: D. Mackenzie, "The automation of proof: a historical and sociological exploration" IEEE Annals of the History of Computing, Volume 17, Number 3, 1995, pp. 7-29.

Meyer85: B. Meyer, "On formalism in specification" IEEE Software, Volume 2, Issue 1, January 1985, pp. 6-26.

Morgan88: C. Morgan, K. Robinson, and P. Gardiner, "On the refinement calculus", Oxford University Computing Laboratory, 1988, ISBN 0-902928-52-X.

ParnasN72: D. L. Parnas, "A technique for software module specification with examples", Communications of the ACM, Volume 15, Number 5, May 1972, pp. 330-336.

ParnasD72: D. L. Parnas, "On the criteria to be used in decomposing system into modules", Communications of the ACM, Volume 15, Number 12, December 1972, pp. 1053-11058.

ParnasM85: D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems", IEEE Transactions on Software Engineering, Volume SE-11, Issue 3, March 1985, pp. 259-266.

Petriu02: D. Petriu and M. Woodside, "Performance analysis in the software lifecycle: Analyzing software requirements specifications for performance", In Proceedings of the third international workshop on Software and performance, July 24-26, 2002, Rome, Italy.

Petters04: S. M. Petters, A. Betts, and G. Betts, "A new timing schema for WECT analysis", In Proceedings of the fourth international workshop on Worst case execution Time Analysis, June 2004.

Petters07: S. M. Petters, P. Zadarnowski, and G. Heiser, "Measurements or static analysis or both", In Proceedings of the 7th international workshop on Worst case execution Time Analysis, July 2007, Pisa, Italy, pp. 5-12.

Pooley00: R. Pooley, "Software engineering and performance: a roadmap", Proceedings of the Conference on The Future of Software Engineering, May 2000.

Popovic02: M. Popovic, V. Kovacevic, and I. Velikic, "A formal software verification concept based on automated theorem proving and reverse engineering", Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based systems (ECBS '02), 2002.

Pour98: G. Pour, "Moving toward component-based software development approach", IEEE Computer Society, In Proceedings of Technology of Object-Oriented Languages, TOOLS 27, 1998, pp. 296 – 300.

Puschner99: P. Puschner and A. Burns, "A review of worst-case execution-time analysis", Kluwer Academic Publisher, 1999, Printed in Netherland.

Ravindar11: A. Ravindar and Y. N. Srikant, "Implications of Program Phase Behavior on Timing Analysis", Interaction between Compilers and Computer Architectures, 2011, pp. 71-79.

Ravindar11G: A. Ravindar and Y. N. Srikant, "Relative roles of instruction count and cycles per instruction in WCET estimation", Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011, ISBN 978-1-4503-0519-8, pp. 55-60.

Ravindar12: A. Ravindar and Y. N. Srikant, "Estimation of probabilistic bounds on phase CPI and relevance in WCET analysis", Proceedings of the tenth ACM international conference on Embedded software, ISBN: 978-1-4503-1425-1, pp. 165-174.

Richter03: K. Richter, M. Jersak, and R. Ernst, "A formal approach to MpSoc performance verification", IEEE Computer Society, April 2003, pp. 60-67.

Schoeberl06: M. Schoeberl, "A time predictable Java processor", In Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006), European Design and Automation Association, Leuven, Belgium: Munich, Germany, March 2006, pp. 800–805.
Schoeberl10: M. Schoeberl, W. Puffitch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a Java processor", Software - Practice & Experience, 2010, Volume 40, Issue 6, pp. 507-542.

Schupp04: S. Schupp, M. Zalewski, and K. Ross, "Rapid performance prediction for library components", In Workshop on Software and Performance archive Proceedings of the 4th international workshop on Software and performance, Redwood Shores, California, 2004, pp. 69-73.

Sevcik07: J. Sevcik, "Proving Resource Consumption of Low-level programs using automated theorem provers", Electronic Notes in Theoretical Computer Science, Volume 190, Issue 1, July 2007, pp. 133-147.

Shaw89: A. C. Shaw, "Reasoning about time in higher-level language software." IEEE Transactions on Software Engineering, Vol 15, No. 7, July 1989, pp. 875-889.

Siddiqui02: K. Siddiqui and M. Woodside, "Performance aware software development (PASD) using resources demand budgets", WOSP, Proceedings of the 3rd International Workshop on Software Performance '02, July 24-26, Rome, Italy, 2002, pp.275-285.
Sitaraman94: M. Sitaraman and B. Weide, "Component-based software using RESOLVE". ACM SIGSOFT Software Engineering Notes, Vol. 19, Issue 4, 1994, pp. 21-22.

SitaramanM01: Sitaraman, M., "Compositional performance reasoning", 4th ICSE Workshop on Component-Based Software Engineering Component Certification and System Prediction. May 14-15, Toronto, Canada, 2001.

Sitaraman11: M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. Weide, "Building a Push button RESOLVE Verifier: Progress and Challenges". In Formal Aspects of Computing, Springer, 2011. pp. 607-626

Smith90: C. U. Smith, "Performance engineering of software systems" Addision-Wesely Publisher, Seventh Edition, 1990.

Smith99: C. U. Smith and C.M. Woodside, "Performance Validation at Early Stages of Software Development", in E. Gelenbe ed. System Performance Evaluation: Methodologies and Applications, CRC Press, 1999.

Smith02: C. U. Smith and L. G. Williams, "Performance solutions: A practical guide to creating responsive, scalable software", Published by Addison-Wesley, 2002.

Smith03: C. U. Smith and L. G. Williams, "Ten best practices for software performance engineering", MeasureIT, Computer Measurement Group online Newsletter, Issue 4.0, June, 2003.

Smith08: Smith, H., Roche, K., Sitaraman, M., Krone, J., and Ogden, W., "Integrating math units and proof checking for specification and verification". In Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008), 2008, pages 59-66.

Smith13: H. W., Smith, "Engineering Specifications and Mathematics for Verified Software". Ph. D. Dissertation, Clemson University, 2013, 240 pages.

Stankovic88: J. A. Stankovic and K. Ramamritham, "Tutorial Hard Real-Time Systems", The computer Society, 1988, ISBN 0-8186-0819-6.

Stata95: R. Stata and J. V. Guttag, "Modular reasoning in the presence of subclassing", ACM SIGPLAN Notices, Volume 30, Issue 10, October 1995, pp. 200-214.

Szyperski98: C. Szyperski, "Component software: Beyond object-oriented programming", Addision_Wesely Publisher, Second edition, 1998, ISBN 0-201-17888-5.

Tetzlaff13: D. Tetzlaff and S. Glesner, "Intelligent Prediction of Execution Times", Informatics and Applications, 2013, Second International Conference on Informatics & Applications, May 2013, pp. 234-239.
Wilhelm08: R., Wilhelm, "The Worst-Case Execution Time Problem." Journal ACM Transactions on Embedded Computing Systems (TECS), April 2008, Vol. 7 Issue 3, No. 36.

Williams95: L.G. Williams and C. U. Smith, "Information Requirements for Software Performance Engineering", in Quantitative Evaluation of Computing and Communication Systems, number 977 in Lecture Notes in Computer Science, Springer-Verlag, 1995.

Woodside02: M. Woodside, D. Petriu, and K. Siddiqui, "Technical papers: software specification: Performance-related completions for software specifications", Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, May 2002, pp. 22-32.

Wu03: X. Wu, D. McMullan, M. Woodside, "Component-based performance engineering", Proceedings of the 6th ICSE workshop on component-based software engineering: Automated reasoning and prediction, Portland, Oregon, USA, May 3-4, 2003.

Wu04: X. Wu and M. Woodside, "Performance modeling from software components", ACM SIGSOFT Software Engineering Notes, Proceedings of the fourth international workshop on Software and performance, Redwood Shores, California, 2004, pp. 290-301.

Yacoub02: S. Yacoub, "Performance analysis of component-based application", Lecture notes in computer science, Springer Berlin Publisher, Proceedings software product line: second international conference, San Diego, CA, USA, August 19-22, 2002, pp.1-5.

Young80: W. D. Young and D. I. Good, "Generics and verification in ADA", ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN symposium on ADA programming language ACM SIGPLAN'80, Volume 15, Issue 11, November 1980.

Webster60: "Webster's new collegiate dictionary", G. &C. Merriam Co., Publishers, 1960.

LIST OF APPENDICES

APPENDIX A: DURATION BASIC THEORY

The performance profiles, also includes one or more of the following term. These terms comes from the *Duration_Basic_Theory*, Fig A.1. The *Duration_Basic_Theory* is a math unit that precisely defines duration theoretic terms.

I_Dur(Entry): The time needed tocreate an initiale an *Entry*.

F_IV_Dur(Entry): The duration to finalize an initial valued *Entry*.

F_Dur(Entry, E): The duration to finalize an *Entry* type element *E*.

Dur_Call (n): The duration to call an operation with *n* parameters.

Dur_Assgn: Duration for the assignment (:=) operation.

Dur_Swap: Duration for the swap (:=:) operation.

```
Theory Duration_Basics_Theory;
    uses Real_Number_Theory;

    -- all definitions are syntactic place holders

    Definition I_Dur (T: MType): RPos;
    Definition F_Dur (T: MType, x: T): RPos;
    Definition F_IV_Dur (T: MType): RPos;

    Definition Dur_Call (n: N): RPos;
    Definition Dur_Assgn: RPos;
    Definition Dur_Swap: RPos;

end Duration_Basics_Theory;
```

Fig A.1: Duration basic theory

APPENDIX B: ADDITIONAL SPECIFICATIONS AND REALIZATIONS

B1: INTEGER_TEMPLATE

```
Concept Integer_Template;
    uses Integer_Theory;

    Definition min_int: Z;
    Definition 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;
    end;

    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 );
```

Integer_Template (continued)

```
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_into(updates i: Integer; evaluates j: Integer);
    requires j /= 0;
    ensures i = ( #i/j ):
```

Integer_Template (continued)

```
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))
           else true;
      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;
      requires j /= 0;
      ensures Rem = i - ( i/j ) * j;

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;
```

Integer_Template (continued)

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

    Operation Clear(clears i: Integer);

end Integer_Template;
```

B2: ITP (PERFORMANCE PROFILE FOR INTEGER_TEMPLATE)

```
Operation Mod(evaluates i, j: Integer): Integer;
    duration ITP_Mod;

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

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

Operation Div(evaluates i, j: Integer): Integer;
    duration ITP_Div;

Operation Are_Equal(evaluates i, j: Integer): Boolean;
    duration ITP_AreEq;

Operation Are_Not_Equal(evaluates i, j: Integer): Boolean;
    duration ITP_AreNotEq;

Operation Replica(restores i: Integer): Integer;
    duration ITP_Replica;

Operation Read(replaces i: Integer);
    duration ITP_Read;

Operation Write(evaluates i: Integer);
    duration ITP_Write;

Operation Write_Line(evaluates i: Integer);
    duration ITP_WrtLn;

Operation Max_Int(): Integer;
    duration ITP_MaxInt;

Operation Min_Int(): Integer;
    duration ITP_MinInt;
```

ITP (continued)

```
Operation Sum(evaluates i, j: Integer): Integer;
    duration ITP_Sum;

Operation Negate(evaluates i: Integer): Integer;
    duration ITP_Ngt;

Operation Difference(evaluates i, j: Integer): Integer;
    duration ITP_Diff;

Operation Product(evaluates i, j: Integer): Integer;
    duration ITP_Prdct;

Operation Power(evaluates i, j: Integer): Integer;
    duration ITP_Power;

Operation Divide(evaluates i, j: Integer; replaces q: Integer);
    duration ITP_Divide;

Operation Divide_into(updates i: Integer; evaluates j: Integer);
    duration ITP_DivideInto;

Operation Mod(evaluates i, j: Integer): Integer;
    duration ITP_Mod;

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

Operation Div(evaluates i, j: Integer): Integer;
    duration ITP_Div;

Operation Are_Equal(evaluates i, j: Integer): Boolean;
    duration ITP_AreEq;
```

ITP (continued)

```
    Operation Are_Not_Equal(evaluates i, j: Integer): Boolean;
        duration ITP_AreNotEq;

    Operation Replica(restores i: Integer): Integer;
        duration ITP_Replica;

    Operation Read(replaces i: Integer);
        duration ITP_Read;

    Operation Write(evaluates i: Integer);
        duration ITP_Write;

    Operation Write_Line(evaluates i: Integer);
        duration ITP_WrtLn;

    Operation Max_Int(): Integer;
        duration ITP_MaxInt;

    Operation Min_Int(): Integer;
        duration ITP_MinInt;

    Operation Clear(clears i: Integer);
        duration ITP_Cr;

end ITP;
```

B3: STACK_TEMPLATE

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

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

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

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

B4: SSC (PERFORMANCE PROFILE FOR STACK_TEMPLATE)

```
Profile SSC short_for Space_Conscious for Stack_Template;
    uses Real_Number_Theory;
    uses  Duration_Basics_Theory;

        Definition Cnts_Dur(S: Str(Entry): RPos = Sigma (
                        x: Entry, Occurs_Ct(x, S) * F_Dur(Entry, x));


    Type Family Stack is modeled by Str( Entry );
        initialization
            duration  SSCI1 +
                                (SSCI2 + I_Dur(Entry)) * Max_Depth;

            finalization
                duration  SSCF1 + Cnts_Dur( #S) +
                 ( SSCF2 + F_IV_Dur(Entry) ) * ( Max_Depth - |#S| );
   end;

    Operation  Push( clears E: Entry; updates S: Stack );
        duration  SSCPu;

    Operation  Pop( replaces E: Entry; updates S: Stack );
        duration  ( SSCPo + I_Dur(Entry) ) + F_Dur (Entry, #E);

    Operation  Depth( restores S: Stack ): Integer;
        duration  SSCDp;

    Operation  Rem_Capacity( restores S: Stack ): Integer;
        duration  SSCRc;

    Operation  Clear( clears S: Stack );
        duration  SSCC1 + Cnts_Dur( #S ) +
            (SSCC2 + F_IV_Dur(Entry)) * ( Max_Depth -|#S| );
```

APPENDIX C: VERIFICATION CONDITIONS FILES

C1: APPEND_REALIZ_1.RB

VCs for Append_Realiz_1.rb generated Wed Mar 04 00:03:45 EST 2015

========== VC(s): ==========

VC 0_1
Base Case of the Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. Entry.Is_Initial(Next)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. ((|P| + |Q|) <= Max_Length)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)
8. (Max_Length > 0)

VC 0_2
Base Case of the Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. Entry.Is_Initial(Next)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. ((|P| + |Q|) <= Max_Length)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)
8. (Max_Length > 0)

VC 0_3
Base Case of the Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
(|Q| = |Q|)

Given(s):
1. Entry.Is_Initial(Next)
2. (|Q| <= Max_Length)

3. (|P| <= Max_Length)
4. ((|P| + |Q|) <= Max_Length)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)
8. (Max_Length > 0)

VC 0_4
Requires Clause of Dequeue in Procedure Append_to: Append_Realiz_1.rb(12)

Goal(s):
(|Q"| /= 0)

Given(s):
1. (Len" /= 0)
2. (P_val' = |Q"|)
3. (Len" = |Q"|)
4. Entry.Is_Initial(Next"')
5. ((P" o Q") = (P o Q))
6. Entry.Is_Initial(Next)
7. (|Q| <= Max_Length)
8. (|P| <= Max_Length)
9. ((|P| + |Q|) <= Max_Length)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_5
Requires Clause of Enqueue in Procedure Append_to: Append_Realiz_1.rb(13)

Goal(s):
(|P"| < Max_Length)

Given(s):
1. (Q" = (<Next"> o Q'))
2. (Len" /= 0)
3. (P_val' = |Q"|)
4. (Len" = |Q"|)
5. Entry.Is_Initial(Next"')
6. ((P" o Q") = (P o Q))
7. Entry.Is_Initial(Next)
8. (|Q| <= Max_Length)
9. (|P| <= Max_Length)
10. ((|P| + |Q|) <= Max_Length)
11. (Last_Char_Num > 0)

12. (0 < max_int)
13. (min_int <= 0)
14. (Max_Length > 0)

VC 0_6
Requires Clause of Decrement in Procedure Append_to: Append_Realiz_1.rb(14)

Goal(s):
(min_int <= (Len" - 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P" o <Next">))
3. (Q" = (<Next"> o Q'))
4. (Len" /= 0)
5. (P_val' = |Q"|)
6. (Len" = |Q"|)
7. Entry.Is_Initial(Next"')
8. ((P" o Q") = (P o Q))
9. Entry.Is_Initial(Next)
10. (|Q| <= Max_Length)
11. (|P| <= Max_Length)
12. ((|P| + |Q|) <= Max_Length)
13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Length > 0)

VC 0_7
Inductive Case of Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
((P' o Q') = (P o Q))

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P" o <Next">))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. (P_val' = |Q"|)
7. (Len" = |Q"|)
8. Entry.Is_Initial(Next"')
9. ((P" o Q") = (P o Q))
10. Entry.Is_Initial(Next)
11. (|Q| <= Max_Length)

12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_8
Inductive Case of Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
Entry.Is_Initial(Next')

Given(s):
1. (Len' = (Len'' - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P'' o <Next''>))
4. (Q'' = (<Next''> o Q'))
5. (Len'' /= 0)
6. (P_val' = |Q''|)
7. (Len'' = |Q''|)
8. Entry.Is_Initial(Next''')
9. ((P'' o Q'') = (P o Q))
10. Entry.Is_Initial(Next)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_9
Inductive Case of Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
(Len' = |Q'|)

Given(s):
1. (Len' = (Len'' - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P'' o <Next''>))
4. (Q'' = (<Next''> o Q'))
5. (Len'' /= 0)
6. (P_val' = |Q''|)
7. (Len'' = |Q''|)

8. Entry.Is_Initial(Next"')
9. ((P" o Q") = (P o Q))
10. Entry.Is_Initial(Next)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_10
Termination of While Statement: Append_Realiz_1.rb(10)

Goal(s):
(|Q'| < P_val')

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P" o <Next">))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. (P_val' = |Q"|)
7. (Len" = |Q"|)
8. Entry.Is_Initial(Next"')
9. ((P" o Q") = (P o Q))
10. Entry.Is_Initial(Next)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 1_1
Base Case of the Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. Entry.Is_Initial(Next)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)

4. ((|P| + |Q|) <= Max_Length)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)
8. (Max_Length > 0)

VC 1_2
Base Case of the Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. Entry.Is_Initial(Next)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. ((|P| + |Q|) <= Max_Length)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)
8. (Max_Length > 0)

VC 1_3
Base Case of the Invariant of While Statement: Append_Realiz_1.rb(9)

Goal(s):
(|Q| = |Q|)

Given(s):
1. Entry.Is_Initial(Next)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. ((|P| + |Q|) <= Max_Length)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)
8. (Max_Length > 0)

VC 1_4
Ensures Clause of Append_to: Append_Realiz_1.rb(3)

Goal(s):
(P' = (P o Q))

Given(s):
1. (Len' = 0)
2. (P_val' = |Q'|)
3. (Len' = |Q'|)
4. Entry.Is_Initial(Next')
5. ((P' o Q') = (P o Q))
6. Entry.Is_Initial(Next)
7. (|Q| <= Max_Length)
8. (|P| <= Max_Length)
9. ((|P| + |Q|) <= Max_Length)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 1_5
Ensures Clause of Append_to: Append_Realiz_1.rb(3)

Goal(s):
(Q' = Empty_String)

Given(s):
1. (Len' = 0)
2. (P_val' = |Q'|)
3. (Len' = |Q'|)
4. Entry.Is_Initial(Next')
5. ((P' o Q') = (P o Q))
6. Entry.Is_Initial(Next)
7. (|Q| <= Max_Length)
8. (|P| <= Max_Length)
9. ((|P| + |Q|) <= Max_Length)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

C2: APPEND_SOME_REALIZ_2.RB

VCs for Append_Some_Realiz_2.rb generated Thu Apr 09 13:44:01 EDT 2015

========== VC(s): ==========

VC 0_1
Ensures Clause of Append_Some: Append_Some_Realiz_2.rb(4)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. (I = 0)
2. (temp_Q = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (|P| <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_2
Duration Clause of Append_Some: Append_Some_Realiz_2.rb(4)

Goal(s):
(((((Cum_Dur + I_Dur(Entry)) + I_Dur(P_Queue)) + I_Dur(Integer)) + ((F_Dur(Entry, Next) + F_Dur(P_Queue, temp_Q)) + F_Dur(Integer, I))) <= ((((C + I_Dur(Entry)) + I_Dur(P_Queue)) + F_IV_Dur(Entry)) + F_IV_Dur(P_Queue)))

Given(s):
1. (I = 0)
2. (temp_Q = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (|P| <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

C3: APPEND_SOME_REALIZ_3.RB

VCs for Append_Some_Realiz_3.rb generated Thu Apr 09 13:49:00 EDT 2015

========== VC(s): ==========

VC 0_1
Ensures Clause of Append_Some: Append_Some_Realiz_3.rb(4)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. (Cum_Dur = 0.0)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. (Last_Char_Num > 0)
5. (0 < max_int)
6. (min_int <= 0)
7. (Max_Length > 0)

VC 0_2
Duration Clause of Append_Some: Append_Some_Realiz_3.rb(4)

Goal(s):
(((Cum_Dur + Dur_Swap) + Dur_Swap) <= C)

Given(s):
1. (Cum_Dur = 0.0)
2. (|Q| <= Max_Length)
3. (|P| <= Max_Length)
4. (Last_Char_Num > 0)
5. (0 < max_int)
6. (min_int <= 0)
7. (Max_Length > 0)

C4: APPEND_ONE_REALIZ_1.RB

VCs for Append_One_Realiz_1.rb generated Thu Apr 09 13:50:20 EDT 2015

========== VC(s): ==========

VC 0_1
Requires Clause of Dequeue in Procedure Append_One: Append_One_Realiz_1.rb(8)

Goal(s):
(|Q| /= 0)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (|Q| <= Max_Length)
4. (|P| <= Max_Length)
5. (|P| < Max_Length)
6. (|Q| /= 0)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_2
Requires Clause of Enqueue in Procedure Append_One: Append_One_Realiz_1.rb(9)

Goal(s):
(|P| < Max_Length)

Given(s):
1. (Q = (<Next"> o Q'))
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. (|P| < Max_Length)
7. (|Q| /= 0)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 0_3
Ensures Clause of Append_One: Append_One_Realiz_1.rb(5)

Goal(s):
((P' o Q') = (P o Q))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. (|P| < Max_Length)
9. (|Q| /= 0)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_4
Ensures Clause of Append_One: Append_One_Realiz_1.rb(5)

Goal(s):
(|P'| = (|P| + 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. (|P| < Max_Length)
9. (|Q| /= 0)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_5
Duration Clause of Append_One: Append_One_Realiz_1.rb(5)

Goal(s):
(((((Cum_Dur + I_Dur(Entry)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next)) + Dur_Call(2))) + (CEn + Dur_Call(2))) + F_Dur(Entry, Next')) <= ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. (|P| < Max_Length)
9. (|Q| /= 0)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

C5: APPEND_TO_REALIZ_1.RB

VCs for Append_to_Realiz_1.rb generated Thu Apr 09 13:51:56 EDT 2015

========== VC(s): ==========

VC 0_1
Requires Clause of Dequeue in Procedure Append_to: Append_to_Realiz_1.rb(19)

Goal(s):
(|Q| /= 0)

Given(s):
1. (|Q| /= 0)
2. (Len = 0)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (|P| <= Max_Length)
7. (|P| < Max_Length)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 0_2
Requires Clause of Enqueue in Procedure Append_to: Append_to_Realiz_1.rb(20)

Goal(s):
(|P| < Max_Length)

Given(s):
1. (Q = (<Next"> o Q'))
2. (|Q| /= 0)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. (|P| < Max_Length)
9. (Last_Char_Num > 0)
10. (0 < max_int)
11. (min_int <= 0)
12. (Max_Length > 0)

VC 0_3
Ensures Clause of Append_to: Append_to_Realiz_1.rb(12)

Goal(s):
((P' o Q') = (P o Q))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. (|Q| /= 0)
5. (Len = 0)
6. Entry.Is_Initial(Next)
7. (Cum_Dur = 0.0)
8. (|Q| <= Max_Length)
9. (|P| <= Max_Length)
10. (|P| < Max_Length)
11. (Last_Char_Num > 0)
12. (0 < max_int)
13. (min_int <= 0)
14. (Max_Length > 0)

VC 0_4
Duration Clause of Append_to: Append_to_Realiz_1.rb(12)

Goal(s):
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next)) + Dur_Call(2))) + (CEn + Dur_Call(2))) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next') + F_Dur(Integer, |Q|))) <= ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P o <Next">))
3. (Q = (<Next"> o Q'))
4. (|Q| /= 0)
5. (Len = 0)
6. Entry.Is_Initial(Next)
7. (Cum_Dur = 0.0)
8. (|Q| <= Max_Length)
9. (|P| <= Max_Length)
10. (|P| < Max_Length)
11. (Last_Char_Num > 0)
12. (0 < max_int)
13. (min_int <= 0)
14. (Max_Length > 0)

VC 1_1
Ensures Clause of Append_to: Append_to_Realiz_1.rb(12)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. (|Q| = 0)
2. (Len = 0)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (|P| <= Max_Length)
7. (|P| < Max_Length)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 1_2
Duration Clause of Append_to: Append_to_Realiz_1.rb(12)

Goal(s):
(((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next) + F_Dur(Integer, |Q|))) <= ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry))))

Given(s):
1. (|Q| = 0)
2. (Len = 0)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (|P| <= Max_Length)
7. (|P| < Max_Length)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

C6: APPEND_REALIZ_2.RB

VCs for Append_Realiz_2.rb generated Thu Apr 09 13:53:26 EDT 2015

========== VC(s): ==========

VC 0_1
Base Case of the Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_2
Base Case of the Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_3
Base Case of the Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
(|Q| = |Q|)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Append_Realiz_2.rb(18)

Goal(s):
(((|Q| - |Q|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_5
Requires Clause of Dequeue in Procedure Append_to: Append_Realiz_2.rb(21)

Goal(s):
(|Q"| /= 0)

Given(s):
1. (Len" /= 0)
2. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

3. (P_val' = |Q"|)
4. (Len" = |Q"|)
5. Entry.Is_Initial(Next"')
6. ((P" o Q") = (P o Q))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (|Q| <= Max_Length)
11. (|P| <= Max_Length)
12. ((|P| + |Q|) <= Max_Length)
13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Length > 0)

VC 0_6
Requires Clause of Enqueue in Procedure Append_to: Append_Realiz_2.rb(22)

Goal(s):
(|P"| < Max_Length)

Given(s):
1. (Q" = (<Next"> o Q'))
2. (Len" /= 0)
3. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
4. (P_val' = |Q"|)
5. (Len" = |Q"|)
6. Entry.Is_Initial(Next"')
7. ((P" o Q") = (P o Q))
8. (Len = 0)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_7
Requires Clause of Decrement in Procedure Append_to: Append_Realiz_2.rb(23)

Goal(s):
(min_int <= (Len" - 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (P' = (P" o <Next">))
3. (Q" = (<Next"> o Q'))
4. (Len" /= 0)
5. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
6. (P_val' = |Q"|)
7. (Len" = |Q"|)
8. Entry.Is_Initial(Next"')
9. ((P" o Q") = (P o Q))
10. (Len = 0)
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (|Q| <= Max_Length)
14. (|P| <= Max_Length)
15. ((|P| + |Q|) <= Max_Length)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 0_8
Inductive Case of Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
((P' o Q') = (P o Q))

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P" o <Next">))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next"')
10. ((P" o Q") = (P o Q))

11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (|Q| <= Max_Length)
15. (|P| <= Max_Length)
16. ((|P| + |Q|) <= Max_Length)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 0_9
Inductive Case of Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
Entry.Is_Initial(Next')

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P" o <Next">))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next"')
10. ((P" o Q") = (P o Q))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (|Q| <= Max_Length)
15. (|P| <= Max_Length)
16. ((|P| + |Q|) <= Max_Length)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 0_10
Inductive Case of Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
(Len' = |Q'|)

Given(s):
1. (Len' = (Len'' - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P'' o <Next''>))
4. (Q'' = (<Next''> o Q'))
5. (Len'' /= 0)
6. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q''|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q''|)
8. (Len'' = |Q''|)
9. Entry.Is_Initial(Next''')
10. ((P'' o Q'') = (P o Q))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (|Q| <= Max_Length)
15. (|P| <= Max_Length)
16. ((|P| + |Q|) <= Max_Length)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 0_11
Termination of While Statement: Append_Realiz_2.rb(16)

Goal(s):
(|Q'| < P_val')
Given(s):
1. (Len' = (Len'' - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P'' o <Next''>))
4. (Q'' = (<Next''> o Q'))
5. (Len'' /= 0)
6. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q''|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q''|)
8. (Len'' = |Q''|)
9. Entry.Is_Initial(Next''')
10. ((P'' o Q'') = (P o Q))

11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (|Q| <= Max_Length)
15. (|P| <= Max_Length)
16. ((|P| + |Q|) <= Max_Length)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 0_12
Termination of While Statement: Append_Realiz_2.rb(16)

Goal(s):
((((((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next'")) + Dur_Call(2))) + (CEn + Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <= ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (P' = (P" o <Next">))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. ((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next'")
10. ((P" o Q") = (P o Q))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (|Q| <= Max_Length)
15. (|P| <= Max_Length)
16. ((|P| + |Q|) <= Max_Length)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 1_1
Base Case of the Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
((P o Q) = (P o Q))

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 1_2
Base Case of the Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 1_3
Base Case of the Invariant of While Statement: Append_Realiz_2.rb(15)

Goal(s):
(|Q| = |Q|)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)

4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 1_4
Base Case of Elapsed Time Duration of While Statement: Append_Realiz_2.rb(18)

Goal(s):

(((|Q| - |Q|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (|Q| <= Max_Length)
5. (|P| <= Max_Length)
6. ((|P| + |Q|) <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 1_5
Ensures Clause of Append_to: Append_Realiz_2.rb(8)

Goal(s):
(P' = (P o Q))

Given(s):
1. (Len' = 0)
2. (Cum_Dur' = ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q'|)
4. (Len' = |Q'|)
5. Entry.Is_Initial(Next')
6. ((P' o Q') = (P o Q))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (|Q| <= Max_Length)
11. (|P| <= Max_Length)
12. ((|P| + |Q|) <= Max_Length)

13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Length > 0)

VC 1_6
Ensures Clause of Append_to (Condition from "clears" parameter mode): Append_Realiz_2.rb(8)

Goal(s):
(Q' = Empty_String)

Given(s):
1. (Len' = 0)
2. (Cum_Dur' = ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q'|)
4. (Len' = |Q'|)
5. Entry.Is_Initial(Next')
6. ((P' o Q') = (P o Q))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (|Q| <= Max_Length)
11. (|P| <= Max_Length)
12. ((|P| + |Q|) <= Max_Length)
13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Length > 0)

VC 1_7
Duration Clause of Append_to: Append_Realiz_2.rb(8)

Goal(s):
(((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + (F_Dur(Entry, Next') + F_Dur(Integer, Len'))) <= (((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) + (|Q| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (Len' = 0)
2. (Cum_Dur' = ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q'|)
4. (Len' = |Q'|)
5. Entry.Is_Initial(Next')
6. ((P' o Q') = (P o Q))

7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (|Q| <= Max_Length)
11. (|P| <= Max_Length)
12. ((|P| + |Q|) <= Max_Length)
13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Length > 0)

C7: HALVING_REALIZ.RB

VCs for Halving_Realiz.rb generated Thu Apr 09 14:30:15 EDT 2015

========== VC(s): ==========

VC 0_1
Requires Clause of Increment in Procedure Halve: Halving_Realiz.rb(8)

Goal(s):
((J + 1) <= max_int)

Given(s):
1. (J = 0)
2. (Cum_Dur = 0.0)
3. (I <= max_int)
4. (min_int <= I)
5. (Last_Char_Num > 0)
6. (0 < max_int)
7. (min_int <= 0)

VC 0_2
Requires Clause of Increment in Procedure Halve: Halving_Realiz.rb(9)

Goal(s):
((J" + 1) <= max_int)

Given(s):
1. (J" = (J + 1))
2. (J = 0)
3. (Cum_Dur = 0.0)
4. (I <= max_int)
5. (min_int <= I)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)

VC 0_3
Requires Clause of Div in Procedure Halve: Halving_Realiz.rb(11)

Goal(s):
(J' /= 0)

Given(s):
1. (J' = (J" + 1))
2. (J" = (J + 1))
3. (J = 0)

4. (Cum_Dur = 0.0)
5. (I <= max_int)
6. (min_int <= I)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)

VC 0_4
Ensures Clause of Halve: Halving_Realiz.rb(5)

Goal(s):
((I / J') = (I / 2))

Given(s):
1. (J' = (J" + 1))
2. (J" = (J + 1))
3. (J = 0)
4. (Cum_Dur = 0.0)
5. (I <= max_int)
6. (min_int <= I)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)

VC 0_5
Duration Clause of Halve: Halving_Realiz.rb(5)

Goal(s):
((((((((((Cum_Dur + I_Dur(Integer)) + (ITP_Incr + Dur_Call(1))) + (ITP_Incr + Dur_Call(1))) + (ITP_Div + Dur_Call(2))) + F_Dur(Integer, I)) + F_Dur(Integer, J')) + Dur_Assgn) + F_Dur(Integer, Halve)) + F_Dur(Integer, J')) <= C)

Given(s):
1. (J' = (J" + 1))
2. (J" = (J + 1))
3. (J = 0)
4. (Cum_Dur = 0.0)
5. (I <= max_int)
6. (min_int <= I)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)

C8: OBVIOUS_FLIP_REALIZ.RB

VCs for Obvious_Flip_Realiz.rb generated Mon Apr 06 21:58:19 EDT 2015

========== VC(s): ==========

VC 0_1
Base Case of the Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
(S = (Reverse(S_Flipped) o S))

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 0_2
Base Case of the Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 0_3
Base Case of the Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
(|S| = |S|)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Obvious_Flip_Realiz.rb(20)

Goal(s):
((|S_Flipped| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 0_5
Requires Clause of Pop in Procedure Flip: Obvious_Flip_Realiz.rb(22)

Goal(s):
(|S"| /= 0)

Given(s):
1. (Dep" /= 0)
2. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |S"|)

4. (Dep" = |S"|)
5. Entry.Is_Initial(Next"')
6. (S = (Reverse(S_Flipped") o S"))
7. (Dep = 0)
8. (S_Flipped = Empty_String)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|S| <= Max_Depth)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Depth > 0)

VC 0_6
Requires Clause of Push in Procedure Flip: Obvious_Flip_Realiz.rb(23)

Goal(s):
(|S_Flipped"| < Max_Depth)

Given(s):
1. (S" = (<Next"> o S'))
2. (Dep" /= 0)
3. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
4. (P_val' = |S"|)
5. (Dep" = |S"|)
6. Entry.Is_Initial(Next"')
7. (S = (Reverse(S_Flipped") o S"))
8. (Dep = 0)
9. (S_Flipped = Empty_String)
10. Entry.Is_Initial(Next)
11. (Cum_Dur = 0.0)
12. (|S| <= Max_Depth)
13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Depth > 0)

VC 0_7
Requires Clause of Decrement in Procedure Flip: Obvious_Flip_Realiz.rb(24)

Goal(s):
(min_int <= (Dep" - 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (S_Flipped' = (<Next"> o S_Flipped"))
3. (S" = (<Next"> o S'))
4. (Dep" /= 0)
5. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
6. (P_val' = |S"|)
7. (Dep" = |S"|)
8. Entry.Is_Initial(Next"')
9. (S = (Reverse(S_Flipped") o S"))
10. (Dep = 0)
11. (S_Flipped = Empty_String)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (|S| <= Max_Depth)
15. (Last_Char_Num > 0)
16. (0 < max_int)
17. (min_int <= 0)
18. (Max_Depth > 0)

VC 0_8
Inductive Case of Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
(S = (Reverse(S_Flipped') o S'))

Given(s):
1. (Dep' = (Dep" - 1))
2. Entry.Is_Initial(Next')
3. (S_Flipped' = (<Next"> o S_Flipped"))
4. (S" = (<Next"> o S'))
5. (Dep" /= 0)
6. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |S"|)
8. (Dep" = |S"|)
9. Entry.Is_Initial(Next"')

10. (S = (Reverse(S_Flipped") o S"))
11. (Dep = 0)
12. (S_Flipped = Empty_String)
13. Entry.Is_Initial(Next)
14. (Cum_Dur = 0.0)
15. (|S| <= Max_Depth)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Depth > 0)
VC 0_9
Inductive Case of Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
Entry.Is_Initial(Next')

Given(s):
1. (Dep' = (Dep" - 1))
2. Entry.Is_Initial(Next')
3. (S_Flipped' = (<Next"> o S_Flipped"))
4. (S" = (<Next"> o S'))
5. (Dep" /= 0)
6. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |S"|)
8. (Dep" = |S"|)
9. Entry.Is_Initial(Next"')
10. (S = (Reverse(S_Flipped") o S"))
11. (Dep = 0)
12. (S_Flipped = Empty_String)
13. Entry.Is_Initial(Next)
14. (Cum_Dur = 0.0)
15. (|S| <= Max_Depth)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Depth > 0)

VC 0_10
Inductive Case of Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
(Dep' = |S'|)

Given(s):
1. (Dep' = (Dep" - 1))
2. Entry.Is_Initial(Next')
3. (S_Flipped' = (<Next"> o S_Flipped"))
4. (S" = (<Next"> o S'))
5. (Dep" /= 0)
6. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |S"|)
8. (Dep" = |S"|)
9. Entry.Is_Initial(Next"')
10. (S = (Reverse(S_Flipped") o S"))
11. (Dep = 0)
12. (S_Flipped = Empty_String)
13. Entry.Is_Initial(Next)
14. (Cum_Dur = 0.0)
15. (|S| <= Max_Depth)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Depth > 0)

VC 0_11
Termination of While Statement: Obvious_Flip_Realiz.rb(19)

Goal(s):
(|S'| < P_val')

Given(s):
1. (Dep' = (Dep" - 1))
2. Entry.Is_Initial(Next')
3. (S_Flipped' = (<Next"> o S_Flipped"))
4. (S" = (<Next"> o S'))
5. (Dep" /= 0)
6. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped"| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |S"|)
8. (Dep" = |S"|)

9. Entry.Is_Initial(Next''')
10. (S = (Reverse(S_Flipped'') o S''))
11. (Dep = 0)
12. (S_Flipped = Empty_String)
13. Entry.Is_Initial(Next)
14. (Cum_Dur = 0.0)
15. (|S| <= Max_Depth)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Depth > 0)

VC 0_12
Termination of While Statement: Obvious_Flip_Realiz.rb(19)

Goal(s):
((((((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) + (((SSCPo + I_Dur(Entry)) + F_Dur(Entry, Next''')) + Dur_Call(2))) + (SSCPu + Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <= (|S_Flipped'| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (Dep' = (Dep'' - 1))
2. Entry.Is_Initial(Next')
3. (S_Flipped' = (<Next''> o S_Flipped''))
4. (S'' = (<Next''> o S'))
5. (Dep'' /= 0)
6. (((((((Cum_Dur' + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) = (|S_Flipped''| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |S''|)
8. (Dep'' = |S''|)
9. Entry.Is_Initial(Next''')
10. (S = (Reverse(S_Flipped'') o S''))
11. (Dep = 0)
12. (S_Flipped = Empty_String)
13. Entry.Is_Initial(Next)
14. (Cum_Dur = 0.0)
15. (|S| <= Max_Depth)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Depth > 0)

VC 1_1
Base Case of the Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
(S = (Reverse(S_Flipped) o S))

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 1_2
Base Case of the Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 1_3
Base Case of the Invariant of While Statement: Obvious_Flip_Realiz.rb(18)

Goal(s):
(|S| = |S|)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)

6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 1_4
Base Case of Elapsed Time Duration of While Statement: Obvious_Flip_Realiz.rb(20)

Goal(s):
((|S_Flipped| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Dep = 0)
2. (S_Flipped = Empty_String)
3. Entry.Is_Initial(Next)
4. (Cum_Dur = 0.0)
5. (|S| <= Max_Depth)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Depth > 0)

VC 1_5
Ensures Clause of Flip: Obvious_Flip_Realiz.rb(10)

Goal(s):
(S_Flipped' = Reverse(S))

Given(s):
1. (Dep' = 0)
2. (Cum_Dur' = (|S_Flipped'| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |S'|)
4. (Dep' = |S'|)
5. Entry.Is_Initial(Next')
6. (S = (Reverse(S_Flipped') o S'))
7. (Dep = 0)
8. (S_Flipped = Empty_String)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|S| <= Max_Depth)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Depth > 0)

VC 1_6
Duration Clause of Flip: Obvious_Flip_Realiz.rb(10)

Goal(s):
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Stack)) + I_Dur(Integer)) + (SSCDp + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Dep)) + (ITP_AreNotEq + Dur_Call(2))) + (|S_Flipped'| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + Dur_Swap) + ((F_Dur(Entry, Next') + F_Dur(Stack, S')) + F_Dur(Integer, Dep'))) <= (((((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) + I_Dur(Stack)) + F_IV_Dur(Stack)) + (|S| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (Dep' = 0)
2. (Cum_Dur' = (|S_Flipped'| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |S'|)
4. (Dep' = |S'|)
5. Entry.Is_Initial(Next')
6. (S = (Reverse(S_Flipped') o S'))
7. (Dep = 0)
8. (S_Flipped = Empty_String)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|S| <= Max_Depth)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Depth > 0)

C9: APPEND_REALIZ_3.RB

VCs for Append_Realiz_3.rb generated Tue Apr 28 12:48:09 EDT 2015

========== VC(s): ==========

VC 0_1
Requires Clause of Dequeue in Procedure Append_to: Append_Realiz_3.rb(16)

Goal(s):
(|Q| /= 0)

Given(s):
1. (|Q| /= 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. ((|P| + |Q|) <= Max_Length)
9. (Last_Char_Num > 0)
10. (0 < max_int)
11. (min_int <= 0)
12. (Max_Length > 0)

VC 0_2
Requires Clause of Enqueue in Procedure Append_to: Append_Realiz_3.rb(17)

Goal(s):
(|P| < Max_Length)

Given(s):
1. (Q = (<Next"> o Q"))
2. (|Q| /= 0)
3. (P_val = |Q|)
4. (Len = 0)
5. Entry.Is_Initial(Next)
6. (Cum_Dur = 0.0)
7. (|Q| <= Max_Length)
8. (|P| <= Max_Length)
9. ((|P| + |Q|) <= Max_Length)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_3
Show Termination of Recursive Call: Append_Realiz_3.rb(9)

Goal(s):
(|Q"| < P_val)

Given(s):
1. Entry.Is_Initial(Next')
2. (P" = (P o <Next">))
3. (Q = (<Next"> o Q"))
4. (|Q| /= 0)
5. (P_val = |Q|)
6. (Len = 0)
7. Entry.Is_Initial(Next)
8. (Cum_Dur = 0.0)
9. (|Q| <= Max_Length)
10. (|P| <= Max_Length)
11. ((|P| + |Q|) <= Max_Length)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Length > 0)

VC 0_4
Requires Clause of Append_to in Procedure Append_to: Append_Realiz_3.rb(18)

Goal(s):
((|P"| + |Q"|) <= Max_Length)

Given(s):
1. Entry.Is_Initial(Next')
2. (P" = (P o <Next">))
3. (Q = (<Next"> o Q"))
4. (|Q| /= 0)
5. (P_val = |Q|)
6. (Len = 0)
7. Entry.Is_Initial(Next)
8. (Cum_Dur = 0.0)
9. (|Q| <= Max_Length)
10. (|P| <= Max_Length)
11. ((|P| + |Q|) <= Max_Length)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Length > 0)

VC 0_5
Ensures Clause of Append_to: Append_Realiz_3.rb(8)

Goal(s):
(P' = (P o Q))

Given(s):
1. (Q' = Empty_String)
2. (P' = (P'' o Q''))
3. Entry.Is_Initial(Next')
4. (P'' = (P o <Next''>))
5. (Q = (<Next''> o Q''))
6. (|Q| /= 0)
7. (P_val = |Q|)
8. (Len = 0)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_6
Ensures Clause of Append_to (Condition from "clears" parameter mode): Append_Realiz_3.rb(8)

Goal(s):
(Q' = Empty_String)

Given(s):
1. (Q' = Empty_String)
2. (P' = (P'' o Q''))
3. Entry.Is_Initial(Next')
4. (P'' = (P o <Next''>))
5. (Q = (<Next''> o Q''))
6. (|Q| /= 0)
7. (P_val = |Q|)
8. (Len = 0)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)

14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_7
Duration Clause of Append_to: Append_Realiz_3.rb(8)

Goal(s):
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next)) + Dur_Call(2))) + (CEn + Dur_Call(2))) + ((|Q"| * ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))) + Dur_Call(2))) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next') + F_Dur(Integer, |Q|))) <= (|Q| * ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))))

Given(s):
1. (Q' = Empty_String)
2. (P' = (P" o Q"))
3. Entry.Is_Initial(Next')
4. (P" = (P o <Next">))
5. (Q = (<Next"> o Q"))
6. (|Q| /= 0)
7. (P_val = |Q|)
8. (Len = 0)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)
11. (|Q| <= Max_Length)
12. (|P| <= Max_Length)
13. ((|P| + |Q|) <= Max_Length)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 1_1
Ensures Clause of Append_to: Append_Realiz_3.rb(8)

Goal(s):
(P = (P o Q))

Given(s):
1. (|Q| = 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)

6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. ((|P| + |Q|) <= Max_Length)
9. (Last_Char_Num > 0)
10. (0 < max_int)
11. (min_int <= 0)
12. (Max_Length > 0)

VC 1_2
Ensures Clause of Append_to (Condition from "clears" parameter mode): Append_Realiz_3.rb(8)

Goal(s):
(Q = Empty_String)

Given(s):
1. (|Q| = 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. ((|P| + |Q|) <= Max_Length)
9. (Last_Char_Num > 0)
10. (0 < max_int)
11. (min_int <= 0)
12. (Max_Length > 0)

VC 1_3
Duration Clause of Append_to: Append_Realiz_3.rb(8)

Goal(s):
(((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next) + F_Dur(Integer, |Q|))) <= (|Q| * ((C + (2 * I_Dur(Entry))) + (2 * F_IV_Dur(Entry)))))

Given(s):
1. (|Q| = 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (|P| <= Max_Length)
8. ((|P| + |Q|) <= Max_Length)

9. (Last_Char_Num > 0)
10. (0 < max_int)
11. (min_int <= 0)
12. (Max_Length > 0)

C10: ROTATE_REALIZ_2.RB

Cs for Rotate_Realiz_2.rb generated Thu Apr 09 14:50:13 EDT 2015

========== VC(s): ==========

VC 0_1
Base Case of the Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
(Q = (Prt_Btwn((n - n), |Q|, Q) o Prt_Btwn(0, (n - n), Q)))

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)
VC 0_2
Base Case of the Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 0_3
Base Case of the Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
(|Q| = |Q|)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Rotate_Realiz_2.rb(14)

Goal(s):
(((n - n) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 0_5
Requires Clause of Dequeue in Procedure Rotate: Rotate_Realiz_2.rb(16)

Goal(s):
(|Q'''| /= 0)

Given(s):
1. (n'' /= 0)
2. ((Cum_Dur' + I_Dur(Entry)) = ((n - n'') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n'')
4. (|Q| = |Q'''|)
5. Entry.Is_Initial(Next''')
6. (Q = (Prt_Btwn((n - n''), |Q|, Q) o Prt_Btwn(0, (n - n''), Q)))
7. Entry.Is_Initial(Next)
8. (Cum_Dur = 0.0)
9. (n <= max_int)
10. (min_int <= n)
11. (|Q| <= Max_Length)
12. (n <= |Q|)
13. (0 <= n)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 0_6
Requires Clause of Enqueue in Procedure Rotate: Rotate_Realiz_2.rb(17)

Goal(s):
(|Q''| < Max_Length)

Given(s):
1. (Q''' = (<Next''> o Q''))
2. (n'' /= 0)
3. ((Cum_Dur' + I_Dur(Entry)) = ((n - n'') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
4. (P_val' = n'')
5. (|Q| = |Q'''|)
6. Entry.Is_Initial(Next''')
7. (Q = (Prt_Btwn((n - n''), |Q|, Q) o Prt_Btwn(0, (n - n''), Q)))
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (n <= max_int)
11. (min_int <= n)
12. (|Q| <= Max_Length)
13. (n <= |Q|)
14. (0 <= n)

15. (Last_Char_Num > 0)
16. (0 < max_int)
17. (min_int <= 0)
18. (Max_Length > 0)

VC 0_7
Requires Clause of Decrement in Procedure Rotate: Rotate_Realiz_2.rb(18)

Goal(s):
(min_int <= (n" - 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (Q' = (Q" o <Next">))
3. (Q"' = (<Next"> o Q"))
4. (n" /= 0)
5. ((Cum_Dur' + I_Dur(Entry)) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
6. (P_val' = n")
7. (|Q| = |Q"'|)
8. Entry.Is_Initial(Next"')
9. (Q = (Prt_Btwn((n - n"), |Q|, Q) o Prt_Btwn(0, (n - n"), Q)))
10. Entry.Is_Initial(Next)
11. (Cum_Dur = 0.0)
12. (n <= max_int)
13. (min_int <= n)
14. (|Q| <= Max_Length)
15. (n <= |Q|)
16. (0 <= n)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 0_8
Inductive Case of Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
(Q = (Prt_Btwn((n - n'), |Q|, Q) o Prt_Btwn(0, (n - n'), Q)))

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (Q"' = (<Next"> o Q"))
5. (n" /= 0)
6. ((Cum_Dur' + I_Dur(Entry)) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

7. (P_val' = n'')
8. (|Q| = |Q'''|)
9. Entry.Is_Initial(Next''')
10. (Q = (Prt_Btwn((n - n''), |Q|, Q) o Prt_Btwn(0, (n - n''), Q)))
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (n <= max_int)
14. (min_int <= n)
15. (|Q| <= Max_Length)
16. (n <= |Q|)
17. (0 <= n)
18. (Last_Char_Num > 0)
19. (0 < max_int)
20. (min_int <= 0)
21. (Max_Length > 0)

VC 0_9
Inductive Case of Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
Entry.Is_Initial(Next')

Given(s):
1. (n' = (n'' - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q'' o <Next''>))
4. (Q''' = (<Next''> o Q''))
5. (n'' /= 0)
6. ((Cum_Dur' + I_Dur(Entry)) = ((n - n'') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n'')
8. (|Q| = |Q'''|)
9. Entry.Is_Initial(Next''')
10. (Q = (Prt_Btwn((n - n''), |Q|, Q) o Prt_Btwn(0, (n - n''), Q)))
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (n <= max_int)
14. (min_int <= n)
15. (|Q| <= Max_Length)
16. (n <= |Q|)
17. (0 <= n)
18. (Last_Char_Num > 0)
19. (0 < max_int)
20. (min_int <= 0)
21. (Max_Length > 0)

VC 0_10
Inductive Case of Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
(|Q| = |Q'|)

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (Q"' = (<Next"> o Q"))
5. (n" /= 0)
6. ((Cum_Dur' + I_Dur(Entry)) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. (|Q| = |Q"'|)
9. Entry.Is_Initial(Next"')
10. (Q = (Prt_Btwn((n - n"), |Q|, Q) o Prt_Btwn(0, (n - n"), Q)))
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (n <= max_int)
14. (min_int <= n)
15. (|Q| <= Max_Length)
16. (n <= |Q|)
17. (0 <= n)
18. (Last_Char_Num > 0)
19. (0 < max_int)
20. (min_int <= 0)
21. (Max_Length > 0)

VC 0_11
Termination of While Statement: Rotate_Realiz_2.rb(13)

Goal(s):
(n' < P_val')

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (Q"' = (<Next"> o Q"))
5. (n" /= 0)
6. ((Cum_Dur' + I_Dur(Entry)) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. (|Q| = |Q"'|)
9. Entry.Is_Initial(Next"')
10. (Q = (Prt_Btwn((n - n"), |Q|, Q) o Prt_Btwn(0, (n - n"), Q)))

11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (n <= max_int)
14. (min_int <= n)
15. (|Q| <= Max_Length)
16. (n <= |Q|)
17. (0 <= n)
18. (Last_Char_Num > 0)
19. (0 < max_int)
20. (min_int <= 0)
21. (Max_Length > 0)

VC 0_12
Termination of While Statement: Rotate_Realiz_2.rb(13)

Goal(s):
((((((Cum_Dur' + I_Dur(Entry)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next'")) + Dur_Call(2))) + (CEn + Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <= ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (Q'" = (<Next"> o Q"))
5. (n" /= 0)
6. ((Cum_Dur' + I_Dur(Entry)) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. (|Q| = |Q'"|)
9. Entry.Is_Initial(Next'")
10. (Q = (Prt_Btwn((n - n"), |Q|, Q) o Prt_Btwn(0, (n - n"), Q)))
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (n <= max_int)
14. (min_int <= n)
15. (|Q| <= Max_Length)
16. (n <= |Q|)
17. (0 <= n)
18. (Last_Char_Num > 0)
19. (0 < max_int)
20. (min_int <= 0)
21. (Max_Length > 0)

VC 1_1
Base Case of the Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
(Q = (Prt_Btwn((n - n), |Q|, Q) o Prt_Btwn(0, (n - n), Q)))

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 1_2
Base Case of the Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 1_3
Base Case of the Invariant of While Statement: Rotate_Realiz_2.rb(12)

Goal(s):
(|Q| = |Q|)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 1_4
Base Case of Elapsed Time Duration of While Statement: Rotate_Realiz_2.rb(14)

Goal(s):
(((n - n) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. Entry.Is_Initial(Next)
2. (Cum_Dur = 0.0)
3. (n <= max_int)
4. (min_int <= n)
5. (|Q| <= Max_Length)
6. (n <= |Q|)
7. (0 <= n)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 1_5
Ensures Clause of Rotate: Rotate_Realiz_2.rb(8)

Goal(s):
(Q' = (Prt_Btwn(n', |Q|, Q) o Prt_Btwn(0, n', Q)))

Given(s):
1. (n' = 0)
2. (Cum_Dur' = ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n')
4. (|Q| = |Q'|)
5. Entry.Is_Initial(Next')
6. (Q = (Prt_Btwn((n - n'), |Q|, Q) o Prt_Btwn(0, (n - n'), Q)))
7. Entry.Is_Initial(Next)
8. (Cum_Dur = 0.0)
9. (n <= max_int)
10. (min_int <= n)
11. (|Q| <= Max_Length)
12. (n <= |Q|)
13. (0 <= n)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

VC 1_6
Duration Clause of Rotate: Rotate_Realiz_2.rb(8)

Goal(s):
(((((Cum_Dur + I_Dur(Entry)) + (ITP_AreNotEq + Dur_Call(2))) + ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + F_Dur(Entry, Next')) <= (((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) + (n * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (n' = 0)
2. (Cum_Dur' = ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n')
4. (|Q| = |Q'|)
5. Entry.Is_Initial(Next')
6. (Q = (Prt_Btwn((n - n'), |Q|, Q) o Prt_Btwn(0, (n - n'), Q)))
7. Entry.Is_Initial(Next)
8. (Cum_Dur = 0.0)
9. (n <= max_int)
10. (min_int <= n)
11. (|Q| <= Max_Length)
12. (n <= |Q|)

13. (0 <= n)
14. (Last_Char_Num > 0)
15. (0 < max_int)
16. (min_int <= 0)
17. (Max_Length > 0)

APPENDIX D: MISCELLANEOUS EXAMPLES

D1: INVERT A QUEUE

```
Enhancement Inverting_Capability for Preemptable_Queue_Template;

    Operation Invert (updates Q: P_Queue);
        ensures Q = Reverse(#Q);

end Inverting_Capability;
```

Fig. D.1: Functionality specification of Inverting_Capability

ITERATIVE IMPLEMENTATION

```
Profile Invt_Prfile_2 short_for Invert_Profile for Inverting_Capability for
                        Preemptable_Queue_Template with_profile PQ_CT_Profile;

    Defines C1: RPos;
    Defines C2: RPos;
    Operation Invert (updates Q: P_Queue);
        duration C1 + I_Dur(P_Queue) + I_Dur(Entry) +
                                        F_IV_Dur(P_Queue) + F_IV_Dur(Entry) +
                                    |#Q| * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));

end Invt_Prfile_2;
```

Fig. D.2a: Performance profile of Invert operation

```
Realization Invert_Realiz_2 with_profile Invt_Prfile_2 for
                              Inverting_Capability of Preemptable_Queue_Template;

    uses ITP, Integer_Theory;
    Definition C1: RPos = Dur_Assgn + CLe + ITP_AreNotEq + Dur_Swap +
              I_Dur(Integer) + 2*F_IV_Dur(Integer) + Dur_Call(1) + Dur_Call(2);
    Definition C2: RPos = CDq + CIn + ITP_Decr +
                                      ITP_AreNotEq + Dur_Call(1) + 3*Dur_Call(2);

    Procedure Invert (updates Q: P_Queue);
        Var Q_Invert: P_Queue;
        Var Next: Entry;
        Var Len: Integer;

        Len := Length(Q);
        While (Len /= 0)
            maintaining (#Q = Reverse(Q_Invert) o Q) and
                                                  Entry.Is_Initial(Next) and Len = |Q|;
            decreasing |Q|;
            elapsed_time (|#Q| - |Q|) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));
        do
            Dequeue(Next, Q);
            Inject(Next, Q_Invert);
            Decrement(Len);
        end;
        Q :=: Q_Invert;
    end Invert;
end Invert_Realiz_2;
```

Fig. D.2b: Iterative realization of the of Invert operation

VC:

VCs for Invert_Realiz_2.rb generated Wed Apr 29 08:28:02 EDT 2015

================================ VC(s): ================================

VC 0_1
Base Case of the Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
(Q = (Reverse(Q_Invert) o Q))

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 0_2
Base Case of the Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 0_3
Base Case of the Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
(|Q| = |Q|)
Given(s):

1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Invert_Realiz_2.rb(17)

Goal(s):
(((|Q| - |Q|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 0_5
Requires Clause of Dequeue in Procedure Invert: Invert_Realiz_2.rb(19)

Goal(s):
(|Q"| /= 0)

Given(s):
1. (Len" /= 0)
2. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q"|)
4. (Len" = |Q"|)
5. Entry.Is_Initial(Next"')
6. (Q = (Reverse(Q_Invert") o Q"))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Q_Invert = Empty_String)
10. (Cum_Dur = 0.0)

11. (|Q| <= Max_Length)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Length > 0)

VC 0_6
Requires Clause of Inject in Procedure Invert: Invert_Realiz_2.rb(20)

Goal(s):
(|Q_Invert"| < Max_Length)

Given(s):
1. (Q" = (<Next"> o Q'))
2. (Len" /= 0)
3. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
4. (P_val' = |Q"|)
5. (Len" = |Q"|)
6. Entry.Is_Initial(Next"')
7. (Q = (Reverse(Q_Invert") o Q"))
8. (Len = 0)
9. Entry.Is_Initial(Next)
10. (Q_Invert = Empty_String)
11. (Cum_Dur = 0.0)
12. (|Q| <= Max_Length)
13. (Last_Char_Num > 0)
14. (0 < max_int)
15. (min_int <= 0)
16. (Max_Length > 0)

VC 0_7
Requires Clause of Decrement in Procedure Invert: Invert_Realiz_2.rb(21)

Goal(s):
(min_int <= (Len" - 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (Q_Invert' = (<Next"> o Q_Invert"))
3. (Q" = (<Next"> o Q'))
4. (Len" /= 0)
5. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

6. (P_val' = |Q"|)
7. (Len" = |Q"|)
8. Entry.Is_Initial(Next"')
9. (Q = (Reverse(Q_Invert") o Q"))
10. (Len = 0)
11. Entry.Is_Initial(Next)
12. (Q_Invert = Empty_String)
13. (Cum_Dur = 0.0)
14. (|Q| <= Max_Length)
15. (Last_Char_Num > 0)
16. (0 < max_int)
17. (min_int <= 0)
18. (Max_Length > 0)

VC 0_8
Inductive Case of Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
(Q = (Reverse(Q_Invert') o Q'))

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (Q_Invert' = (<Next"> o Q_Invert"))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next"')
10. (Q = (Reverse(Q_Invert") o Q"))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Q_Invert = Empty_String)
14. (Cum_Dur = 0.0)
15. (|Q| <= Max_Length)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 0_9
Inductive Case of Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
Entry.Is_Initial(Next')

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (Q_Invert' = (<Next"> o Q_Invert"))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next"')
10. (Q = (Reverse(Q_Invert") o Q"))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Q_Invert = Empty_String)
14. (Cum_Dur = 0.0)
15. (|Q| <= Max_Length)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 0_10
Inductive Case of Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
(Len' = |Q'|)

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (Q_Invert' = (<Next"> o Q_Invert"))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)

9. Entry.Is_Initial(Next''')
10. (Q = (Reverse(Q_Invert'') o Q''))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Q_Invert = Empty_String)
14. (Cum_Dur = 0.0)
15. (|Q| <= Max_Length)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 0_11
Termination of While Statement: Invert_Realiz_2.rb(16)

Goal(s):
(|Q'| < P_val')

Given(s):
1. (Len' = (Len'' - 1))
2. Entry.Is_Initial(Next')
3. (Q_Invert' = (<Next''> o Q_Invert''))
4. (Q'' = (<Next''> o Q'))
5. (Len'' /= 0)
6. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q''|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q''|)
8. (Len'' = |Q''|)
9. Entry.Is_Initial(Next''')
10. (Q = (Reverse(Q_Invert'') o Q''))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Q_Invert = Empty_String)
14. (Cum_Dur = 0.0)
15. (|Q| <= Max_Length)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 0_12
Termination of While Statement: Invert_Realiz_2.rb(16)

Goal(s):
((((((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next'")) + Dur_Call(2))) + (CIn + Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <= ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (Len' = (Len" - 1))
2. Entry.Is_Initial(Next')
3. (Q_Invert' = (<Next"> o Q_Invert"))
4. (Q" = (<Next"> o Q'))
5. (Len" /= 0)
6. (((((((Cum_Dur' + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) = ((|Q| - |Q"|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = |Q"|)
8. (Len" = |Q"|)
9. Entry.Is_Initial(Next'")
10. (Q = (Reverse(Q_Invert") o Q"))
11. (Len = 0)
12. Entry.Is_Initial(Next)
13. (Q_Invert = Empty_String)
14. (Cum_Dur = 0.0)
15. (|Q| <= Max_Length)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 1_1
Base Case of the Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
(Q = (Reverse(Q_Invert) o Q))

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)

8. (min_int <= 0)
9. (Max_Length > 0)

VC 1_2
Base Case of the Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 1_3
Base Case of the Invariant of While Statement: Invert_Realiz_2.rb(15)

Goal(s):
(|Q| = |Q|)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)
3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 1_4
Base Case of Elapsed Time Duration of While Statement: Invert_Realiz_2.rb(17)

Goal(s):
(((|Q| - |Q|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Len = 0)
2. Entry.Is_Initial(Next)

3. (Q_Invert = Empty_String)
4. (Cum_Dur = 0.0)
5. (|Q| <= Max_Length)
6. (Last_Char_Num > 0)
7. (0 < max_int)
8. (min_int <= 0)
9. (Max_Length > 0)

VC 1_5
Ensures Clause of Invert: Invert_Realiz_2.rb(8)

Goal(s):
(Q_Invert' = Reverse(Q))

Given(s):
1. (Len' = 0)
2. (Cum_Dur' = ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q'|)
4. (Len' = |Q'|)
5. Entry.Is_Initial(Next')
6. (Q = (Reverse(Q_Invert') o Q'))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Q_Invert = Empty_String)
10. (Cum_Dur = 0.0)
11. (|Q| <= Max_Length)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Length > 0)

VC 1_6
Duration Clause of Invert: Invert_Realiz_2.rb(8)

Goal(s):
((((((((((Cum_Dur + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + Dur_Swap) + ((F_Dur(P_Queue, Q') + F_Dur(Entry, Next')) + F_Dur(Integer, Len'))) <= (((((C1 + I_Dur(P_Queue)) + I_Dur(Entry)) + F_IV_Dur(P_Queue)) + F_IV_Dur(Entry)) + (|Q| * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (Len' = 0)
2. (Cum_Dur' = ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = |Q'|)
4. (Len' = |Q'|)

5. Entry.Is_Initial(Next')
6. (Q = (Reverse(Q_Invert') o Q'))
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Q_Invert = Empty_String)
10. (Cum_Dur = 0.0)
11. (|Q| <= Max_Length)
12. (Last_Char_Num > 0)
13. (0 < max_int)
14. (min_int <= 0)
15. (Max_Length > 0)

Table D.1: Proof of duration VC 1_6 for Invert operation

LHS	((((((((((Cum_Dur + I_Dur(P_Queue)) + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + ((|Q| - |Q'|) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + Dur_Swap) + ((F_Dur(P_Queue, Q') + F_Dur(Entry, Next')) + F_Dur(Integer, Len')))	Simplify and Given #10
	0.0 + I_Dur(P_Queue) + I_Dur(Entry) + I_Dur(Integer) + CLe + Dur_Call(1)) + Dur_Assgn + F_Dur(Integer, Len) + ITP_AreNotEq + Dur_Call(2) + (|Q| - |Q'|) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry)) + Dur_Swap + F_Dur(P_Queue, Q') + F_Dur(Entry, Next') + F_Dur(Integer, Len')	Simplify, Given #4 and #1, and rear-range
	Dur_Assgn + CLe + ITP_AreNotEq + Dur_Swap + I_Dur(Integer) + F_Dur(Integer, Len) + F_Dur(Integer, Len') + Dur_Call(1) + Dur_Call(2) + I_Dur(P_Queue) + I_Dur(Entry) + F_Dur(P_Queue, Q') + F_Dur(Entry, Next') + (|Q|) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Given #7 and #1, and simplify
	Dur_Assgn + CLe + ITP_AreNotEq + Dur_Swap + I_Dur(Integer) + 2*F_IV_Dur(Integer) + Dur_Call(1) + Dur_Call(2) + I_Dur(P_Queue) + I_Dur(Entry) + F_Dur(P_Queue, Q') + F_Dur(Entry, Next') + (|Q|) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Definition from Fig 5.12
	C1 + I_Dur(P_Queue) + I_Dur(Entry) + F_Dur(P_Queue, Q') + F_Dur(Entry, Next') + (|Q|) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	Given #4, #1, and #5
	C1 + I_Dur(P_Queue) + I_Dur(Entry) + F_IV_Dur(P_Queue) + F_IV_Dur(Entry) + (|Q|) * (C2 + I_Dur(Entry) + F_IV_Dur(Entry))	<= RHS
	Hence proved.	

RECURSIVE IMPLEMENTATION

```
Profile Invt_Prfile_3 short_for Invert_Profile for
                Inverting_Capability for Preemptable_Queue_Template
                                        with_profile PQ_CT_Profile;
    Defines C: RPos;
    Operation Invert (updates Q: P_Queue);
            duration |#Q|*(C + 2*I_Dur(Entry) +
                                                2*F_IV_Dur(Entry));
end Invt_Prfile_3;
```

Fig. D.3a: Performance profile of Invert operation

```
Realization Invert_Realiz_3 with_profile Invt_Prfile_3 for
                    Inverting_Capability of Preemptable_Queue_Template;

    uses ITP, Integer_Theory;
    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C: RPos = CLe + Dur_Assgn + ITP_AreNotEq + CDq +
                                    CEn + 2*Dur_Call(1) + 3*Dur_Call(2) +
                    I_Dur(Integer) + F_IV_Dur(Integer) + F_Dur_Integer;

    Procedure Invert (updates Q: P_Queue);
        decreasing |Q|;
        Var Next: Entry;
        Var Len: Integer;

        Len := Length(Q);
        if (Len /= 0) then
            Dequeue(Next, Q);
            Invert(Q);
            Enqueue(Next, Q);
        end;
    end Invert;
end Invert_Realiz_3;
```

Fig. D.3b: Realization (Recursive) of the of Invert operation

VC

VCs for Invert_Realiz_3.rb generated Wed Apr 29 09:48:26 EDT 2015

================================ VC(s): ================================

VC 0_1
Requires Clause of Dequeue in Procedure Invert: Invert_Realiz_3.rb(16)

Goal(s):
(|Q| /= 0)

Given(s):
1. (|Q| /= 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 0_2
Show Termination of Recursive Call: Invert_Realiz_3.rb(9)

Goal(s):
(|Q'"| < P_val)

Given(s):
1. (Q = (<Next"> o Q'"))
2. (|Q| /= 0)
3. (P_val = |Q|)
4. (Len = 0)
5. Entry.Is_Initial(Next)
6. (Cum_Dur = 0.0)
7. (|Q| <= Max_Length)
8. (Last_Char_Num > 0)
9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)

VC 0_3
Requires Clause of Enqueue in Procedure Invert: Invert_Realiz_3.rb(18)

Goal(s):
(|Q''| < Max_Length)

Given(s):
1. (Q'' = Reverse(Q'''))
2. (Q = (<Next''> o Q'''))
3. (|Q| /= 0)
4. (P_val = |Q|)
5. (Len = 0)
6. Entry.Is_Initial(Next)
7. (Cum_Dur = 0.0)
8. (|Q| <= Max_Length)
9. (Last_Char_Num > 0)
10. (0 < max_int)
11. (min_int <= 0)
12. (Max_Length > 0)

VC 0_4
Ensures Clause of Invert: Invert_Realiz_3.rb(8)

Goal(s):
(Q' = Reverse(Q))

Given(s):
1. Entry.Is_Initial(Next')
2. (Q' = (Q'' o <Next''>))
3. (Q'' = Reverse(Q'''))
4. (Q = (<Next''> o Q'''))
5. (|Q| /= 0)
6. (P_val = |Q|)
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (|Q| <= Max_Length)
11. (Last_Char_Num > 0)
12. (0 < max_int)
13. (min_int <= 0)
14. (Max_Length > 0)

VC 0_5
Duration Clause of Invert: Invert_Realiz_3.rb(8)

Goal(s):
((((((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next)) + Dur_Call(2))) + ((|Q'''| * ((C + (2 * I_Dur(Entry))) + F_IV_Dur(Entry))) + Dur_Call(1))) + (CEn + Dur_Call(2))) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next') + F_Dur(Integer, |Q|))) <= (|Q| * ((C + (2 * I_Dur(Entry))) + F_IV_Dur(Entry))))

Given(s):
1. Entry.Is_Initial(Next')
2. (Q' = (Q'' o <Next''>))
3. (Q'' = Reverse(Q'''))
4. (Q = (<Next''> o Q'''))
5. (|Q| /= 0)
6. (P_val = |Q|)
7. (Len = 0)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (|Q| <= Max_Length)
11. (Last_Char_Num > 0)
12. (0 < max_int)
13. (min_int <= 0)
14. (Max_Length > 0)

VC 1_1
Ensures Clause of Invert: Invert_Realiz_3.rb(8)

Goal(s):
(Q = Reverse(Q))

Given(s):
1. (|Q| = 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

VC 1_2
Duration Clause of Invert: Invert_Realiz_3.rb(8)

Goal(s):
(((((((Cum_Dur + I_Dur(Entry)) + I_Dur(Integer)) + (CLe + Dur_Call(1))) + Dur_Assgn) + F_Dur(Integer, Len)) + (ITP_AreNotEq + Dur_Call(2))) + (F_Dur(Entry, Next) + F_Dur(Integer, |Q|))) <= (|Q| * ((C + (2 * I_Dur(Entry))) + F_IV_Dur(Entry))))

Given(s):
1. (|Q| = 0)
2. (P_val = |Q|)
3. (Len = 0)
4. Entry.Is_Initial(Next)
5. (Cum_Dur = 0.0)
6. (|Q| <= Max_Length)
7. (Last_Char_Num > 0)
8. (0 < max_int)
9. (min_int <= 0)
10. (Max_Length > 0)

D2: SPLIT A QUEUE

```
Enhancement Splitting_Capability for Preemptable_Queue_Template;

    Operation Split (updates P: P_Queue;
                                    replaces Q: P_Queue; evaluates n: Integer);
        requires 0 <= n <= |P|;
        ensures Q = Prt_Btwn(0, n, #P) and P = Prt_Btwn(n, |#P|, #P);

end Splitting_Capability;
```

Fig. D.4a: Functionality specification of Splitting_Capability

```
Profile Splt_Prfile_2 short_for Split_Profile for Splitting_Capability
                                for Preemptable_Queue_Template with_profile
                                                                PQ_CT_Profile;

    Defines C1: RPos;
    Defines C2: RPos;
    Operation Split (updates P: P_Queue; replaces Q: P_Queue;
                                                    evaluates n: Integer);
        duration C1 + I_Dur(Entry) + F_IV_Dur(Entry) +
                Cnts_Dur(#Q) + (Max_Length - |#Q|)*F_IV_Dur(Entry) +
                                #n * (C2 + I_Dur(Entry) + F_IV_Dur(Entry));

ends Splt_Prfile_2;
```

Fig. D.4b: Performance profile of Split operation

```
Realization Split_Realiz_2 with_profile Splt_Prfile_2 for
                    Splitting_Capability of Preemptable_Queue_Template;

    uses ITP, Integer_Theory;

    Definition F_Dur_Integer: RPos = F_Dur(Integer, max_int);
    Definition C1: RPos = CCl1 + ITP_AreNotEq + Dur_Call(1) +
                                    Dur_Call(2) + F_Dur_Integer*CCl2;
    Definition C2: RPos = CDq + CEn + ITP_Decr + ITP_AreNotEq +
                                    Dur_Call(1) + 3*Dur_Call(2);

    Procedure Split (updates P: P_Queue; replaces Q: P_Queue;
                                                evaluates n: Integer);
        Var Next: Entry;

        Clear(Q);
        While (n /= 0)
            maintaining #P = Q o P and |Q| = #n - n and
                                                Entry.Is_Initial(Next);
            decreasing n;
                        elapsed_time (#n - n) * (C2 + I_Dur(Entry) +
                                                F_IV_Dur(Entry));
        do
            Dequeue(Next, P);
            Enqueue(Next, Q);
            Decrement(n);
        end;
    end Split;
end Split_Realiz_2;
```

Fig. D.4c: Realization of the of Split operation

VC:

VCs for Split_Realiz_2.rb generated Thu Apr 09 19:05:30 EDT 2015

========== VC(s): ==========

VC 0_1
Base Case of the Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
(P = (Q''' o P))

Given(s):
1. (Q''' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_2
Base Case of the Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
(|Q'''| = (n - n))

Given(s):
1. (Q''' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)

13. (Max_Length > 0)

VC 0_3
Base Case of the Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Q''' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_4
Base Case of Elapsed Time Duration of While Statement: Split_Realiz_2.rb(16)

Goal(s):
(((n - n) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Q''' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 0_5
Requires Clause of Dequeue in Procedure Split: Split_Realiz_2.rb(18)

Goal(s):
(|P"| /= 0)

Given(s):
1. (n" /= 0)
2. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n")
4. Entry.Is_Initial(Next"')
5. (|Q"| = (n - n"))
6. (P = (Q" o P"))
7. (Q"' = Empty_String)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (n <= max_int)
11. (min_int <= n)
12. (Q = Empty_String)
13. (|P| <= Max_Length)
14. (n <= |P|)
15. (0 <= n)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 0_6
Requires Clause of Enqueue in Procedure Split: Split_Realiz_2.rb(19)

Goal(s):
(|Q"| < Max_Length)

Given(s):
1. (P" = (<Next"> o P'))
2. (n" /= 0)
3. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
4. (P_val' = n")
5. Entry.Is_Initial(Next"')
6. (|Q"| = (n - n"))
7. (P = (Q" o P"))
8. (Q"' = Empty_String)
9. Entry.Is_Initial(Next)
10. (Cum_Dur = 0.0)

11. (n <= max_int)
12. (min_int <= n)
13. (Q = Empty_String)
14. (|P| <= Max_Length)
15. (n <= |P|)
16. (0 <= n)
17. (Last_Char_Num > 0)
18. (0 < max_int)
19. (min_int <= 0)
20. (Max_Length > 0)

VC 0_7
Requires Clause of Decrement in Procedure Split: Split_Realiz_2.rb(20)

Goal(s):
(min_int <= (n" - 1))

Given(s):
1. Entry.Is_Initial(Next')
2. (Q' = (Q" o <Next">))
3. (P" = (<Next"> o P'))
4. (n" /= 0)
5. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
6. (P_val' = n")
7. Entry.Is_Initial(Next"')
8. (|Q"| = (n - n"))
9. (P = (Q" o P"))
10. (Q"' = Empty_String)
11. Entry.Is_Initial(Next)
12. (Cum_Dur = 0.0)
13. (n <= max_int)
14. (min_int <= n)
15. (Q = Empty_String)
16. (|P| <= Max_Length)
17. (n <= |P|)
18. (0 <= n)
19. (Last_Char_Num > 0)
20. (0 < max_int)
21. (min_int <= 0)
22. (Max_Length > 0)

VC 0_8
Inductive Case of Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
(P = (Q' o P'))

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (P" = (<Next"> o P'))
5. (n" /= 0)
6. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. Entry.Is_Initial(Next"')
9. (|Q"| = (n - n"))
10. (P = (Q" o P"))
11. (Q"' = Empty_String)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (n <= max_int)
15. (min_int <= n)
16. (Q = Empty_String)
17. (|P| <= Max_Length)
18. (n <= |P|)
19. (0 <= n)
20. (Last_Char_Num > 0)
21. (0 < max_int)
22. (min_int <= 0)
23. (Max_Length > 0)

VC 0_9
Inductive Case of Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
(|Q'| = (n - n'))

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (P" = (<Next"> o P'))
5. (n" /= 0)
6. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

7. (P_val' = n")
8. Entry.Is_Initial(Next"')
9. (|Q"| = (n - n"))
10. (P = (Q" o P"))
11. (Q"' = Empty_String)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (n <= max_int)
15. (min_int <= n)
16. (Q = Empty_String)
17. (|P| <= Max_Length)
18. (n <= |P|)
19. (0 <= n)
20. (Last_Char_Num > 0)
21. (0 < max_int)
22. (min_int <= 0)
23. (Max_Length > 0)

VC 0_10
Inductive Case of Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
Entry.Is_Initial(Next')

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (P" = (<Next"> o P'))
5. (n" /= 0)
6. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. Entry.Is_Initial(Next"')
9. (|Q"| = (n - n"))
10. (P = (Q" o P"))
11. (Q"' = Empty_String)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (n <= max_int)
15. (min_int <= n)
16. (Q = Empty_String)
17. (|P| <= Max_Length)
18. (n <= |P|)
19. (0 <= n)
20. (Last_Char_Num > 0)

21. (0 < max_int)
22. (min_int <= 0)
23. (Max_Length > 0)

VC 0_11
Termination of While Statement: Split_Realiz_2.rb(15)

Goal(s):
(n' < P_val')

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (P" = (<Next"> o P'))
5. (n" /= 0)
6. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. Entry.Is_Initial(Next"')
9. (|Q"| = (n - n"))
10. (P = (Q" o P"))
11. (Q"' = Empty_String)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (n <= max_int)
15. (min_int <= n)
16. (Q = Empty_String)
17. (|P| <= Max_Length)
18. (n <= |P|)
19. (0 <= n)
20. (Last_Char_Num > 0)
21. (0 < max_int)
22. (min_int <= 0)
23. (Max_Length > 0)

VC 0_12
Termination of While Statement: Split_Realiz_2.rb(15)

Goal(s):
(((((((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) + (((CDq + I_Dur(Entry)) + F_Dur(Entry, Next"')) + Dur_Call(2))) + (CEn + Dur_Call(2))) + (ITP_Decr + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) <= ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))

Given(s):
1. (n' = (n" - 1))
2. Entry.Is_Initial(Next')
3. (Q' = (Q" o <Next">))
4. (P" = (<Next"> o P'))
5. (n" /= 0)
6. (((Cum_Dur' + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) = ((n - n") * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
7. (P_val' = n")
8. Entry.Is_Initial(Next"')
9. (|Q"| = (n - n"))
10. (P = (Q" o P"))
11. (Q"' = Empty_String)
12. Entry.Is_Initial(Next)
13. (Cum_Dur = 0.0)
14. (n <= max_int)
15. (min_int <= n)
16. (Q = Empty_String)
17. (|P| <= Max_Length)
18. (n <= |P|)
19. (0 <= n)
20. (Last_Char_Num > 0)
21. (0 < max_int)
22. (min_int <= 0)
23. (Max_Length > 0)

VC 1_1
Base Case of the Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
(P = (Q" o P))

Given(s):
1. (Q" = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 1_2
Base Case of the Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
(|Q''| = (n - n))

Given(s):
1. (Q'' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 1_3
Base Case of the Invariant of While Statement: Split_Realiz_2.rb(14)

Goal(s):
Entry.Is_Initial(Next)

Given(s):
1. (Q'' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 1_4
Base Case of Elapsed Time Duration of While Statement: Split_Realiz_2.rb(16)

Goal(s):
(((n - n) * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))) = 0.0)

Given(s):
1. (Q'' = Empty_String)
2. Entry.Is_Initial(Next)
3. (Cum_Dur = 0.0)
4. (n <= max_int)
5. (min_int <= n)
6. (Q = Empty_String)
7. (|P| <= Max_Length)
8. (n <= |P|)
9. (0 <= n)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)
13. (Max_Length > 0)

VC 1_5
Ensures Clause of Split: Split_Realiz_2.rb(9)

Goal(s):
(Q' = Prt_Btwn(0, n', P))

Given(s):
1. (n' = 0)
2. (Cum_Dur' = ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n')
4. Entry.Is_Initial(Next')
5. (|Q'| = (n - n'))
6. (P = (Q' o P'))
7. (Q'' = Empty_String)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (n <= max_int)
11. (min_int <= n)
12. (Q = Empty_String)
13. (|P| <= Max_Length)
14. (n <= |P|)
15. (0 <= n)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)

19. (Max_Length > 0)

VC 1_6
Ensures Clause of Split: Split_Realiz_2.rb(9)

Goal(s):
(P' = Prt_Btwn(n', |P|, P))

Given(s):
1. (n' = 0)
2. (Cum_Dur' = ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n')
4. Entry.Is_Initial(Next')
5. (|Q'| = (n - n'))
6. (P = (Q' o P'))
7. (Q'' = Empty_String)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (n <= max_int)
11. (min_int <= n)
12. (Q = Empty_String)
13. (|P| <= Max_Length)
14. (n <= |P|)
15. (0 <= n)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VC 1_7
Duration Clause of Split: Split_Realiz_2.rb(9)

Goal(s):
((((((Cum_Dur + I_Dur(Entry)) + (((CCl1 + Cnts_Dur(Q)) + ((CCl2 + F_IV_Dur(Entry)) * (Max_Length - |Q|))) + Dur_Call(1))) + (ITP_AreNotEq + Dur_Call(2))) + ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))) + F_Dur(Entry, Next')) <= (((((C1 + I_Dur(Entry)) + F_IV_Dur(Entry)) + Cnts_Dur(Q)) + ((Max_Length - |Q|) * F_IV_Dur(Entry))) + (n * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry)))))

Given(s):
1. (n' = 0)
2. (Cum_Dur' = ((n - n') * ((C2 + I_Dur(Entry)) + F_IV_Dur(Entry))))
3. (P_val' = n')
4. Entry.Is_Initial(Next')
5. (|Q'| = (n - n'))
6. (P = (Q' o P'))

7. (Q'' = Empty_String)
8. Entry.Is_Initial(Next)
9. (Cum_Dur = 0.0)
10. (n <= max_int)
11. (min_int <= n)
12. (Q = Empty_String)
13. (|P| <= Max_Length)
14. (n <= |P|)
15. (0 <= n)
16. (Last_Char_Num > 0)
17. (0 < max_int)
18. (min_int <= 0)
19. (Max_Length > 0)

VITA

NIGHAT YASMIN

EDUCATION

M.S., Computer and Information Science, University of Mississippi, 1999
M.Sc, Civil Engineering, University of Alberta, Canada, 1993
B.Sc, Civil Engineering, Peshawar, Pakistan, 1986
B.Sc, Physics and Mathematics, Peshawar, Pakistan, 1981

ACADEMIC EXPERIENCE

Clemson University - Senior Lecturer (2012-present) - Full-time
Clemson University - Lecturer (2006 - 2012) - Full-time
Clemson University - Lecturer (2004 - 2006) - Part-time

NON-ACADEMIC EXPERIENCE

University of Mississippi (NCCHE) - (1999 - 2003) – Part-time
Research Associate
University of Mississippi (NCCHE) - (1997 - 1999) – Part-time
Research Assistant