2001

The Class Validation System

Marybeth Gurski
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
The Class Validation System

by

Marybeth Gurski

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science
Major Professor: Albert L. Baker

Iowa State University
Ames, Iowa
2001
Copyright © Marybeth Gurski, 2001. All rights reserved.
Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of
Marybeth Gurski
has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Committee Member
Signature was redacted for privacy.

Committee Member
Signature was redacted for privacy.

Committee Member
Signature was redacted for privacy.

Committee Member
Signature was redacted for privacy.

Major Professor
Signature was redacted for privacy.

For the Major Program
Signature was redacted for privacy.

For the Graduate College
TABLE OF CONTENTS

ABSTRACT .................................................................................................................. ix

CHAPTER 1 SOFTWARE DESIGN AND TESTING ............................................... 1
   Formal Specifications ............................................................................................ 3
   Software Testing .................................................................................................. 4
      Functional Testing ............................................................................................ 7
   Goals of This Research ....................................................................................... 9

CHAPTER 2 SPECS-C++ ...................................................................................... 11
   Executability of SPECS-C++ ........................................................................... 16
   Implementation of SPECS-C++ Prototyping Compiler ...................................... 18

CHAPTER 3 THE SPECS-C++ COMPILER ......................................................... 26
   Enhancements to the SPECS-C++ Compiler .................................................... 27
   Limitations of Current Compiler ...................................................................... 29
   Representation Mapping Compiler ................................................................... 29
      Support for Representation Mapping in the SPECS-C++ Compiler ............... 31
   Example ............................................................................................................. 32

CHAPTER 4 THE CLASS VALIDATION SYSTEM .................................................. 36
   Abstract Test System ......................................................................................... 38
      Design of the Abstract Test Tool .................................................................... 41
   The Class Validation System ............................................................................ 43
      Class Development with the Class Validation System ................................. 44
   Oracle - Equivalence Testing ............................................................................ 48
      Implementation Details .................................................................................. 51

CHAPTER 5 ABSTRACT AND CONCRETE VALUES ........................................... 57
   Model-Based Value Generation ......................................................................... 57
Method-Based Value Generation ........................................ 61
Values in the Class Validation System ................................. 63

CHAPTER 6 RELATED WORK ............................................ 67
Automated Testing of Classes ......................................... 67
Integrating Black-Box and White-Box Testing ....................... 68
Axiomatic Specifications ................................................ 69
Using Axiomatic Specifications ........................................ 71
Automatically Testing Using Axiomatic Specifications ............ 72

CHAPTER 7 CONCLUSION .................................................. 74
Additional Enhancements ................................................ 75

CHAPTER A SPECIFICATIONS FOR SPECS-C++ CLASSES ........ 76
Specification for class T ............................................... 76
Specification for class Sequence ..................................... 82
Specification for class Set ............................................. 86
Specification for class Tuple .......................................... 90
Specification for class Enumeration ................................ 94
Specification for class Alternative ................................... 100

APPENDIX B CLASS PRIORITY QUEUE ............................... 104
pqqueue.pre .......................................................... 104
pqqueue.h ............................................................. 104
pqqueue.pri .......................................................... 107
pqqueue.C ............................................................. 108

APPENDIX C SPECS-C++ VALIDATION TOOLS .................... 112
Engine Interfaces ...................................................... 112
Abstract Engine Interface ........................................... 112
Implementation Engine Interface .................................. 113
Return Value Containers ............................................. 114
ReturnVals.h .......................................................... 114
ReturnValsConcrete.h .............................................. 116
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5.1</td>
<td>Maximum values possible for set of tuple(char, int, int)</td>
<td>60</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Number of values generated for int</td>
<td>60</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Number of values generated for set of int</td>
<td>61</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

| Figure 1.1 | Process for Component Validation | 6 |
| Figure 2.1 | Class PriorityQueue, formal model | 13 |
| Figure 2.2 | Class PriorityQueue, abstract functions | 14 |
| Figure 2.3 | Class PriorityQueue, constraints | 14 |
| Figure 2.4 | Class PriorityQueue, constructors | 15 |
| Figure 2.5 | Class PriorityQueue, destructor | 15 |
| Figure 2.6 | Class PriorityQueue, mutator functions | 16 |
| Figure 2.7 | Class PriorityQueue, observer functions | 16 |
| Figure 2.8 | Hierarchy of classes used to implement SPECS-C++ types | 19 |
| Figure 2.9 | Model for class T | 20 |
| Figure 2.10 | Constructors for class T | 21 |
| Figure 2.11 | Observers for class T | 22 |
| Figure 2.12 | Model for class Sequence | 23 |
| Figure 2.13 | Operations of class Sequence | 24 |
| Figure 2.14 | Sequence primitive operations for class Sequence | 25 |
| Figure 3.1 | Specification for function AddEntry | 27 |
| Figure 3.2 | Parse tree for AddEntry postconditions | 28 |
| Figure 3.3 | Implementation type definitions for PriorityQueue | 30 |
| Figure 3.4 | Implementation data members for PriorityQueue | 30 |
| Figure 3.5 | Representation mapping function PriorityQueue | 31 |
| Figure 3.6 | Code generated for domains section of the model by SPECS-C++ Compiler | 33 |
| Figure 3.7 | Generated code for abstract function LatestTime | 34 |
| Figure 3.8 | Code generated for member function AddEntry | 35 |
| Figure 3.9 | Code generated for representation mapping function | 35 |
| Figure 4.1  | One test case in a Class Validation Tool | 37 |
| Figure 4.2  | Main Menu for the Abstract Test Tool for class PriorityQueue | 38 |
| Figure 4.3  | Interactive testing of class member function AddEntry, text based tool | 39 |
| Figure 4.4  | Excerpts from log file for automatic testing of class PriorityQueue | 40 |
| Figure 4.5  | Interface for Abstract Test Tool engine | 42 |
| Figure 4.6  | Implementation of an Abstract Test Tool | 43 |
| Figure 4.7  | Design of a Class Validation Tool | 45 |
| Figure 4.8  | Interface for the Class Validation Tool engine | 46 |
| Figure 4.9  | Excerpts from the log file for the Class Validation Tool | 49 |
| Figure 4.10 | Excerpts from summary of automated testing for Class Validation Tool | 50 |
| Figure 4.11 | Abstract function Equivalent | 52 |
| Figure 4.12 | Alias problem in implementation - pre-state | 53 |
| Figure 4.13 | Alias problem in implementation - post-state | 54 |
| Figure 4.14 | Example tree of values | 55 |
| Figure 5.1  | Parse tree for the model of PriorityQueue | 58 |
| Figure 5.2  | Tree of values formed from values generated for sequence of integer | 59 |
| Figure 5.3  | Model for CompoundValue | 65 |
| Figure 5.4  | Model for CompoundValueSet | 66 |
| Figure 6.1  | Example axiomatic specification | 70 |
ABSTRACT

Formal model-based specifications provide precise descriptions of the behavior of software components. These formal specifications are written using pre- and post-condition assertions. They can serve as a basis for formally verifying the correctness of an implementation. But a formal specification is really only useful when it captures the desired functionality. How can the specifier be confident that the specification is correct?

The Abstract Test Tool supports the direct execution of C++ class specifications through the incremental development and automated generation of abstract test cases and the display of abstract results - both in terms of the abstract model used in the specification. The Class Validation System builds upon the Abstract Test Tool to support the automated and extensive testing of C++ class implementations. The Class Validation System provides a potentially one-to-many mapping from abstract test cases to implementation test cases. The results of executing the implementation are automatically compared to the results of executing the specification by mapping the implementation results to the abstract model. Thus, the Class Validation System supports fully automated implementation testing.
CHAPTER 1 SOFTWARE DESIGN AND TESTING

Rapid development of reliable software systems requires libraries of reusable software components. The most useful components are well-specified and well-tested. The Class Validation System, described in this dissertation, integrates formal software specifications and testing to help produce these software components. This system provides an environment to interact with a specification (the Abstract Test System), automate key aspects of testing of the specification, automate generation of class instances to be used in testing, and automate validation of a corresponding implementation.

Computer scientists have long recognized the importance of formal specifications to describe the correct behavior of software, but the abstract concepts and the tedious proofs associated with formal specifications have prevented extensive use outside of an academic setting. Recently, many publications have begun to address the application of formal specifications to practical applications[12, 10, 25, 28, 30].

The contribution of the research described in this dissertation is the synthesis of formal specifications, software prototyping from the formal specifications, and testing with and against formal specifications. This combination of formality and practical application strives to make formal methods accessible and useful outside of the academic world.

Software specifications may do more than describe the desired behavior of the implementation. A specification may describe constraints upon the performance of the system, interaction with data created by other software, or may describe the necessary robustness of the software. These, and other, aspects are critical to system design and must be taken into consideration when developing a software system. However, the research described here focuses upon the behavior of software components. Existing tools have been designed, and methodologies created to test the other software system issues mentioned here.

Planning a test strategy in conjunction with specification and design has been useful in producing software that exhibits the desired behavior [27, 29]. A correct specification of the software is needed to construct software that does the right job. It is challenging to write a correct specification and develop software that implements it. Current testing techniques neither test using the formal specification, nor automatically validate with respect to the formal specification. The Class Validation System addresses
this issue through the use of formal software specifications integrated with a variation on black box software testing.

The result of this research is a software development process which synthesizes formal specifications and testing into the traditional software methods. The Class Validation System, a tool developed as part of this research, demonstrates the various aspects of the integrated development process.

The Class Validation System creates a testing tool, a Class Validation Tool for a particular class, which allows software developers to combine testing techniques in a manner which is most appropriate for the particular software component being developed. This tool provides interactive and automatic testing of a formal specification of a C++ class in SPECS-C++, a formal specification language developed at Iowa State University[4, 3, 19]. Details about SPECS-C++ will be discussed in Chapter 2.

Automatic testing is a simple way to apply a large number of test cases to a class, and interactive testing allows the software developer to use knowledge about the domain of the software component to apply specific tests to exercise the specification in ways that automatic testing may miss. The software developer can concentrate on these few test cases, while the tool generates test cases which exercise the specification based on the structure of the model\textsuperscript{1} used in the specification.

Once the software developer has tested the specification to his satisfaction, the formal specification becomes the oracle for testing the implementation of this class. The oracle is the source of the correct abstract result. The implementation data structures are typically different from the model used in the class specification. This makes it difficult to automate validation of instances of the implementation data structure against the specified result. The Class Validation System is able to automatically create an oracle to test the implementation of a class by creating instances of this class, applying class functions to the instances and evaluating the results according to the formal specification using a representation mapping (repmap) provided by the implementation developer in terms of the interface defined in the SPECS-C++ specification. The repmap function defines the mapping from implementation values to the domain of values defined in the specification. Details of the repmap function are discussed in Chapters 2, 3 and 4.

The Class Validation Tool does many jobs. It runs tests against the implementation and it compares the results of the tests against the expected results defined in the specification, determining if the test passed. The software developer does not need to create any additional test programs to test the functionality of this class in isolation from the rest of the software system.

The Abstract Test System [17] and Class Validation System form a testing solution developed to test

\textsuperscript{1}The model defines the set of values which form the domain of the class.
formal specifications of software components, and their corresponding implementations. The Abstract Test System produces an abstract testing tool for a formal class specification written in SPECS-C++. An Abstract Test Tool for a class provides an environment in which the specifier can test a formal class specification without writing a prototype from the specification. The Class Validation System extends the Abstract Test System to create tools which test a class implementation against its specification. The testing environment provided by the Class Validation System presents test cases and results in terms of the model given in the formal specification; no knowledge of the implementation is necessary.

**Formal Specifications**

Formal specifications describe, without ambiguity, software functional behavior. Many formal specification languages, such as Larch [18], VDM [6], Z [34] and SPECS-C++ [19, 15] are based on first order predicate logic. A model-based specification uses discrete mathematical entities, like sets and sequences, to model the data. Operations on the values of the domain are specified with assertions over the model. Model-based specification languages lend themselves well to object oriented languages. Formal model-based specifications give precise descriptions of desired software behaviors, and provide an abstract model of the object to be implemented. Both these factors are useful in testing the software. The precise description of the behavior defines the expected output for a given input. The abstract model is useful in designing test cases from the specification. The model defines the domain of values which are instances of the class. These values form the set from which all test cases can be selected.

Use of formal specifications generally increases the quality of the software [28, 24]. When formal specifications are used, design errors are more likely to be revealed in the specification process. The writing of an informal specification may not catch the design errors that writing a formal specification would. However, the use of formal specifications does have some costs. For example, training software developers and engineers to use a formal specification language in their software design, and tools supporting the formal specification language takes time and financial resources. The combination of initial costs, the association of formal specifications with formal proofs, plus the limited amount of good tool support for formal specification languages keeps formal specifications from widespread use in industrial development of software.

Increasing the use of formal specification in the software development industry requires development tools and techniques which support traditional testing techniques. They must fit into existing development processes, provide a link between the specification and the implementation, and allow the users of the tool to specify those parts of the software which they feel are most important [24, 12, 25].
The Class Validation System satisfies these requirements by allowing the software developer to test a formal specification to see if it meets its informal requirements, to generate test cases (both for the specification, and for the implementation), and to test an implementation against its formal specification. The Class Validation System fits into the software development process described in the waterfall model, a common software paradigm. As will be described in the following section, the Class Validation System does not require a new development paradigm, but allows additional steps to be inserted into an existing paradigm.

Software Testing

People test computer programs to answer the question “Is this program correct?” This question encompasses many issues, including:

- “Does the program perform the desired function?”
- “Does the program produce the correct output for a given input?”
- “Does the program handle error conditions correctly?”
- “Is the program free of errors?”

Testing alone cannot ensure that a program has no flaws in it; testing can only reveal some of the errors which might exist in a program [5]. Extensive testing on a piece of software may only give the tester a high amount of confidence that there are very few errors in a program and that the program performs its intended function correctly. Combining a variety of techniques can increase the confidence in the correctness of the software.

Software testing techniques can be broken down into two main categories: functional testing and structural testing. The goal of functional testing, also known as behavioral or “black-box testing,” is to demonstrate that the software performs the desired function, by executing a program with typical inputs and examining the outputs. The goal of structural testing, also known as “white-box testing,” is to find faults in the software [23] by forcing program execution to meet particular coverage criteria, e.g. “all statements.”

Many software development paradigms exist, including the waterfall approach, exploratory programming, formal transformation and system assembly from reusable components [33]. While each is distinct and beneficial in its own way, the most common all software development processes follow the waterfall method to some degree. This method consists of the following common stages [33]:
1. Gather requirements

2. Design and specify data structures

3. Implement and unit test

4. Perform integration testing

5. Perform system testing

6. Maintain the system

Because testing typically occurs late in the software development cycle (after the software has been fully implemented) errors that are introduced at the requirements, or design and specification stages might not be caught until well into the testing process. At these latter stages correcting errors often has heavy costs [1, 7].

Use of the Class Validation System supports the introduction new steps in the development process, providing the opportunity to locate errors earlier. One such development strategy may consist of the following steps:

1. Gather requirements

2. Design and specify data structures

3. Test the specification and design with the Abstract Test Tool

4. Implement and perform unit testing with the Class Validation Tool

5. Perform integration testing

6. Perform system testing

7. Maintain the system

This process provides the opportunity to find errors which are introduced at early stages before implementation. These errors can be corrected, and carrying the errors through to the implementation can be avoided. Figure 1.1 demonstrates how the Abstract Test Tool and Class Validation Tool aid in relating formal specifications and implementation of software components with the requirements. The Abstract Test Tool allows testing through prototypes generated automatically from a specification. It also provides a means to locate errors in a specification. The Class Validation System links the
Figure 1.1 Process for Component Validation
implementation to the specification, and indirectly to the requirements. This provides automatic testing of an implementation by generating test cases from abstract values based upon the abstract model.

The marriage of testing and formal specifications is new, but is gaining popularity as researchers attempt to make formal specifications more "user-friendly." This research uses testing, a technique familiar to software developers, to integrate the rigor of formal specifications into the software development process. Recent work demonstrates this trend [17, 30, 10, 2].

System testing tends to be functional, since it is based on the user's view of the system. Testing the various classes that make up the system can be difficult because the class is not a program on its own. Three techniques of class testing that have been used are compilation (identifying syntactic and type errors), code walk-throughs, and script based testing[27]. The Class Validation System creates a program to exercise and test the class outside of the scope of the entire system. Tool support for automated class testing has particular importance when the model specifications may change. Any tests which have been written by hand would have to be maintained in parallel with the model. Automated testing removes the need to maintain test cases. The tests are performed for each evolution of the specification, taking into account the latest changes to the specification [27, 32].

Many testing techniques and tools have been used to ensure software correctness. Some methods, including those supported with the Class Validation System, test using only knowledge of the specification. Other methods use knowledge of the implementation to exercise sections, or even individual lines, of code ensuring that no part of the program goes untried. Structural testing describes this type of implementation testing. Ensuring that each code path through the code has been executed is the basic goal of this testing. The Class Validation System does not support testing of the implementation in this manner. It focuses is on how the implementation supports the specification.

The next sections identify and describe several types of specification-based testing.

Functional Testing

Functional testing requires that the tester understand the desired functionality of the software before testing. Generally, the specification supplies this information. The specification, either formal or informal, indicates how the software should behave, and becomes the basis for testing. For structural testing the tester should have intimate knowledge of the implementation of the program, and know how to arrange program input to force program execution through various segments of the code. This research focuses on formal specifications and functional testing.

According to Ostrand and Belcer [29], the goal of functional testing, also known as "black box
testing," is to find discrepancies between the specified and the actual behavior of a program. Functional testing can be performed in several often used ways, such as Boundary Value Testing, Equivalence Class Testing, and Decision Based Testing [23]. Black box testing is not the only technique of testing, and it is also not sufficient to be used alone. Functional testing presents an important first step in the testing process, but the most effective testing combines many techniques [5].

Boundary Value Testing is probably the best known functional testing technique. This type of testing involves selecting test case values which lie on or near the edges of the input domain. Many software faults occur with boundary value inputs. Programs where the input consists of independent variables with bounded physical quantities require boundary value testing [23]. Typical boundary value testing selects five values for each input. For example, if the input domain were a set of integers, the five values would be \( \text{min} \), \( \text{min} + 1 \), a median value, \( \text{max} - 1 \), and \( \text{max} \). One input variable is exercised at a time, while a median value is used for the others [26].

At least two subclasses of Boundary Value Testing exist. \textit{Worst-case testing} involves creating a set of inputs from the cartesian product of the boundary values for each input variable. This creates input combinations where multiple inputs have values on the edge of the input domain, while in traditional boundary value testing only one input at a time is on the edge of the input domain. \textit{Special-case testing} uses the intuition and the domain knowledge of the tester to create test cases which are not, by definition, boundary value tests, but may constitute a boundary value according to the domain of the program or software component [23].

Equivalence class testing, also known as partition testing, partitions the domain of input values into a finite number of disjoint subsets in which each subset represents an equivalence class, and the union of all the subsets equals the original input domain. One value selected from each partition is used as input for the test. A single value from an equivalence class represents each value in that class. Equivalence class testing provides completeness and non-redundancy to the testing process. Since the union of all the partitions yields the domain, each type of value is represented. Since each equivalence class only provides one value no redundancy exists among the input values. Equivalence class testing can be strong or weak. \textit{Weak equivalence class testing} only one value from each partition is selected for each input value. Therefore, the number of test cases equals the maximum number of equivalence classes. \textit{Strong equivalence class testing} also selects one value from each partition for each input, forms the Cartesian Product of those values and produces the inputs for the test [23, 21, 5, 26].

\textit{Decision table based testing} uses conditions and actions to determine test cases to use as input. Relationships among the inputs form the conditions, the specification determines the actions that result
when some, or all of the conditions are satisfied, or not satisfied. The combinations of conditions and actions form a decision table which determine the expected outputs for a given set of inputs. The test cases are formed by selecting input values which satisfy or do not satisfy the conditions as indicated by the table. This functional testing technique relies heavily on the domains of the input values, and on the specification [23].

**Goals of This Research**

The Class Validation System supports automated testing of SPECS-C++ class specifications and their corresponding implementations. This test system provides software developers the opportunity to test formal model-based specifications in a manner similar to traditional software testing, and an automated way to use the specification as the basis for testing the implementation. The Class Validation System allows the software developer to tie the steps of the software development process more closely together than steps in the traditional waterfall model. An Abstract Test Tool tests a specification against informal requirements and a corresponding Class Validation Tool ties an implementation back to its specification. Most defects occur because of insufficient or incomplete requirements [31]. The Abstract Test System is designed to find these sorts of errors early in the development process.

Software development cycles are shrinking dramatically as software providers try to enter an increasingly competitive market. This means less time is available for development and testing. Traditional development cycles do not support this rapid software development cycle [31]. The use of formal specifications and the Class Validation System should prove to be beneficial in the new environment. Taking the time to design using formal specifications has been demonstrated to improve quality and reduce the number of problems found in testing [24]. The Class Validation System integrates testing into the design and development process. The tool has been designed to identify errors in the specifications before implementation is even started, as well as errors in the implementation where it does not meet specifications. Development of the example used in this paper has shown that it effectively pinpoints the root of the problem, not just the visible side effects. Details of the Abstract Test System and Class Validation System will be presented in Chapter 4.

The next chapter gives an introduction to SPECS-C++, the compiler originally designed for this language, and introduces the example used throughout this dissertation. Chapter 3 discusses the enhancement to the SPECS-C++ compiler contributed by this research. Chapters 4 and 5 discuss in detail the tools and techniques which are the concrete results of this research, including the Class Validation
System. Chapter 6 discusses research and software tools related to this work. The final chapter includes the benefits and impacts of this research.
CHAPTER 2 SPECS-C++

Model-based specification languages are often used to specify abstract data types (ADTs)\(^1\) in a manner consistent with the object-oriented programming paradigm [33]. The model in a model-based specification defines the domain of the ADT being specified; it is an abstraction of the set of the instances of the class. The operations on the ADT are then specified in terms of the model. The primary purpose of a class specification is to define and describe the interface behavior of a class. The interface behavior to a class is the detailed design specification for that class. It defines the operations on the class, and how those operations will act upon instances of the class. A class specification is used by both the class implementor and clients of the class, to know how the class and its objects should behave. A class client, any other code which uses that class, need only understand the behavior interface of the class to use it. The ADT priority queue, described below, provides an example.

A priority queue is an ADT based upon the first-in-first-out ADT queue. Items are removed from a priority queue based both upon the order in which they were placed in the priority queue (like a regular queue) and on a priority level associated with each item. Items in a priority queue with the highest priority are removed first and, among the items with the highest priority, the one that was placed in the queue first will be the first to be removed [36].

There are numerous reasonable models that might be used to specify priority queue. One possible model is a sequence of pairs. Each pair would contain the item and its associated priority level. The sequence would model the order in which the items were inserted into the structure with the most recently inserted items at the end of the sequence and oldest items the beginning. Alternatively, a priority queue can be modeled as an unordered collection of triples, in which each triple contains the item, its associated priority level, and a time stamp to model the order in which items were inserted into the collection.

SPECS-C++ is a formal model-based specification language which is well-suited for specifying C++ classes that implement ADTs like priority queue. It was developed at Iowa State University by Baker

\(^1\)An abstract data type is a domain of values, and a set of operations upon those values.
and defined by Coleman [11]. A SPECS-C++ class specification appears as comments in the C++
header file for the class. This allows the specification to appear interspersed with the actual class
declaration.

Models in SPECS-C++ are built upon primitive, discrete, mathematical types which are built into
the language. There are two kinds of primitive types: simple types and structured types. The simple
types are integer, real, character, boolean, the enumerated type, and string. The structured
types are set, sequence, tuple, and an alternatively defined type (similar to a union in C). In
addition, the specifier can compose more complex types from these primitives. All values in a SPECS-
C++ are interpreted as pure mathematical values, and not as objects. Equality of values does not
indicate equality of object identity.

The actual model component in SPECS-C++ has four parts. The domains section declares types
used in the rest of the specification. These abstract types are simple compositions of the primitive
types of the language. The data members section contains declarations of abstract data members in
terms of either the types declared in the domains section, or the primitive types of SPECS-C++. The
abstract functions section contains definitions of mathematical functions that are used to modularize
and simplify the assertions over the model which appear as pre- and postconditions to the operations,
and in the class constraint. The abstract functions defined in the model are parameterized definitions
of a value. The constraints section is a first-order predicate logic assertion written over the abstract
data members that limit the abstract values that make up the domain of the class.

Class operations are specified using pre- and postcondition assertions over the model. An opera-
tion specification has three parts. The first part which starts with the prefix preA:, is the abstract
precondition. The precondition is an assertion over the pre-state values of the operation and defines
the pre-states in which the operation may be used. If the precondition is omitted it is assumed to be
true, indicating that any state is a valid pre-state for this operation. The modifies clause identifies
the data members and parameters that may change as a result of the operation. Items not listed in
this clause will not change the abstract value as a result of this function. If this clause is omitted it is
assumed that no parameters change state as a result of the operation. The final part of an operation
specification is the postcondition, and starts with postA:. It describes the state which results from the
execution of the operation. The post-state is defined in terms of the pre-state values. The post-state
values are identified primed identifiers, and the return value of the function by the keyword result.
Any identifier which is primed in the postcondition must be listed in the modifies clause of the operation
class PriorityQueue {
    /* model
    ** domains
    ** char ElemType
    ** int PriorityType
    ** int TimeType
    ** tuple (ElemType data
    **     PriorityType priLevel
    **     TimeType entryTime) EntryType
    ** set of EntryType QueueType
    **
    ** data members
    ** QueueType thePQ
    
Figure 2.1 Class PriorityQueue, formal model

specification.²

The following example is a specification of a class representing ADT priority queue described above. In addition to common C++ class functions [35] (a default constructor, a copy constructor, a destructor and an assignment operator), the interface to this class implementing a priority queue provides functions to add an item, remove the first item, look at the first item, and view the priority of the first item [36].

The entire SPECS-C++ model for class PriorityQueue appears across Figures 2.1, 2.2, and 2.3.³ Figure 2.1 contains the domains and data members sections. The type EntryType defined in the domains section is the type of the items contained in the priority queue.⁴ In this example, EntryType is modeled as char, and PriorityType and TimeType are modeled as int. EntryType, PriorityType, and TimeType are the types of the components of the tuple EntryType. The entryTime field in the tuple is used to model the order of insertion into a priority queue. A set of EntryTypes makes up a QueueType. There is only one data member used to model a priority queue, thePQ, of type QueueType. Thus, a client can think of an instance of class PriorityQueue as having a single data member which is a set of triples.

The abstract functions for class PriorityQueue is in Figure 2.2. UniqueTimes defines a boolean value which is true if no two distinct EntryTypes in the parameter q have the the same value entryTime field of the tuple. The abstract function HighestEntry defines the EntryType from the parameter q which is "first". This value is determined by the definition of a priority queue, namely the item with the

²A full grammar of SPECS-C++ can be found in [19, 15].
³The full specification appears also in Appendix B.
⁴In the example class PriorityQueue the types for EntryType, PriorityType, and TimeType have been predefined. To make this class reusable, a template class would be used. This allows the class to be created in a manner which allows any types to be substituted for the elements in the queue, and the priority. SPECS-C++ does not support template classes.
abstract functions

- `define UniqueTimes(QueueType q) as boolean such that UniqueTimes(q) = \forall(EntryType e1) [ (e1 \in q) => \forall (EntryType e2) [ (e2 \in q) => ((e1 = e2) V entryTime(e1) \neq entryTime(e2)) ] ]`

- `define HighestEntry(QueueType q) as EntryType such that \exists(EntryType e) [ (e \in q) /\ \forall(EntryType e1) [ (e1 \in q) => (e = e1)\/ ((priLevel(e) > priLevel(e1)) \land (entryTime(e) < entryTime(e1))) ] /\ HighestEntry(q) = e ]`

- `define LatestTime(QueueType q) as TimeType such that (q = {} => LatestTime(q) = 0) /\ \exists(EntryType e) [ (e \in q) /\ \forall(EntryType e1) [ (e1 \in q) => (e = e1)\/ (entryTime(e) > entryTime(e1)) ] /\ LatestTime(q) = entryTime(e) ]`

Figure 2.2 Class PriorityQueue, abstract functions

constraints

- `UniqueTimes(thePQ) /\ \forall(EntryType e) [(e \in thePQ) => priLevel(e) \geq 0 ] /\ \forall(EntryType e) [(e \in thePQ) => entryTime(e) \geq 0 ]`

Figure 2.3 Class PriorityQueue, constraints

highest priority which has been in the queue the longest. In this specification `HighestEntry` contains the information that distinguishes this class ADT as priority queue. `LatestTime` defines the `entryTime` of the `EntryType` in the parameter `q` which has the most recent timestamp.

This constraint (i.e. invariant) for the class PriorityQueue indicates that each `EntryType in thePQ` must have a unique `entryTime` value indicated by the abstract function `UniqueOrder`, and that each `orderOfEntry` and `priLevel` must be non-negative.

The operations described earlier are specified in Figures 2.4, 2.6, and 2.7. The member functions which appear in Figure 2.4 are the constructors for the class. These functions create new instances of the class. The first constructor is the default constructor, and creates an empty PriorityQueue. The second creates a PriorityQueue which is a copy of the argument `pq`. The copy is a "value" copy, meaning that the values in `thePQ` are copied into the newly created instance, not referenced by it.
** operations
*/
public:

PriorityQueue();
    /* modifies: thePQ
    ** postA: thePQ' = {}
    */

PriorityQueue(const PriorityQueue^ pq);  
    /* modifies: thePQ
    ** postA: thePQ' = pq.thePQ
    */

Figure 2.4  Class PriorityQueue, constructors

~PriorityQueue();
    /* modifies: thePQ
    ** postA: trashed(thePQ)
    */

Figure 2.5  Class PriorityQueue, destructor

Figure 2.5 contains the specification for the class destructor. This method is invoked whenever an instance of the class is destroyed. The predefined function trashed indicates that the value has been destroyed and is no longer usable.

Class functions which are not constructors can be classified as mutators, or observers. The mutators are the functions which change the state of the class instance, observers do not. Figure 2.6 shows the specification for the two member functions which add and remove items from the priority queue, altering its state. The function AddEntry inserts a new item into the priority queue. The postcondition describes the post-state value of thePQ as the union of the pre-state value of thePQ and the set containing the single tuple constructed from the new item, its priority, and order of entry into the PriorityQueue. RemoveEntry describes the post-state of thePQ as the pre-state value of thePQ with the first entry, as defined by the abstract function HighestEntry, removed.

The three functions specified in Figure 2.7 are the observer functions for this class. They allow clients of the class to observe the state of the class instance without changing it. The function FirstEntry indicates that the return value of this function is the data item from the HighestEntry in thePQ. The value of this function is only defined when the precondition is satisfied, when thePQ is not empty. HighestPriority returns the priority of the first entry in the priority queue. The boolean valued
void AddEntry(const EntryType& e, const PriorityType& level);
// modifies: thePQ
** postA: thePQ' = thePQ \union \{ (e, level, LatestTime(thePQ) + 1) \}
/*

void RemoveEntry();
// preA: thePQ != \{ \}
** modifies: thePQ
** postA: thePQ' = thePQ - \{ HighestEntry(thePQ) \}
*/

Figure 2.6 Class PriorityQueue, mutator functions

EntryType FirstEntry() const;
// preA: thePQ != \{ \}
** postA: result = data(HighestEntry(thePQ))
*/

PriorityType HighestPriority() const;
// preA: thePQ != \{ \}
** postA: result = priLevel(HighestEntry(thePQ))
*/

bool IsEmpty() const;
// postA: result = (thePQ = \{ \})
*/

} ; // end class PriorityQueue

Figure 2.7 Class PriorityQueue, observer functions

function IsEmpty is included to provide a way to test the preconditions of the functions in this class. The result of the function is true if the value of thePQ is the same as the empty set, and false otherwise.

This example class PriorityQueue demonstrates a portion of the grammar of SPECS-C++ and its usage. A thorough treatment of the language is found in [19, 15].

**Executability of SPECS-C++**

Testing of a specification is dependent upon the executability of the specification language. With an executable specification language a specification can be tested, and be used as an oracle when testing an implementation. In [37], Wahls defines the significant and useful subset of SPECS-C++ that is executable and gives an algorithm for executing assertions over models in SPECS-C++. The subset of assertions over the intrinsic types in SPECS-C++ that can be executed using Wahls' approach are
called constructive assertions. A constructive assertion is one from which post-state values that satisfy the assertion can be created. In practice, most of the assertions written to specify C++ class member functions are constructive. Wahls describes the restrictions on universal and existential quantification assertions. These restrictions apply to the structure of these assertions, and rarely impact the expressiveness of the specification. His algorithm for executing assertions has two main steps: first to check the actual arguments with the precondition, and second, to construct post-state values that satisfy the postcondition.

Preconditions describe the required state before the function may be executed. By definition they cannot contain post-state values. Since there are no post-state values to construct, and the pre-state values are known at the time of the call, preconditions require only evaluations. Set comprehensions (known as “set builder notation” in [19, 15]) and universal and existential quantifications are challenging to evaluate in both preconditions, and postconditions. For a universally quantified assertion in SPECS-C++ to be executable it must be of the form:

$$\forall T(x) \left[ (BP(x)) \land P(x) \Rightarrow Q(x) \right]$$

and existentially quantified assertions must be of the form:

$$\exists T(x) \left[ (BP(x)) \lor P(x) \right]$$

where $T$ is the type of the bound variable, $x$, and $BP(x)$ is either $x \in E$, where $E$ is a finite set or sequence, or $BP(x)$ is $low \leq x \leq high$, for some integer valued expressions $low$ and $high$. The assertion $BP(x)$, is the bounding predicate which defines the finite domain of $x$. The assertions $P(x)$ and $Q(x)$ are executable assertions over the quantified variable $x$. The quantifications are evaluated by evaluating the predicates $P(x) \Rightarrow Q(x)$ and $P(x)$ once for each value of $x$ in the finite domain defined by $BP(x)$, and using logical and ($\land$) for universal quantification and or ($\lor$) for existential quantification to evaluate the entire quantification. To ensure that a universal quantification of the above form is constructive as well as executable, neither the bounding predicate $BP(x)$, nor the predicate $P(x)$ may contain post-state values, and the predicate $Q(x)$ must be constructive. Similarly, existential quantifications are constructive if the bounding predicate $BP(x)$ does not contain post-state values, and the predicate $P(x)$ is constructive [37].

Set comprehensions are evaluated in a similar manner. A set comprehension will have the form:

$$\{ F(x) \mid (BP(x)) \land P(x) \}$$
where $BP(x)$ represents the same type of assertions above. $P(x)$ is a filter of $x$ and $F(x)$ is a function of $x$ building on $P(x)$. The set defined by this comprehension is constructed by creating a domain set of values defined by the bounding predicate $BP(x)$ which also satisfy the predicate $P(x)$, and constructing the desired set from the values which result by applying the filter $F(x)$ to each value in the domain set. Set comprehensions are constructive if they do not contain post-state values [37].

Constructing post-state values that satisfy the postcondition is the involved aspect of the algorithm. An overview of the algorithm is given in the following five steps from [37].

1. Split the postcondition into constructive and non-constructive parts. The constructive parts are those which will be helpful in constructing the post-state values which satisfy the postcondition.

2. Generate a list of constraints from the constructive part. Each constraint defines a post-state value, or some part of a post-state value.

3. Solve the constraints to construct a portion of the post-state that differs from the pre-state.

4. Check the non-constructive portion of the postcondition by evaluating it in a state the pre-state values for the unprimed identifiers, and the values constructed in the previous step for the post-state values. If this check fails, then the execution of the postcondition fails.

5. Construct the post-state which results from the function call. This state reflects changes to the default parameters, actual parameters to the function, and the return value (if any) of the function.

Wahls’ execution method is incorporated in an interpreter written in Standard ML. The interpreter uses an abstract syntax for SPECS-C++ that is similar to ML syntax. As will be described in the next section, a compiler which generates C++ code from a SPECS-C++ specification implements Wahls’ algorithm has been implemented.

**Implementation of SPECS-C++ Prototyping Compiler**

The compiler used in this research was based upon the compiler begun by Haverdink [19] and Goodman [15]. Details regarding additions to the compiler to support the Class Validation System are described in Chapter 3. The Abstract Test Tool and Class Validation System described in Chapter 4 rely on this compiler to generate code the the specified class as well as generate parts of the tools used to test the specifications. This section will describe the details of the data structures used by the compiler to to produce C++ code which implements Wahl’s execution technique for SPECS-C++ assertions.
The internal data structures used by the compiler to implement the primitive types of SPECS-C++ have been specified in SPECS-C++, and implemented in C++. They were developed in conjunction with the Abstract Test Tool by Gurski and Haerdink [17]. The semantics of the SPEC-C++ compiler, and the code it generate, depend on the implementation of the primitive types. The design details of the implementation follows.

A "meta class", class T, has been implemented to be a base class for each of the types in SPECS-C++. An instance of the class T has (in the model) two parts: a type and a value. The type part can be any one of the SPECS-C++ primitive types listed in the beginning to this chapter, and the value is of the type that is in the type part of the model. Figure 2.8 shows the hierarchy of the data structures used to implement the SPECS-C++ data types.

A partial header of meta-class class T is provided with an explanation in Figures 2.9 and 2.10. The full specification for class T may be found in Appendix A. At this time the compiler does not support the specifications of the classes used to implement the code it generates.

Defined in the domains section (see Figure 2.9) are two types that will be used in the class specification. The type Tnames is an enumeration of the types that a value can take. Ttype is an alternatively defined type of the allowable types. The type UNKNOWN is used when an empty set or sequence is encountered and its type cannot be determined from context. The constraint ensures that the data members type and val are consistent (i.e. the label type correctly indicates the type of val). The type containedType is used in some of the derived types.
class T {
/* domains */
** ( INT, FLOAT, CHAR, STRING, BOOL, ENUMERATED, 
* SET, SEQUENCE, TUPLE, ALTERNATE, UNKNOWN) Tnames 
**
** integer | real | character | string | boolean | Enumerated | 
** Set | Sequence | Tuple | Alternate Ttype 
**
** data members 
** Tnames type 
** Ttype val 
** Tnames containedType -- used in 'container classes' like set 
**
** abstract functions 
** define HasType(T value, Tnames typename) as boolean such that 
** HasType(value, typename) = 
** ( ((value ofType integer) \/ typename = INT) \/
** ( ((value ofType real) \/ typename = FLOAT) \/
** ( ((value ofType character) \/ typename = CHAR) \/
** ( ((value ofType string) \/ typename = STRING) \/
** ( ((value ofType boolean) \/ typename = BOOL) \/
** ( ((value ofType enumerated) \/ typename = ENUMERATED) \/
** ( ((value ofType set) \/ typename = SET) \/
** ( ((value ofType sequence) \/ typename = SEQUENCE) \/
** ( ((value ofType tuple) \/ typename = TUPLE) \/
** ( ((value ofType alternate) \/ typename = ALTERNATE)) 
**
** constraints 
** HasType (val, type) 
**
** operations */

Figure 2.9  Model for class T
There are a variety of constructors for class T. A subset of these constructors may be found in Figure 2.10. The default constructor creates an integer.\(^5\) A constructor that takes an argument of type Tnames is included to create an uninitialized value of a certain type. Values of the types integer, real, boolean, character and string can be promoted to an instance of class T with constructors.\(^6\) An example of such a constructor is provided for the type real (implemented as double).

Figure 2.11 contains a subset of the observer functions for class T. The function GetType returns the value contained in the data member type (which represents the type of the value contained in the abstract data member val). The pure virtual function getKind is used by some of the classes that are derived from class T and will be examined more closely later in this section. For each of the types integer, real, boolean, character and string there is a function that will return the value contained in val. The example given is for real. The function OfType is implemented to aid the SPECS-C++ tools and returns a string that contains a representation of the most-specific type of val. For example, the result of OfType for the value \{1, 2, 3\} would be 'set of sequence of int'.

A collection of operations used to compare instances of class T that are of the same type are given. The example of one of these operators in Figure 2.11 is the equality operator. The notion of equality is defined for all SPECS-C++ primitive types. It is defined to be value equality, not object equality.\(^7\) Two sets which contain the same values are equal under the SPECS-C++ definition of equality. This is the notion of equality which is defined for the specification of the classes which implement the SPECS-C++

---

\(^5\)The implementation of the enumeration Tnames contains values of the form T.typeName, where typeName is one of the enumerands of Tnames. For example, T.Int is the implementation value which corresponds to the specification value INT.

\(^6\)In SPECS-C++ the types integer and real represent mathematical integers and real numbers. The SPECS-C++ compiler implements these types as int and double respectively, ignoring the limitations of the programming language.

\(^7\)Two objects are equal (using value equality) if they have the same structure, and the same attributes. For example, two apples are equal if they are the same variety, have the same weight and size. Under object equality, two objects are equal if they are the same object. Using the apple example, no two distinct apples can be the same apple since each is a distinct instance of apple.
Tnames GetType() const;
/* postA: result = type */

virtual Tnames getKind() const { };
/* preA: type \in \{ SET, SEQUENCE \}
** postA: result = containedType */

double GetFloatVal() const;
/* preA: type = FLOAT
** postA: result = val */

virtual String OfType() const;
/* postA: result = strtype(val)
** -- where strtype is a string representation of the type of val
** -- for example, strtype({1,2,3}) is set of int */

virtual bool operator == (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self = t2) */
}; // end of class T

Figure 2.11 Observers for class T

primitive types.

Each of the structured types is implemented as a class derived from class T. The operations that are defined for each of the structured types are included as member and friend functions of the class. The partial specification of class Sequence illustrates this.

Sequences in SPECS-C++ are homogeneous mathematical structures. Since an object of type T can essentially have any type as its value, we need a way to indicate that all the T's that make up the Seq data member have the same underlying type. This is accomplished with the data member containedType inherited from class T, and the class constraint (Figure 2.12). The constraint indicates that each item in the sequence is of the same type, and that the data member containedType (inherited from the base class T) correctly identifies this type.

The default constructor (Figure 2.13) creates an empty sequence. If a type is not provided, containedType is UNKNOWN until the type can be inferred from context. The constructor that takes an argument of type T creates a new sequence which contains that value. The function AssignToIndex

8The enumerated type which is a simple primitive type in SPECS-C++ is also implemented as a class derived from class T.
class Sequence : public T {
/* model */
** domains **
** sequence of T SeqModel **
** **
** data members **
** SeqModel theSeq **
**
** constraints **
** \forall (T t1) [ (t1 \item theSeq) => **
** \forall (T t2) [ (t2 \item theSeq) => **
** t1.type = t2.type /\ containedType = t1.Type ]] **
** operations */
public:

Figure 2.12 Model for class Sequence

was provided to simplify the code generated by the compiler. The function allows the value at a particular location in the sequence to be changed, or for a value to be added at the end of a sequence. The function getKind returns the type of the values contained in the sequence by traversing the structure of the value to determine the most-specific type.

All the built-in operations for sequences are supported in the implementation for class Sequence. They include First, Last, Header, Trailer, Length, and Index. The specifications for First, Index, Length, and Header are shown in Figure 2.14. The function IsItemOf implements the \in function for sequences (a membership test). The concatenation operator allows two sequences to be connected to create a new sequence.

There are similar class specifications for class Set, class Tuple, class Enumerated and class Alternative that implement the rest of the primitive types of SPECS-C++. Full specifications for all the classes used to implement the primitive types of SPECS-C++ are given in Appendix A. These classes are used by the SPECS-C++ compiler to implement abstract data members. Further enhancements to the SPECS-C++ compiler produced for the Abstract Test Tool and Class Validation System are described in the next chapter.
Sequence(Tnames tn = TUnknown);
/* modifies: self
** postA: theSeq' = [] \ containedType' = tn */

Sequence(const T& val);
/* modifies: self
** postA: theSeq' = [val] \ containedType' = val.type */

void AssignToIndex(int where, const T& what);
/* preA: (containedType = what.type \ containedType = UNKNOWN) \*/
** ( 1 <= where \ where <= length(theSeq) + 1)
** -- where = (length + 1) means append the value to the end of
** -- the sequence.
** modifies: self
** postA: ((where <= length(theSeq) =>
** \forall(int i) [ 1 <= i \ i <= length(theSeq) \/
** i != where =>
** theSeq'[i] = theSeq[i]] \/
** theSeq'[where] = what) \/
** (where = length(theSeq) + 1) =>
** (theSeq' = theSeq [\ [what]] ) \/
** (containedType' = what.type)
*/

virtual Tnames getKind() const;
/* postA: result = containedType */

Figure 2.13 Operations of class Sequence
virtual T First() const;
/* preA: theSeq != []
** postA: result = first(theSeq)
*/

virtual Sequence Header() const;
/* preA: theSeq != []
** postA: result = header(theSeq)
*/

virtual T& Index(int i) const;
/* preA: 1 <= i <= length(theSeq)
** postA: result = theSeq[i]
*/

virtual int Length() const;
/* postA: result = length(theSeq)
*/

virtual bool IsItemOf(const Tfc val) const;
/* preA: \exists(T t) [ t \elem theSeq /\ t.type = val.type ]
** postA: \exists (int i) [ 1 <= i <= length(theSeq) /\ theSeq[i] = val ]
*/

virtual Sequence operator || (const Sequence& s2) const;
/* preA: \forall(T t1) [ \forall(T t2) [ (t1 \elem s1 /\ t2 \elem s2) \implies t1.type = t2.type ]]
** postA: result = s1.theSeq || s2.theSeq
*/

}; // end of class Sequence

Figure 2.14  Sequence primitive operations for class Sequence
CHAPTER 3 THE SPECS-C++ COMPILER

As described in Chapter 2, Wahls developed a technique for directly executing SPECS-C++ specifications and implemented this technique in a SPECS-C++ interpreter in Standard ML. The SPECS-C++ compiler was developed to generated C++ code from specifications. This generated code is a prototype of the class and can be linked with clients of the class. The final implementation of the class can be linked to the client in the place of the prototype with no changes to the functional behavior of the client program. In [19], Haverdink developed a parser for SPECS-C++ and provided a formal definition of the language. Goodman [15] incorporated Wahls' methodology for executing constructive assertions into a preliminary compiler that generates prototypes of classes specified in SPECS-C++. These prototypes are generated C++ code which can be compiled and linked with clients of the specified class. The compiler, built upon the parser and type-checker for SPECS-C++ developed by Haverdink[19], generates C++ code that uses data structures implementing the primitive types of SPECS-C++. Specific implementation details of the SPECS-C++ compiler can be found in [15]. In general terms, it is a single pass, compiler with bottom-up parsing implemented using the compiler tools Flex [13] and Bison [14] which are alternate implementations of lex and yacc [22]. The compiler does not fully implement Wahls' execution algorithm, but it does generate code which constructs valid post-state values which satisfy function postconditions for many specifications. It does not evaluate preconditions, class constraints, or non-constructive assertions. The development of the Class Validation System, discussed in Chapter 4, resulted in enhancement to the compiler, including support for the evaluation of preconditions and class constraints. These enhancements are discussed later in this chapter.

The preliminary compiler produced code for many specifications, but it was not complete. Development of the Abstract Test Tool and the Class Validation System required a more complete compiler. Enhancements made to the compiler include support for the SPECS-C++ type Tuple, abstract functions, and nested quantifications. The enhancements to the compiler are described in the next section.
void AddEntry(const ItemType& e, const PriorityType& level);
/* modifies: thePQ
** postA: thePQ' = thePQ \union \{ (e, level, LastEntry(thePQ) + 1) \}
*/

Figure 3.1 Specification for function AddEntry

Enhancements to the SPECS-C++ Compiler

Enhancements to the SPECS-C++ compiler included both minor changes, and substantial additions. All of these additions made it possible to compile specifications to be used by both the Abstract Test System and the Class Validation System.

Support for the SPECS-C++ primitive type Tuple had been begun by Goodman. The support for this type was completed by propagating the generated tuple field access code up the parse tree. For example, the postcondition of the function AddEntry in the class PriorityQueue (see Figure 3.1) contains a tuple which is created in a set which is part of another expression. The parse tree for the example appears in Figure 3.2. The box in the figure contains the part of the tree where the code to evaluate the tuple field access is generated. To ensure that this code is generated with the whole expression it must be propagated up through all the expressions above it in the tree: +, {}, \union, and =. When the generated code for the entire expression is written to the class prototype implementation the code for the tuple will be written out as well. This propagation was not carried out by the earlier version of the compiler.

Support to generate code for abstract functions was added. The generated code is a friend function to the class which evaluates the expression defined by the abstract function. The SPECS-C++ compiler will correctly parse and generate code for recursive and non-recursive abstract functions.

To fully support all parts of the SPECS-C++ model section, the grammar for SPECS-C++ was slightly modified by switching the positions of the abstract functions and the class constraint. The class constraint is now the last part of the model. This allows constraints which refer to abstract functions to be easily parsed by the SPECS-C++ compiler. Abstract Test Tools contain a feature which evaluates the class constraint of each abstract value they generate. To do this, a boolean function is generated in the abstract class which returns true if and only if the default parameter satisfies the class constraint.

The Abstract Test System has an automatic test feature which invokes functions on the abstract values. To ensure that each function's precondition is satisfied, a feature was added to generate boolean functions which will evaluate the preconditions of each function in the class.
Figure 3.2 Parse tree for AddEntry postconditions
As documented in [15], not all quantifications in expressions were supported in the SPECS-C++ compiler, specifically nested quantifications and quantifications that contain no post-state values. Since assertions in preconditions and constraints never contain post-state values, and nested quantifications are very useful, these limitations have been removed from the compiler.

Some errors in the compiler were corrected as well. The code that was generated for friend functions was corrected to not use the keyword friend in the implementation of the abstract class. The original design of the symbol table did not support enough information to generate correct code for tuples. The symbol table was enhanced to complete the support of tuples by the compiler described earlier.

**Limitations of Current Compiler**

Although many limitations of the SPECS-C++ compiler were removed because of the needs of the Abstract Test Tool and The Class Validation System, some limitations remain. The largest known limitation is that there are certain kinds of quantifications that cannot be parsed by the current grammar.

The following example is used to illustrate the problem. Class Digraph can be specified as a set of nodes, and a set of directed arcs. A node may be any simple type, an arc is a tuple containing two nodes, a "to" node and a "from" node, and an ArcSet is a set of arcs. The abstract function Sources contains a quantification that cannot be parsed by the current grammar defined by the compiler.

\[
\text{define Sources(NodeType } n, \text{ ArcSet } \text{arcset) as ArcSet such that }\\
\forall (\text{ArcType } a) [ (a \in \text{arcset }/\ n = \text{from(a)}) => (a \in \text{Sources}(a, \text{arcset})) ]
\]

In the quantification above, \(a\) is known as the *bound variable*. The grammar used to generate the C++ code expects the bound variable to be used alone in the bounding expression \(a \in \text{arcset }/\ n = \text{from(a)}\). In this case, the bounding expression is based upon one of the tuple fields of the bound variable, and the bound variable is not used alone.\(^1\)

**Representation Mapping Compiler**

Part of a systematic implementation design for a class specified using SPECS-C++ is a representation function[20] that maps the implementation data structures to the corresponding abstract models. The representation mapping function is the key which ties this research together. It joins the

---

\(^1\)Updating the SPECS-C++ compiler to accommodate expressions of this type was outside the scope of this work, and not done. Implementation of these expressions in SPECS-C++ is not required to demonstrate the work in this thesis. The current SPECS-C++ compiler provides sufficient functionality.
typedef int PriorityType;
typedef char ElemType;

struct Entry {
    ElemType element;
    PriorityType priority;
    Entry *next;
};
typedef Entry* EntryPointer;

Figure 3.3 Implementation type definitions for PriorityQueue

/ *— data members */

private:
    Entry* head;

Figure 3.4 Implementation data members for PriorityQueue

implementation data structure to the specification domain. To illustrate the representation mapping function in SPECS-C++, suppose the implementation of class PriorityQueue were a singly-linked list of structures which contained a priority, and an element. The structure definition of Entry, and the implementation data members are given in Figures 3.3 and 3.4.

The representation mapping in Figure 3.5 defines an abstract instance (as denoted by the type PriorityQueue_A) of the class given a concrete (implementation) instance (denoted by PriorityQueue) of the class. This mapping function is defined in terms of a recursive mapping function of a pointer to an Entry to QueueType. Each Entry in the linked list is mapped to an EntryType in the set of EntryType.

Since the mapping from abstract values to implementation representations is a one-to-many mapping, it would be expected that many implementation values may map to the same abstract value. This is clear since a linked list (the implementation design of class PriorityQueue) is an ordered structure, and a set (the model for class PriorityQueue) is unordered. The abstract value represented by the linked list pointed to by head is not dependent upon the order of the list, but only on the values contained in it. In fact, this implementation and model participate in a many-to-many mapping. The reasons for this, and the techniques used to handle it are described in Chapter 4.
/* abstract functions */
**
** define Repmap_QueueType(EntryPointer p, int t) as QueueType such that
** (p = NULL => Repmap_QueueType(p, t) = { } ) /
** (p != NULL => \exists (EntryType e)
**  [ (e = (p->element, p->priority, t)) /
**   (Repmap_QueueType(p, t) = ({e} \union
**    Repmap_QueueType (p->next, (t+1))))])
**
** define Repmap_PriorityQueue(PriorityQueue pq_c) as PriorityQueue_A
** such that
** Repmap_PriorityQueue(pq_c).thePQ = Repmap_QueueType(pq_c.head, 1)
*/

Figure 3.5 Representation mapping function PriorityQueue

Support for Representation Mapping in the SPECS-C++ Compiler

Adding the support to test the implementation of a class, in addition to testing the specification of the class, required the SPECS-C++ compiler to parse and generate code for more than SPECS-C++ class specifications. Although it is not necessary to parse the entire implementation in C++, it is necessary to parse the implementation type declarations, and the representation mapping function which combines SPECS-C++ types and assertions with C++ implementation types. To achieve this, a second parser was added to the SPECS-C++ compiler to parse the implementation data members and representation mapping functions. The second parser is known as the representation mapping compiler. Limited type checking support was added to parse the representation mapping function.

To mark the distinction between specifications and implementations a certain formatting style has been adopted. Although recommended, it is not required to conform to this style in SPECS-C++ specifications to use the tools described in the following chapter. Typically, when additional types are needed in a C++ class implementation, types are added in the header file before the class definition. The style employed in this research separates this information out into its own file, called the .pre file (denoted by the filename extension .pre). The implementation data members must appear in the class definition (according to the C++ grammar). To separate this information away from the specification, it is located in the .pri file (denoted by the filename extension .pri). The information in both the .pre and .pri files are included into the class header with the C++ preprocessor directive #include. This style successfully allows the reader of a SPECS-C++ class specification to ignore implementation issues of the class. The representation mapping function, and any other abstract functions that go with it are located in the .pri file following the implementation data members.
The .pre file contains the type definitions that are used to define the implementation data members, or the types used as parameters to functions. The types defined in this file constitute a part of the implementation which is analogous to the domains section of the specification model. This section consists of C++ typedef statements. The types defined by the typedefs must be parsed and stored in the compiler's symbol table. Parsing the type information allows for some simple type checking in the representation mapping functions. Simple types, like ElemType and PriorityType (see Figure 3.3) are straightforward to parse as they are quite similar to the simple types defined in the domains section of the specification. Pointer types, such as EntryPointer, are also allowed in this section.

The challenging type to parse is struct. For simplicity, they may only be defined by naming the struct (as in Figure 3.3). It is common for a struct to contain a field whose type is a pointer to the same struct. Therefore, it is necessary to cache the structure’s name in the symbol table before parsing of struct is complete.

The specification model is parsed by the representation mapping compiler just as it was by the original compiler. To avoid conflicts, each symbol in the specification model has an _A appended to it. The types defined in the model are referred to as the abstract types. The abstract types are referred to in the representation mapping functions with the suffix _A.

Use of the representation mapping compiler requires that only simple types, or types defined in the .pre file be used to declare the implementation data members. This requirement simplifies the work to be done by the parser gathering information for the implementation data members and does not limit the implementation. The types defined in the .pre file are the only ones that are necessary to parse the representation mapping function.

The representation mapping function is compiled in a manner much like abstract functions in the specification. The only additions to the grammar are operations on pointers and structs. Pointer operations -> and * translate into the corresponding C++ operation. Fields of a struct are accessed in the same manner as they are in C++ code.

The representation mapping compiler expects to find a representation mapping function. It will produce an error message if it does not find one. The representation mapping function must be named following the convention Repmap_{Classname}.

Example

Figures 3.6, 3.7, 3.8, and 3.9 are examples of some of the code generated by the SPECS-C++ compiler, and the representation mapping compiler. All the code generated for the class PriorityQueue by
both compilers is in Appendix D. These figures illustrate the code generated to implement specification domain types, abstract functions, function specifications, and the representation mapping function.

Recall from Figure 2.1 class PriorityQueue only has one data member, a set of tuples called thPQ. The types defined in the model are translated into typedef statements by the SPECS-C++ compiler. Figure 3.6 illustrates that the implementation of the SPECS-C++ primitives allow for a direct mapping in the generated code.

The abstract function LatestTime (see Figure 2.2) demonstrates both existential and universal quantifications, as well as nested quantifications. The generated code (Figure 3.7) implements Wahls' algorithm [37] for executing expressions with quantifications. The outer while loop, which implements the evaluation of the existential quantification, (following the comment exists._expr) iterates the a copy of the collection q until it reaches the end, or until eresult is true, indicating that the a time has been found which is later than all others. The inner while loop which implements the universal quantification (following the comment forall._expr), iterates a different copy of q. The value e5 is the current value for the outer loop, and the value e16 is the current value in the inner loop. The inner loop selects the entryTime field (it is the third field in the tuple) from both e5 and e16 and compares them. The boolean value solutionSelect17 is true if the entryTime of e5 is greater than the entryTime of all the other EntryType value in q. At the end of the inner loop, if solutionSelect17 is true, then there does exist an EntryType with the specified value for entryTime and we can exit the outer loop, setting the return value of the function in result to be the entryTime of e5.

The code generated to implement the postcondition of AddEntry appears in Figure 3.8. It constructs a new tuple, assigning the fields with the parameters, and the calculated time stamp. It then creates a new set to serve as the post-state value. It is assigned the value of the pre-state value of thePQ union-ed with the new tuple created from the parameters. The post-state value is then assigned to the data member.

The code generated to implement the representation mapping function is shown in Figure 3.9. This
myModal
void PriorityQueue_A::AddEntry (const ElemType_A& e,  
          const PriorityType_A& level) 
{
    QueueType_A _thePQ;

    Tuple tup3;
    T* tvalue22 = new T(e);
    tup3.AddComponent(T_char, "", tvalue22);
    T* tvalue23 = new T(level);
    tup3.AddComponent(T_int, "", tvalue23);
    T* tvalue24 = new T(LatestTime_abs_fn(thePQ) + 1);
    tup3.AddComponent(T_int, "", tvalue24);
    Set temp4( tup3 );
    _thePQ = ( thePQ ).Union(temp4);
    thePQ = _thePQ;
}

Figure 3.8 Code generated for member function AddEntry

PriorityQueue_A Repmap_pqueue::Repmap_PriorityQueue_abs_fn  
          (PriorityQueue pq_c) {
    PriorityQueue_A result;

    (result.thePQ) = ( Repmap_QueueType_abs_fn((pq_c.head), 1) );
    return (result);

}

Figure 3.9 Code generated for representation mapping function

is generated in the same manner as abstract functions in the abstract model. In this case, the function
refers to another abstract function in the representation mapping section.

The next chapter discusses how the Abstract Test System and Class Validation System use the
compiler to create an Abstract Test Tool and a Class Validation Tool.
CHAPTER 4  THE CLASS VALIDATION SYSTEM

The Class Validation System is a testing system which creates automated testing tools for C++
class specifications written in SPECS-C++ and corresponding C++ implementations of those classes.
Each tool created is composed of two parts, an Abstract Test Tool and a Class Validation Tool. The
Abstract Test Tool (ATT) [16] is the testing environment for the class specification, intended for use
after the class is specified, and before it is implemented. The Class Validation Tool (CVT) is a testing
environment which tests the class implementation against the specification. Generally, the specification
should be validated before it is used as the basis for the oracle generated by the CVS. The test cases
for the CVT are viewed in the same format as the test cases for the ATT. The user of the CVT need
only understand the test cases in terms of the specification since details of the class implementation are
hidden by the CVT.

The Abstract Test Tool is used to test a C++ class specification. In the automated mode the tool
generates instances of the class, as defined by the class model, containing abstract values. Chapter 5
describes techniques used to generate the abstract values. In the interactive mode, the abstract values
are built by the user through the application of class member functions, or entered in as text. The
class member functions are applied to values which satisfy function preconditions. The user validates
the post-state values. The process of processing one test case is shown in the top portion of Figure 4.1
labeled Abstract Test Tool. The abstract value is chosen (either by the user or the tool) and the tool
applies the function to this value. The resulting post-state value is the abstract result.

After the user has validated the specification to his satisfaction and implemented the class, the Class
Validation Tool is used to validate the implementation with respect to the specification. The previously
tested specification is the source of the expected output. Therefore, there is no need for user validation
and the Class Validation Tool is fully automatic. Similar to the ATT, the CVT generates values which
are called implementation values (see Chapter 5 for details regarding generation of implementation
values). Each implementation value is associated with an abstract value which it represents.¹

¹Typically, there is only one abstract value for any implementation value. The example used later in this chapter
shows a class specification and implementation where this is not the case, and how the tools support it.
A single test case in a CVT is more complex than the corresponding test case in an ATT. Testing begins with the selection of an abstract value. The known implementation values which represent this abstract value are retrieved. The left side of Figure 4.1 depicts this step. The CVT applies the function specification to the abstract value, and saves the abstract result (see the top of Figure 4.1). The function implementation is applied to each of the implementation values. The resulting values are saved as the implementation results. The multiple arrows represent the multiple potential implementation test cases. The right side of Figure 4.1 depicts the key portion of the test. The representation mapping function (discussed in Chapter 2) maps each implementation result back to the abstract value it represents. The test passes if each implementation result maps to an abstract result which is equivalent to the abstract result saved earlier. Equivalence to the abstract result is discussed later in this chapter. Testing terminates at the first error found since further tests would be corrupted if the incorrect value were saved.
Abstract Test Tool:

1) PriorityQueue_A ()
2) PriorityQueue_A (const PriorityQueue_A& pq)
3) "PriorityQueue_A ()
4) PriorityQueue_A& operator = (const PriorityQueue_A& pq)
5) void AddEntry (const ElemType_A& e, const PriorityType_A& level)
6) ElemType_A FirstEntry () const
7) PriorityType_A HighestPriority () const
8) void RemoveEntry ()
9) bool IsEmpty () const
0) exit

Selection:

Figure 4.2  Main Menu for the Abstract Test Tool for class PriorityQueue

Abstract Test System

The Abstract Test Tool operates in two modes: interactive and automatic. In the interactive mode the user selects the functions to test, and the values to use. The user chooses to automatically generate values, build the values through the functions, or enter them in as text. Abstract values are generated by the tool using one of the techniques described in the next chapter. The class constraint is applied to all values generated as class instances. Those values which satisfy the constraint are kept and the rest are discarded. The values are listed for the user to choose from when testing functions. As mentioned above, values may also be created through using the class functions. The user would start with the value which results from the execution of a constructor and apply various functions to the values. This produces only values which can be created through the interface of the class. The user may enter a text string which represents a value. The string is parsed and converted into a SPECS-C++ value for use as an abstract value. An example string representing an abstract instance of PriorityQueue would be \{ ('u', 411, 92 ), ('m', 671, 868 ) \}. Figure 4.2 is the main menu of the interactive ATT for class PriorityQueue. From this menu the user selects the function to test. Selecting option 5 will test the function AddEntry. The user selects the pre-state values. Figure 4.3 shows a display of selection of pre-state values and the display of post-state values. After examining the results the user may continue testing, or choose to stop and change the specification.

In the automatic mode the user selects parameters for generating the values (see Chapter 5) and each operation is executed with combinations of values matching the function signature. The result of each function execution is logged reporting the results of each test, including all errors. Examination
Executing: void AddEntry (const ElemType_A& e, const PriorityType_A& level)

----------------------
Pre-state Value Menu - self:
----------------------
1) { }  
2) { ( 'a', 5, 1 ) }  
0) edit new value

Selection: 2

----------------------
Pre-state Value Menu - const ElemType_A& e:
----------------------
1) 'a'  
0) edit new value

Selection: 0
Enter a value for const ElemType_A& e: 'b'

----------------------
Pre-state Value Menu - const PriorityType_A& level:
----------------------
1) 5  
0) edit new value

Selection: 0
Enter a value for const PriorityType_A& level: 3

----------------------
Results (Post-state values):
----------------------
Self': { ( 'a', 5, 1 ), ( 'b', 3, 2 ) }  
Params:  
'b'  
3

Figure 4.3 Interactive testing of class member function AddEntry, text based tool
Test Case #1
================================================================================================
Function: PriorityQueue_A ()
================================================================================================
Results (Test Case #1)

self': { }

Test Case #471
================================================================================================
self: { ( 'u', 411, 92 ), ( 'm', 671, 868 ) }
Function: void AddEntry (const ElemType_A& e, const PriorityType_A& level)
Parameters:
const ElemType_A& e: 'J'
const PriorityType_A& level: 4
================================================================================================
Results (Test Case #471)

self': { ( 'u', 411, 92 ), ( 'm', 671, 868 ), ( 'J', 4, 869 ) }
Params:
'J'
4

Test Case #477
================================================================================================
self: { }
Function: ElemType_A FirstEntry () const
================================================================================================
Results (Test Case #477)

self': { }
Message: "ERROR: precondition for FirstEntry not satisfied"

Test Case #487
================================================================================================
self: { ( 'm', 671, 92 ), ( 'u', 411, 868 ) }
Function: ElemType_A FirstEntry () const
================================================================================================
Results (Test Case #487)

self': { ( 'm', 671, 92 ), ( 'u', 411, 868 ) }
Result: 'm'

Figure 4.4 Excerpts from log file for automatic testing of class PriorityQueue
of the log determines if the specification is correct. Figure 4.4 contains four test case entries from a log file created during automated testing. The first entry Test Case #1 shows the testing of the default constructor. Each entry has two parts, separated by a dashed line. Above the line is the pre-state values, below it are the post-state values. Test Case #1 has no parameters, and no default parameters (it is a constructor). The only post-state value is the newly created default parameter. Test Case #471 is an example of a log entry for a function with both pre- and post-state values. The next entry shows how errors are reported, and the last (Test Case #487) how functions with return values are treated.

Design of the Abstract Test Tool

The Abstract Test System for SPECS-C++ provides a facility for testing SPECS-C++ class specifications. The test system generates a specialized tool for each class specification. Each Abstract Test Tool generated for a class specification by the Abstract Test System can be viewed as interactive client code of the class. Each tool allows the user to create and store instances of the class, and to execute member and friend function specifications. Each instance value is represented abstractly, according to the types defined in the class model.

Given a specified class, class C, there are several steps required to create an Abstract Test Tool for class C:

1. Compile the class specification for class C to C++ code with the SPECS-C++ compiler.

2. Use the Abstract Test Tool for SPECS-C++ to parse the class specification for class C to obtain the data model and function signatures. With this information, the test system produces C++ code for an Abstract Test Tool for class C. This C++ code contains the actual calls to member and friend functions of the class.\(^2\)

3. Compile and link the C++ code generated by the SPECS-C++ compiler and the code generated by the test system creating the Abstract Test Tool for class C.

First described in [17], the implementation design of the Abstract Test Tool (ATT) is composed of two main parts (see Figure 4.6). The user interface and the engine. The user interface communicates with the engine though the method Execute shown in Figure 4.5.\(^3\) The engine is generated by the SPECS-C++ compiler (with the -att option) and contains all the class specific code necessary to call

\(^2\)The SPECS-C++ compiler supports combining this step with step 1.

\(^3\)The interface for ReturnVals can be found in Appendix C.
#include "String.h"
#include "Array.h"
#include "ReturnVals.h"

extern String className;

ReturnVals *Execute(const String& defaultParam,
                    const StringArray& actualParams,
                    const String& memberFuncName);

/* This procedure will execute the member function identified by
   * memberFuncName on the defaultParam, with formal arguments
   * replaced by actualParams. Results of the execution will be
   * returned the result structure.
   * If an error occurs in the execution of the function then all
   * values in the result will be empty strings.
*/

StringArray FilterConstraintSatisfying(const StringArray& s);

/* This function takes an array of strings which represent values
   ** which are instances of the class being testing. The values in
   ** s which satisfy the constraint of the class are returned in the
   ** result.
*/

Figure 4.5 Interface for Abstract Test Tool engine

the functions of the class (created in step 2 above). The Abstract Test Tool is built by linking the
user interface, the engine, the compiled C++ code generated by the SPECS-C++ compiler, and the
SPECS-C++ library together. When the user of an ATT tests a function, the parameters are gathered
by the interface and sent to the engine. The engine then evaluates the precondition of the function,
executes the function if the precondition is satisfied, and evaluates the class constraint the resulting
default parameter and any other parameters of the same type as the class being tested.

Abstract values used in testing are stored and managed in the interface part of an Abstract Test Tool.
The values are stored as string representations of the SPECS-C++ values. The engine for a tool converts
each string value to the SPECS-C++ value. The interface validates class instance values through the
method FilterConstraintSatisfying method on the engine. This method takes a collection of string
representations of abstract values and returns a collection containing the values in the passed collection
which satisfy the class constraint. This method takes each value in the collection and applies the class
constraint to it. Only those values which satisfy the constraint are returned in the new collection.
The Class Validation System

Once a class specification has been tested so that it is deemed correct, and it has been subsequently implemented, the implementation can be validated against the specification to increase the confidence that the software correctly implements the specified class. The Class Validation System produces a tool greatly improves the thoroughness and efficiency of this validation process.

The tool produced by the Class Validation System has a design similar to the Abstract Test Tool. The main components of a Class Validation Tool are the user interface, which is independent of the class being tested, the Abstract Test Tool engine, the class specification, the class implementation, the Class Validation Tool engine, the representation mapping function, and equivalence function. Figure 4.7 shows the components of a Class Validation Tool.

The user interface of the Class Validation Tool integrates the pieces of the specification and implementation into a single tool. It controls the generation of values based upon user input, it selects the test cases and logs test results. The abstract engine and class specification are the same as generated for the Abstract Test Tool. The implementation engine is similar to the abstract engine except that it invokes the class implementation, rather than the class specification for each method. The representation
mapping compiler generates the `Repmap` and `Equivalent` functions which are also invoked through the implementation engine. The interface to the implementation engine is shown in Figure 4.8. Interfaces for other classes references can be found in Appendix C.

The `ExecuteImpl` method is implemented in a manner similar to the `Execute` function in the ATT. It takes a default parameter and an array of actual parameters. These values are held by void pointers. In C++, a void pointer is a reference to an object of any type. This makes it a convenient type to use for this generic interface. The use of pointers in this manner did result in some restrictions on classes which may be tested using the Class Validation System. The restrictions and explanations are discussed later in this chapter. `FilterConstraintSatisfyingImpl` performs the same function as in the ATT. The implementation constraint is evaluated using each of the passed values. Only those which satisfy the constraints are returned. The last two methods are used to evaluate the representation mapping function. The first, `PassRepmap` takes an abstract value and an implementation value. The implementation value is mapped to the abstract domain using the representation mapping function and compared to the passed abstract value using the `Equivalent` function. The result is the result of evaluating the `Equivalent` function. The `GetRepMapValue` function returns a string representation of the abstract value to which the implementation value mapped. This method is provided to aid in the reporting of errors when the `PassRepmap` function returns false.

Class Development with the Class Validation System

Introduction of the Class Validation System into the software development process provides the opportunity to test a software specification for correctness and then validate the corresponding implementation with respect to that specification. The benefits of this include more accurate software specifications and implementations which adhere closely to the specification. Formal specifications allow software developers writing client software to clearly understand the behavior of the specified component.

One possible development process of a software component with the Class Validation System would be composed of the following steps (see Figure 1.1). The process allows each component to be tested in isolation before it is introduced into the rest of the system. For each component

1. Gather requirements
2. Formally specify
3. Initial interactive testing with Abstract Test Tool
Figure 4.7 Design of a Class Validation Tool
#include "String.h"
#include "Array.h"
#include "ReturnValsConcrete.h"

extern String classNameImpl;
extern String typeInfoFilename;

ReturnValsConcrete *ExecuteImpl(const void* defaultParam,
 const ImplArray& actualParams,
 const String& memberFuncName);

/* This procedure will execute the member function identified by
** memberFuncName on the defaultParam, with formal arguments
** replaced by actualParams. Results of the execution will be
** returned the result structure.
** If an error occurs in the execution of the function then all
** values in the result will be empty.
*/

ImplArray FilterConstraintSatisfyingImpl(const ImplArray& vals);

/* This function takes an array of strings which represent values
** which are instances of the class being testing. The values in
** vals which satisfy the constraint of the class are returned
** in the result.
*/

bool PassRepmap(const String& abstractValue,
 const void* concreteValue);

/* postA: result = FromString(abstractValue) == Repmap(concreteValue)
** -- This is for the appropriate class
*/

String GetRepMapValue(const void* concreteValue);

/* postA: result = toString(Repmap(concreteValue)
** -- This is for the appropriate class
*/

Figure 4.8 Interface for the Class Validation Tool engine
4. Thorough automatic testing with Abstract Test Tool

5. Review Test Results (return to step 2 until results are as desired)

6. Implement

7. Automatic testing with Class Validation Tool

8. Review test results (return to step 6 until results are as desired)

The above process defines the preferred methodology using the Class Validation System. Steps 3 and 5 ensure that the formal specification created in step 2 properly capture the requirements set forth in the initial evaluation. Implementation does not begin until the specification is tested and accepted. Because the specification describes the correct behavior, the Class Validation Tool uses the specification to validate the results of testing the implementation. Once testing is complete and the software is behaving as specified the component is ready to be integrated into the system.

Excerpts of test results from a test of class PriorityQueue are shown in Figures 4.9 and 4.10. The log file produced (Figure 4.9) reports each test that was performed and its outcome. This figure shows three tests, the default constructor, the function IsEmpty and AddEntry. These log entries are very similar to those produced by an Abstract Test Tool. Figure 4.10 shows excerpts from the summary of a run of the CVT for class PriorityQueue. The first section shows the class name, and the parameters used to generate values. The results include the date the test was run, how many values (abstract and concrete) were generated, and the most operations applied to a single value. Method coverage, and all the values generated are also included. The excerpts included in Figure 4.10 show three abstract values, and five concrete values. Each abstract value is shown in as its string representation, and each concrete value is shown as the list of methods and parameters used to create that value.

The summary contains three sections: settings, value and function count, and generated values. Included the Figure 4.10 are the settings, summary and excerpts from the generated values.

The settings section includes information such as classname and value generation parameters. In this figure, values were generated using a depth of 5, and a breadth of 5. The parameters MaxAbstract and MaxConcrete were not used. If supplied, these parameters indicate that testing continues until the specified number of abstract or concrete values have been generated.

The second section contains a time stamp, the number of values generated and the largest number of operations applied to an instance. It also contains a list of the class functions, and how many times each

---

4The technique Method-based value generation used to generate values is described in Chapter 5.
was used. In this example, the urn of the test on May 13, 2000 did not invoke several of the operations even once. Analysis of the corresponding log (excerpts of which appear in Figure 4.9) indicated that testing of these operations was attempted, but the values used did not satisfy the preconditions of the operations.

Excerpts from the list of values generated make up the last part of the figure. Shown here are two of the four abstract values. The first is one which has been destroyed and has only one corresponding implementation value. The second is shown with two of the implementation values which correspond to it. Each sequence of operations represents one implementation value.

**Oracle - Equivalence Testing**

One of the benefits of automatic testing of an implementation with respect to the formal specification is the formal specification supplies the oracle of correctness. The Class Validation Tool uses the specification to determine results of executing the implementation are correct. When the results are compared to the results of executing the specification the test is considered to pass. Figure 4.1 (shown earlier this chapter) shows the process a Class Validation Tool uses to run each test case. For the typical class, each abstract value can be represented by one or more implementation values. Comparing the abstract values represented by the implementation values to the abstract values determines the correctness of the test.

Abstract values are compared using value equality as defined by SPECS-C++. Equality of abstract values is observational equality. Values are the same if they are observed to produce the same results for any function applied to them. For example, the sets \{1,2,3\} and \{3,2,1\} are equal because they contain the same values. All functions on sets rely on which elements are in a set, and the order is not relevant. Functions on sequences do rely on order, therefore \[1,2\] and \[2,1\] are not equal.

In many cases equality of abstract values is a sufficient comparison for test validation. This is not the case with all classes. The example used in this paper, class PriorityQueue, has been specified and implemented such that there are equivalence classes of abstract values. Each equivalence class is defined by the order the items are selected for removal from the queue. The implementation of PriorityQueue does not implement the time stamp that is used in the abstract model. In the specification, the time stamp is used to determine the order of insertion into the queue; but in the implementation, an ordered list is adequate to represent this information. This means the results of a correct mapping of an implementation value may not produce results identical to abstract values produced by the specification, but produces an equivalent value. To support classes of this nature, the CVT provides an opportunity
Function: PriorityQueue_A ()
self:
Actual Params:
-----------------------------
Results:
--------

self': { }

Function: bool IsEmpty () const
self: { }
Actual Params:
-----------------------------
Results:
--------

self': { }
Result: true

Function: void AddEntry (const ElemType_A& e, const PriorityType_A& level)
self: { }
Actual Params:
'+'
318
-----------------------------
Results:
--------

self': { ( '+', 318, 1 ) }
Params:
'+'
318

Figure 4.9 Excerpts from the log file for the Class Validation Tool
Current Settings:
  Classname: PriorityQueue_A
  Depth: 5
  Breadth: 5
  MaxAbstract: 0
  MaxConcrete: 0

Method based value generation report
May 13, 2000 10:36:34

Summary
  Abstract Values: 4
  Concrete Values: 28
  Length of longest operation application sequence: 6

Function Usage
  1 PriorityQueue_A ()
  0 PriorityQueue_A (const PriorityQueue_A& pq)
  2 ~PriorityQueue_A ()
  1 PriorityQueue_A& operator = (const PriorityQueue_A& pq)
  2 void AddEntry (const ElemType_A& e, const PriorityType_A& level)
  0 ElemType_A FirstEntry () const
  0 PriorityType_A HighestPriority () const
  0 void RemoveEntry ()
  3 bool IsEmpty () const

Values Generated
  trashed

Operation Application Sequence:
  (1) PriorityQueue () ;
  (2) PriorityQueue& operator = (const PriorityQueue& pq) ; { }
  (3) bool IsEmpty () const ;
  (4) ~PriorityQueue () ;

{ ( '4', 318, 1 ) } Operation Application Sequence:
  (1) PriorityQueue () ;
  (2) PriorityQueue& operator = (const PriorityQueue& pq) ; { }
  (3) bool IsEmpty () const ;
  (4) bool IsEmpty () const ;
  (5) void AddEntry (const ElemType& e, const PriorityType& level) ;

'4', 318 Operation Application Sequence:
  (1) PriorityQueue () ;
  (2) bool IsEmpty () const ;
  (3) bool IsEmpty () const ;
  (4) void AddEntry (const ElemType& e, const PriorityType& level) ;

'4', 318

Figure 4.10 Excerpts from summary of automated testing for Class Validation Tool
for the implementor or specifier to provide an abstract function which defines the equivalence classes of abstract values. Equivalent\(^5\) is a boolean function parameterized by two abstract instances of the class whose value is true if and only if the two abstract values are equivalent. The oracle of the Class Validation Tool for class PriorityQueue uses the function Equivalent (see Figure 4.11 for an example) to determine if the implementation values produced the correct results for the test case. The Equivalent function, a property of the abstract specification, partitions the abstract domain into equivalence classes. Because there is no automated validation in the Abstract Test System, it is not necessary to define the Equivalent function until the implementation of the class is developed and may be needed by the Class Validation System.

The abstract function Equivalent, if supplied, will be used by the CVT to determine if the implementation results are correct. The result of applying the repmap function to each implementation result must be equivalent to the abstract result. If the abstract function Equivalent is not provided, the CVT compares each abstract data member using the equality methods defined by the SPECS-C++ types. If there is an error in the Equivalent function, test cases may fail. In general, test case failures may be caused by errors in the implementation, the representation mapping function, or the equivalence function.

The abstract function Equivalent is not the only solution to this problem. The compiler could generate code which would evaluate a postcondition, given pre- and post-state values. The results of applying the repmap function to the implementation results could be applied to the postcondition. A successful test case would produce results satisfying the postcondition in addition to satisfying the precondition.

**Implementation Details**

The implementation of the Class Validation Tool is very similar to the implementation of the Abstract Test Tool. As shown in Figure 4.7, there are four basic parts to a Class Validation Tool. The first is the user interface. The user interface, like the user interface for the CVT, is independent of the specific class being tested. The second part consists of the abstract engine and the executable class specification, produced by the SPECS-C++ compiler. The third part is produced by the representation mapping compiler and contains the implementation engine, the representation mapping function, and the equivalence function. The final part is the class implementation. To achieve a generic design, the

\(^5\)The Equivalent function is located in the .pri file of the class. It is translated to C++ by the representation mapping compiler.
/* abstract functions
** define EquivSets(QueueType q1, QueueType q2) as boolean such that
**  (((q1 = {}) \ (q2 = {})) => EquivSets(q1, q2) = (q1 = q2)) /
**  (((q1 !== {}) \ (q2 !== {})) =>
**   EquivSets(q1, q2) =
**   ((priLevel(HighestEntry(q1)) = priLevel(HighestEntry(q2))) /
**    (data(HighestEntry(q1)) = data(HighestEntry(q2))) /
**    EquivSets((q1 - {HighestEntry(q1)}), (q2 - {HighestEntry(q2)})))))
**
** define Equivalent (PriorityQueue_A pql, PriorityQueue_A pq2) as boolean
** such that
**   Equivalent(pql, pq2) = EquivSets(pql.thePQ, pq2.thePQ)
*/

Figure 4.11  Abstract function Equivalent

pointers were used in the user interface and in the classes which support value generation. The method employed to manage the abstract and implementation values generated will be discussed in Chapter 5.

The CVT builds a collection of values using the class methods. This collection consists of abstract values and the implementation values which represent them. The tool ensures that each implementation value maps to the abstract value with which it is associated using the representation mapping function. This collection of values is discussed in detail in Chapter 5.

Implementation Alias Problem

As discussed earlier, the Class Validation Tool is implemented in C++. Implementation of the tool includes the use of void pointer. A void pointer is a reference to an object of any type. When a pointer value is copied only the object reference is copied producing an alias. An alias occurs when two references exist to the same object.

The use of pointers in the Class Validation Tool places a restriction on the classes that can be tested. Each class must implement a "deep" copy constructor. A "deep" copy constructor copies the class instance following all pointers, and copying all the data along the chain. The restriction arose because pointers are used to indicate implementation representations, these pointers reference the actual object. If a function which mutates the object is invoked, the object will change, altering the implementation value in the pre-state as well as the post-state.

To illustrate, suppose an abstract instance of the class a is created, and its corresponding implementation c is associated with it, Figure 4.12. This object satisfies the constraint that c must map to a using the representation mapping function. Invoking a method on values a and c results in new
values \( a' \) and \( c' \). The object \( a' \) is distinct from the object \( a \) because the implementation of abstract values follows the semantics of the specification, indicating that values are immutable and the implementation of the SPECS-C++ compiler supports this by copying \( a \) to a new object \( a' \) before applying the function, resulting in pre-state objects which are distinct from post-state objects. The CVT uses a pointer to reference the implementation value \( c \). When that pointer is copied to a pointer which will reference \( c' \) both \( c \) and \( c' \) refer to the same object. When the method is invoked on \( c' \) and it’s value is mutated, the value reference by \( c \) is changed as well. The object referenced by \( c \) no longer represents abstract value \( a \). The object \( c' \) represents \( a' \), \( c' \) does not map to \( a \) (assuming \( a \) does not equal \( a' \)) (see Figure 4.13), leaving the values in an inconsistent state.

One possible solution to the aliasing problem in the CVT was to not keep a collection of values as in the original CVT algorithm. The proposed algorithm would follow one implementation value from the original constructor through the last test on that value (determined by testing parameters). If all the values tested were viewed as a tree with the value created by the constructor at the root, and values created from it as its children (an example appears as Figure 4.14), this approach would navigate the tree one path at a time from root to leaf, looking at the generation of values in a linear manner.

The primary advantage of this approach is that there are no added requirements of the classes that can be tested with the CVT. The class does not have to implement a deep copy constructor. Additionally, if the class were thread safe, the validation could proceed in a multi-threaded manner.

A significant disadvantage of this approach is that aliasing in the copy constructor is unlikely to
be identified. Because of the tree-like nature of the values generated using this approach, a number of values would be regenerated multiple times. Those values which appear close to the root of the tree are part of most of the paths from root to leaf in the tree. The tool would duplicate much work.

The second proposed solution requires that each class to be tested implement a deep copy constructor. The testing algorithm invokes the copy constructor to copy the pre-state value of the implementation representation of the object and invokes the method on the copy. This copy will become the post-state object. Immediately following the method invocation, a consistency check of the original object is done. This check evaluates the representation mapping function on the implementation pre-state value, and passes if the result is equal to the abstract pre-state value.

The primary advantage of this method is that the client receives immediate feedback regarding aliasing errors in the copy constructor. Without the consistency check of the pre-state implementation value, these aliasing errors in the copy constructor could not be identified by the tool and could cause errors to appear in later tests which are difficult to diagnose. The requirement of the deep copy constructor is both an advantage and a disadvantage. It is an extra requirement upon the class developer, but the advantage of identifying these types of errors is significant.

After careful evaluation, the option to require the class to implement the deep copy constructor was selected. Both options are described below. Experience and anecdotal evidence indicate that a copy constructor in a C++ class is a useful thing, but is frequently the source of errors.

Traditionally, the copy constructor makes a deep copy of the object, and does not alias the new
Figure 4.14 Example tree of values
object to the copied object[35]. The copy constructor in C++ may be invoked implicitly, and errors in
the copy constructor may appear to be errors in other methods. Because the CVT requires that the
copy constructor be implemented, and explicitly invokes it to make a copy of the implementation value,
it can test for errors in the copy constructor, and identify those which are not correctly making a deep
copy of the object, but are instead creating aliases. In fact, errors in the implementation of the example
class PriorityQueue were immediately found when the alias checking functionality was added to the
Class Validation System.
CHAPTER 5 ABSTRACT AND CONCRETE VALUES

The Class Validation System generates a testing tool for a class. Each Class Validation Tool generates and manages both abstract and concrete values. This chapter discusses two techniques used to generate and manage abstract and concrete values by the Abstract Test Tools, and in the Class Validation Tools. The first, model-based value generation, is used by an Abstract Test Tool to generate values from the data model supplied in the specification. Method-based value generation is used by both an Abstract Test Tool and a Class Validation Tool. It generates values from a sequence of method invocations.

Model-Based Value Generation

Model-based value generation creates values based on the structure of the model which satisfy the constraints of the class. This technique does not yield any knowledge of how these abstract class instances could be built using the class operations. Rather, it uses the structure of the model to generate values. The abstract types are first identified in terms of the the SPECS-C++ types from which they are composed. For example, recall that a PriorityQueue is modeled as a QueueType, and that a QueueType is a set of EntryType. This can be further simplified to set of tuple (char, int, int). The first step in generating values for this type is to parse the type, forming a parse tree, where the leaves represent the types of the first values to be generated. Figure 5.1 shows the graphical representation of the parse tree for the model of PriorityQueue.

In the example, the leaves are char, int, and int. Any technique could be used to generate values for the simple types of SPECS-C++. The CVS uses the random number generation utility available in the programming language. The values generated are used in constructing values of more complex types.

Once values for the leaves have been generated, these values can be used to construct the values for the type in the next level of the structure. In this case, tuples will be generated. Tuples are constructed from the cross product of the values generated for its fields. For example, let the set of values generated
for the char field be \{ 'a', 'b' \}, and the sets of values for the two int fields be \{1, 2\} and \{3, 4\} then the set of tuple values generated is \{('a', 1, 3), ('a', 1, 4), ('a', 2, 3), ('a', 2, 4), ('b', 1, 3), ('b', 1, 4), ('b', 2, 3), ('b', 2, 4)\}. This technique is continued to the root of the parse tree. For sets the generated values are the power set of the set of values generated for its elements. Values of type sequence are the set of all permutations of the set of values generated for its elements.

It is possible to conceive a system which may generate an unlimited number of values. For practical purposes, the number of values generated must remain finite to ensure the tests complete in a finite amount of time. Bounds are placed on the value generation to achieve this goal. The bounds defined are breadth and depth. The values generated in model-based value generation may be viewed in tree form. Breadth is the term which reflects the number of children from each internal node in the tree, and depth reflects the number of levels in the tree. Consider the set of values generated for sequence of integer: \{ [], [3], [9], [3, 3], [3, 9], [9, 3], [9, 9] \}. These may be viewed in the tree shown in Figure 5.2. In this example breadth is 2, and depth is 2 (the root of the tree has depth 0). In other words, breadth defines the maximum number of values generated for a simple type and depth defines the maximum cardinality of a set or length of a sequence. Given the generation techniques and the bounds on the number of values to be generated, the following formulas define the largest number of values that may be generated for a given type with the given bounding parameters.

For the simple types int, real, and char the maximum cardinality of the generated set of values, $GS$, is $b$, where $b$ is the breadth parameter. The simple types bool and the user defined enumerated
types will have $\min(b, \mid \text{type} \mid)$. The formulas for the structured types follow. In the following formulas $d$ is the depth parameter, and $b$ is the breadth parameter.

As mentioned earlier, the set of values generated for type set if the power set of the $\mathcal{GS}$ for the elements. Equation 5.1 defines that the cardinality of the set of generated values if less than or equal to the cardinality of the power set of $\mathcal{GS}$. The set of values generated for type sequence is the set of all permutations with lengths ranging from zero to $d$ of the set of values generated for its elements. The definition of the maximum cardinality appears as Equation 5.2. Equation 5.3 defines the maximum cardinality of $\mathcal{GS}$ for tuples which is constructed from the cross product of values generated for the elements of the tuple. As described in Chapter 2, an alternatively defined type may have a value of one or more types. The size of the set of generated values is defined by summing the cardinalities of the sets of generated values for each type defined in the alternatively defined type. This is shown in Equation 5.4.

$$
|\mathcal{GS} \text{(set of } T, d, b)\mid \leq \sum_{i=0}^{d} \left( |\mathcal{GS}(T, d, b)\mid \right) 
$$

$$
|\mathcal{GS} \text{(sequence of } T, d, b)\mid \leq \sum_{i=0}^{d} |\mathcal{GS}(T, d, b)\mid^i = \frac{|\mathcal{GS}(T, d, b)\mid^{d+1} - 1}{|\mathcal{GS}(T, d, b)\mid - 1} 
$$

$$
|\mathcal{GS} \text{(tuple } T_1, T_2, \ldots, T_n, d, b)\mid \leq \prod_{i=1}^{n} |\mathcal{GS}(T_i, d, b)\mid 
$$
Table 5.1 Maximum values possible for set of tuple (char, int, int)

<table>
<thead>
<tr>
<th>depth</th>
<th>breadth</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>37</td>
<td>379</td>
<td>2,081</td>
<td>7,876</td>
<td>23,437</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>163</td>
<td>20,854</td>
<td>679,121</td>
<td>10,017,001</td>
<td>89,880,967</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>247</td>
<td>397,594</td>
<td>83,278,001</td>
<td>4,935,173,776</td>
<td>135,343,435,123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>256</td>
<td>3,505,699</td>
<td>5,130,659,561</td>
<td>1,260,850,151,401</td>
<td>107,154,718,161,058</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>256</td>
<td>16,628,809</td>
<td>184,144,458,889</td>
<td>21,081,650,591,065</td>
<td>152,260,906,744,688</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 Number of values generated for int

<table>
<thead>
<tr>
<th>depth</th>
<th>breadth</th>
<th>Experimental</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Maximum Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

\[ |G_S((\text{alternately defined } T_1, T_2, \ldots, T_n), d, b)| \leq \sum_{i=1}^{n} |G_S(T_i, d, b)| \quad \text{(5.4)} \]

Given these formulas the maximum number of values that may be generated for set of tuple (char, int, int) may be calculated, as shown in Table 5.1. Note that with even small numbers values for depth and breadth the number of values which can theoretically be generated is extremely large.

In practice, not all these values will satisfy the class constraint. All values which are generated must be filtered through the class constraint prior to use in testing to ensure valid class values.

Actual value generation is constrained by the limits of the memory and speed of the computer running the generation software, as well as the performance and memory management of the value generation software itself. Limits of the software and hardware have restricted the data included here to the values generated for int (see Table 5.2) and set of int (see Table 5.3). These values are generated from models with no class constraint. The data show that the actual number of values are close to the maximum number. The slightly lower number of actual values is attributed to duplicate values being generated.
Model-Based Value Generation

Model-based value generation is used in the Abstract Test System to generate abstract values to test the specification. Applying this technique to the Class Validation System would require the implementor of the class to implement an additional function in C++ which maps abstract values to implementation values, a reverse representation mapping function. As discussed earlier, the mapping from the abstract domain of values to the implementation domain of values is a one-to-many (or many-to-many) mapping, and capturing this mapping in a reverse representation mapping function is challenging. The Class Validation System is designed to validate an implementation with respect to the specification. Requiring an additional complicated implementation function is unnatural to the system. A different approach to generating implementation values was designed to satisfy the restriction that the Class Validation System require no implementation other than the implementation of the class to be validated. Method-based value generation is the technique used. Method-based value generation uses the methods of the class to build values, both abstract and concrete.

For some of the primitive structured types of SPECS-C++ the concept of size of a value is obvious. When comparing values of the same type, one typically knows that one set is bigger than another, but the concept of size does not readily transfer to some types, such as tuples. In this discussion, the size of a value is thought of only in terms of its top-most type. For example, if a class \( T \) were modeled as a set of sequences of integers, the size of a value of type \( T \) would be the number of sequences in the set. The size of each sequence, or the integers within each sequence, are irrelevant.\(^1\) Therefore, to define size, cardinality is used for sets, and length is used for sequences. All values of a simple type in SPECS-C++, such as integer or boolean, are defined to have the same size. All tuples have the same size in this partial order as well. The smallest value is the uninitialized value (of any type). The partial

\(^1\) Alternative definitions of size which considers all levels of a type may be developed and used here as well.
order defined by the size of values of a particular type forms a lattice of values for that type.

With a definition of size established, it is possible to define a non-negative operation as an operation that increases the size of value. The change in size will depend upon the operation and the actual parameters to the operation. The types of operations that should be considered non-negative operation are those that make an instance bigger, or at least not smaller, every time it is applied a value. For example, in the class \texttt{PriorityQueue} (see Chapter 2) non-negative operation would be \texttt{AddEntry}. \texttt{RemoveEntry} would not be a non-negative operation. It is important to note that since the smallest value defined in the lattice is the uninitialized value the constructors of the class are non-negative operations.

The operations that are contained in the set of non-negative operations fall into one of the following categories: those that modify the default parameter, those that produce a new object without modifying the default parameter, and those that modify the default parameter and produce a new object as well (a new object is one that is of the same type as the default parameter). If the result of an operation that modifies the default parameter is not of the type under current consideration, that result is ignored in this discussion. Non-negative operations include all the constructors (in the C++ sense), the assignment operator, as well as other member and friend functions which satisfy the definition of non-negative operations.

There are clues in the post-conditions of operations that would help to classify a non-negative operation. For example, in classes that use sets to model the objects, non-negative operations are the default constructor, the copy constructor, the assignment operator, (any other implicit constructors), and any operation with a result (or post-state of the default parameter) produced through the operation \texttt{\textbackslash union}. Those operations with post-states resulting from set difference, or intersection most likely are not non-negative. In terms of the current definition of the partial order on values, the only non-negative operations for models based on tuples are the class constructors.

The set of non-negative operations for a class, can be defined as the set of all operations \texttt{op} in a class \texttt{C}, such that the result of applying \texttt{op} to some value \texttt{v} (of class \texttt{C}) is bigger than, or the same as \texttt{v}. In other words, this means that non-negative operations result in values that are the same as the value to which they are applied, or at a higher level in the lattice than the value to which they are applied.

The application of a non-negative operation can be identified by two "nodes" in the lattice. It follows that each node in the lattice is built through applications of non-negative operations involving a subset of nodes that are smaller than it. Each series of operations is an operation application sequence.

\footnote{The term \texttt{operation} may refer to a function, or a function with actual arguments, depending on the context.}
sequence. An operation application is an operation and its actual arguments. Arguments are selected from the set of previously generated values of the appropriate type. For example, \texttt{AddEntry(1,7)} and \texttt{AddEntry(6,12)} are different operation applications. An operation application has no meaning without a value to which it is applied; the default parameter is obtained from the operation application sequence. An operation application sequence is a finite sequence of operation applications that when applied, in order, result in a particular value.

The value of an operation application sequence is obtained by applying the uninitialized value to the first operation application (this should be a constructor) in the sequence. The value of each operation application is used as the default parameter to the next operation application in the sequence. The value of the last operation application is the value of the whole sequence.

Each abstract value \(v\) corresponds to a set of operation application sequences. The value of each sequence in the set is \(v\). Each operation application sequence in the set corresponds to an implementation value. Because the Class Validation System focuses on abstract values the implementation result of any unique operation application sequence is considered to be a unique implementation value. It is possible to have unique operation application sequences produce the same implementation value, but without knowledge of the implementation the Class Validation System is unable to recognize this. Treating each implementation value as unique does not limit the Class Validation System because the duplication of values only causes redundancy in the test. The technique of considering implementation values as unique if they have unique operation application sequences ensures that truly unique values will be maintained in the collection, and that errors resulting from a particular operation application sequence are more likely to be discovered.

Capturing the information about which class operations could be used to build a value serves at least two purposes. The first is the ability to test operations and generate values simultaneously. The other advantage is to aid the Class Validation System in creating the implementation representations of each abstract value.

**Values in the Class Validation System**

The Class Validation System will use two techniques to generate implementation representations of abstract values. The first, used in the interactive mode of the system, requires that the user build values from scratch through the methods of the class. This will create one implementation representation for

\(^3\)Either model-based or method-based value generation may be used to generate values for actual arguments to operations.
each set of operations the user selects to build the abstract value.

The second technique is method-based value generation. Each operation application sequence represents a potentially unique implementation representation of that value. This method of generating implementation values is used in the automatic testing mode. Each operation application sequence for a value \( v \) was a step in building an implementation representation of \( v \). These implementation representations are used to test the implementations of the operations.

Method-based value generation was selected to be used in the Class Validation System for several reasons. Implementation values can be generated without knowledge of the implementation data structure. Each value generated has been created through the class interface, as a client would create the values. Error reporting and defect detection are simplified because each implementation value has a history of operation applications associated with it. That history can be used to back-trace to the source of the defect.

Values generated in the Class Validation System are held in a set of tuples, each of which maintains the abstract value, operation application sequence and implementation value. The classes which provide support for managing these values are CompoundValue and CompoundValueSet. A CompoundValue (see the class model in Figure 5.3) represents a single abstract value, and implementation values which represent it. Each concrete value is associated with a unique operation application sequence. This design assumes that operation applications produce deterministic results. An alternative design would need to be used to support non-deterministic methods. A CompoundValueSet (Figure 5.4) is a table of CompoundValues. The lookup key in this table is the abstract value of the CompoundValue. If a new value is added to the table with an abstract value which matches the abstract value of a CompoundValue which already exists in the table the two CompoundValues are merged into a single CompoundValue which replaces the existing one in the table. The abstract function Merge defines the result of merging two CompoundValues.
/* model */
** domains 
**  tuple (void* value_c StringArray history ) ConcreteInstanceType 
**  String AbstractInstanceType 
**  set of ConcreteInstanceType ConcreteSet 
**  
** data members 
**  AbstractInstanceType AbstractValue 
**  ConcreteSet ConcreteValues 
**  ConcreteSet IteratedValues 
**  
** abstract functions 
**  define IsThere(ConcreteSet s, ConcreteInstanceType elem) as 
**    boolean such that 
**    IsThere(s, elem) = \exists(ConcreteInstanceType e) [ 
**      (e \in s) /\ history(e) = history(elem) ] 
**  
**  define AddOne(ConcreteSet s, ConcreteInstanceType elem) as 
**    ConcreteSet such that 
**    ( IsThere(s, elem) => AddOne(s, elem) = s ) /\ 
**    (!IsThere(s, elem) => AddOne(s, elem) = s \union \{ elem \} ) 
**  
**  define Merge(ConcreteSet a, ConcreteSet b) as 
**    ConcreteSet such that 
**    ( b = \{} => Merge(a, b) = a ) /\ 
**    ( b != \{} => \exists(ConcreteInstanceType e) [ 
**      (e \in b) /\ Merge(a, b) = 
**      Merge(AddOne(a, e), (b - \{ e \} )) 
**    ] 
**  
**  define HistoryLength(ConcreteInstanceType elem) as int such that 
**    HistoryLength(elem) = length(history(elem)) 
**  
**  define LongestHistory(ConcreteSet s) as int such that 
**    \exists (ConcreteInstanceType e) [ (e \in s) /\ 
**    \forall(ConcreteInstanceType e2) [ (e2 \in s) => 
**      HistoryLength(e2) <= HistoryLength(e) ] /\ 
**    LongestHistory(s) = HistoryLength(e) ] 
**  
** constraints 
**  \forall(ConcreteInstanceType c) [ (c \in ConcreteValues) => 
**  RepMap(value_c(c)) = AbstractValue /\ 
**  \forall(ConcreteInstanceType c2) 
**  [ ((c2 \in ConcreteValues) /\ (c != c2)) => 
**  (history(c) != history(c2)) ] /\ 
**  IteratedValues \subset subset ConcreteValues 
**  
Figure 5.3 Model for CompoundValue
/* model */
** uses CompoundValue */
**
** data members */
**
** set of CompoundValues theSet */
**
** abstract functions */
**
define max(int x, int y) as int such that */
** (x >= y => max(x,y) = x) */
** (x < y => max(x,y) = y) */
**
define UniqueOnKey (CompoundValueSet s) as boolean such that */
** UniqueOnKey(s) = */
** \
** (e1 \in s) => */
** \forall(CompoundValue e2) [ (e2 \in s) /\ */
** (e1 != e2) => */
** (e1.AbstractValue != e2.AbstractValue)] ] */
**
define Contains(String abstractValue, CompoundValueSet s) */
** as boolean such that */
** Contains(abstractValue, s) = */
** \exists(CompoundValue e) [ e \in s /\ */
** abstractValue = (e.AbstractValue) ] */
**
define SumCardinality(set of CompoundValue s) as int such that */
** (s != { } => */
** \exists(CompoundValue cv) [ (cv \in s) /\ */
** SumCardinality(s) = |ConcreteValues(cv)| + */
** SumCardinality(s - { cv }) ] ) /\ */
** (s = { } => SumCardinality(s) = 0) */
**
define LongestHistory(set of CompoundValue s) as int such that */
** (s != { } => */
** \exists(CompoundValue cv) [ (cv \in s) /\ */
** LongestHistory(s) = max(LongestHistory(cv), */
** LongestHistory(s - { cv })) ] ) */
** (s = { } => LongestHistory(s) = 0) */
**
** constraints */
**
** UniqueOnKey(theSet) */

Figure 5.4 Model for CompoundValueSet
CHAPTER 6 RELATED WORK

As the need for high quality software in short periods of time increases, the amount of research related to automated software testing, formal specifications and the use of formal specifications in testing slowly increases. Until recently, supporters of formal specification have been primarily interested in formal proofs of correctness. The focus on formal proofs kept formal specifications from being integrated into practical testing techniques. This chapter attempts to address some of the relevant testing research which is beginning to integrate formal specification and validation.

Automated Testing of Classes

Programs developed using object-oriented technology have special needs that traditional testing methodologies do not necessarily satisfy. The research of Buy, Orso, and Pezzè [9] addresses the special issues related to testing object-oriented classes. Unlike the techniques used in the Class Validation System, and the other approaches discussed in this chapter, their technique does not rely on any formal specification of the class being tested. Given that there is no formal specification, no test oracles are generated. Validation of test cases must be done manually, or test oracles must be created through some other technique.

To identify state-based errors (errors which appear when a method is applied to an instance in a particular pre-state), Buy, et al. developed a technique which identifies pairs of statements, definition-use pairs (du-pairs), in which one statement defines an instance variable and the other uses it. The du-pairs are identified during Dataflow Analysis, the first phase of this technique. The second phase identifies all execution paths through each method and the inputs required for each path. The third phase, Automated Deduction, identifies those paths from the second phase which exercise the du-pairs found in the first phase. Each sequence is then guaranteed to use the instance variables of the du-pairs. Additional refinement using output from the sequences will further exercise the class. The authors identify some disadvantages, the most significant of which is the computational complexity of the three steps, of this technique, but these do not prevent it from being a usable approach to testing classes.
which are not formally specified. In the cases where only a partial set of du-pairs is generated because of unbounded loops or complex expressions additional information provided by the user on a case-by-case basis may allow testing to continue.

This technique is useful for testing classes which are not formally specified. The benefit supplied by formal specification which is most useful in testing is that the specification can serve as an oracle. Without an oracle all test results must be validated manually. Since the goal of the Class Validation System is to integrate formal specifications in to the development process, this technique does not meet the same goals as the CVS.

**Integrating Black-Box and White-Box Testing**

Advantages and disadvantages appear in both functional (black-box) testing and structural (white-box) testing. Chen, et al. [10] propose an approach to class-level testing which integrates both methods. Their technique for identifying two equivalent values is derived through distinct operation application sequences. The equivalence of these values is determined through normalization of axiomatic specifications. To describe the technique, assume two terms \( \{u_1, u_2\} \) are determined to be equivalent using the axiomatic specification. Each term \( u_1, u_2 \) represents a sequence of methods in a class, \( s_1, s_2 \). Applying those sequences of methods to objects \( O_1, O_2 \), results in objects \( O'_1 \) and \( O'_2 \). If \( O'_1 \) and \( O'_2 \) are observationally equivalent, then the test passes. Chen, et al. use white-box techniques to determine observational equivalence.

In this paper they prove that one cannot determine observational equivalence through pure black-box testing techniques. One cannot use the observable contexts of a class to determine equivalence, because the set of all observable contexts is infinite. They state, as part of their theorem, “there exist objects that are not observationally equivalent, but appear to be so when only a finite subset of the observational contexts are applied.” Using what is called a heuristic white-box technique they are able to determine a subset of observational contexts, a relevant observable context, to determine observational equivalence [10]. Based on this result, they developed the Relevant Observable Context Technique to determine the relevant observable context used to determined that two values are observably equivalent.

The Relevant Observable Context Technique depends upon the Data-member Relevance Graph (DRG) to determine the relevant observable context of any given data member. The graph is constructed with thick boxes to represent each implementation data member, and a thin box for each constant. These are the nodes. If a data member \( d_1 \) directly affects data member \( d_2 \) under condition \( p \) in method \( m \) then there is a arc between \( d_1 \) and \( d_2 \) labeled \( (p, m) \). Each arc with an observer method as the second
argument terminates in a special node called observed. The algorithm for comparing two instances of a class \( O_1 \) and \( O_2 \) can be summarized in the following steps.

1. Construct the DRG.

2. If all data members, \( d_1, \ldots, d_n \) of \( O_1 \) and \( O_2 \) are equal, then return.

3. For each data member \( d_j \) of \( O_1 \) and \( O_2 \) which is not equal
   
   (a) If there is no path from this data member, satisfying the conditions on the arcs, to the node observed, skip this data member as it is not part of the observable context by the client.
   
   (b) Traverse every acyclic path to obtain the relevant observable context. This is straightforward for simple types, compound types are broken down and each part is treated as an individual data member. Check if any relevant observable contexts fail (such that the data members \( d_j \) of \( O_1 \) and \( O_2 \) are not observable equivalent. Stop, otherwise this data member passes.

4. If all data members pass, then \( O_1 \) is observably equivalent to \( O_2 \) in this relevant observable context.

The disadvantage of this approach is that creating the DRG for this technique is much more work than writing the representation mapping and equivalent function. It requires intricate knowledge of the entire implementation, not just the implementation data model. Errors in the code may be translated into the graph and missed the this technique. In the Class Validation System errors in the representation mapping function, equivalence function or implementation are detected.

**Axiomatic Specifications**

Some of the recent research involving the testing formal specifications uses axiomatic specifications. Since axiomatic specifications are a different type of specification, this section provides an brief introduction to axiomatic specifications. An axiomatic specification of an ADT describes the operations on a class using a set of axioms [33]. These axioms specify the behavior of an operation in terms of the other operations of the class. Figure 6.1 is an example of an axiomatic specification for a simple class IntStack. An IntStack is a last in, first out (LIFO) data structure containing integers [36].

This specification has three parts: operation definitions, variables, and axioms. The operations on an IntStack are new, IsEmpty, Push, Pop, and Top. The operation new creates an IntStack. IsEmpty takes an IntStack and returns a boolean value. Push takes two parameters, an IntStack and an
IntStack

operations

new: \rightarrow \text{IntStack}
.IsEmpty(): \text{IntStack} \rightarrow \text{boolean}
.Push(\cdot): \text{IntStack int} \rightarrow \text{IntStack}
.Pop(): \text{IntStack} \rightarrow \text{IntStack}
.Top(): \text{IntStack} \rightarrow \text{int}

variables

S: \text{IntStack}
x: \text{int}

axioms

a_1: \text{new}.IsEmpty() = \text{true}
a_2: S.Push(x).IsEmpty() = \text{false}
a_3: \text{new}.Pop() = \text{undefined}
a_4: \text{new}.Top() = \text{undefined}
a_5: S.Push(x).Pop() = S
a_6: S.Push(x).Top() = x

Figure 6.1 Example axiomatic specification
integer, and returns a new IntStack. Pop takes an IntStack and returns another IntStack. Top takes an IntStack and returns an integer. The two variables used in the axioms are $S$ which represents any IntStack, and $x$ which represents any integer.

The six axioms define the behavior of an IntStack. Axioms $a_1$ and $a_2$ define the method IsEmpty. Only a new IntStack is empty; any stack which has had something pushed on it is not empty. Operations Pop and Top are not defined for a new IntStack as described by axioms $a_3$ and $a_4$. Axiom $a_5$ describes the LIFO behavior of the IntStack. If Pop is applied to any IntStack with a value pushed onto it, the stack without that value is the result (the last value put onto the stack is the first one removed). The behavior of Top is defined by $a_6$: Top returns the last value pushed onto the IntStack.

**Using Axiomatic Specifications**

A technique using axiomatic specifications was published by Özcan. This research also uses the programming language PARADOX PASCAL to prototype software systems from axiomatic specifications [30]. A compiler generates Pascal code from the specification. This methodology consists of three inter-related stages:

1. prototype design
2. prototype implementation
3. prototype evaluation

Özcan's research supports specifying the entire software system, and generates the prototype for it. The tester uses a set of scenarios to validate the system. Like the Abstract Test System, the methodology relies heavily on user feedback. This approach is well-suited to whole applications rather than components (which is the focus of the Class Validation System). This system allows swapping out prototyped components and replacing them with implementations to improve performance [30].

While this technique is useful for system integration, it does not satisfy the goals of the Class Validation System. It does not take advantage of the tested specification which could be used as a test oracle for the implementation. Individual components are not tested independently from the system, and the implementation is not validated against the specification. The Class Validation Tool approaches validation with a greater degree of granularity. Applying the focus to the component level, the Class Validation System provides testing to support reliable, reusable components for larger systems.
Automatically Testing Using Axiomatic Specifications

Antoy and Hamlet [2] propose a system which automatically validates an implementation against its axiomatic specification. A single test, in their system, begins with an implementation value. Using the representation mapping, supplied in the programming language of the implementation, to produce the corresponding abstract value. The executable version of the function specification is applied to the abstract value and the function implementation is applied to the implementation value. The representation mapping function is applied to the result of the implementation function. The resulting abstract value is compared to the abstract result.

Three classes are involved in the automated checking of the implementation: the by-hand implementation, the direct implementation, and the self-checking implementation. The by-hand implementation is the actual implementation of the class. The direct implementation is the executable version of the specification. The self-checking implementation joins the direct and by-hand implementations with some additional checking code, including the representation mapping function. The self-checking implementation contains the calls to the representation mapping function and to verify the abstract result equals the result of applying the representation mapping to the implementation result. Using the self-checking implementation in an application supplies the automated validation of the class.

The work published by Antoy and Hamlet is closely related to the work supporting the Class Validation System. Their technique uses an axiomatic formal specification of an ADT, and a representation mapping function written in the programming language of the implementation, rather than in the specification language, as in SPECS-C++. Like the CVS, the oracle is the executable version of the specification, which may by produced by a tool similar to the SPECS-C++ compiler.

The most obvious differences between this approach and the one used in the Class Validation System is the type of technique used to specify the ADT, and the representation mapping function. Additionally, the techniques vary in the processing of a single test. The approach used by Antoy and Hamlet uses the representation mapping function to translate the implementation values both in the pre-state, and in the post-state. The CVS uses values from the abstract pre-state and the implementation values previously associated with them. This gives the CVS the advantage of testing potentially multiple implementation value representations of the abstract value in a single test.

Another difference is the tool support. The CVS provides a testing environment in which to exercise the class specification and implementation. Antoy and Hamlet’s work does not provide such a tool, but does provide a self-checking class that can be used by any application code. The self-checking implementation automatically verifies the post-state of the implementation with the expected output.
as determined by the executable specification.

While useful, the self-checking implementation does not supply the same functionality as the Class Validation System. There is no support to test the specification, and there is no automated test framework. Both of these are supported by the CVS. By requiring the representation mapping as part of the specification, the CVS requires no additional implementation beyond the implementation of the class. The Abstract Test System and Class Validation System provide a consistent view of class instances: values from the abstract model. This is possible because of the model-based specification language used by the CVS. Techniques that rely on axiomatic specifications do not have a data model, and cannot supply this type of information to the tester, as the Class Validation System can.
CHAPTER 7 CONCLUSION

Producing quality software (reliable software which correctly performs its intended task) is the primary goal of software developers. Catching errors early in the development process reduces the cost of maintenance and increases the correctness of the software [1, 8, 24].

Traditional software testing has taken many forms, but all these forms strive to answer the basic question "Is the software right?". The question in itself is ambiguous. "Is it right?" can mean many things, such as does the software solve the problem it intended to solve, and did it solve it in a way that is consistent with a solution to the problem? (Did we build it right? Did we build the right thing?) This is the basic difference between verification and validation [33].

The software industry is changing. New demand for software technology has created an environment where the development cycle for software has been reduced. The traditional software development models (like the waterfall model) aren't going to stand up to the strain of short development cycles. The needs of the industry require new development models to keep up to demand without reducing the quality of software. The Class Validation System is a development tool which gets testing done quickly, automatically, and early. This will provide reliable, reusable components with which developers can build software systems.

The Class Validation System, which is the product of this research, provides support for a realistic development methodology. The requirement of a formal specification introduces additional rigor [24, 28] to the development process.

The Class Validation System has brought together formal software specifications and testing in a unique and practical manner. The ability to test an abstract specification against implicit requirements, and then use the validated specification as the oracle against which the implementation is tested is unique to this tool and constitutes the contribution of this research.

Reusable software components must be reliable to be useful, and through specifications and testing is needed in a software market when development time is shortened to speed up the time to market. Techniques like those developed to produce the Class Validation System are a sound way to achieve
this.

**Additional Enhancements**

Components of the Class Validation System use only one technique to generate values and select test case. Enhancements may be made to the system to allow for other techniques of generating abstract values in an Abstract Test Tool. Alternative approaches to value generation, and test case selection could greatly improve the effectiveness of the tool. Such approaches may select values from the edges of the domain, or use the specifications of the functions to determine the values which would best exercise the function. Modularization of the test case selection and value generation may allow the tester to select which technique to use.

Enhancements such as these do not change the basic effectiveness of the Class Validation System. Even with unsophisticated techniques for value generation and test case selection, it is an effective tool for testing classes and their specifications automatically.
APPENDIX A SPECIFICATIONS FOR SPECS-C++ CLASSES

The following specifications are for the classes which implement the SPECS-C++ primitive types, as described in Chapter 2. Class T is the meta-class which serves as a super class for all types. It also implements the simple types. The structured types are Sequence, Set, Tuple, Enumerated, and Alternate.

Specification for class T

```c++
#ifndef _T_H
#define _T_H

#include <iostream.h>
#include "String.h"

typedef void * PointerType;

#include "T.pre"

#ifdef _GARBAGE_COLLECT
#include "gc_cpp.h"
#endif

#ifdef _GARBAGE_COLLECT
class T :public gc {
#else
class T {
#endif
    /* domains
    ** ( int, float, char, string, bool,
    ** Enumerated, Set, Sequence, Tuple, Alternate, Unknown, Pointer ) Tnames
```
** int | float | char | string | bool | Enumerated
** Set | Sequence | Tuple | Alternate | Unknown | Pointer Ttype
**
** data members
** Tnames type
** Ttype val
**
** constraints
** val \\ of type Ttype
**
** abstract functions
** - Valid string representations of SPECS-C++ types are defined
** - by the grammar of SPECS-C++, and abstract functions to
** - re-define them are not included here.
**
** define IsValid(string str) as bool such that
** IsValid(str) =
** IsValidSet(str) \lor IsValidSeq(str) \lor IsValidTuple(str)
** \lor IsValidInt(str) \lor IsValidFloat(str) \lor IsValidChar(str)
** \lor IsValidStr(str) \lor IsValidBool(str) \lor IsValidEnum(str)
** \lor IsValidAlt(str)
**
** operations
*/

public:

//. Constructors -------------------

T();
/* modifies: self
** postA: type' = T.Int
*/

T(const Tnames& kind);
/* modifies: self
** postA: type' = kind
*/

T(double x);
/* modifies: self
 ** postA: type' = float \ val'= z
 */

T(bool x);
/* modifies: self
 ** postA: type' = bool \ val'= x
 */

T(char x);
/* modifies: self
 ** postA: type' = char \ val'= x
 */

T(int x);
/* modifies: self
 ** postA: type' = int \ val'= x
 */

T(const char* x);
/* modifies: self
 ** postA: type' = string \ val'= x
 */

T(const String& s);
/* modifies: self
 ** postA: type' = string \ val'= s
 */

T(PointerType x);
/* modifies: self
 ** postA: type' = Pointer \ val'= x
 */

T(const T& t); // this is "virtual"
/* modifies: self
 ** postA: self' = t
 */

// Assignment Operator ————
virtual T& operator = (const T& t);
    /* modifies: self
    ** postA: self' = t \ result = self'
    */

T& operator = (const double& x);
    /* modifies: self
    ** postA: type' = float \ val' = x
    */

T& operator = (const int& x);
    /* modifies: self
    ** postA: type' = int \ val' = x
    */

T& operator = (const PointerType& x);
    /* modifies: self
    ** postA: type' = Pointer \ val' = x
    */

T& operator = (const String& s);
    /* modifies: self
    ** postA: type' = string \ val' = s
    */

T& operator = (const char& x);
    /* modifies: self
    ** postA: type' = char \ val' = x
    */

T& operator = (const bool& x);
    /* modifies: self
    ** postA: type' = bool \ val' = x
    */

    // Destructor ------------------------------------------

virtual T();
    /* postA: trashed(self')
Observers

Names GetType() const;
/* postA: result = type */

virtual Names getKind() const;
/* this is used by derived classes that are "container" classes */

int GetIntVal() const;
/* preA: type = int */
** postA: result = val */

PointerType GetPointerVal() const;
/* preA: type = Pointer */
** postA: result = val */

String GetStringVal() const;
/* preA: type = string */
** postA: result = val */

double GetFloatVal() const;
/* preA: type = float */
** postA: result = val */

char GetCharVal() const;
/* preA: type = char */
** postA: result = val */

bool GetBooleanVal() const;
/* preA: type = bool */
** postA: result = val */
virtual bool operator == (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self == t2)
*/

virtual bool operator != (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self != t2)
*/

virtual bool operator <= (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self <= t2)
*/

virtual bool operator < (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self < t2)
*/

virtual bool operator >= (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self >= t2)
*/

virtual bool operator > (const T& t2) const;
/* preA: type = t2.type
** postA: result = (self > t2)
*/

// Misc. ----------------------------------

virtual StringOfType() const;
/* postA: result = string version of type information */
// Friends ---------------------

friend ostream & operator << (ostream & out, const T & t);
/* modifies: out
   ** postA: out' = str(t) \ result = out'
*/

friend T* FromString(String str);
/* preA: IsValid(str)
   ** postA: str = str(result)
*/

virtual String ToString() const;
/* postA: result = str(self)
*/

#include "T.pri"

}; // end of class T
#endif

Specification for class Sequence

#if !defined _SEQUENCE_H
#define _SEQUENCE_H

#include <iostream.h>
#include "String.h"
#include "T.h"
#include "Sequence.pre"

class Sequence : public T {
    
    /*
     ** uses class T
     **
     ** domains
     ** sequence of T SeqModel
     **
     ** data members
     */

#endif
** SeqModel theSeq
** Tnames seqOf
**
** constraints
** \forall (T t1) \forall (T t2) [ 
** (t1 elem self) \land (t2 \text{\textit{elem}} \text{\textit{self}}) \Rightarrow t1.type = t2.type \land 
** seqOf = t1.Type ]
** /
**/

public:

// Constructors
---
Sequence(Tnames tn = T.unknown);
// tn is the type of elements in the sequence
/* modifies: self */
** postA: theSeq' = [] \land seqOf' = tn */

Sequence(const Sequence& s);
/* modifies: self */
** postA: self' = s */

Sequence(const T& val);
/* modifies: self */
** postA: theSeq' = [val] \land seqOf' = val.type */

// Destructors
---
virtual ~Sequence();
/* modifies: self */
** postA: trashed(self); */

// Assignment operator
---
virtual Sequence& operator = (const Sequence& s2); 
/* preA: seqOf = Unknown \lor seqOf = s2.seqOf */
** modifies: self
** postA: self' = result \and result = s2
*/

Sequence& operator = (const T& s2);
/** preA: seqOf = Unknown \or seqOf = s2.seqOf
** modifies: self
** postA: self' = result \and result = s2
*/

void AssignToIndex(int where, const T& what);
/** preA: (seqOf = what.type \or seqOf = Unknown) \and
** ( 1 <= where \and where <= length(theSeq) + 1)
** where = length + 1 means tack it on the end
** modifies: self
** postA: ((where <= length(theSeq) =>
** \forall(int i) ( i <= i \and i <= length(theSeq) \and i != where
** => theSeq'[ij = theSeq[ij] \and
** theSeq'[where] = what) \and
** (where = length(theSeq) + 1) =>
** (theSeq' = theSeq | | [what]) ) \and
** (seqOf' = what.type)
*/

// Observers
virtual Tnames getKind() const;
/** postA: result = seqOf
*/

virtual T First() const;
/** preA: theSeq != []
** postA: result = first(theSeq)
*/

virtual T Last() const;
/** preA: theSeq != []
** postA: result = last(theSeq)
*/
virtual Sequence Header() const;
/* preA: theSeq !== []
** postA: result = header(theSeq)
*/

virtual Sequence Trailer() const;
/* preA: theSeq !== []
** postA: result = trailer(theSeq)
*/

virtual int Length() const;
/* postA: result = length(theSeq)
*/

virtual T& Index(int i) const ;
/* preA: 1 <= i <= length(theSeq)
** postA: result = theSeq[i]
*/

virtual T& operator[](int i) const ;
/* preA: 1 <= i <= length(theSeq)
** postA: result = theSeq[i]
*/

virtual bool operator == (const Sequence& s2) const;
/* postA: result = (self == s2)
*/

virtual bool operator == (const T& t2) const;
/* postA: result = (self == t2)
*/

virtual bool IsElemOf(const T& val) const ;
/* preA: \exists(T t) [ t \elem theSeq \& t.type = val.type ]
** postA: \exists (int i) [ 1 <= i <= length(theSeq) \& theSeq[i] = val ]
*/

virtual String OfType() const;
/* postA: result = string version of type information
*/
virtual Sequence Concatenate(const Sequence& s2) const;
/* preA: \forall(T t1) [ \forall(T t2) [ (t1 \elem s1 \land \\
** t2 \elem s2) \Rightarrow t1.type = t2.type ]] 
** postA: result = s1.theSeq || s2.theSeq */

virtual Sequence operator || (const Sequence& s2) const;
/* preA: \forall(T t1) [ \forall(T t2) [ (t1 \elem s1 \land \\
** t2 \elem s2) \Rightarrow t1.type = t2.type ]] 
** postA: result = s1.theSeq || s2.theSeq */

friend ostream& operator << (ostream& out, const Sequence& s);
/* modifies: out 
** postA: result = out' \land out' = out || str(s) */

virtual String ToString() const;
/* postA: result = str(s) */

#include "Sequence.pri"

}: // end of class Sequence

#endif

Specifications for class Set

#if !defined .SET_H
#define .SET_H

#include <iostream.h>
# include "T.h"

# include "Set.pre"

class Set : public T {

/*
 ** domains
 ** set of T SetModel
 **
 ** data members
 ** SetModel theSet
 ** Tnames setOf
 ** T currentItem
 **
 ** constraints
 ** \forall (T t1) \ (t1 \in theSet \land \ \land (theSet \neq \emptyset \land \exists (T t1) / t1 = currentItem) )
 ** operations
 */

public:

Set(Tnames tn = T.unknown); // tn is the type of the set elements
/* modifies: self
 ** postA: theSet' = \emptyset \land setOf' = tn
 */

Set(const Set& s); // copy constructor
/* modifies: self
 ** postA: theSet' = s.theSet
 */

Set(const T& t);
/* modifies: self
 ** postA: theSet' = \{ t \} \land setOf' = t.type
 */

virtual ~Set();
/* modifies: self
** postA: trashed(self);
 */

Set& operator = (const Set& s);
/* modifies: self
** postA: theSet' = s.theSet \ result = self'
 */

Set& operator = (const T& s);
/* modifies: self
** postA: theSet' = s.theSet \ result = self'
 */

Tnames getKind() const ;
/* postA: result = s.setKind
 */

Set Union(const Set& s2) const ;
/* preA: \forall (T t1) [ \forall (T t2) [ (t1 \ elem theSet) \ /
** (t2 \ elem s2.theSet) \Rightarrow t1.type = t2.type ]]
** postA: result.theSet = theSet \ union s2.theSet
 */

Set Intersect(const Set& s2) const ;
/* preA: \forall (T t1) [ \forall (T t2) [ (t1 \ elem theSet) \ /
** (t2 \ elem s2.theSet) \Rightarrow t1.type = t2.type ]]
** postA: result.theSet = theSet \ intersect s2.theSet
 */

Set operator - (const Set& s2) const ;
/* preA: \forall (T t1) [ \forall (T t2) [ (t1 \ elem theSet) \ /
** (t2 \ elem s2.theSet) \Rightarrow t1.type = t2.type ]]
** postA: result.theSet = theSet - s2.theSet
 */

bool IsElcm(const T& elem) const ;
/* preA: \ exists(T t1) [ (t1 \ elem theSet) \ /
** (t1.type = elem.type) ]
** postA: result = elem \ elem theSet
 */
void Insert(const T& newItem);
/* preA: (theSet > 0) => \exists(t1 \in theSet) /
** (t1.type = newItem.type)
** modifies: self
** postA: theSet' = theSet \union {newItem}
*/

void Remove(const T& elem);
/* modifies: self
** postA: theSet' = theSet - {elem}
*/

bool IsSubset(const Set& s2) const;
/* preA: \forall (T tl) \forall (T t2) (tl \in theSet) /
** (t2 \in s2.theSet) => tl.type = t2.type
** postA: result = theSet \subset s2.theSet
*/

int Cardinality() const;
/* postA: result = | theSet |
*/

friend ostream& operator << (ostream& out, const Set& s);
/* modifies: out
** postA: result = out' \and out' = out \and s = str(s)
*/

virtual bool operator == (const Set& s2) const;
/* postA: result = (self = s2)
*/

virtual bool operator == (const T& s2) const;
/* preA: s2.type = T.set
** postA: result = (self = s2)
*/

// Misc.
virtual String OfType() const;
/* postA: result = string version of type information */

virtual String ToString() const;
/* postA: result = str(s) */

// TB specified later
void BeginIteration();
bool EndOfIteration();
T GetNext();
T& GetNextReference();

#include "Set.pri"

}; // end of class Set
#endif

---

**Specification for class Tuple**

#include "T.h"
#include "String.h"
#include "Tuple.pre"

class Tuple:public T {

/*
 ** uses class T
 **
 ** domains
 ** tuple ( string compName
 **    T    compVal ) tupComponent
 ** sequence of tupComponent TupModel
 ** - The following are modeled, but need to be implemented by the client.
** sequence of string  List<String>
** sequence of Tnames  List<Tnames>
** sequence of T  List<T>

**

** data members
** TupModel theTuple
** integer numComponents
** boolean isDefault
**

** constraints
** (\forall (int i) \forall (int j) \cdot (1 <= i < j <= length(theTuple))
** => compName(theTuple[ij]) = compName(theTuple[jj]))
** \land (length(theTuple) = numComponents)
** \land (numComponents > 0)

** operations
*/

public:
Tuple();
/* modifies: self
** postA: isDefault \land theTuple = []
*/

Tuple(NameList names, TypcList types, ValList vals);
/* preA: length(names) = length(vals) \land length(names) = length(types) \land
** length(names) > 0 \land
** \forall (int i) \forall (int j) \cdot (1 <= i < j <= length(names))
** => names[ij] != names[jj] \land
** \forall (int i) \cdot I <= i <= length(types) =>
** types[i] = vals[i].type
** modifies: self
** postA: numComponents' = length(names) \land
** \forall (int i) \cdot (1 <= i <= length(names)) =>
** compName(theTuple'[i]) = names[i] \land
** compVal(theTuple'[i]).type = vals[i].type \land
** compVal(theTuple'[i]).val = vals[i].val \land isDefault'
*/
Tuple(TypeList types, ValList vals):
/* preA: length(types) = length(vals) \land length(vals) > 0 */
   \forall (int i) \ [ 1 <= i <= length(types) ]
   types[i] = vals[i].type
** modifies: self
** postA: numComponents' = length(names) \land
   \forall (int i) \ [ (1 <= i <= length(names))] 
   compName(theTuple'[ij]) = "" \land
   compVal(theTuple'[ij]).type = vals[i].type \land
   compVal(theTuple'[ij]).val = vals[i].val \land isDefault'
*/

Tuple(const Tuple& t):
/* modifies: self
** postA: self' = t \land isDefault'
*/

Tuple();
/* modifies: self
** postA: trashed(self)
*/

Tuple& operator = (const Tuple& t);
/* modifies: self
** postA: self' = t \land result = self'
*/

Tuple& operator = (const T& t);
/* modifies: self
** postA: self' = t \land result = self'
*/

void Set(String cname, const T& val);
/* preA: \exists (int i) \ [ 1 <= i <= length(theTuple) ] \land
   \forall (int i) \ [ 1 <= i <= length(theTuple) ]
   compName(theTuple[i]) = cname \land
   compVal(theTuple[i]).type = val.type /
** modifies: self
** postA: \exists (int i) \ [ 1 <= i <= length(theTuple) ] \land
   compName(theTuple[i]) = cname \land
   compVal(theTuple[i]) = val /
*/
void Set(int pos, const T& val);
/* preA: pos <= length(theTuple)
** modifies: self
** postA: compVal(theTuple[pos]) = val */

T* Select(String cname) const;
/* preA: \exists(i) 1 <= i <= length (theTuple) /\ 
** compName(theTuple[i]) = cname/ 
** postA: \exists(i) 1 <= i <= length (theTuple) /\ 
** compName(theTuple[i]) = cname /\ 
** result = compVal(theTuple[i]) */

T* Select(int pos) const;
/* preA: pos <= length(theTuple)
** postA: result = compVal(theTuple[pos]) */

bool IsDefault() const;
/* postA: result = isDefault */

void AddComponent(Tnames type, String name = "", T* val = NULL);
/* preA: isDefault
** modifies: self
** postA: theTuple' = theTuple || (name, ('val)) /\ 
** numComponents' = numComponents + 1 */

void MakeNotDefault();
/* modifies: self
** postA: isDefault' = false */

virtual bool operator == (const TupleSt t2) const;
/* postA: result = (self = t2) */
virtual bool operator == (const Tuple & t2) const;

friend ostream& operator << (ostream& out, const Tuple & t);

#include "Tuple.pri"

} ; // end of class Tuple

#include "Enumerated.h"

/* $Header: Enumerated.h,v 1.31 97/07/17 10:28:40 gurski Exp $ */

#include "T.h"
#include "String.h"
#include <iostream.h>

static char Enumerated_ident[] = "$Header: Enumerated.h,v 1.31 97/07/17 10:28:40 gurski Exp $";

#include "T.h"
#include "String.h"
#include <iostream.h>
class Enumerated : public T {

/*
 ** model
 ** data members
 ** sequence of string possible Vals
 ** string current Val
 **
 ** constraints
 ** current Val \in possible Vals
 */

/* operations */

public:

Enumerated (StringList vals);
/* preA: length(vals) >= l
 ** modifies: self
 ** postA: possible Vals' = vals.theSeq
 ** \land current Val' = first(possible Vals') */

Enumerated (const Enumerated& e);
/* modifies: self
 ** postA: possible Vals' = e.possible Vals \land
 ** current Val' = e.current Val */

Enumerated (const String& s);
/* modifies: self
 ** postA: current Val' = s \land possible Vals = \{ s \} */

~Enumerated();
/* modifies: self */
** postA: trashed(self') */

Enumerated& operator = (const Enumerated& e);
/* preA: possibleVals = e.possibleVals */
** modifies: self */
** postA: currentVal' = e.currentVal /
** resul = self' */

void SetVal(String val);
/* preA: val \in possibleVals */
** modifies: self */
** postA: currentVal' = val */

String GetVal() const;
/* postA: result = currentVal */
*/

bool IsVal(String val) const;
/* postA: result = val \in possibleVals */
*/

Enumerated First() const;
/* postA: result.possibleVals = possibleVals /
** result.current = first(possibleVals) /
** result.numVals = numVals */

Enumerated Last() const;
/* postA: result.possibleVals = possibleVals /
** result.current = last(possibleVals) /
** result.numVals = numVals */

Enumerated operator ++() ; // pre-increment
/* preA: currentVal < last(possibleVals) */
** modifies: self
** postA: \exists (integer i) [ i <= i <= length(possible Vals) \wedge
  ** current Val = possible Vals[i] \wedge
  ** current Val' = possible Vals[i] + 1] \wedge
  ** result = self'
*/

Enumerated operator ++(int) : // post-increment
/* preA: current Val < last(possible Vals)
** modifies: self
** postA: \exists (integer i) [ i <= i <= length(possible Vals) \wedge
  ** current Val = possible Vals[i] \wedge
  ** current Val' = possible Vals[i] + 1] \wedge
  ** result = self'
*/

Enumerated operator --() : // pre-decrement
/* preA: current Val > 1
** modifies: self
** postA: \exists (integer i) [ i <= i <= length(possible Vals) \wedge
  ** current Val = possible Vals[i] \wedge
  ** current Val' = possible Vals[i] - 1] \wedge
  ** result = self'
*/

Enumerated operator --(int) : // post-decrement
/* preA: current Val > 1
** modifies: self
** postA: \exists (integer i) [ i <= i <= length(possible Vals) \wedge
  ** current Val = possible Vals[i] \wedge
  ** current Val' = possible Vals[i] - 1] \wedge
  ** result = self'
*/

/* - overload ==, !=
*/

bool operator == (const Enumerated& e2) const;
/* postA: result = (self = e2)
*/
bool operator == (const T& e2) const;
/* postA: result = (self == e2)
 */

bool operator != (const Enumerated& e2) const;
/* postA: result = (self != e2)
 */

/* - overload < , <=, >, >= */

bool operator < (const Enumerated& e2) const;
/* preA: possibleVals = e2.possibleVals
 ** postA: \exists(integer i) [ 1 <= i <= length(possibleVals) \land 
 ** \exists (integer j) [ 1 <= j <= length(e2.possibleVals) \land 
 ** currentVal = possibleVals[i] \land 
 ** e2.currentVal = e2.possibleVals[j] \land 
 ** result = (i < j) ]
 */

bool operator <= (const Enumerated& e2) const;
/* preA: possibleVals = e2.possibleVals
 ** postA: \exists(integer i) [ 1 <= i <= length(possibleVals) \land 
 ** \exists (integer j) [ 1 <= j <= length(e2.possibleVals) \land 
 ** currentVal = possibleVals[i] \land 
 ** e2.currentVal = e2.possibleVals[j] \land 
 ** result = (i <= j) ]
 */
bool operator <= (const T& e2) const;
/* preA: possibleVals = e2.possibleVals
** postA: \exists(integer i) \[ 1 <= i <= length(possibleVals) \]
** \exists (integer j) \[ 1 <= j <= length(e2.possibleVals) \]
** currentVal = possibleVals[i] \land
** e2.currentVal = e2.possibleVals[j] \land
** result = (i <= j) */

bool operator > (const Enumerated& e2) const;
/* preA: possibleVals = e2.possibleVals
** postA: \exists(integer i) \[ 1 <= i <= length(possibleVals) \]
** \exists (integer j) \[ 1 <= j <= length(e2.possibleVals) \]
** currentVal = possibleVals[i] \land
** e2.currentVal = e2.possibleVals[j] \land
** result = (i > j) */

bool operator > (const T& e2) const;
/* preA: possibleVals = e2.possibleVals
** postA: \exists(integer i) \[ 1 <= i <= length(possibleVals) \]
** \exists (integer j) \[ 1 <= j <= length(e2.possibleVals) \]
** currentVal = possibleVals[i] \land
** e2.currentVal = e2.possibleVals[j] \land
** result = (i > j) */

bool operator >= (const Enumerated& e2) const;
/* preA: possibleVals = e2.possibleVals
** postA: \exists(integer i) \[ 1 <= i <= length(possibleVals) \]
** \exists (integer j) \[ 1 <= j <= length(e2.possibleVals) \]
** currentVal = possibleVals[i] \land
** e2.currentVal = e2.possibleVals[j] \land
** result = (i >= j) */

bool operator >= (const T& e2) const;
/* preA: possibleVals = e2.possibleVals
** postA: \exists(integer i) \[ 1 <= i <= length(possibleVals) \]
** \exists (integer j) \[ 1 <= j <= length(e2.possibleVals) \]
** currentVal = possibleVals[i] \land
** e2.currentVal = e2.possibleVals[j] \land
** result = (i >= j) */
\exists (integer j) | 1 <= j <= length(e2.possibleVals) /
**
**
currentVal = possibleVals[j] /
**
e2.currentVal = e2.possibleVals[j] /
**
result = (i >= j)]]
*/

// Misc. --------------

virtual String OfType() const;
/* postA: result = string version of type information
*/

virtual String ToString() const;
/* postA: result = str(e)
*/

friend ostream& operator << (ostream& out, const Enumerated& e);
/* modifies: out
** postA: result = out' /\ out' = out || str(e)
*/

#include "Enumerated.pri"

} ; // end class Enumeration

#endif

Specification for class Alternative

-----------------------------


#ifndef _ALTERNATETYPE
#define _ALTERNATETYPE

static char Alternate_ident[] = "$Header: Alternate.h,v 1.12 97/07/17 10:28:41 gurski Exp $";

#include "T.h"
#include <iostream.h>
#include "Alternate.pre"

class Alternative : public T {

  /*
   ** domains
   ** (Tnames type
   **  T  val)    union
   ** sequence of Tnames TypeList
   **
   ** data members
   ** set of Tnames possible
   ** union   current
   **
   ** constraints
   ** type(current) \in possible /
   ** val(current) \oftype type(current)
   **
   ** abstract functions
   ** define SeqToSet(sequence of Tnames seq) as set of Tnames such that
   ** (seq = [ ] => SeqToSet(seq) = { }) /
   ** (seq != [ ] => SeqToSeq(seq) = [first(seq)] \union
   **                 SeqToSet(trailer(seq))
   **
   ** operations
   */

public:

  Alternative();
  /* modifies: self
   ** postA: possible' = {} */

  Alternative (TypeList types, const T& curVal);
  /* preA: \exists(Tnames tn)[curVal \oftype tn \\in types]*/
  /* modifies: self
   ** postA: possible' = SeqToSet(types) /
   ** \exists(Tnames tn)[tn \in possible' \\and
   ** curVal \oftype tn */

** type(current') = tn \land

** val(current') = curVal /\n
*/

Alternative (const T& t);

/* modifies: self

** postA: possible' = { type(t) } \land

** type(current') = type(t) \land

** val(current') = val(t)

*/

Alternative(const Alternative& a);

/* modifies: self

** postA: self' = a

*/

virtual Alternative();

/* modifies: self

** postA: trashed(self)

*/

Alternative& operator = (const Alternative& a);

/* preA: type(val(a.current)) \in possible

** modifies: current

** postA: result = self' \land val(current') = val(a.current) \land

** type(current') = type(a.current)

*/

Alternative& operator = (const T& t);

/* preA: type(t) \in possible

** modifies: current

** postA: result = self' \land val(current') = val(t)

*/

virtual bool operator == (const Alternative& a) const;

/* postA: result = (val(a.current) = val(current))

*/

virtual bool operator == (const T& t) const;

/* postA: result = (val(t) = val(current))
StringOfType() const:
   /* postA: string of the current type */

Tnames GetType() const:
   /* postA: result = type(current) */

void SetVal(const T& v):
   /* preA: \exists (Tnames tn)[ v \ ofType tn /\ tn \in \possible ]
   ** modifies: current
   ** postA: \text{val}(current') = v /\ 
            \text{\exists (Tnames tn)} [\text{type}(current') = tn /\ 
            ** 
            v \ ofType tn] */

T* GetVal() const:
   /* postA: result = 	ext{val}(current) */

friend ostream& operator << (ostream& out, const Alternative& a):
   /* modifies: out
   ** postA: result = out' /\ out' = out || \text{str}(	ext{val}(a.current)) */

virtual String ToString() const:
   /* postA: result = \text{str}(	ext{val}(a.current)) */

#include "Alternative.pri"

}; // end class Alternative

#endif
APPENDIX B  CLASS PRIORITY QUEUE

This appendix contains the specification and implementation for the class PriorityQueue used as an example throughout this paper.

pqueue.pre

// pqueue.pre

typedef int PriorityType;
typedef char ElemType;

struct Entry {
    ElemType element;
    PriorityType priority;
    Entry *next;
};
typedef Entry* EntryPointer;

#include "Pqueue_A.h" // sc++ -repmap: 08/25/98 10:09 PM

pqueue.h

// PQueue.h

#include <iostream.h>
#if !defined _P.QUEUE_H
#define _P.QUEUE_H
#include "pqueue.pre"
class PriorityQueue {
    /* model */
    ** domains
    ** char ElemType
    ** int PriorityType

    ** methods

    /* public */
    PriorityQueue();
    ~PriorityQueue();
    int add(ElemType, PriorityType);
    int removeMin();
    int size();
    bool isEmpty();

    ** private */
};
** int TimeType
**
** tuple (ElemType data
**   PriorityType priLevel
**   TimeType entryTime) EntryType
**
** set of EntryType QueueType
**
** data members
**
** QueueType thePQ
**
** abstract functions
**
** define UniqueTimes(QueueType q) as boolean such that
**
** UniqueTimes(q) =
**
** \( \forall (\text{EntryType } el) \ (el \in q) =>
**
** \( \forall (\text{EntryType } e2) \ (e2 \in q) =>
**
** ((el = e2) \lor entryTime(el) \neq entryTime(e2)) \)
**
**
** define HighestEntry(QueueType q) as EntryType such that
**
** \( \exists (\text{EntryType } e) \ (e \in q) \land
**
** \( \forall (\text{EntryType } el) \ (el \in q) => (e = el) \lor
**
** ((priLevel(e) > priLevel(el)) \lor
**
** ((priLevel(e) = priLevel(el)) \land
**
** (entryTime(e) < entryTime(el)))) \)
**
** \( \land \text{HighestEntry}(q) = e \)
**
**
** define LatestTime(QueueType q) as TimeType such that
**
** \( q = {} => \text{LatestTime}(q) = 0 \land
**
** \( q \neq {} => \]
**
** \( \exists (\text{EntryType } e) \ (e \in q) \land
**
** \( \forall (\text{EntryType } el) \ (el \in q) => (e = el) \lor
**
** (entryTime(e) > entryTime(el)) \land
**
** \( \text{LatestTime}(q) = entryTime(e) \) \))))
**
**
** constraints
**
** UniqueTimes(thePQ) \land
**
** \( \forall (\text{EntryType } e) \ (e \in thePQ) => priLevel(e) \geq 0 \) \land
**
** \( \forall (\text{EntryType } e) \ (e \in thePQ) => entryTime(e) \geq 0 \)
**
**
** operations
*/

public:
PriorityQueue();
/* modifies: thePQ
** postA: thePQ' = {}
*/

PriorityQueue(const PriorityQueue& pq);
/* modifies: thePQ
** postA: thePQ' = (pq.thePQ)
*/

"PriorityQueue();
/* modifies: thePQ
** postA: trashed(thePQ)
*/

PriorityQueue& operator = (const PriorityQueue& pq);
/* modifies: thePQ
** postA: thePQ' = (pq.thePQ) ∧ result = self'
*/

void AddEntry(const ElemType& e, const PriorityType& level);
/* modifies: thePQ
** postA: thePQ' = thePQ \ union \{ (e, level, LatestTime(thePQ) + 1) \}
*/

ElemType FirstEntry() const;
/* preA: thePQ != \{ \}
** postA: result = data(HighestEntry(thePQ))
*/

PriorityType HighestPriority() const;
/* preA: thePQ != \{ \}
** postA: result = priLevel(HighestEntry(thePQ))
*/

void RemoveEntry();
/* preA: thePQ != \{ \}
** modifies: thePQ
** postA: thePQ' = thePQ - { HighestEntry(thePQ) }
bool IsEmpty() const;
    // postA: result = (thePQ = { })
    */
#endif

#include "pqueue.pri"
} ; // end template <class ElemType, class PriorityType> class PriorityQueue

pqueue.pri

// pqueue.pri -----

/** data members */

private:
Entry* head;

/** abstract functions
**
** define EquivSets(QueueType q1, QueueType q2) as boolean such that
** ((q1 = {}) V (q2 = {})) => EquivSets(q1, q2) = (q1 = q2) \&
** ((q1 != {}) \& (q2 != {})) =>
** EquivSets(q1, q2) =
** ((priLevel(HighestEntry(q1)) = priLevel(HighestEntry(q2))) \&
** (data(HighestEntry(q1)) = data(HighestEntry(q2))) \&
** EquivSets(q1 - {HighestEntry(q1)}, (q2 - {HighestEntry(q2)})))
**
** define Equivalent (PriorityQueue_A pql, PriorityQueue_A pq2) as boolean
** such that
** Equivalent(pql, pq2) = EquivSets(pql.thePQ, pq2.thePQ)
**
** define Repmap.QueueType(EntryPointer p, int t) as QueueType such that
** (p = NULL => Repmap.QueueType(p, t) = { }) \&
** (p != NULL => \exists (EntryType e)
** [ (e = (p->element, p->priority, t)) \&
** (Repmap.QueueType(p, t) = ((e) \union
** Repmap.QueueType (p->next, (t+1))))])
** define Repmap-PriorityQueue(PriorityQueue pq.c) as PriorityQueue.A
** such that
** Repmap-PriorityQueue(pq.c).thePQ = Repmap.QueueType(pq.c.head, t)
*/

friend class Repmap.pqueue; // sc++ -rempaq: 03/30/95 10:04 PM

pqueue.C

// pqueue.C ----------------------

#include "pqueue.h"

// ==============================================================
// Helper Functions
// ==============================================================

Entry* makeNode(ElemType elem,  
                PriorityType level,  
                Entry* n)  
{  
    Entry* node = new Entry;  
    node->element = elem;  
    node->priority = level;  
    node->next = n;  
    return node;  
} // end makeNode

// ==============================================================

Entry* copyList(Entry* original) {
    Entry *copy = NULL;
    Entry *origP = original,
    *copyP = NULL;

    while (origP != NULL) {
        if (copyP == NULL) {
            copyP = makeNode(origP->element, origP->priority, NULL);
            copy = copyP;
        }
        // ...
} else {
    copyP->next = makeNode(origP->element, origP->priority, NULL);
    copyP = copyP->next;
}
origP = origP->next;
} // end while

return copy;
} // end copyList

// ==

// Public Member Functions

PriorityQueue::PriorityQueue() {
    head = NULL;
} // end PriorityQueue()

//

PriorityQueue::PriorityQueue(const PriorityQueue& pq) {
    head = copyList(pq.head);
} // end PriorityQueue(const PriorityQueue& pq)

//

PriorityQueue::~PriorityQueue() {
    Entry *deleteThis;
    Entry *current;
    current = head;
    head = NULL;
    while (current != NULL) {
        deleteThis = current;
        current = current->next;
        delete deleteThis;
    } // end while
} // end ~PriorityQueue

//
PriorityQueue& PriorityQueue::operator = (const PriorityQueue& pq) {
if (head != pq.head) {
    head = copyList(pq.head);
}
return(*this);
} // end operator = (const PriorityQueue& pq)

void PriorityQueue::AddEntry(const ElemType& e, const PriorityType& level) {
#define .DEBUG cout << "AddEntry: pre state Adding: (" << e << ", " << level << ")\n"
    cout << "Head = " << head << endl;
    for (current = head; current != NULL; current = current->next) {
        cout << "\tNode Addr: " << current << "\tElem: " << current->element << "\tPriLevel: " << current->priority << "\tNext: " << current->next << endl;  
    } // end for
#define .DEBUG if (((head == NULL) || (head->priority < level)) {  
    head = makeNode(e, level, head); // this line fixed because of CVS
} else {
    Entry* current, *follower;
    current = head;
    while ((current != NULL) && (current->priority >= level)) {
        follower = current;
        current = current->next;
    } // end while
    follower->next = makeNode(e, level, current);
#undef .DEBUG
#undef .DEBUG
}
#undef .DEBUG
#define .DEBUG cout << "AddEntry: post state\n"
    cout << "Head = " << head << endl;
    for (current = head; current != NULL; current = current->next) {
        cout << "\tNode Addr: " << current << "\tElem: " << current->element << "\tPriLevel: " << current->priority
    } // end for
#undef .DEBUG
#undef .DEBUG
"\tNext: " << current->next << endl << flush;
} // end for
#endif
} // end AddEntry(const ElemType& e, const PriorityType& level)

// -----------------------------------

ElemType PriorityQueue::FirstEntry() const {
  return (head->element);
} // end FirstEntry() const

// -----------------------------------

PriorityType PriorityQueue::HighestPriority() const {
  return (head->priority);
} // end HighestPriority() const

// -----------------------------------

void PriorityQueue::RemoveEntry() {
  Entry* old = head;
  head = head->next;
  delete old;
} // end RemoveEntry()

// -----------------------------------

bool PriorityQueue::IsEmpty() const {
  return (head == NULL);
} // end IsEmpty() const

// -----------------------------------
APPENDIX C  SPECS-C++ VALIDATION TOOLS

The first section contains the interfaces implemented by the generated code for an ATT and a CVT. The second section contains the classes used to return values in string form to the user interface from the tool engine.

Engine Interfaces

Abstract Engine Interface

```c
ifndef DO.IT.H
#define DO.IT.H

#include "String.h"
#include "Array.h"
#include "ReturnVa.h"

extern String className;

ReturnVals *Execute(const String& defaultParam,
const StringArray& actualParams,
const String&   memberFuncName);
/* This procedure will execute the member function identified by
** memberFuncName on the defaultParam, with formal arguments
** replaced by actualParams. Results of the execution will be
** returned the result structure.
** If an error occurs in the execution of the function then all
** values in the result will be empty strings.
*/

StringArray FilterConstraintSatisfying(const StringArray& s);
/* This function takes an array of strings which represent values
** which are instances of the class being testing. The values in
```
**s which satisfy the constraint of the class are returned in the
** result.
 */

```
#endif

Implementation Engine Interface
```

```
// $Header: DoItImpl.h,v 1.5 99/04/18 14:45:53 gurski Exp $

static char id_DolImpl[] = "0(0)";

 ifndef DO.IT.IMPL.H
 define DO.IT.IMPL.H

 #include "String.h"
 #include "Array.h"
 #include "ReturnValsConcrete.h"

 extern String classNamelmpl;
 extern String typeInfoFilename;

 ReturnValsConcrete *ExecuteImpl(const void* defaultParam,
 const ImplArray& actualParams,
 const String& memberFuncName);

 /* This procedure will execute the member function identified by
 ** memberFuncName on the defaultParam, with formal arguments
 ** replaced by actualParams. Results of the execution will be
 ** returned the result structure.
 ** If an error occurs in the execution of the function then all
 ** values in the result will be empty strings.
 */

 ImplArray FilterConstraintSatisfyingImpl(const ImplArray& vals);
 /* This function takes an array of strings which represent values
 ** which are instances of the class being testing. The values in
 ** s which satisfy the constraint of the class are returned in the
 ** result.
 */
```
bool PassRepmap(const String& abstractValue,
    const void* concreteValue);
/* postA: result = FromString(abstractValue) == Repmap(concreteValue)
** - This is for the appropriate class */

String GetRepMapValue(const void* concreteValue);
/* postA: result = toString(Repmap(concreteValue))
** - This is for the appropriate class */
#endif

---

**Return Value Containers**

**ReturnVals.h**

```cpp
#ifndef RETURN_VALS_H
#define RETURN_VALS_H

#include "String.h"
#include "Array.h"
#include <iostream.h>

typedef Array<class String> StringArray;

class ReturnVals {
    /* model
    ** data members
    ** string deflt
    ** sequence of string parameters
    ** string return
    ** string message */

    /* operations */
}

public:
    ReturnVals(const String& defl,
```
const StringArray& params,
const String& rslt,
const String& msg = (String)"");

ReturnVals(const ReturnVals& rv);

ReturnVals& operator = (const ReturnVals& rv);

String GetDefault() const;

StringArray GetParams() const;

String GetResult() const;

String GetMessage() const;

friend ostream& operator << (ostream& out, const ReturnVals& r);
```cpp
typedef Array<void*> ImplArray;

class ReturnValsConcrete: public gc {  
  /* model */  
  /** data members */  
  /** pointer deft */  
  /** sequence of pointer parameters */  
  /** pointer return */  
  /** pointer message */  
  /** operations */  
  
  public:  
    ReturnValsConcrete(const void* def,
```
const ImplArray& params.
const void* rslt.
const String msg = (String)"");

/* modifies: self */
** postA: deflt' = def \ parameters' = params \ 
** return' = rslt \ message' = msg */

ReturnValsConcrete(const ReturnValsConcrete& rv);

/* modifies: self */
** postA: deflt' = rv.deflt \ parameters' = rv.parameters \ 
** return' = rv.return \ message' = rv.message */

ReturnValsConcrete& operator = (const ReturnValsConcrete& rv);
/* modifies: self */
** postA: deflt' = rc.deflt \ parameters' = rv.parameters \ 
** return' = rv.return \ message' = rv.message 
** \ result = self' */

void* GetDefaultQ const;
/* postA: result = deflt */

ImplArray GetParamsQ const;
/* postA: result = parameters */

void* GetResultQ const;
/* postA: result = return */

String GetMessageQ const;
/* postA: result = message */

friend ostream& operator << (ostream& out, const ReturnValsConcrete& r);

#include "ReturnValsConcrete.pri"
): 
#endif
APPENDIX D  COMPILER GENERATED CODE

These sections contain the code generated by the ATS and CVS to create the ATT and CVT for PriorityQueue shown in Appendix B.

SPECS-C++ Compiler

The files in this section were generated by sc++ -att. The first four files, pqueue_A.pre, pqueue_A.h, pqueue_A.pri, pqueue_A.C are the executable version of the specification. DoIt-PriorityQueue_A.C is the engine for the tool and Types-PriorityQueue_A.txt lists the types defined in the model.

pqueue_A.pre

```c
// Pqueue_A.pre (compiler-generated) -------

typedef char ElemType_A;
typedef int PriorityType_A;
typedef int TimeType_A;
typedef Tuple EntryType_A;
typedef Set QueueType_A;
```

pqueue_A.h

```c
// Pqueue_A.h (compiler-generated) -------

#ifndef _Pqueue_A_H
#define _Pqueue_A_H
#include <T.h>
#include <Set.h>
#include <Sequence.h>

#endif
```
```cpp
#include <Tuple.h>
#include <String.h>
#include "Pqueue_A.pre"

class PriorityQueue_A {
friend bool UniqueTimes_abs.fn (QueueType_A q);
friend EntryType_A HighestEntry_abs.fn (QueueType_A q);
friend TimeType_A LatestTime_abs.fn (QueueType_A q);
public:
  PriorityQueue_A () ;
  bool PriorityQueue_A.preA() const ;
  PriorityQueue_A (const PriorityQueue_A& pq) ;
  bool PriorityQueue_A.preA(const PriorityQueue_A& pq) const ;
  PriorityQueue_A () ;
  PriorityQueue_A& operator = (const PriorityQueue_A& pq) ;
  bool operator.eq.preA(const PriorityQueue_A& pq) const;
  void AddEntry (const ElemType_A& e, const PriorityType_A& level) ;
  bool AddEntry.preA(const ElemType_A& e, const PriorityType_A& level) const ;
  ElemType_A FirstEntry () const ;
  bool FirstEntry.preA() const ;
  PriorityType_A HighestPriority () const ;
  bool HighestPriority.preA() const ;
  void RemoveEntry () ;
  bool RemoveEntry.preA() const ;
  bool IsEmpty () const ;
  bool IsEmpty.preA() const ;

#include "Pqueue_A.pri"
};  // end of class PriorityQueue_A
```
private:
QueueType_A thePQ;

public:
bool ConstraintFilter() const;

#include "Pqueue_A.h"

bool UniqueTimes_abs_fn(QueueType_A q) {
#ifndef _DEBUG
    cout << "bool UniqueTimes_abs_fn (QueueType_A q)\n";
#endif
    bool result;
    EntryType_A e11;
    Set q5 = q;
    bool solutionForall5 = true;
    T* tempForall5 = NULL;
    q5.BeginIteration();
#ifndef _DEBUG
    cout << "for all expr '/\n";
#endif
    while(!q5.EndOfIteration()) {
        tempForall5 = &(q5.GetNextReference());
        Tuple e11;
        e11 = (*tempForall5);
        EntryType_A e22;
        Set q4 = q;
    }
}
bool solutionForall4 = true;
T* tempForall4 = NULL;
q4.BeginIteration();
/** foral_expr */
while(!q4.EndOfIteration()) {
    tempForall4 = &(q4.GetNextReference());
    Tuple e22;
    e22 = (*tempForall4);
    T *tempSelect1;
    tempSelect1 = (e11).Select(3);
    int solutionSelect1 = (*tempSelect1).GetIntVal();
    T *tempSelect2;
    tempSelect2 = (e22).Select(3);
    int solutionSelect2 = (*tempSelect2).GetIntVal();
    solutionForall4 = solutionForall4 &&
    ((e11 == e22) ||
    (solutionSelect1 != solutionSelect2 ));
}

solutionForall5 = solutionForall5 && (solutionForall4);
}
result = (solutionForall5);
return(result);
;

/]***********
EntryType_A HighestEntry.abs_fn (QueueType_A q) {
#endif .DEBUG
    cout << "EntryType_A HighestEntry.abs_fn (QueueType_A q)\n";
#endif
    Tuple result;
    EntryType_A e3;

    bool result = false;
    Set q13 = q;
    q13.BeginIteration();
    T* tempExists13;
/* exists_expr */
while(!ql3.EndOfIteration() && leresult) {
    tempExists13 = &(q13.GetNextReference());
    Tuple solutionExists13;
    solutionExists13 = (*tempExists13);
    e3 = solutionExists13;
    EntryType_A e14;
    Set ql2 = q;
    bool solutionForall12 = true;
    T* tempForall12 = NULL;
    ql2.BeginIteration();
    /* forall_expr */
    while(!ql2.EndOfIteration()) {
        tempForall12 = &(ql2.GetNextReference());
        Tuple e14;
        e14 = (*tempForall12);
        T *tempSelect5;
        tempSelect5 = (e3).Select(2);
        int solutionSelect5 = (*tempSelect5).GetIntVal();
        T *tempSelect6;
        tempSelect6 = (e4).Select(2);
        int solutionSelect6 = (*tempSelect6).GetIntVal();
        T *tempSelect7;
        tempSelect7 = (e3).Select(2);
        int solutionSelect7 = (*tempSelect7).GetIntVal();
        T *tempSelect8;
        tempSelect8 = (e4).Select(2);
        int solutionSelect8 = (*tempSelect8).GetIntVal();
        T *tempSelect9;
        tempSelect9 = (e3).Select(3);
        int solutionSelect9 = (*tempSelect9).GetIntVal();
        T *tempSelect10;
        tempSelect10 = (e4).Select(3);
        int solutionSelect10 = (*tempSelect10).GetIntVal();

        solutionForall12 = solutionForall12 &&
            ((e3==e14) || ( (solutionSelect5 > solutionSelect6) ||
                (solutionSelect7 == solutionSelect8) &&
                (solutionSelect9 < solutionSelect10) ));
    }
}
if(( solutionForall12 )) {
    result = (e3);
    eresult = true;
}
}
return (result);

TimeType_A LatestTime_abs_fn (QueueType_A q) {
#ifndef _DEBUG
  cout << "TimeType_A LatestTime_abs_fn (QueueType_A q)\n";
#endif
  int result;
  Set tempO;
  Set temp1;
  EntryType_A e5;

  if (q == tempO) {
    result = (0);
  }
  if (q != temp1) {
    bool eresult = false;
    Set q18 = q;
    q18.BeginIteration();
    T* tempExists18;
    /*exists_expr*/
    while(q18.EndOfIteration() && !eresult) {
      tempExists18 = &(q18.GetNextReference());
      Tuple solutionExists18;
      solutionExists18 = (*tempExists18);
      e5 = solutionExists18;
      EntryType_A e16;
      Set q17 = q;
      bool solutionForall17 = true;
  
}
T* tempForall17 = NULL;
q17.BeginIteration();
/* forall_expr */
while(!q17.EndOfIteration()) {
    tempForall17 = &(q17.GetNextReference());
    Tuple e16;
    e16 = (*tempForall17);
    T *tempSelect13;
    tempSelect13 = (e5).Select(3);
    int solutionSelect13 = (*tempSelect13).GetIntVal();
    T *tempSelect14;
    tempSelect14 = (e16).Select(3);
    int solutionSelect14 = (*tempSelect14).GetIntVal();

    solutionForall17 = solutionForall17 &&
        ((e5==e16) ||
        ( ( solutionSelect13 > solutionSelect14 ) ));
}
if(solutionForall17) {
    T *tempSelect15;
    tempSelect15 = (e5).Select(3);
    int solutionSelect15 = (*tempSelect15).GetIntVal();
    result = ( solutionSelect15 );

    result = true;
}
}

return (result);
;
}

/***********************
/
/* Constraint Expression Code ———*/
bool PriorityQueue_A::ConstraintFilter() const {
    #ifdef _DEBUG
        cout << "bool PriorityQueue_A::ConstraintFilter() const
\n";
typedef

EntryType_A e7;
Set thePQ20 = thePQ;
bool solutionForall20 = true;
T* tempForall20 = NULL;
thePQ20.BeginIteration();

while(!thePQ20.EndOfIteration()) {
    tempForall20 = &thePQ20.GetNextReference();
    Tuple e7;
    e7 = (*tempForall20);
    T *tempSelect18;
    tempSelect18 = e7.Select(2);
    int solutionSelect18 = (*tempSelect18).GetIntVal();

    solutionForall20 = solutionForall20 && (solutionSelect18 >= 0);
}

EntryType_A e8;
Set thePQ22 = thePQ;
bool solutionForall22 = true;
T* tempForall22 = NULL;
thePQ22.BeginIteration();

while(!thePQ22.EndOfIteration()) {
    tempForall22 = &thePQ22.GetNextReference();
    Tuple e8;
    e8 = (*tempForall22);
    T *tempSelect20;
    tempSelect20 = e8.Select(3);
    int solutionSelect20 = (*tempSelect20).GetIntVal();

    solutionForall22 = solutionForall22 && (solutionSelect20 >= 0);
}

return (( UniqueTimes.abs_fn(thePQ) ) &&
    ( solutionForall20 ) ) && ( solutionForall22 ));

} /* ---------------- */
PriorityQueue A::PriorityQueue A () {

#ifdef _DEBUG
    cout << "PriorityQueue A::PriorityQueue A () \n";
#endif
    QueueType A _thePQ:

    Set temp2;
    _thePQ = temp2;
    thePQ = _thePQ;
};

bool PriorityQueue A::PriorityQueue A preA () const {

#ifdef _DEBUG
    cout << "bool PriorityQueue A::PriorityQueue A preA () const \n";
#endif
    return (true);
} /* ---------------- */

PriorityQueue A::PriorityQueue A (const PriorityQueue A & pq) {

#ifdef _DEBUG
    cout << "PriorityQueue A::PriorityQueue A (const PriorityQueue A & pq) \n";
#endif
    QueueType A _thePQ:

    _thePQ = (pq.thePQ);
    thePQ = _thePQ;
};

bool PriorityQueue A::PriorityQueue A preA (const PriorityQueue A & pq) const {

#ifdef _DEBUG
    cout << "bool PriorityQueue A::PriorityQueue A preA (const PriorityQueue A & pq) const \n";
#endif
    return (true);
} /* ---------------- */
PriorityQueue.A::PriorityQueue.A () {
    
    #ifdef .DEBUG
    cout << "PriorityQueue.A::PriorityQueue.A () \n";
    #endif

    ;
}

/*/ --------------- */

    #ifdef .DEBUG
    cout << "PriorityQueue.A& PriorityQueue.A::operator = (const PriorityQueue.A& pq) \n";
    #endif
    QueueType.A __thePQ;

    __thePQ = (pq.__thePQ);
    __thePQ = __thePQ;
    return ( *this );
}

bool PriorityQueue.A::operator_eq.preA(const PriorityQueue.A& pq) const {
    #ifdef .DEBUG
    cout << "bool PriorityQueue.A::operator_eq.preA(const PriorityQueue.A& pq) const\n";
    #endif
    return (true);
}

/*/ --------------- */

void PriorityQueue.A::AddEntry (const ElemType.A& c, const PriorityType.A& level) {
    #ifdef .DEBUG
    cout << "void PriorityQueue.A::AddEntry (const ElemType.A& c, const PriorityType.A& level) \n";
    #endif

QueueType_A thePQ;

Tuple tup3;
T* tvalue22 = new T(e);
tup3.AddComponent(T.char, "", tvalue22);
T* tvalue23 = new T(level);
tup3.AddComponent(T.int, "", tvalue23);
T* tvalue24 = new T(LatestTime_abs_fn(thePQ) + 1);
tup3.AddComponent(T.int, "", tvalue24);
Set temp4( tup3 );
.thePQ = ( thePQ ).Union(temp4);
thePQ = ..thePQ;

bool PriorityQueue_A::AddEntry_preA(const ElemType_A& e, const PriorityType_A& level) const {
  ifndef .DEBUG
    cout << "bool PriorityQueue_A::AddEntry_preA(const ElemType_A& e, const PriorityType_A& level) const:\n";
  endif

    return (true);
}
/* ---------------*/

ElemType_A PriorityQueue_A::FirstEntry () const {

#ifndef .DEBUG
    cout << "ElemType_A PriorityQueue_A::FirstEntry () const: \n";
#endif

    T *tempSelect25;
    tempSelect25 = ( HighestEntry_abs_fn(thePQ) ).Select(1);
    char solutionSelect25 = (*tempSelect25).GetCharVal();
    return( solutionSelect25 );
}

bool PriorityQueue_A::FirstEntry_preA() const {
  ifndef .DEBUG
    ifndef .DEBUG
      cout << "bool PriorityQueue_A::FirstEntry_preA() const:\n";
  endif
#endif
}
Set temp5;
    return (thePQ != temp5);
}
/* ------------*/

PriorityType_A PriorityQueue_A::HighestPriority () const {

    #ifdef _DEBUG
    cout << "PriorityType_A PriorityQueue_A::HighestPriority () const \n";
    #endif
    
    T *tempSelect26;
    tempSelect26 = ( HighestEntry.abs.fn(thePQ) ).Select(2);
    int solutionSelect26 = (*tempSelect26).GetIntVal();
    return( solutionSelect26 );
}

bool PriorityQueue_A::HighestPriority_preA() const {

    #ifdef _DEBUG
    cout << "bool PriorityQueue_A::HighestPriority_preA() const\n";
    #endif
    
    Set temp6;
    return (thePQ != temp6);
}
/* ------------*/

void PriorityQueue_A::RemoveEntry () {

    #ifdef _DEBUG
    cout << "void PriorityQueue_A::RemoveEntry () \n";
    #endif
    
    QueueType_A __thePQ;

    Set temp8( HighestEntry.abs.fn(thePQ) );
    __thePQ = thePQ - temp8;
    thePQ = __thePQ;
    ;
}
bool PriorityQueue::RemoveEntry_prcA() const {
    #ifdef DEBUG
        cout << "bool PriorityQueue::RemoveEntry_prcA() const \n";
    #endif
    Set temp7;
    return (thePQ != temp7);
} /* ---------*/

bool PriorityQueue::IsEmpty_prcA() const {
    #ifdef DEBUG
        cout << "bool PriorityQueue::IsEmpty_prcA() const \n";
    #endif
    bool result;

    Set temp9;
    result = (thePQ == temp9);
    return(result);
}

bool PriorityQueue::IsEmpty_prcA() const {
    #ifdef DEBUG
        cout << "bool PriorityQueue::IsEmpty_prcA() const \n";
    #endif
    return (true);
} /* ---------*/

// for the Abstract Test Tool
PriorityQueue::PriorityQueue (QueueType _thePQ, T* dummy)
{
    thePQ = _thePQ;
}

Tuple PriorityQueue::AbsDataMembers() const 
{
Tuple *result = new Tuple;

#ifdef .DEBUG
  cout << "\nPriorityQueue_A::AbsDataMembers:" << endl;
#endif
  Set* _thePQ = new Set(_thePQ);

#ifdef .DEBUG
  cout << " _thePQ = new Set(_thePQ);
#endif

result->AddComponent(_thePQ->GetType(), "thePQ", _thePQ);
return (*result);

---

DoIt-PriorityQueue_A.C

#include "Dolt.h"
#include "Pqueue_A.h"
#include <StringAux.h>
#include <stdio.h>

String className = "PriorityQueue_A":

PriorityQueue_A* StringToClass(String val) {
  val = (String) "(" || val;
  val = val || ")";
  Tuple* tupVal = (Tuple*) FromString(val);
  if (tupVal == NULL) {
    return NULL;
  } else {
    Set* param1 = (Set*)tupVal->Select(1);
// delete tupVal;
    return new PriorityQueue_A(*param1, (T*)NULL);

    }// end if
    }// end StringToClass

// -----------------------------

StringArray FilterConstraintSatisfying(const StringArray& s){
    PriorityQueue_A* temp;
    StringArray result(O);
    for(int i = 1; i <= s.Length(); i++){
        temp = StringToClass(s[i]);
        if (temp->ConstraintFilter()) {
            result.Append(s[i]);
        }
    // delete temp;
    }
    return result;
}

// -----------------------------

ReturnVals* Execute(const String& defaultParam,
    const StringArray& actualParams,
    const String& memberFuncName)
{
    String tempString;
    Tuple* def;
    ReturnVals* rv = NULL;
    PriorityQueue_A* __PriorityQueue_A;
    tempString = GetSubstringTo(memberFuncName, '(');
    tempString.Trim();
    bool isConstructor = (tempString == className);
    if (!memberFuncName.IsPrefixOf("friend") &&
        !isConstructor) {
        __PriorityQueue_A = StringToClass(defaultParam);
        if (!__PriorityQueue_A->ConstraintFilter()) {
StringArray tempArray(0);
rv = new ReturnVals(defaultParam, tempArray, "",
    "Error: default parameter does not satisfy constraints.");
return rv;
} // end if

if (memberFuncName == "") {
    fprintf(stderr,"Empty member function name\n");
    return NULL;
} else if (memberFuncName == "PriorityQueue.A ()") {
if (!..PriorityQueue.A->PriorityQueue_A.preA()) {
    rv = new ReturnVals(defaultParam, actualParams,"",
    "ERROR: precondition for PriorityQueue.A not satisfied");
    return rv;
}

if (isConstructor)
cerr << "OOPS! This is supposed to be a constructor"
    << endl << "Please report this as a bug.\n";
..PriorityQueue_A = new PriorityQueue_A();

    Tuple dataMembers(....PriorityQueue_A->AbsDataMembers());
tempString = dataMembers.ToString();
tempString[tempString.FindFirst(' ') = ' ';
tempString[tempString.FindLast(' ')] = '\0';
tempString.Trim();

if (!...PriorityQueue.A->ConstraintFilter()) {
    StringArray tempArray(0);
    rv = new ReturnVals(tempString, tempArray, "",
    "Error: default parameter (post-state) does not satisfy constraints.");
    return rv;
} // end if

    // Trim the parameters
    // Trim the parameters
for (int trimCount = 1; trimCount <= actualParams.Length();trimCount++) {
    actualParams[trimCount].Trim();
rv = new ReturnVals(tempString, actualParams, "");

else if (memberFuncName == "PriorityQueue.A (const PriorityQueue.A& pq)") {
    PriorityQueue.A param1(*(StringToClass(actualParams[l])));
    if (!param1.ConstraintFilter()) {
        rv = new ReturnVals("", actualParams,"",
            "ERROR: param 1 does not satisfy constraint");
        return rv;
    }
    PriorityQueue.A param2(*(StringToClass(actualParams[l + 1])));
    if (!PriorityQueue.A->PriorityQueue.A.preA(param1)) {
        rv = new ReturnVals(defaultParam,
            actualParams,"",
            "ERROR: precondition for PriorityQueue.A not satisfied");
        return rv;
    }
    if (!isConstructor)
        cerr << "OOPS! This is supposed to be a constructor"
        << endl << "Please report this as a bug.\n";
    PriorityQueue.A = new PriorityQueue.A(param1);
    Tuple dataMembers(PriorityQueue.A->AbsDataMembers());
    tempString = dataMembers.ToString();
    tempString[tempString.FindFirst(\'\')I = '\';
    tempString[tempString.FindLast(\'\')J = '\0';
    tempString.Trim();

    if (!PriorityQueue.A->ConstraintFilter()) {
        StringArray tempArray(0);
        rv = new ReturnVals(tempString, tempArray, "");
        "Error: default parameter (post-state) does not satisfy constraints."
        return rv;
    } // end if
    // Trim the parameters
    for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
        actualParams[trimCount].Trim();
    } // end for
rv = new ReturnVals(tempString, actualParams, "");

} else if (memberFuncName == "PriorityQueue_A ()") {

    PriorityQueue_A -> "PriorityQueue_A();

    tempString = "trashed";
    // Trim the parameters
    for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
        actualParams[trimCount].Trim();
    } // end for
    rv = new ReturnVals(tempString, actualParams, "");

} else if (memberFuncName == "PriorityQueue_A & operator = (const PriorityQueue_A & pq)") {
    PriorityQueue_A param1 = *(StringToClass(actualParams[1]));
    if (!param1.ConstraintFilter()) {
        rv = new ReturnVals("", actualParams, "",
            "ERROR: param 1 does not satisfy constraint");
        return rv;
    }
    if (!PriorityQueue_A->operator.eq.preA(param1)) {
        rv = new ReturnVals(defaultParam, actualParams, "",
            "ERROR: precondition for operator = not satisfied");
        return rv;
    }
    PriorityQueue_A result = ((PriorityQueue_A) = param1);

    Tuple resultTup(result.AbsDataMembers());
    String resultString = resultTup.ToString();
    resultString[resultString.FindFirst('(')] = ' ';
    resultString[resultString.FindLast(')')] = '\0';
    if (!result.ConstraintFilter()) {
        rv = new ReturnVals("", actualParams, resultString,
            "ERROR: result does not satisfy constraint");
        return rv;
    }
}
Tuple dataMembers(_._PriorityQueue.A—>AbsDataMembers());

tempString = dataMembers.ToString();

tempString[tempString.FindFirst('(')] = ' ';

tempString[tempString.FindLast(')')] = '\0';

tempString.Trim();

if (! _._PriorityQueue.A—>ConstraintFilter()) {
    StringArray tempArray(0);
    rv = new ReturnVals(tempString, tempArray, "";
        "Error: default parameter (post-state) does not satisfy constraints.");
        return rv;
    }

    // Trim the parameters

for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
    actualParams[trimCount].Trim();
    } // end for

rv = new ReturnVals(tempString, actualParams, resultString);

} else if (memberFuncName == "void AddEntry (const ElemType.A& e, const PriorityType.A& level)";) {

    T* temp1 = FromString(actualParams[1]);
    char param1 = (*temp1).GetCharVal();

    T* temp2 = FromString(actualParams[2]);
    int param2 = (*temp2).GetIntVal();

    if (! _._PriorityQueue.A—>AddEntry.preA(param1, param2)) {
        rv = new ReturnVals(defaultParam, actualParams,"",
            "ERROR: precondition for AddEntry not satisfied");
        return rv;
    }

    _._PriorityQueue.A—>AddEntry(param1, param2);

    Tuple dataMembers(_._PriorityQueue.A—>AbsDataMembers());
    tempString = dataMembers.ToString();
    tempString[tempString.FindFirst('(')] = ' ';
    tempString[tempString.FindLast(')')] = '\0';
    tempString.Trim();
if (!PriorityQueue_A->ConstraintFilter()) {
    StringArray tempArray(0);
    rv = new ReturnVals(tempString, tempArray, "",
            "Error: default parameter (post-state) does not satisfy constraints.");
    return rv;
} // end if

// Trim the parameters
for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
    actualParams[trimCount].Trim();
} // end for
rv = new ReturnVals(tempString, actualParams, "");
}

else if (memberFuncName == "ElemType_A FirstEntry ( ) const") {
    if (!PriorityQueue_A->FirstEntry().preA()) {
        rv = new ReturnVals(defaultParam, actualParams,"",
                "ERROR: precondition for FirstEntry not satisfied");
        return rv;
    }
    T result(T_char);
    result = PriorityQueue_A->FirstEntry();

    String resultString = result.ToString();
    resultString.Trim();
    
    Tuple dataMembers(PriorityQueue_A->AbsDataMembers());
    tempString = dataMembers.ToString();
    tempString[tempString.FindFirst(' ')] = ' ';
    tempString[tempString.FindLast(') ')] = '\0';
    tempString.Trim();

    if (!PriorityQueue_A->ConstraintFilter()) {
        StringArray tempArray(0);
        rv = new ReturnVals(tempString, tempArray, "",
                "Error: default parameter (post-state) does not satisfy constraints.");
        return rv;
    } // end if
    // Trim the parameters
for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
    actualParams[trimCount].Trim();
} // end for
rv = new ReturnVals(tempString, actualParams, resultString);

} else if (memberFuncName == "PriorityType_A HighestPriority () const") {
if (PriorityQueue_A->HighestPriority_prcA()) {
    rv = new ReturnVals(defaultParam,
                        actualParams,"",
                        "ERROR: precondition for HighestPriority not satisfied");
    return rv;
}
T result(T::int);
result = PriorityQueue_A->HighestPriority();
String resultString = result.ToString();
resultString.Trim();

    Tuple dataMembers(PriorityQueue_A->AbsDataMembers());
    tempString = dataMembers.ToString();
    tempString[tempString.FindFirst(1) = ' ';
    tempString[tempString.FindLast(1) = '\0';
    tempString.Trim();

    if (!PriorityQueue_A->ConstraintFilter()) {
        StringArray tempArray(0);
        rv = new ReturnVals(tempString, tempArray, "",
                            "Error: default parameter (post-state) does not satisfy constraints.");
        return rv;
    } // end if
    // Trim the parameters
    for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
        actualParams[trimCount].Trim();
    } // end for
    rv = new ReturnVals(tempString, actualParams, resultString);

} else if (memberFuncName == "void RemoveEntry ()") {
if (!PriorityQueue.A->RemoveEntry_preA()) {
    rv = new ReturnVals(defaultParam, actualParams, "\n    "ERROR: precondition for RemoveEntry not satisfied"\n    
    
    return rv;
}

..PriorityQueue.A—> RemoveEntry();

Tuple dataMembers(../PriorityQueue..A—>AbsDataMembers());
    tempString = dataMembers.ToString();
    tempString[tempString.FindFirst('"')] = '\"';
    tempString[tempString.FindLast('"')] = '\\0';
    tempString.Trim();

if (!../PriorityQueue..A—>ConstraintFilter()) {
    StringArray tempArray(O);
    rv = new ReturnVals(tempString, tempArray, "\n    "Error: default parameter (post-state) does not satisfy constraints.\n    
    
    return rv;
} // end if
    // Trim the parameters
for (int trimCount = 1; trimCount <= actualParams.Length();trimCount++) {
    actualParams[trimCount].Trim();
} // end for
rv = new ReturnVals(tempString,actualParams, "\n
} else if (memberFuncName == "bool IsEmpty () const") {
    if (!../PriorityQueue..A—>IsEmpty.preAQ) {
        rv = new ReturnVals(defaultParam, actualParams, "\n        "ERROR: precondition for IsEmpty not satisfied"\n        
        return rv;
    }
    T result(T.bool);
    result = ../PriorityQueue..A—>IsEmpty();
}
String resultString = result.ToString();
resultString.Trim();

Tuple dataMembers(PriorityQueue.A->AbsDataMembers());
tempString = dataMembers.ToString();
tempString[tempString.FindFirst('‘')] = ' ';
tempString[tempString.FindLast('’')] = '\0';
tempString.Trim();

if (!PriorityQueue.A->ConstraintFilter()) {
    StringArray tempArray(0);
    rv = new ReturnVals(tempString, tempArray, "Error: default parameter (post-state) does not satisfy constraints.");
    return rv;
} // end if

// Trim the parameters
for (int trimCount = 1; trimCount <= actualParams.Length(); trimCount++) {
    actualParams[trimCount].Trim();
} // end for
rv = new ReturnVals(tempString, actualParams, resultString);

} else {
    cerr << "Unknown member function: " << memberFuncName << "\n" << endl;
    return NULL;
} // end if
return rv;
} // end Execute....

---

**Types-PriorityQueue_A.txt**

ElemType.A: char
PriorityType.A: int
TimeType_A: int
EntryType.A: (tuple char int int)
QueueType.A: set of (tuple char int int)
PriorityQueue.A: set of (tuple char int int)
Representation Mapping Compiler

The following files were generated by sc++ -repmap. The first two are the executable versions of the abstract functions defined in the .pri file. This includes the repmap function, and the equivalence function. DoItImpl-PriorityQueue.C is the engine for the CVT, and TypesImpl-pqueue.txt contains the types defined in the .pre file.

```cpp
pqueue_repmap.h

// pqueue_repmap.h
#include "pqueue.h"
#include "Pqueue.A.h"
ifndef .repmap.pqueue
#define .repmap.pqueue

class Repmap.pqueue {
   public:

   static bool EquivSets.abs.fn (QueueType.A q1, QueueType.A q2);
   static bool Equivalent.abs.fn (PriorityQueue.A pq1, PriorityQueue.A pq2);
   static QueueType.A Repmap.QueueType.abs.fn (EntryPointer p, int t);
   static PriorityQueue.A Repmap.PriorityQueue.abs fn (PriorityQueue pq.c);
}; // end class Repmap.pqueue
#endif

pqueue_repmap.C

// pqueue_repmap.C
#include "pqueue_repmap.h"

bool Repmap.pqueue::EquivSets.abs.fn (QueueType.A q1, QueueType.A q2) {
   ifndef _DEBUG
      cout << "bool EquivSets.abs.fn (QueueType.A q1, QueueType.A q2)\n";
   endif
   bool result;
   Set temp10;
   Set temp11;
   Set temp12;
```
Set temp13;

if (( q1 == temp10 ) || ( q2 == temp11 )) {
    result = ( q1 == q2 );
    return(result);
}

if (( q1 != temp12 ) && ( q2 != temp13 )) {
    T *tempSelect27;
    tempSelect27 = ( HighestEntry.abs.fn(q1) ).Select(2);
    int solutionSelect27 = (*tempSelect27).GetIntVal();
    T *tempSelect28;
    tempSelect28 = ( HighestEntry.abs.fn(q2) ).Select(2);
    int solutionSelect28 = (*tempSelect28).GetIntVal();
    T *tempSelect29;
    tempSelect29 = ( HighestEntry.abs.fn(q1) ).Select(1);
    char solutionSelect29 = (*tempSelect29).GetCharVal();
    T *tempSelect30;
    tempSelect30 = ( HighestEntry.abs.fn(q2) ).Select(1);
    char solutionSelect30 = (*tempSelect30).GetCharVal();

    Set temp14( HighestEntry.abs.fn(q1) );
    Set temp15( HighestEntry.abs.fn(q2) );
    result = ( ( ( solutionSelect27 == solutionSelect28 ) &&
                    ( solutionSelect29 == solutionSelect30 ) ) &&
                    ( EquivSets.abs.fn(q1 — temp14, q2 — temp15) ) );
    return(result);
}

return (result);
;
}

bool Repmap.pqueue::Equivalent.abs.fn (PriorityQueue.A pql, PriorityQueue.A pq2) {
#endif .DEBUG
    cout << "bool Equivalent.abs.fn (PriorityQueue.A pql, PriorityQueue.A pq2)\n";
#endif

    bool result;

    result = ( EquivSets.abs.fn((pql.thePQ), (pq2.thePQ)) );
return(result);
;
}

QueueType_A Repmap.pqueue::Repmap.QueueType_abs_fn (EntryPointer p, int t) {
#ifndef _DEBUG
    cout << "QueueType_A Repmap.QueueType_abs_fn (EntryPointer p, int t)\n";
#endif
    Set result;
    EntryType_A e10;

    if (p == NULL) {
        Set temp16;
        result = ( temp16 );
    }
    if (p != NULL) {
        Tuple tup17;
        T* tvalue31 = new T(p -> element);
        tup17.AddComponent(T.char, "", tvalue31);
        T* tvalue32 = new T(p -> priority);
        tup17.AddComponent(T.int, "", tvalue32);
        T* tvalue33 = new T(t);
        tup17.AddComponent(T.int, "", tvalue33);

        Set _spec_9(tup17);
        bool ercsult = false;
        Set _spec_935 = _spec_9;
        _spec_935.BeginIteration();
        T* tempExists35;
        /*exists.expr*/
        while(!_spec_935.EndOfIteration() && !result) {
            tempExists35 = &(_spec_935.GetNextReference());
            Tuple solutionExists35;
            solutionExists35 = (*tempExists35);
            e10 = solutionExists35;
            Set templ8(e10);
            result = ( ( temp18 ).Union(Repmap.QueueType.abs_fn(p -> next, t + 1) ) );
eresult = true;

return (result);

PriorityQueue_A Repmap::Repmap_PriorityQueue_abs_fn (PriorityQueue pq_c) {
    #ifdef _DEBUG
        cout << "PriorityQueue_A Repmap_PriorityQueue_abs_fn (PriorityQueue pq_c)\n";
    #endif
    PriorityQueue_A result;

    (result.thePQ) = (Repmap_QueueType_abs_fn((pq_c.head), 1));
    return (result);
}

DoItImpl-PriorityQueue.C

#include "DoltImpl.h"
#include "pqueue_repmap.h"
#include <StringAux.h>
#include <stdio.h>

String classNameImpl = "PriorityQueue";
String TypeInfoFilename = "TypesImpl-pqueue.txt";

PriorityQueue_A* StringToClassImplVersion(String val) {
    val = (String) "(" ++ val;
    val = val || ")";
    Tuple* tupVal = (Tuple*) FromString(val);
    if (tupVal == NULL) {
        return NULL;
    } else {

Set* param1 = (Set*)tupVal->Select(1);

// delete tupVal;
return new PriorityQueue_A(*param1, (T*)NULL);
}

// end if
}

// end StringToClassImplVersion

ImplArray FilterConstraintSatisfying(const ImplArray& s) {

/* PriorityQueue* temp;
StringArray result(0);
for (int i = 1; i <= s.Length(); i++) {
    temp = StringToClassImplVersion(s[i]);
    if (temp->ConstraintFilter()) {
        result.Append(s[i]);
    }
    // delete temp;
}*/
return s;
}

//

bool PassRepmap(const String& absVal,
const void* concreteVal) {
PriorityQueue_A* classVal_A = StringToClassImplVersion(absVal);
PriorityQueue_A* repmapResult = new PriorityQueue_A(Repmap_pqueue::Repmap_PriorityQueue_abs_fn
        (*((PriorityQueue*)concreteVal)));
bool same = Repmap_pqueue::Equivalent_abs_fn(*classVal_A, *repmapResult);

#include <iostream>

if (_DEBUG)
    Tuple origDataMembers = classVal_A->AbsDataMembers();
    Tuple repmapDataMembers = repmapResult->AbsDataMembers();
    cout << "PassRepmap: class PriorityQueue" << endl;
    cout << "abstractValue: " << origDataMembers << endl;
    cout << "repmapValue: " << repmapDataMembers << endl;
    cout << "Equivalent: " << same << endl;
cout << "same? " /* boolalpha */ << (origDataMembers == repmapDataMembers)
    << endl;
#endif

return (same);
}

// -------------------------------
String GetRepMapValue(const void* concreteVal) {
    PriorityQueue.A* repmapResult = new PriorityQueue.A(Repmap_queue::Repmap_PriorityQueue_abs_fn
        ((*((PriorityQueue*)concreteVal)));)
    Tuple repmapDataMembers = repmapResult->AbsDataMembers();
    if (.DEBUG
        cout << "GetRepMapValue: class PriorityQueue" << endl;
        cout << "\trepmapValue: " << repmapDataMembers << endl;
    #endif
    return (repmapDataMembers.ToString());
}

// -------------------------------
ReturnValsConcrete* ExecuteImpl(const void* .defaultParam,
    const mplArray& actualParams,
    const String& memberFuncName){

String tempString;
ReturnValsConcrete* rv = NULL;
PriorityQueue *defaultParam = NULL;
if (.defaultParam != NULL) {
    defaultParam = new PriorityQueue (*((PriorityQueue*).defaultParam));
} else {
    defaultParam = NULL;
} // end if
    tempString = GetSubstringTo(memberFuncName, ' ');
    tempString.Trim();
    bool isConstructor = (tempString == classNameImpl);
if (!memberFuncName.IsPrefixOf("friend") &&
    !isConstructor) {

/* if (!defaultParam->ConstraintFilter()) {
    StringArray tempArray(0);
    rv = new ReturnValsConcrete(defaultParam, tempArray, "",
        "Error: default parameter does not satisfy constraints.");
    return rv;
} // end if */

if (memberFuncName == "") {
    fprintf(stderr,"Empty member function name\n");
    return NULL;
} else if (memberFuncName == "PriorityQueue ()") {

    if (!isConstructor)
        cerr << "OOPS! This is supposed to be a constructor"
            << endl << "Please report this as a bug.\n";
    defaultParam = new PriorityQueue();

    rv = new ReturnValsConcrete(defaultParam, actualParams, NULL);
}

else if (memberFuncName == "PriorityQueue (const PriorityQueue* pq)") {

    if (!isConstructor)
        cerr << "OOPS! This is supposed to be a constructor"
            << endl << "Please report this as a bug.\n";
    defaultParam = new PriorityQueue((PriorityQueue*)actualParams[1]));

    rv = new ReturnValsConcrete(defaultParam, actualParams, NULL);
}

else if (memberFuncName == "PriorityQueue ()") {
    defaultParam->PriorityQueue();

    rv = new ReturnValsConcrete(defaultParam, actualParams, NULL);
}

else if (memberFuncName == "PriorityQueue& operator = (const PriorityQueue& pq)"
PriorityQueue *result = &((defaultParam) = (*((PriorityQueue *)actualParams[1])));

rv = new ReturnValsConcrete(defaultParam, actualParams, result);

} else if (memberFuncName == "void AddEntry (const ElemType& e, const PriorityType& level)") {

defaultParam->AddEntry( (*((char *)actualParams[1])), (*((int *)actualParams[2])));

rv = new ReturnValsConcrete(defaultParam, actualParams, NULL);

} else if (memberFuncName == "ElemType FirstEntry () const") {

cchar *result;

result = new char (defaultParam->FirstEntry());

rv = new ReturnValsConcrete(defaultParam, actualParams, result);

} else if (memberFuncName == "PriorityType HighestPriority () const") {

int *result;

result = new int (defaultParam->HighestPriority());

rv = new ReturnValsConcrete(defaultParam, actualParams, result);

} else if (memberFuncName == "void RemoveEntry ()") {

defaultParam->RemoveEntry();

rv = new ReturnValsConcrete(defaultParam, actualParams, NULL);

} else if (memberFuncName == "bool IsEmpty () const") {

bool *result;

result = new bool (defaultParam->IsEmpty());

rv = new ReturnValsConcrete(defaultParam, actualParams, result);

} else {
    cerr << "Unknown member function: \"" << memberFuncName << "\" " << endl;
    return NULL;
} // end if
return rv;
} // end Execute....

---

TypesImpl-pqueue.txt

PriorityType: int
ElemType: char
EntryPointer: pointer to (struct char int pointer to struct)
APPENDIX E  COLLECTION OF VALUES

The classes specified here support value generation in the Class Validation System.

Model-based Value Generation

These classes support Model-based value generation in the Abstract Test System. SetOfVal.h contains the method invoked by an ATT to generated values from the type defined in the model. ValueTable.h specifies the collection used to maintain the collection of values.

SetOfVal.h

```
// $Header: SetOfValues.h,v 1.4 99/12/05 17:35:57 gurski Exp $

#ifndef _SETOFVALUES_H
#define _SETOFVALUES_H

static char id_SetOfValues_h[] = "0(«)$Header: SetOfValues.h,v 1.4 99/12/05 17:35:57 gurski Exp $";

#include "ValueType.h"
#include <String.h>
#include <List.h>

ValueType* GenRandomVals(String signature, int depth, int breadth) {

/* preA: signature is of the form:
**   - [name:]dataMember.1 (, dataMember.1)*
**   depth > 0 \ breadth > 0
**
** postA: result->typeName = name or ""
**   result->type = dataMember.1 (, dataMember.1)*
**   result->value = set of values of type result->type
*/
// The following functions are useful to other modules:

List<String> GetTupleFields(String tupSig);
String GetNextType(String type);

ValueTable.h

// ValueTable.h

ifndef_.VALUE.TABLE.H
#define_.VALUE.TABLE.H

#include <iostream.h>
#include "String.h"

struct bucketEntry {
    int size;
    String value;
    bucketEntry* next;
};//end struct

#ifdef _GARBAGE_COLLECT
#include "gc.cpp.h"
#endif

#ifdef _GARBAGE_COLLECT
class ValueTable : public gc {
#else
class ValueTable {
#endif

/* model
** uses T - specification only
** domains
**
** data members
**
** string theType
** set of T theSet
** sequence of T iteratedVals
**
** abstract functions
**
** define sequenceToSet(sequence of T theSeq) as set of T such that
**
** \( \forall(T x) \[ x \in theSeq \Rightarrow x \in \text{sequenceToSet}(theSeq) \] /\ \forall(T x) \[ x \in \text{sequenceToSet}(theSeq) \Rightarrow x \in \text{theSeq} \] **
**
** define containsSameVals(sequence of T theSeq, set of T theSet) as boolean such that
**
** \( \text{containsSameVals}(theSeq, theSet) = \text{sequenceToSet}(theSeq) = theSet \)
**
** constraints
**
** \( \text{sequenceToSet}(\text{iteratedVals}) \subset \text{theSet} \land \)
**
** \( \forall(\text{int} i) \[ i < j \land i \leq \text{length}(\text{iteratedVals}) \Rightarrow \)
**
** \( \forall(\text{int} j) \[ i != j \land i < j \leq \text{length}(\text{iteratedVals}) \Rightarrow \)
**
**
** - /\ informally:
**
** \( \forall(T x) \[ x \in \text{theSet} \Rightarrow x \text{ ofType theType} \] **
**
** operations
*/

public:

ValueTable(String type);
/* modifies: self
** postA: theType' = type \land
** theSet' = { } \land iteratedVals' = [ ] */

ValueTable();
/* modifies: self
** postA: self' = trashed */

ValueTable(const ValueTable& other);
/* modifies: self
** postA: theSet' = other.theSet \land theType' = other.theType \land
** iteratedVals' = [ ] */
ValueTable& operator=(const ValueTable& vt);
/* modifies: self
** postA: theSet' = other.theSet \& \ theType' = other.theType \&
**        iteratedVals' = [ ] \& result = self'
*/

void clearTable();
/* modifies: self
** postA: theSet' = { } \& iteratedVals' = [ ]
*/

String getTypeName() const;
/* postA: result = theType
*/

void insert(String s, int length = -1);
/* modifies: self
** postA: theSet' = theSet \union { s }
** - length is optionally used to indicate the length of the
** - value represented by s. This is used to speed up the
** - time it takes to check for duplicates
*/

void beginIteration();
/* modifies: self
** postA: \length(iteratedVals) = 1 \&
**         \exists(T x)[x \in theSet \& iteratedVals'[1] = x]
*/

int currentSize() const;
/* postA: ((theType = "int" || theType = "real" ||
**          theType = "boolean" || theType = "char") => result = 0) \&
**          ((theType = "set") =>
**           result = |iteratedVals|\length(iteratedVals)|) \&
**          ((theType = "tuple") => result = -1) \&
**          ((theType = "sequence" || theType = "string") =>
**           result = \length(iteratedVals|\length(iteratedVals|)))
*/
String currentValue() const;
/* preA: iteratedVals != [] */
** postA: result = iteratedVals[length(iteratedVals)] */

void next();
/* preA: ! containsSameVais(iteratedVals, theSet) */
** modifies: iteratedVals */
** postA: length(iteratedVals') = length(iteratedVals) + 1 /
** \forall i : \langle i < length(iteratedVals) \rangle \rightarrow
** \forall j : \langle i <= j < length(iteratedVals) \rangle \rightarrow
** iteratedVals[i] = iteratedVals'[j] /
** \exists x \in theSet \land ! (x \in iteratedVals) \land
** iteratedVals'[length(iteratedVals')] = x */

void removeCurrent(); // sets cursor at next element.
/* preA: iteratedVals != [] */
** modifies: self */
** postA: theSet' = theSet - \{ iteratedVals[length(iteratedVals)] \} /
** iteratedVals' = header(iteratedVals) */

bool doneIterating() const;
/* postA: result = containsSameVais(iteratedVals, theSet) */

int tableSize() const;
/* postA: result = | theSet| */

void printTable() const:
/* modifies: cout */
** postA: cout' = cout | | ToString(theSet) */

private:

const int HASHSIZE = 13;
bucketEntry* hashTable[HASHSIZE];
int currentBucketIndex;
bucketEntry* currentElement;
String typeName;
int hashValue(String myString);

} // end class ValueTable

#endif
Method-based Value Generation

As described in Chapter 5, the classes CompoundValue and CompoundValueSet manage the values generated using Method-based value generation.

CompoundValue.pre

```c
// $Header: CompoundValue.pre,v 1.3 98/02/21 11:08:17 gurski Exp$

struct ConcreteType{
    void    *value;
    StringArray history;
    bool     iterated;
    ConcreteType* next;
};
```

CompoundValue.h

```c
// $Header: CompoundValue.h,v 1.7 98/04/12 13:43:28 gurski Exp$

#ifndef CompoundValue_H
#define CompoundValue_H
static char id_CompoundValue_H[] = "0(#$Header: CompoundValue.h,v 1.7 98/04/12 13:43:28 gurski Exp $";

#include "Array.h"
#include "String.h"
#include <iostream.h>

typedef Array<String> StringArray;

#include "CompoundValue.pre"

#ifdef _GARBAGE_COLLECT
class CompoundValue: public gc {
#else

class CompoundValue {

}
**
**
** constraints
**
** \forall(ConcreteInstanceType c) \ (c \in ConcreteValues) \Rightarrow
**
** RepMap(value.c(c)) = AbstractValue \land
**
** \forall(ConcreteInstanceType c2) \ (c2 \in ConcreteValues) \Rightarrow
**
** (c = c2) \lor (history(c) \neq history(c2)) \lor \land
**
** IteratedValues \subset ConcreteValues
**
**
** operations
*/

public:

CompoundValue(const String& value.a):

/* modifies: self
** postA: AbstractValue' = value.a \land
** ConcreteValues' = { } \land
** IteratedValues' = { }
*/

CompoundValue(const String& val.a, void* val.c, const StringArray& h):

/* preA: RepMap(val.c) = val.a
** modifies: self
** postA: AbstractValue' = val.a \land
** ConcreteValues' = { (val.c, h) } \land
** IteratedValues' = { }
*/

CompoundValue(const CompoundValue& cv):

/* modifies: self
** postA: AbstractValue' = cv.AbstractValue \land
** ConcreteValues' = cv.ConcreteValues \land - note: copy of pointers
** IteratedValues' = cv.IteratedValues
*/

~CompoundValue():

/* modifies: self
** postA: trashed(self)
*/
void Merge(const CompoundValued& cv):
  /* preA: AbstractValue = cv.AbstractValue
       ** modifies: self
       ** postA: ConcreteValues' = UnionOnKey(ConcreteValues, cv.ConcreteValues)
   */

int LongestHistory() const;
  /* postA: result = LongestHistory(ConcreteValues)
   */

int CardinalityConcrete() const;
  /* postA: result = | ConcreteValues | 
   */

void BeginIteration();
  /* modifies: self
     ** postA: IteratedValues' = { }
   */

bool DoneIterating() const;
  /* postA: result = (IteratedValues = ConcreteValues)
   */

void GetNext(void*& currentValue, StringArray& currentHistory);
  /* preA: (IteratedValue ! = ConcreteValue)
     ** modifies: self
     ** postA: \exists(ConcreteInstanceType c) [ (c \in ConcreteValues) \land
               **       (c \notin IteratedValues) \land currentValue' = value_c(c) \land
               **       currentHistory' = history(c) \land
               **       IteratedValues' = IteratedValues \union \{ c \}]
   */

String GetAbstractValue() const;
  /* postA: result = AbstractValue
   */

friend ostream& operator << (ostream& out, const CompoundValue& cv);
  /* modifies: out
     ** postA: write (*AbstractValue) and list of histories to out
   */
include "CompoundValue.pri"
}
}; // end CompoundValue
#endif

CompoundValue.pri

// $Header: CompoundValue.pri,v 1.3 98/04/12 13:45:36 gurski Exp $
private:

String absValue;
ConcreteType* concreteValues;

// Private Functions

bool ValueInList(ConcreteType* cv) const:
/* postA: result = lsThere(ConcreteValues, (cv.value, cv.history)) */

CompoundValueSet.pre

// $Header: CompoundValueSet.pre,v 1.2 98/02/21 11:54:15 gurski Exp $

struct cvSetNode {
    CompoundValue* value;
    cvSetNode* next;
} ; // end struct

CompoundValueSet.h

// $Header: CompoundValueSet.h,v 1.10 98/04/19 09:30:13 gurski Exp $

#ifndef CompoundValueSet_H
#define CompoundValueSet_H

static char id_OpSet_H[] = "$Header: CompoundValueSet.h,v 1.10 98/04/19 09:30:13 gurski Exp $";

#include "CompoundValue.h"
```cpp
#include <iostream.h>

#include "CompoundValueSet.pra"

#if defined _GARBAGE_COLLECT
class CompoundValueSet: public gc {
#else
class CompoundValueSet {
#endif

    /* model */
    /* uses CompoundValue */
    /* */
    /* data members */
    /* set of CompoundValues theSet */
    /* */
    /* abstract functions */
    /* define max(int x, int y) as int such that */
    /* (x >= y => max(x,y) = x) */
    /* (x < y => max(x,y) = y) */
    /* */
    /* define UniqueOnKey (CompoundValueSet s) as boolean such that */
    /* UniqueOnKey(s) = */
    /* \forall(CompoundValue el) [ (el \in s) => */
    /* \forall(CompoundValue e2) [ (e2 \in s) \land */
    /* (el != e2) => */
    /* (el.AbstractValue != e2.AbstractValue) ] */
    /* */
    /* define Contains(String abstractValue, CompoundValueSet s) */
    /* as boolean such that */
    /* Contains(abstractValue, s) = */
    /* \exists(CompoundValue e) [ e \in s \land */
    /* abstractValue = e.AbstractValue ] */
    /* */
    /* define SumCardinality(set of CompoundValue s) as int such that */
    /* (s != {} => */
    /* \exists(CompoundValue cv) [ (cv \in s) \land */
    /* SumCardinality(s) = i ConvetaeValues(cv) ] */
    /* SumCardinality(s - { cv } ) ] \land */
    /* (s = {} => SumCardinality(s) = 0) */
```

**
** define LongestHistory(set of CompoundValue s) as int such that
** (s != { } =>
** \exists(CompoundValue cv) ( (cv \in s) /\ 
** LongestHistory(s) = max(LongestHistory(cv),
** LongestHistory(s - { cv })) )
** (s = { } => LongestHistory(s) = 0)
**
** constraints
** UniqueOnKey(theSet)
**
**
** operations
**/

public:

CompoundValueSet():
 /* modifies: self
 ** postA: theSet' = {}
 */

CompoundValueSet(const CompoundValueSet& s):
 /* modifies: self
 ** postA: theSet' = s.theSet
 */

CompoundValueSet& operator = (const CompoundValueSet& s):
 /* modifies: self
 ** postA: theSet' = s.theSet /\ result = theSet'
 */

~CompoundValueSet():
 /* postA: trashed(self)
 */

void Insert(CompoundValue* e):
 /* preA: UniqueOnKey({e} \ union theSet)
 ** modifies: self
 **
void Update(CompoundValue* e);
/* modifies: self */
** postA: theSet' = \{ e \} \cup \text{theSet} */

void InsertHistory(const String& abstractValue,

    String h,

    void* ConcreteValue = NULL);
/* modifies: self */
** postA: (Contains(abstractValue, theSet) =>

    \exists (CompoundValue e) \in \text{theSet} /\$

    (*e.\text{AbstractValue} = ^{abstractValue} /\$

    \exists (CompoundValue new) [

    new.\text{AbstractValue} = abstractValue /\$

    new.\text{ConcreteValue} = AddOne(

    theSet' = \text{theSet} - \{ e \} \cup \text{union} */

CompoundValue* Lookup(const String& abstractValue) const;
/* preA: true */
** postA: \exists(ConcreteValue cv) \in \text{theSet} /\$

    cv.\text{AbstractValue} = ^{abstractValue} /\$

    result = &cv /\$

    \exists(ConcreteValue cv) \in \text{theSet} /\$

    cv.\text{AbstractValue} = abstractValue /\$

    result = NULL /\$
*/
bool isEmpty() const;
    /* postA: result = (theSet == {}) */
}

void MakeEmpty();
    /* modifies: theSet */
    /* postA: theSet' = {} */

int CardinalityAbstract() const;
    /* postA: result = | theSet | */

int CardinalityConcrete() const;
    /* postA: result = SumCardinality(theSet) */

int Depth() const;
    /* postA: result = LongestHistory(theSet) */

String GetRandomAbstractValue() const;
    /* postA: \exists(CompoundValue e) ( (e \in theSet) \&
    ** result = AbstractValue(e) ) */

void* GetRandomConcreteValue() const;
    /* postA: \exists(CompoundValue e) ( (e \in theSet) \&
    ** \exists(ConcreteInstanceType val) ( val \in ConcreteValues(e) \&
    ** result = value_c(val) ) */

void PrintSet(ostream& out = cout) const:
    void BeginIteration();
    bool DoneIterating();
    CompoundValue* GetNextCompoundValue();

    void Union(const CompoundValueSet& s);
private:

friend cvSetNode* makeNode(CompoundValue* data,
    cvSetNode* nextOne = NULL);

cvSetNode* head;

cvSetNode* current;

friend void PrintInfo(const CompoundValueSet& s,
    const String& name = "");
BIBLIOGRAPHY


[27] Gail C. Murphy, Paul Townsend, and Pok Sze Wong. Experiences with cluster and class testing. 

July 1996.


