National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

ROLE OF LARCH/C++ SPECIFICATIONS IN BLACK-BOX TESTING OF OBJECT-ORIENTED SOFTWARE

Alicja B. Celer

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

March 1995

ISBN 0-612-01362-6

Canada

Abstract

Role of Larch/C++ Specifications in Black-Box Testing of Object-Oriented Software

Alicja B. Celer

The reusability of software components provides the basis for fast and efficient software development. Software reuse is most effective when it is conducted in black-box fashion. Object-oriented paradigm by polymorphism, inheritance, and encapsulation allows for reuse of software components in such a fashion. However, to be reusable, an object-oriented software component has to have a precisely defined interface and functionalities, and be thoroughly tested. To satisfactorily validate and verify an implemented software, a clear and unambiguous definition of its expected behaviour must be given. Informal specifications, which use common everyday language, are ambiguous and incomplete. Formal specifications, which are based on mathematical abstractions and have precise semantics, provide precise definitions of functionalities, properties and the interface of a software component. However, to support reuse of the software module, formal specifications have to be correct with respect to the requirements and be complete. The same criteria applies while they are used to support testing of the implementation against their specifications. For implementation testing, not all formal specification languages are appropriate. The most suitable ones are those which efficiently support test case preparation, test data derivation, and test result evaluation.

The research, presented in the thesis, concentrates around the problems related to the use of formal specifications written in Larch/C++ for testing software implementation in a black-box fashion. Consequently, a methodology of deriving a Finite State Automaton (FSA) from the formal specifications, written in Larch/C++, was created. Larch/C++ has a two tiered structure: (1) in the LSL layer, the abstract data type (distinguished sort) is defined; while (2) in the interface layer, implementation language requirements are incorporated. Each LSL trait used in the interface specification layer has to be analyzed for completeness. LSL traits analysis provides

the *basic distinguishable domains* of the distinguished sort. Then, based on the domain partitions obtained, the states and valid transitions of the FSA are derived from the interface specifications. FSA derivation is based on Disjunctive Normal Form (DNF) analysis. The methodology also incorporates class invariant derivation and its analysis.

The methodology can be applied to all types of C++ classes. Besides creation of a FSA, the analysis conducted in accordance with the methodology provides systematic information about exceptional cases and behavioural conditions of a class. The methodology was tested on several examples of academic interest, as well as on several classes from Rogue Wave Tools.h++, the commercial library of object-oriented classes. An overview of the tool which implements the methodology is also presented in the thesis.

Mamie i Tacie

Acknowledgments

My greatest acknowledgment is extended to my supervisor Dr. V.S. Alagar. His academic insights and wisdom were instrumental in the successful completion of this thesis. Equally of importance, his patience and humility allowed this research to be a unique experience. I am also grateful to Ilya Umansky for being such a good research companion and friend.

I would like to thank all my friends, who stood by me in my moments of despair, when the work seemed as too much, and home too far, by giving me advice, and supporting me throughout the duration of my studies. I would especially like to mention my 'Canadian Mom' Mrs. Heather Keith-Ryan, Halina Zerko, Gabrielle Kocken, and Carole Biondic.

Finally, to Salem Bouhairie for his love, sacrifices, moral support, and patience to put up with my every whim during all the time that I spent working on the degree.

Last but never the least (in fact, the most), I would like to thank my whole family in Poland for being with me, loving me, and believing in me. Without their continuous support and encouragement, I would not be able to complete my work.

Life is brutal and full of 'zasadzkas'.

Contents

List of Tables xi

List of Figures xii

1 Introduction 1

2 Software Reuse and Motivation for the Thesis 4

2.1 Reuse 4

2.2 Formal Specifications and Reuse 5

2.3 Testing and Reuse 5

2.4 Choice of Specification Language 6

2.5 Scope of Thesis 7

3 Larch/C++ - Quick overview 9

4 Testing Object-Oriented Software 14

4.1 Object Oriented Paradigm 15

4.2 Testing Issues 16

4.2.1 Levels of Testing 18

4.2.2 Errors 19

4.3 Classes as State Machines 20

4.3.1 Coverage Strategies 22

4.3.2 A Hierarchy of Coverage Strategies 24

5 Role of Formal Specifications in Testing 26

5.1 Role of Formal Specifications in the Software Development Process . 26

5.2 Black-Box Testing 28

5.3 Advantages of Using Formal Specifications in Testing 28
5.4 Problems with Using Formal Specifications for Testing Software . . . 30

6 Formal Specifications - Relevant Work **32**
6.1 Formal Specification Languages 32
6.2 Testing Issues 35
6.2.1 Defining Tests 35
6.2.2 Sequencing Tests 36
6.2.3 Test Oracles 36
6.2.4 Selecting Test Data 37
6.2.5 Evaluating Tests 39
6.2.6 Tool Support 41
6.2.7 Representation Mappings 42
6.3 Dick and Faivre Methodology 43

7 Error Detection and Formal Specifications **46**
7.1 Limitations and Advantages in Error Detecting 46
7.2 Larch/C++ and Testing - Errors 48

8 Creation of Finite State Automaton **50**
8.1 Methodology when Applied to VDM 51
8.2 Terminology 53
8.3 Adaptability of Methodology 55
8.4 Class Invariant 57
8.5 Derivation of Information from LSL Traits 59
8.5.1 Taxonomy of LSL Operations 60
8.5.2 Analysis of LSL Primitives 61
8.5.3 Proposed Methodology of LSL Analysis 63
8.6 Derivation of Information from Interface Specifications 65
8.7 Algorithm of Finite State Automaton Derivation from Larch/C++ . 67
8.8 Examples to Explain Methodology 69
8.8.1 Stack 69
8.8.2 LimitedStack 79
8.8.3 Screen 82
8.9 Critique of the Methodology 91

9 Validating Correctness of Tools.h++ Library **94**
9.1 Rogue Wave Tools.h++ Library 94
9.2 Analysis of RWFile Class 95
9.2.1 File.lsl Trait Analysis 96
9.2.2 Interface Specification Analysis 105
9.3 Analysis of BtreeOnDisk Class 111
9.3.1 Btree.lsl Trait Analysis 112
9.3.2 BtreeAux.lsl Trait Analysis 117
9.3.3 FileManager.lsl Trait Analysis 118
9.3.4 BtreOnDisk.lsl Trait Analysis 119
9.3.5 Interface Specification Analysis 123
9.4 Analysis of RWHashTable Class 129
9.4.1 HashTable_V.lsl Trait Analysis 129
9.4.2 HashTable.lsl Trait Analysis 134
9.4.3 Interface Specification Analysis 135
9.5 Sample Tests for RWFile Class 139
9.6 Concluding Remarks 143

10 Concluding Remarks and Future Research **144**
10.1 Summary 144
10.2 Future Work 146
10.3 An Overview of the Tool Based on the Proposed Methodology 146
10.3.1 Subsystems Decomposition 148
10.3.2 Data Dictionary 155
10.3.3 Human Assistance 158

Bibliography **161**

A Analysis of Selected LSL Traits **169**
A.1 SetBasic.lsl Trait 169
A.2 Set.lsl Trait 171
A.3 BagBasics.lsl Trait 174
A.4 Bag.lsl Trait 175
A.5 Array.lsl Trait 178
A.6 Deque.lsl Trait 180

B RWFile Class Tests 184

List of Tables

1 DST: Data Dictionary 155
2 DST: Data Dictionary cont. 156
3 DST: Data Dictionary cont. 157
4 DST: Data Dictionary cont. 158

List of Figures

1 `SetTrait.lsl` definition 12
2 Interface specification for `IntSet` class 13
3 `StackTrait.lsl` definition 70
4 Interface specification of `IntStack` class 73
5 Finite State Automaton for `IntStack` class 78
6 `LStackTrait.lsl` definition 81
7 `Screen.lsl` definition 82
8 Interface specification of `Screen` class 84
9 Finite State Automaton for `Screen` class 90
10 `File.lsl` definition 97
11 All Valid and not valid states for `File` trait 104
12 FSA for `RWFile` class, open for READ_WRITE or WRITE_ONLY . . 109
13 FSA for `RWFile` class, open for READ_ONLY 110
14 `Btree.lsl` definition 114
15 `BtreeAux.lsl` definition 117
16 `FileManager.lsl` definition 118
17 `BtreeOnDisk.lsl` definition 120
18 Valid states for `Btree` trait 122
19 Finite State Automaton for `RWBtreeOnDisk` class 127
20 FSA for `RWBtreeOnDisk` class: Initial States 128
21 `HashTable_V.lsl` definition 131
22 `HashTable.lsl` definition 134
23 FSA for `RWHashTable` class 138
24 DST: System decomposition 153
25 DST: Architectural Design 154
26 `BasicSet.lsl` definition 170

27 `Set.lsl` definition 172
28 `BagBasics.lsl` definition 174
29 `Bag.lsl` definition 176
30 `Array.lsl` definition 178
31 `Deque.lsl` definition 181

Chapter 1

Introduction

Software development benefits from the reuse of well tested software components. Reuse provides a means of achieving economy, in time and resources. However, there are a number of requirements which software components must fulfill to be reusable. Among them are the following: software behaviour and properties should be clearly specified; its implementations should be throughly tested; and any modifications to the internal structure of a software module should not influence its interface and behaviour.

The reuse of a software component may be enhanced, if its properties are clearly specified. Formal specifications provide unambiguous and precise definition of the component's behaviour and interface, subsequently, allowing software reuse [Col93, Wing90]. Additionally, formal specifications ensure correctness; that is, it is possible to develop a methodology to determine if an implementation of the component corresponds to its formal specification.

The dependability of software largely rests on its reliability. The reliability of a developed software can be improved by testing implemented ideas. It is not only important that the software works correctly (software verification), but also that the implementation reflects what it was intended, and later designed to do (software validation). To validate and verify an implementation, developers and designers need tools and theory. Formal specifications, by nature, exclude casual informalism from the software design. The existence of formal specifications has a powerful influence in the software life-cycle. Its influence is evident in the requirement specification phase, design phase, and less evident in all other phases of the software development process.

This thesis work was motivated by a growing interest to use formal specifications in all phases of the software development process, as well as to find ways of stating the proper behaviour of reusable software components.

The language chosen to examine the possibility of supporting the use of formal specifications of software module for reuse, is Larch/C++. Since object-oriented methodology, by definition, supports reuse, the choice of formal specification language was motivated by its ability to follow an object-oriented methodology. In addition to supporting object-orientation, Larch/C++, by its two tier approach and separation of abstraction of abstract data types from implementation language requirements, provides an interesting test-bed for our research. During the first phase of the research work, formal specifications in Larch/C++ were written for several classes from Rogue Wave Tools.h++ commercial library of C++ classes [rogue]. That work was conducted to provide sufficient experience in writing formal specifications, and to create a number of formally specified classes for testing the proposed methodology in the research. The results of that work can be found in [ACCUA94].

Issues related to testing object-oriented software and the use of formal specifications for testing an implementation are analyzed, and presented in the thesis, since they form the basis for our further analysis. The main contribution of this thesis is the methodology of Finite State Automaton derivation, based on Larch/C++ formal specifications of object-oriented classes. For that, both layers of Larch/C++ specifications are analyzed. From an analysis of LSL layer, initial partition of input/output domain is obtained. Then, based on that information, the states and valid transitions of the FSA are derived from interface specifications. FSA derivation is based on Disjunctive Normal Form (DNF) analysis. The methodology incorporates also class invariant derivation and analysis. The methodology is illustrated with a number of examples from Rogue Wave commercial library of C++ classes.

This thesis addresses only one part of a larger and on-going research on black-box reuse of object-oriented software: testing an implementation based on formal specifications. This research is sponsored by BNR-NSERC under a Collaborative Research and Development Grant to Dr. V.S. Alagar. There are a number of students working on different aspects: for example, Ilya Umansky, with whom I collaborated to develop Larch/C++ specifications of Rogue Wave class libraries [rogue], is just completing his thesis devoted to another part of the project (algorithmical testing of the completeness criteria developed by Pierro Colagrosso [Col93]); there are four other

students who are writing specifications of those classes from [rogue] not included in [ACCUA94]. They plan to build several tools based upon the work of this group to make software reuse feasible in the context of commercially used C++ class libraries. Thus, my thesis is just a small contribution to the overall goals; yet, the results of this thesis are quite significant: this is the first attempt, to my knowledge, to provide a sound basis for test based on Larch/C++ specifications. It is also my hope and wish that my results will be used and tested widely by the other members of the group in their research work.

The organization of the remainder of this thesis is as follows. Chapter 2, presents the motivation and the scope for the thesis research. A brief overview of Larch/C++ is given in Chapter 3. In Chapter 4, the issues related to testing object oriented software are addressed. Chapter 5 discusses the role of formal specifications in testing process. Chapter 6 provides commentary on the results of ongoing research in the application of formal specifications to test the implementation. Chapter 7 is an analysis of the error detection power of formal specifications; in particular, we emphasize the error detection capabilities in Larch/C++. Chapter 8 shows the methodology in applying Larch/C++ specifications for FSA derivation. That methodology is illustrated by applying it to several formal specifications. Chapter 9 contains the results of application of the proposed methodology to a number of classes chosen from commercial library of C++ classes [rogue]. Chapter 10 summarizes the work done during this phase of research and identifies some future work. Appendix A contains the analysis of LSL traits conducted according to presented methodology. Appendix B contains the listing of the tests conducted for RWFile class.

Chapter 2

Software Reuse and Motivation for the Thesis

This chapter surveys the critical issues involved with reuse and testing object oriented modules in relation to formal specifications. The motivation behind the research and a description of the scope of this thesis is also given.

In Section 2.1, the concept of software reuse is presented. In Section 2.2, the advantages and role of formal specifications in software development process is introduced. Section 2.3 contains the introduction of the issues related to testing reusable components. The choice of specification language is discussed in Section 2.4. In Section 2.5, the scope of the thesis is defined and the issues which will be researched are presented.

2.1 Reuse

One of the biggest dreams of software designers and implementors is to be able to compose the software systems from reusable components. This would not only speed-up a development process, but also assure better reliability of the final product. However, to reach the point where the new software system will be created from an existing collection of reusable modules, a methodology to create and determine reliability of these components (modules) has to be established.

For the software component to be reusable it has to be precisely specified, implementation independent, and complete with respect to specification. If programmers

are to reuse a software module effectively, the precise behaviour of the module should be understood with minimum effort. The most promising form of the reuse of software modules is black-box reuse. In black-box reuse, from the point of view of the user, the implementation details are hidden where only the services provided are of concern.

The goals of an object-oriented system design usually include reusability, extendibility, and reconfigurability. Object-oriented methodology by use of data abstraction, encapsulation, inheritance and polymorphism is most suitable for the creation of reusable components.

2.2 Formal Specifications and Reuse

The software development process heavily relies on the correctness of software specifications. Hence formal specifications, which are based on mathematical apparatus, are gaining popularity. The formally specified software component/system, can be validated before any implementation is written. Validation can be achieved using either theorem provers to prove their properties, or by executing the specifications in order to verify the designed functionality of the software module.

Formal specifications allow for reuse of a software component by precisely specifying its interface. It is possible to reuse the module by knowing only its formal specifications. In [Col93], the issue of completeness of formal specifications to support black-box reuse of software component was analyzed. In the case of specifications written in Larch/C++, the information contained in the interface layer, in the form of pre- and post-conditions, provides sufficient information about a module's functionality. However, as mentioned in the thesis, to fully support reuse the specifications have to be validated for completeness and correctness.

2.3 Testing and Reuse

The reusable component should be thoroughly tested before it is used. The testing would assure the proper intended behaviour of the module.

In addition, the reusable component needs to be tested in a new environment. Therefore, its specifications should provide the necessary information which can be

used in the test derivation process. There are several methods of testing software components and several levels of testing incorporated into a testing process. More detailed information on testing object-oriented modules is included in Chapter 4.

Since formal specifications provide an unambiguous and precise definition of the software module behaviour and its interface, they can also be helpful while assessing the correctness of an implementation of the module. The use of formal specifications in a testing process is shown to be more and more beneficial [Wing90, HB94a]. However, this area of software engineering has not received as much attention as it deserves. A detailed analysis of these aspects of software testing are presented in Chapters 5, 6, and 7.

2.4 Choice of Specification Language

This research concentrates on the reuse of object-oriented components in a black-box fashion. Therefore, a chosen formal specification language would have to support that design paradigm. There are several formal specification languages which have been proposed for the specification of object-oriented modules; among them are: Fresco [Wil91], Eiffel [Mey88], Anna [Luc91], A++ [CL90a], Larch/C++ [LC92], Larch/M3 [Jon91].

Larch/C++ is the most suitable for research on black-box reuse for the following reasons: (1) Larch/C++ can be used to write implementation independent specifications while providing a sufficiently expressive language; (2) it provides built-in syntactic and semantic support for specifying C++ class interfaces; (3) written specifications can be shorter than those written in any *universal* specification languages, e.g. VDM [Jon90]; (4) the two-tiered approach of Larch/C++ allows for the separation of concerns. In addition, Larch/C++ was chosen due to the lack of sufficient research done in its use for testing purposes.

One of the drawbacks, from the point of view of automating the test suite preparation process, is the lack of a parser for Larch/C++. However, the creators of the language [GHW85,GH93] plan to make it available in future.

A brief overview of the general characteristics and primitives of Larch/C++ is given in Chapter 3.

2.5 Scope of Thesis

The growing popularity of reuse of software components, in particular black-box reuse, in the software development process has prompted this presented research. In particular, the main reasons for the research are the following: (1) the growing popularity of object-oriented approach, (2) the embedded/intended reusability of software components created according to an object-oriented approach, (3) the reliability required for reuse, (4) testing as a method to validate the correctness and reliability of software component, (5) the use of formal specifications for designing software components, (6) the lack of tools and research to support the testing issues of reusable software components; in particular, using their formal specifications.

The goals to achieve during this research were the following:

- Explore the ways to verify the correctness of the implementation of software system/module with respect to their formal interface specifications; especially, for the purpose of testing using formal specifications. Demonstrate the use and importance of formal specifications in the software development process, especially test preparation and execution.

- Analyze the adequacy of specifications for supporting black box reuse, and identify the types of errors which can be found using formal specifications. Analyze the advantages and possible inadequacy of formal specifications with respect to testing.

- Develop a methodology for verifying/testing the correctness of Larch/C++ class specifications vs. its implementation; in particular, the algorithm for derivation of a Finite State Automaton (FSA) from Larch/C++ specifications.

- Validate the methodology by using it to test representative classes from Rogue Wave Tools.h++ library [rogue]. Based on the work done while formally specifying C++ classes from Rogue Wave Tools.h++ library, give the examples of the implementation of the proposed methodology to test several classes from the library.

The main research presented in the thesis is related to the use of Larch/C++ specifications for test suite preparation. At this phase of the research, only class level testing is analyzed. However, the other members of the Black-Box Research Group are studying the other testing aspects of object-oriented paradigm, such as inheritance and sub-typing.

Chapter 3

Larch/C++ - Quick overview

Larch/C++ [LC92, LC94] is a part of the Larch family of specification languages. Larch languages are formal specification languages geared towards the specification of the observable effects of program modules, particularly modules which implement abstract data type. Larch provides a two-tiered approach to specification:

- In one tier, a Larch Interface Language (LIL) is used to describe the semantics of a program module written in a particular programming language. LIL specifications provide the information needed to understand and use a module interface. LIL doesn't refer to a single specification language but to a family of specification languages. Each specification language in the LIL family is designed for a specific programming language.

  LIL specifications are used to specify the abstract state transformations resulting from the invocation of the operations of a module. These specifications are written in a predicative language using pre- and post-conditions.

- In the other tier, the Larch Shared Language (LSL) is used to specify state-independent, mathematical abstractions which can be referred to in LIL specifications. These underlying abstractions, called *traits*, are written in the style of an equational algebraic specification.

  LSL is programming language independent and is shared by all LILs.

Larch/C++ is an interface specification language for specifying C++ classes and functions. The restriction to C++ allows Larch/C++ to have a syntax and semantics that is tailored to C++; for example, the Larch/C++ specification of a C++ function

specifies not only the behavior of the function, but exactly how that function is called from C++ code. The details of how to call a C++ function, the name, return type, and argument types, are called the interface of the function.

Interface specifications rely on definitions from auxiliary specifications, written in LSL, to provide a semantics for the primitive terms they use. Larch encourages a separation of concerns, with basic constructs in the LSL tier and programming details in the interface tier.

Functions are specified in Larch/C++ using Hoare-style pre- and post- conditions. The header of a function specification is the same as that of C++ function definitions. The body describes the effect of function invocation using a pair of predicates following the keywords **requires** and **ensures**. The predicate following **requires** is a pre-condition that must be satisfied to invoke the specified function. The predicate following **ensures** is a post-condition that the specified function establishes upon termination. The notations '$\wedge$' and '$\vee$' denote conjunction and logical disjunction respectively. All the logic notations such as $\forall$ - universal quantifier, $\exists$ - existential quantifier are also valid. The semantics of function specification is that the pre-condition of the state transformation must logically imply the post-condition of the state transformation. For functions that change the values of objects, the body of the function specification must include a **modifies** clause. Only objects listed in the **modifies** clause are allowed to change their values as the result of function invocation. In a function that mutates an object or a variable, there are two different values for the same object; the value in a pre-state and the one in the post-state. The value of the object in the pre-state is denoted by a hat-ed (^) identifier, while the post-state value is represented by a primed (') identifier. If neither of (^) nor (') is used with the object name then the object itself is considered as a memory location and not the object value.

The syntax for data members and member functions in interface specifications are almost the same as in a C++ program. The Larch/C++ reserved word *this* is used in the member function specifications and means the same thing as the C++ reserved word *this*, a pointer to the object of the specified class. The Larch/C++ reserved word *self* is a shorthand for $*(this \backslash any)$. The suffix *any* is like (') or (^), and extracts the value of *this* in some visible state. As in C++, Larch/C++ member functions can be public, protected, and private.

Figure 2 shows a Larch/C++ class interface specification for the class Set. The **uses** clause indicates that the Larch/C++ interface is expressed with the vocabulary of the LSL trait `SetTrait` (see Figure 1). The trait `SetTrait` defines the terms used to denote abstract values of the set as well as the mathematical properties of the set. All the terms in the pre- and post- conditions of the function specifications come from this trait. The type-to-sort mapping, which is given between the parentheses following the names of the used trait, says that the abstract values of the C++ `Set` objects are specified to be those of the LSL sort C in `SetTrait`. (The LSL sort is the "type" of an LSL term; the word type is used only to refer to C++ types). The type to sort mapping makes the connection between the C++ world and the LSL (mathematical) world. The abstract values of Set objects are denoted by equivalence classes of LSL terms of sort C from `SetTrait`.

Figure 2 specifies a constructor, a destructor, and three public member functions: *insert, delete* and *isEmpty.* The destructor uses the Larch/C++ reserved word **trashed** to state that the object **self** is no longer available. The terms *insert, isEmpty* and *delete* which appear in the pre- and post-conditions refer to the LSL operators, not to the member functions having the same name. All C++ declarations are legal in Larch/C++ interface specifications; for example, member functions can be virtual, static, friend, or inline; they all have their C++ meaning. The Larch/C++ keyword **result** can only be used in post-conditions and it denotes the function return value. The sort of **result** is the sort associated with the return type specified for the function. For example, in the interface specification for class `IntSet` the member function **isEmpty** returns sort Bool.

$SetTrait(E, C)$: **trait**

% Essential finite-set operators

introduces

$\{\} :\rightarrow C$

$insert : E, C \rightarrow C$

$delete : E, C \rightarrow C$

$\_\_ \in \_\_ : E, C \rightarrow Bool$

$isEmpty : C \rightarrow Bool$

asserts

C **generated by** $\{\}, insert$

C **partitioned by** $\in$

$\forall\ s : C, e, e_1, e_2 : E$

$\neg(e \in \{\})$

$e_1 \in insert(e_2, s) == e_1 = e_2 \vee e_1 \in s$

$isEmpty(\{\})$

$\neg isEmpty(insert(e, s))$

$e \in s \Rightarrow \neg isEmpty(s)$

$delete(e, \{\}) == \{\}$

$delete(e_1, insert(e_2, s)) ==$ **if** $e_1 = e_2$ **then** s

else $insert(e_2, delete(e_1, s))$

Figure 1: `SetTrait.lsl` definition

```
class IntSet
{
    uses SetTrait(IntSet for C, int for E);
 public:
    IntSet()
    {
            modifies self;
            ensures self' = empty;
    }
    ~IntSet()
    {
            modifies self;
            ensures trashed(self);
    }
    IntSet& insert (int i)
    {
            modifies self;
            ensures self' = insert(self^, i) ∧ result = self;
    }
    IntSet& delete (int i)
    {
            modifies self;
            ensures delete(self^, i) = self' ∧ result = self;
    }
    Bool isEmpty ()
    {
            ensures if isEmpty(self^) then result = TRUE else result = FALSE;
    }
};
```

Figure 2: Interface specification for `IntSet` class

Chapter 4

Testing Object-Oriented Software

The overall goal of testing is to assure the proper functioning of a program. With testing, the only way to guarantee a program's correctness is to execute it for all possible inputs, which is usually impossible. Systematic testing techniques generate a representative set of test cases to provide test coverage of the program according to some selected criteria. There are two general forms of test case coverage: *specification-based* and *program-based.* In specification-based or black-box testing, test cases are generated to show that a program satisfies its functional and performance specifications. Specification-based test cases are usually developed only manually by considering a program's requirements. In program-based or white-box testing, the program's implementation is used to select test cases to exercise certain aspects of the code such as all its statements, branches, data dependencies or paths. Since specification-based and program-based testing complement each other, both types are usually used to test a program. The detailed information on black-box testing is discussed elsewhere [Woit92b].

The growing popularity of object-oriented methods in software development requires the development of adequate software validation techniques. Research shows that traditional testing techniques can be applied to test some aspects of object-oriented software. However, some of the characteristics of the object-oriented paradigm impose problems which should be approached differently.

One of the most promising approaches to assess the validity and correctness of the object-oriented software is representing it as a finite state machine. In this research, that approach is analyzed to establish the usability of formal specifications written in Larch/C++ for testing the implementation.

This chapter is organized as follows: Section 4.1 discusses the object-oriented paradigm. Section 4.2 presents problems related to testing object-oriented software. Section 4.3 presents the issues related to testing the class represented as a finite state machine.

4.1 Object Oriented Paradigm

The emergence of the object-oriented paradigm for designing and implementing software systems [Boo86] is a relatively new development in software engineering. Object-oriented design leads to software architectures based on objects every system and subsystem manipulates. The object-oriented approach involves (1) defining abstract data types (ADT), and (2) organizing software around the collection of ADTs with a view toward exploiting common features. This approach enables the developer to remain close to the conceptual, high level model of a real world problem, which is to be solved. In addition, the modularity of objects and the ability to implement a program as relatively independent units that are easy to maintain and extend are advantages of object-oriented design. The distinctive characteristics of object-oriented paradigm are the following:

- Data abstraction and encapsulation

  Data abstraction is a process of defining an abstract data type. An abstract data type is described by its external view: the available resources and properties of these service, without indicating concrete implementation representations.

- Inheritance and sub-typing

  Inheritance is a mechanism to define a new class in terms of an old one. Inheritance imposes a hierarchical relationship among classes in which a child class inherits data and behaviour from its parent.

- Polymorphism

  Polymorphism means a feature of having more than one form. In the context of object-oriented design, polymorphism refers to the fact that a single operation can have a different behaviour in different objects. In other words, different objects react differently to the same message. In object-oriented programming,

polymorphism is supported in two ways: (1) overloading, and (2) dynamic binding.

4.2 Testing Issues

The object-oriented approach differs from the traditional approaches to design and implementation. Hence the differences should be acknowledged and analyzed not only from the point of view of design and coding techniques, but also in software verification and testing. From the point of view of testing, the major differences between traditional and object-oriented software design methodologies are as follows:

Computational model

The object oriented approach assumes active data and a passive processor, while the traditional (Von Neumann) model uses passive data and active processor.

Testing techniques

Many traditional testing techniques can be applied for testing object-oriented software. However, the nature of object-oriented software puts additional requirements and restrictions on them. This follows from the fact that the objects and classes of the object-oriented model contain routines and possible data structures which describe the state of an object. In addition, in object-oriented models, there is no predefined order of invocation of operations after the object was created, and no sequential input, process, output method can be applied to test the software. Therefore, for an object-oriented model, the functional testing techniques are not applicable because there is no test set to run and similarly, the structural test techniques are not applicable. Consequently, there are many difficulties in analyzing control flow or data flow in an object-oriented model.

This research explores the possibility of applying formal specifications written in Larch/C++ to conduct specification-based (black-box) testing of a reusable software component.

Inheritance of classes

Inheritance introduces special testing concerns. In particular, not only do the redefined or introduced functions have to be tested, but also the inherited ones. Inherited functions can produce errors while interacting with a new environment.

Each lower level in an inheritance hierarchy is a new context for inherited features; the correct behaviour at an upper level in no way guarantees the correct behaviour at a lower level. Polymorphism with dynamic binding dramatically increases the number of possible execution paths. The static analysis of source code to identify paths is relatively of little help here. In addition, while limiting the scope of effect, encapsulation is an obstacle to the controllability and observability of an implementation state.

Many researchers point out the need to assume 'strict' inheritance [McGD93].

McGregor et al. [McGHK92] propose a framework HIT which provides a decision criteria for determining how test suites should be inherited from parent classes and for determining how much re-testing is needed at each level in an inheritance hierarchy. HIT provides a technique for constructing a minimal number of subclass test cases that are needed to supplement the test cases inherited from a parent class. HIT also shows how to identify those test cases that do not need to be rerun.

Reuse of classes

One of the main advantages of the object-oriented paradigm is the reuse of code. This introduces special requirements on the testing of reusable components. Components offered for reuse should be highly reliable; therefore, extensive testing is warranted when reuse is intended. However, each reuse is a new context of usage and, consequently, re-testing is prudent. Therefore, the tests for a reusable class C, must be repeated many times. This is a result of the demand placed by the object-oriented approach, where a subclass may inherit from a superclass, or be reused in a different software module. The three main levels of testing object-oriented classes are identified as follows: (1) testing the initial version of class C, (2) testing after each modification to class C, and (3) testing when class C is used in a new environment (e.g. a new operating system, compiler).

Cheatham and Mellings [CM90] state that inheritance makes reuse feasible by providing a mechanism to add new private data and new operations, and modify

existing operations. At the same time, the existing class is also treated as a black box. They suggest some solutions to the problem of multiple inheritance (inheritance-tree). Structural testing can be done for member functions and functional testing can be done against the requirements and interface description of the class. Testing benefits also occur through inheritance of fully tested classes.

According to Murphy et al. [MTW94] testing for reuse should involve more and different boundary conditions on arguments, and more and different sequences of method invocation.

4.2.1 Levels of Testing

The testing of code is organized around the recognized units of object-oriented design. While approaching the problem of testing, the level of abstraction should be taken into consideration. From the tester's point of view, there are several levels of abstraction:

- *algorithmic level* - tester considers code at the routine level.
- *class level* - tester considers interactions of routines and data encapsulated within a class. Class testing is the first level of integration testing. This level of testing is comparable to the unit test of the procedural approach.
- *cluster level* - tester considers interactions of groups of co-operating classes. A cluster represents a second level of integration. It is assumed that each class in a cluster was tested individually. The effects of polymorphism, dynamic binding, and object state on testing the interactions of classes are analyzed in [McD].
- *system level* - tester analyzes all the code in all the classes and main programs necessary to run an entire system. The system is tested functionally using test cases derived from the user defined cases and other system requirements. In general, system level testing of object-oriented systems has the same objectives as those for systems developed using other techniques. However, the process of developing test cases is somewhat different.

Regardless of the testing level, the goals of the employed testing algorithms remain the same. A testing algorithm should create a minimal number of test cases necessary to achieve a given level of test coverage. It should also, at any point in the development

process, execute a minimal number of test cases necessary to achieve a given adequacy criteria. The choice of the adequacy criteria should allow incremental increases in the level of testing coverage.

This research concentrates on the problems related to class level testing. The algorithmic level is assumed to be of concern for the implementor who tests the written procedures to ensure their correct behaviour. The cluster and system levels are left for future analysis.

4.2.2 Errors

The object-oriented routines may have two classes of errors [SR91], for which they should be tested: inter-routine errors, and intra-routine errors. The former are related to the interaction between classes, while the latter are related to the conceptual misunderstanding of the class behaviour. However, both classes of errors can be analyzed by the impact of the error on the software quality. The conceptual meaning of an error may differ from its actual cause.

In the case of inter-routine errors, the conceptual assumption is that routines may overlap in functionality. This may not initially seem to be an error, but it is certain that the programmer of the class would wish to identify it, since this increases the complexity and hence decreases the quality of the class. However, the actual cause of an error could be due to the fact that the order of interactions between routines may yield an error due to incorrect access to the state.

Similarly, intra-routine errors may have as a conceptual source, the knowledge that routines designed to perform a certain task may not perform the task correctly or may perform a slightly different task. This may or may not preserve the integrity of the state. Either way, the error can only be caught by an external oracle. However, in reality routines may not be implemented correctly and may have simple errors in them that either yield run-time error or corrupt the state of the object, so that error occurs upon invocation of further routines.

Usually, searching for an error is motivated by two factors: (1) which part of the object is under test, and (2) what sort of errors are searched for. The former is based on checking for integrity of the state, or checking for the correct operation of routines. The latter tries to uncover actual run-time errors in the code or produce errors in which implementation performs a task which yields no run-time error, due to

syntactically correct code and preservation of the integrity of the state (but program does not function according to specifications). A number of strategies have been incorporated for testing software in the framework designed by Robson et al. [SR]. The testing framework FOOT supports the following strategies: minimal, exhaustive, tester guided, inheritance testing, memory testing, data flow testing, identity testing, and set and examine.

McGregor [McG] proposes a systematic use of the components of the class to uncover the errors. That approach is based on the specifications of the methods. McGregor defines requirements for class specification which include the following three parts/aspects: (1) a specification for each method, (2) a statement of class invariant, and (3) a description of the dynamic behaviour of an object.

Fiedler [Fie89] points out that the following should also be analyzed: (1) boundary-values which dictate that tests are built to create objects of extremes; (2) test to invoke exception handling; (3) implicit and explicit invocation constructors and destructors; (4) consistency initialization and casting operations; and (5) application of associativity rules to member functions.

4.3 Classes as State Machines

The representation of a class as a Finite State Machine (FSM) 'borrows' from protocol testing techniques. This approach to testing can be adopted because the following holds: (1) objects have states; after object creation, each message either accesses or modifies that state; (2) an object is the target of a stream of messages occuring in a somewhat arbitrary order, and (3) in many cases each method can be activated in any state. The states of an object and the set of possible transitions among the states are useful means of representing partitions of the behaviour of object-oriented systems.

McGregor et. al. [McGD93] postulate that the important states are those which are meaningful, i.e. carry useful testing information. Therefore, in many cases the value of the state variable belongs to the set of values for one particular state.

A state machine is specified as a set of states, represented by particular combinations of attribute values and a set of transitions. The transitions may be labeled with names for the transitions, and defined by an input and output state. The specification

of a tested component includes a state space, S, and an event space, E, which together define the behaviour of the component. A transition space, T, assists in modeling the dynamics of how objects move from one state to another. The state must be based on the observable behaviour of the object. An object is designed to conceal the details of implementation and to present a behavioural interface to potential clients. The state should also be defined in terms of the values returned by the public methods of the object.

There are several restrictions on the FSM which is created for the class. McGregor and Dyer [McGD93] specify the following restrictions on FSM:

1. The behaviour of the object can be defined with a finite number of states.

2. There are a finite number of transitions between states.

3. All states are reachable from the initial state.

4. Each state is uniquely identifiable.

The loss of formalism is discussed with respect to the requirements for FSM [McGD93] as follows: (1) 'global' access to data of each method (i.e. state variables); (2) a class has to be fully connected (i.e. it contains the method to set the class in the initial state); (3) any method can be invoked in any state; (4) an object has to be able to respond to an event in any state, and (5) all states should be observable (i.e., the accessor returns state information or returns the values of modifiers which provide sufficient information to determine the state of an object).

However, the following restrictions apply to the interpretation of FSM:

- there is no assurance about having a finite automaton (FSA)

- multiple branching is allowed in the testing structure, for the same method and for the same node

- there may be possible existence of multiple initial states, and

- modifier methods are completely specified in all states, while accessors are not always completely specified.

However, McGregor [McG2] mentions that, although the dynamics of most objects can be represented by a set of states and their inter-state transitions, most objects

do not constitute formal finite automata. In addition, a state machine describes a transition for every possible event from every state. This includes these event/state combinations which the pre-condition explicitly prohibits.

The idea of representing a class as a state machine for testing purposes was incorporated in the research done by D. Hoffman. Hoffman and Strooper [HS93] propose the following methodology for sufficient testing: (1) test base classes C, (2) test derived classes C0 under the assumption that class C was thoroughly tested. That proposed method is called *graph-based module testing.* It is based on the comparison of the results of the tests done on the class under test (CUT) and the executable oracle built for that purpose. An oracle is built based on the testgraph. A testgraph is a partial model of the states and transitions of a class implementation. Each testgraph node corresponds to a CUT state, and the start node to the initial state. Each arc corresponds to the state transition in the CUT. The oracle is developed, because manual checking of test outputs is typically infeasible in the case of automated testing.

4.3.1 Coverage Strategies

Coverage strategies define adequacy criteria for the testing process. They describe the methodology for selecting test cases to be used in the process. More importantly, they define a criteria for stopping test case selection.

For functional testing (specification-based), the goal is to prove that the software performs according to its specification. In order to achieve this goal, test cases are constructed, executed, and the results are compared with the expected behaviour. Coverage strategies describe techniques for creating test cases so that the cases exercise 'all' of the specified behaviours of the software, where 'all' is defined by coverage strategy.

Missing/ Extra States

A missing state is one that is described in the specification but is not found in the implementation. An extra state is one that exists only in the implementation. After each transition, the object should be in a state described in the specification. If not, the current state is then an extra state, which may mean an error in the transition or it may mean that the specification overlooked a necessary state. If the test cases

intended to place an object in a particular state are unable to do so, then that state is considered to be missing even if some other test cases might have been able to place the object in that state.

Missing/Extra Transitions

The execution of an event results in some transition being traversed, even if the transition returns to the same state. Consider a test case that is intended to take an object in one state and transform it into another state. If after execution of the test case the object is not in that target state, there is evidence of a missing transition or a missing state.

Transitions with incorrect input/output

The input to a transition are the parameters to the event that implement the transition and any globally accessible data. If the incorrect input still results in a transition to the appropriate state, this error will not be detected by functional methods. However, any 'missing transition' errors should be examined to see if the event is simply receiving the wrong input transition to the expected state.

Incorrect output will be directly observable from the execution of the events. Any incorrect output should be investigated and corrected.

Corruption

Corruption is a type of error that occurs when a piece of code appears to execute correctly during one run, but side-effects occur and modify the state in such a way that subsequent operations will not perform correctly. For example, the enqueue message might correctly place an item at the back of the queue but 'lose' the pointer. Subsequently, the second attempt at en-queuing will encounter a corrupt reference. These are not specification errors, however, they may be found in the execution of any test suite.

4.3.2 A Hierarchy of Coverage Strategies

A number of coverage strategies were proposed in the literature for state-based testing [McGD93], but they have varying error detection capabilities and require a wide varying number of test cases.

State coverage

The ultimate goal of testing object oriented software is to develop sufficient test cases when the object is placed in every state defined in the initial specification. Assuming that the objects have more than one state, this strategy is sufficient to detect the following types of errors: (1) all missing states, (2) possibly some extra states, (3) possibly some errors in transitions, and (4) possibly some instances of corruption.

Event coverage

The event coverage guarantees that every method will be executed at least once. Note that event coverage does not subsume state coverage. It is quite possible to exercise each event without being in any state at all. The following are the types of errors which can be found while conducting event coverage: (1) possibly some state errors, (2) possibly some transition errors, and (3) possibly some instances of corruption.

Transition coverage

This level of coverage guarantees that every transition is traversed at least once. This coverage subsumes both state and event coverage. The following can be achieved: (1) detection of all missing states, (2) improved detection of extra states, (3) detection of all missing transitions, (4) detection of some transitions with incorrect input/output, and (5) improved detection of corruption.

Switch coverage

Switch coverage is a general term for the coverage of sequences of transitions [Chow78]. In addition to subsuming the transition coverage strategy, this strategy will improve the possibilities of detecting corruption and extra states. It does this at the cost of many more test cases and represents a small improvement in detection power.

However, it does define a systematic approach to gradually improving detection power until resources are exhausted.

Exhaustive coverage

Exhaustive coverage exercises each event over its entire range of parameter values. This results in all possible paths in the state representation being covered. However, even one cycle in the representation can make this an impossible level to achieve, since the number of cases required may be infinite.

Chapter 5

Role of Formal Specifications in Testing

Testing is a vital part of software development; it is the only widely used method for assuring software quality, and ensuring that a software module adequately meets the requirements of the user. Many software development strategies acknowledge the role of testing the design and implementation of the module during all phases of its development process. Testing, as well as software implementation, is prone to human error. Currently, much research is being conducted in finding ways to automate and formalize testing processes. Formal specifications of software modules provide the most reliable basis for accomplishing these goals.

This chapter is organized as follows: in Section 5.1, the role of formal specifications in the software development process is discussed. Section 5.2 contains a brief introduction into issues related to black-box testing. In Section 5.3, the advantages of using formal specification in testing are analyzed. Problems with using formal specifications for testing are discussed in Section 5.4.

5.1 Role of Formal Specifications in the Software Development Process

Formal specifications provide information about the intended behaviour of a specified software. They provide the basis for reasoning about the software design before

its implementation. As a mathematically based apparatus, formal specification languages can be used in all phases of a software module's development process. Their growing importance in software development has resulted in a growing number of specification languages for a variety of software design paradigms. Formal specification languages exist for traditional functional program development, to support object-oriented paradigm, and to develop concurrent software systems.

Formal specifications are used to reveal the ambiguity, incompleteness, and inconsistency in a software system. In the early phases of its design process, the formal specifications help discover design flaws. In the later phases, they can help determine the correctness of a system's implementation. The application of formal specifications in various phases of software development are as follows:

* In the system requirement analysis phase, the application of formal methods can help the customer clarify a set of informally stated requirements. It helps to reveal contradictions, ambiguities, and incompleteness in requirements.

* In the system design phase, formal specifications support two of the most important activities of that phase: (1) design and decomposition, and (2) refinement. Each interface specification provides a module's client with the information needed to use the module without any knowledge of its implementation. At the same time, it provides the information needed to implement the module without any knowledge of the demands of its clients.

* In the system verification phase, formal specifications allow formal verification of the software module. This process shows how a system satisfies its specifications.

* Formal methods can aid in system testing and debugging. Specifications alone can be used to generate test cases for black-box testing. Specifications, which explicitly state the assumptions on a module's use identify the test cases for boundary conditions. Specifications with implementations can be used for testing analyses such as path testing, unit testing, and integration testing.

* In addition, formal specifications provide an unambiguous documentation of the software system.

Formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation.

5.2 Black-Box Testing

As some studies show [McGK94] testing can be successfully incorporated into the software development process. The definition of test cases, based on the software system requirements and specifications, leads to black-box testing. When the module is implemented, black-box testing is replaced by white-box testing. In these test cases, formal specifications provide a systematic approach to test derivation, which allows for the prediction of testing problems and directions.

The main aspect of black-box testing is to completely cover the system's requirements described within the specifications. For this, we have to extract from the specification different situations for which the specification requires different behaviour. The behaviour is determined by the input values, and the logical structure of the module's requirements. The use of formal specifications allows a systematic analysis of the logical structure to be performed automatically. If the formal specification is separate from the implementation details, it is possible to focus on the intended behaviour of the software module. Nevertheless, all the implementation details should be tested once the software module is written. Formal specifications support a testing process, but they do not replace standard testing methods. Several formal specification languages allow for the incremental development of specifications. The resulting formal specification is as close to the implementation as possible. Therefore, black-box testing evolves towards white-box testing when the specifications are less abstract.

5.3 Advantages of Using Formal Specifications in Testing

The formal specification of a software module used for testing its implementation, provides the following benefits:

* A systematic approach towards testing.

  Formal specifications give guidelines on the intended behaviour of a software system. That formality allows to detect specific aspects of the software module, which should be tested. In addition, the availability of tools allowing for

verification of a specification's completeness and correctness makes formal specifications a reliable basis for software testing. Formal specifications of software can be used for the following aspects of testing: (1) test case derivation, (2) test analysis, (3) oracle development for test purposes, and (4) trace analysis and testing.

* Tests developed in parallel with the implementation.

  Formal specifications evolve from very abstract to detailed ones. In every phase of their development, they provide the necessary information to conduct both software validation and verification activities.

* Abstraction.

  The advantage of deriving test cases from formal specifications rather than from any other source is that formal specifications are precise, yet abstract, descriptions of the system. The structure of input and output is clearly defined. At the same time, its abstraction suppresses irrelevant details. Hence, formal specifications are free of implementation bias. This is especially important when conducting black-box testing.

* Formally specified intended behaviour of a software module.

  Formal specifications use mathematical models to define the properties of the system. Hence, mathematical models can be used to prove the correctness of the system. Therefore, the intended behaviour of the software module is clearly defined. Based on that definition, test cases can be specified. In addition, use of formally specified input and output conditions allows for the detection of hidden relations among consecutive operation invocations.

* Formally specified exception conditions for the software module.

  All formal specification languages provide the means to specify an exceptional behaviour. This is done, for example by using pre-conditions, or, as in case of the Larch LSL layer, using the **exempting** clause. That information can be used to prepare test cases, and to verify that the software system behaves correctly in such situations. In a traditional informal development process, which is not supported by formalism of the specification languages, detection of exception conditions is rather difficult.

* Systematic derivation of test input data for the integration test to achieve complete requirements coverage.

  The formally defined input requirements of the specified operations enables the analysis of input space and external conditions, for the input data derivation. These specifications give the necessary information to determine boundary conditions and partition input space.

* Validation of test results using specification as an oracle to predict a system's behaviour, or as a predicate to identify legal state transitions.

  The test case consists of test data and the means to determine whether the implementation behaves correctly with respect to this data (the test oracle). Test cases for system features are one of the most important pieces of test information.

* Early detection of error and inconsistencies during software development process, optionally leading to a complete formal verification of each development step.

  The act of deriving certain test cases exposes errors in the specifications. In addition, no-domains, those for which the operation is not defined, are able to show what the system will not do. This fact is helpful in spotting subtle errors in either specification or requirements.

5.4 Problems with Using Formal Specifications for Testing Software

The use of formal specifications has not yet got enough attention from software engineering group. Several reasons for the lack of attention can be pointed out:

* Formal systems are not easily verifiable. The use of assisting verification tools (e.g., proof systems) requires knowledge of formal verification methods and expertise.

* The cost of a system's development, based on its formal specifications, is high. In addition, writing formal specifications for complex systems may not always be feasible.

* Every formal specification language has to develop its own mapping to a particular programming language. This complicates the standardization of formal specifications for testing.

* There is inadequate amount of research on the use of formal specifications for the testing. However, most of the work is done in designing formal specification languages, and providing tools for formal verification of completeness and correctness of specifications. Testing issues are not sufficiently researched.

* There is no 'standard' formal specification language. There are several specification languages for each programming paradigm. These languages are based on different logical systems or methods. This fact complicates the unified approach of using formal specifications for testing. Several languages provide only executable specifications, which do not evolve into implementation. Therefore, their use for testing implementation is minimal.

* There are not enough tools to the automate analysis of formally specified software. Testing is a cumbersome activity. The biggest effort is in designing test cases and evaluating test results. These aspects of testing would benefit most from automation. However, till now, formal specifications cannot be fully used to provide the automated tools. Several research groups did attempt to write tools for test suite development. However, no tool has been created for test result evaluation.

* People with no or little mathematical background are afraid of formal specifications. Formal specifications require a mathematical background, certain knowledge of theoretical computer science and logical systems.

Chapter 6

Formal Specifications - Relevant Work

This chapter briefly reviews the results of work done by various research groups in the field of testing software components based on formal specifications. From an analysis of the published results, it can be concluded that this branch of software engineering is in its infancy. However, the beneficial aspects of using formal specifications for testing are widely recognized.

Various research groups have put efforts into developing methodologies and tools to validate the completeness and correctness of formal specifications with respect to software requirements. In this field, special interest was shown towards the creation and use of automated theorem provers (e.g., Larch Theorem Prover (LP) [GG93]). The validation process is indirectly related to testing, because the specifications have to be correct and complete to provide a reliable basis for testing implementation.

The chapter is organized as follows: Section 6.1 briefly describes different types of specification languages; Section 6.2 discusses the various issues related to specification based testing; and Section 6.3 contains a presentation of the work done by Dick and Faivre [DF92]; the methodology in the thesis is partially based on their approach.

6.1 Formal Specification Languages

The main purpose of using formal specifications in a testing process is to guide and help in test case generation and test result evaluation. However, the broad range

of existing specification languages promote a variety of approaches towards the use of formal specifications in testing. Each specification language gives different restrictions on the different aspects of the testing process: test oracle definitions, test sequencing, test data derivation, test results evaluation. The efforts of the research groups working in that area of development concentrate on the issues which can be addressed by analyzing a particular formal specification language. These languages can be categorized as follows:

* Axiomatic specifications are assertions in a predicate logic about the implementation behaviour. They are based on Hoare's work on proofs of correctness of implementation of abstract data types, where first-order predicate logic pre and post-conditions are used for the specification of each operation of a type. Axiomatic languages use a property-oriented method. Anna [Luc91] is an example of specification language that supports the axiomatic method.

  Richardson and O'Malley [ROT89] researched error detection based on the use of formal specifications written in Anna.

* Algebraic specifications are a subset of axiomatic specifications where each assertion is an equality [GH87]. An algebraic specification defines a mathematical object in terms of the relations among the operations defined over the object. This approach uses axioms to specify the properties of systems, but the axioms are restricted to equations. One of well known algebraic specification language is OBJ [GW88].

  OBJ is useful for type checking and experimenting with specifications before the implementation is built, but it does not involve representation mappings because the implementations are not used. A number of research groups based their work on algebraic specification languages. Gerrand et al. [GCM90] used algebraic specifications to show how design time testing may be used in conjunction with formal specifications.

* Larch specification language is a composition of two tiers: the first layer LSL is an algebraic specification of general traits and, the second layer, the interface layer, is an axiomatic specification language with pre- and post-conditions relating the shared language to the implementation. The axiomatic component specifies the

state-dependent behaviour of programs. The algebraic component specifies the state-independent properties of data accessed by programs.

Up-till now, none of the research groups reported any work which uses Larch family languages to test the formally specified implementation. Only Richardson and O'Malley [ROT89] have done some analysis on the applicability of error testing techniques for the Larch/Ada language.

* Model-based specification languages, for example Z [Spi] and VDM [Jon90], construct a model of a software system's behaviour using well defined primitives. Using a model-oriented method, a specifier defines the system's behaviour directly by constructing a model of the system in terms of the basic mathematical structures such as tuples, relations, functions, sets and sequences. VDM supports a model-oriented specification style and defines a set of built-in data types such as lists, sets, mappings, which can be used to define other data types. Z is a formal method based on set theory.

  Stocks and Carrington [SC, SC91, SC93] report work on the creation of the framework to formally specify tests based on their formal specifications. Z language is used to formally specify tests.

  Dick and Faivre [DF93] developed a methodology, based on VDM formal specifications, to create a Finite State Automaton (FSA). Based on their approach, test suites can be prepared and test results may be evaluated. Their FSA derivation methodology is based on obtaining a Disjunctive Normal Form (DNF) from pre- and post-conditions of all operations.

  Horcher and Peleska [HP94] used Z to present an approach towards test result verification. They transform the results of a test from implementation to specification space, and evaluate the satisfiability of any post-condition.

  Richardson at al. [ROA92] use the Z specification language (along with RTIL concurrent specification language) for the analysis of a complex concurrent system. Based on their proposed approach, they create the test oracle and acceptance criteria.

* State-based specifications languages, for example Statecharts [Har87, HLN+90], focus on the system's states and transitions between them. These languages are

intended to specify systems that are highly state dependent, such as reactive or embedded systems. Statecharts reduce the number of specified states at any particular level of the specification with abstraction. Abstract states are composed of a set of parallel finite state machines, where each state machine is in one state.

* Concurrent specification languages (CSL) and distributed systems are used to specify state sequences, event sequences, state and transition sequences, streams, synchronization trees, partial orders, and state machines. Temporal logic is a property-oriented method for specifying the properties of concurrent and distributed systems. For a given temporal logic inference system, special modal operators concisely state assertions about system behaviour. CSP uses a model-oriented method for specifying concurrent processes and a property-oriented method for stating and proving properties about the model.

  Richardson et al. [ROA92] present a methodology for deriving test oracles based on specifications written in the concurrent specification language, RTIL. The oracle information is represented as a set of time intervals expected in response to the stimuli and the assertions that must hold within those intervals.

6.2 Testing Issues

The testing process of a software module involves several issues: e.g., test case derivation, test data preparation, test results evaluation. Formal specifications of software can be applied for each of these issues. During the last few years, several research groups conducted research on the use of formal specifications for testing. However, for any one research group, much of their work concentrates on a limited number of testing issues for one particular formal specification language. Therefore, we will not compare the results of the various groups. Consequently, this section contains a brief description of the results obtained for particular testing issues.

6.2.1 Defining Tests

Formal specifications ease the process of writing the software features. The complexity of a testing process requires a systematic approach to test preparation, execution,

and test results evaluation. Formally specified tests successfully bring the notion of a systematic approach to testing. However, this issue was researched by only one group.

To define test suites, Stocks and Carrington [SC, SC91, SC93] construct abstract definitions of test classes. These test classes guide the test derivation process. The other requirements of testing (e.g, concrete test data, oracles and sequencing) can be synthesized from this information and the specifications. The formal definition of the test is done in the Z specification language. The authors point out it is not necessary to formally specify tests in the same specification language, as the software module is specified.

6.2.2 Sequencing Tests

The role of sequencing tests in a testing process is to test the 'dynamic' composition of operation activations.

Dick and Faivre [DF93] use a reduction to disjunctive normal form (DNF) in test sequencing. For that, the before and after state expressions for all specified operations are disjointed and reduced to DNF. From the resulting expression, a finite state automaton representing the behaviour of the system can be derived, and the relevant paths showing how to conduct the tests can be shown. When the FSA is created, the degree of coverage for all possible states and transitions within the FSA can be used to define a new completeness criterion for test data selection. This FSA might also be used to further supply tool support for test sequence creation.
Horcher and Peleska [HP94] point out that test sequences should not be too long, since the continuation of a sequence may become too difficult once an error is detected. Optimizing the length and number of test sequences becomes important when the test execution is expensive.

6.2.3 Test Oracles

Test oracles prescribe the acceptable behaviour for test execution. That issue was studied by a number of research groups. The results of their work show that the form and usage of an oracle strongly depend on the chosen formal specification language. Richardson et al. [ROA92] argue that test oracles should be derived from specifications in conjunction with testing criteria represented in a common form. However,

Horcher and Peleska [HP94] claim that in some cases the oracles may fail for the following reasons: (1) the implementation of the retrieval function for real systems can be too complex to be carried out manually; (2) the implicit character of specifications may be concerned more with properties than algorithms; (3) formal specifications may be non-deterministic or under-specified (for a given output space, a large set of possible concrete values in the implementation state space may be possible), and (4) generic test data are based on the values from a system's actual state.

Richardson at al. [ROA92] developed a detailed approach to the problem of providing test oracles. The approach is based on the construction of mappings from the name spaces of the specification and implementation to the name space of the oracle. Usually, the oracle name space is the same as the specification name space. Mappings between implementation and corresponding specification control points are established, as are data mappings between the implementation state and the specification state. The implementation state and the specification state changes are monitored at determined control points. The implementation state is checked using the data mappings to the corresponding specification space as an oracle. This approach assumes that the operation sequencing information is available and requires substantial and additional development effort in various mappings.

The key element of Stocks and Carrington's [SC, SC91, SC93] framework is the construction of abstract descriptions of test data, called test templates. This facilitates a fairly simple approach to providing abstract oracles. The input-output relation, defined in the specification to construct abstract descriptions of output from given input templates, is used to obtain a description of the expected output for certain input. One feature of such descriptions in Z is that the state components of the input and output are clearly distinguished from the parameter components. This is helpful if the states need to be constructed and checked, using existing operations.

6.2.4 Selecting Test Data

The quality of the test mainly depends on the test data selected. Most of the approaches dealing with automatic test data generation are based on the implementation code, either using stochastical methods for generation or symbolic execution. Ideally, the requirements should be covered as complete as possible. Therefore, the selection of test data should consider a whole range of possible values to cover possible

discontinuities within the legal range of values, taking into account upper and lower boundaries. Concrete test values are extracted from those domains. The existence of an invariant may limit possible combinations of state variables obtained from the predicate part of the test definition. In addition, the structure of the input data strongly depends on the environment in which the test procedures are executed. The mapping of the abstract operation onto the external interface of the system comes into account, defining the relationship between the abstract operations from the specifications and concrete events actually triggered by the system. Exhaustive testing is usually not feasible; hence, the test should be performed for a representative set of values from allowable domains. In addition, boundary and exception cases testing should be performed. The formal specification of a software module might provide guidelines for selecting test data by defining input spaces and their boundaries. Several research groups conducted research on test data derivation using the specifications.

Hall [H88] uses a general approach to derive test data from Z specifications. Simple partitions of an input space are constructed by examining the obvious divisions of the input defined in the predicates of the operations. This approach is highly structured but not rigorous.

Dick and Faivre [DF93], as a strategy in testing, use partition testing. The input space is partitioned into sub-domains, and one test is drawn from each sub-domain. The goal of this approach is to select sub-domains, so that if the program displays an error for one input from the sub-domain, it will display an error for all inputs from that sub-domain. In simple terms, this generally means choosing input sub-domains which are 'treated the same way' by the program. To partition the input space of an operation specified in model-based notation, the input expression is reduced to a DNF.

Richardson and al. [ROT89] examine strategies for selecting tests by extending implementation based testing techniques to be applicable to formal specifications. Testing strategies are classified into error-based and fault-based. Error-based testing attempts to detect errors in the results of program execution; this usually involves some path or partition analysis of the input and selection of a test sensitive to certain types of errors. Fault-based testing attempts to detect faults in the source code; this usually involves applying rules to elements of source code to produce tests sensitive to commonly introduced code faults. Implementation-based strategies are also extended by actively using the specification as an oracle to be violated. This work does not

define new ways to select tests; it defines, the elements of the input and acceptance criteria in general terms for each of these broad classes of testing approaches.

Stocks and Carrington [SC, SC91, SC93] advocate the use of as many testing strategies as possible. Their framework is a vehicle for using strategies. However, they developed two strategies which they commonly use in testing. The first strategy, called domain propagation, is based on reduction to a DNF, but also considers the effects of sub-operations on the domain partitions. Input domains for each Z operator are constructed[1] and propagated to the higher level operations to see what effect they have on the inputs of the higher level operations. Domains are propagated to the level of the operation under test. The second strategy results from adapting mutation testing to the specification level. This form of mutation testing has a different focus than mutation testing at the implementation level, where the goal is to provide an assessment of a test suite based on the number of program mutants it can kill. Here, specification mutants are considered, and tests are chosen to distinguish these mutants from the original specifications. This sort of testing is aimed at showing that certain errors do not exist in the implementation. Specification mutation testing is aimed at detecting misunderstandings of the specification in the implementation, and has some informal impact on specification validation.

6.2.5 Evaluating Tests

In the classical theory of testing, calculating expected results is a part of the test data selection process. This is generally not possible, even with a formal specification available for automatic evaluation. Although it is not possible to supply concrete values for the expected results, the specification exactly describes the intended behaviour of the system. If there is a test oracle, it can compute the expected outcome of a test case before it is executed. The task of test result evaluation then would simply consist of comparing the expected with the observed outputs.

Some preliminary ideas on test evaluation were presented by Hall [H88]. They form some of the basis for a later paper [H91], extending the relationship between specification and testing in the context of what is learnt about the software by its performance on a test suite. The contrast lies between deducing software correctness and

[1] Not always using reduction to DNF. Most are a combination of experience, good sense, and fine-tuning.

between inducing software correctness based on test results. Dijkstra's classic statements on software testing identify the limits of inducing software correctness based on its performance on a test suite and imply a deductive approach, thus associating the necessity of proving software correctness. Hall argues that, apart from the possibility of introducing errors into a proof, an inductive approach still is of importance because it is acknowledged that software may be faulty when shipped, yet may indeed be satisfactory despite any residual faults. An inductive approach can give insight into a measure of software quality. Hall suggests an approximation measure of software quality by measuring the difference between specification and implementation in the output domain, weighted by the portability of the inputs occuring.

The unifying framework developed by Stocks and Carrington [SC, SC91, SC93] facilitates various analyses of test suites and test strategies. For example, the adequacy criteria of test suites can be shown to hold. Though specification mutation focuses on selecting tests, test suites can be evaluated using mutation analysis and given a rating based on the number of specification mutants distinguished by the suite. Also, test suites derived using different strategies can be compared for overlap and performance, to give insight into deciding which strategies are better under which circumstances.

Horcher and Peleska [HP94] propose a different approach towards the evaluation of test results. Their approach is based on the fact that the Z specifications of an operation may be regarded as the description of the expected output of the operations by means of predicates, and the implementation contains the algorithm to calculate the post-state for a given pre-state of the operation. Their idea is as follows:

> *"take the state transitions computed by the implementation for each test case, transform it back into the specification's space and check it against the specifications by evaluating the predicted using the values observed".*

They claim, this approach is more successful than the traditional approach because of the following: (1) observed outputs are checked in a specification's state space, eliminating possible non-determinism within the data abstraction, and (2) the specification is not used to actively calculate legal state transitions from pre- and post-conditions.

Doong and Frankl [DF91] proposed to construct test cases as a pair of sequences of operations along with a tag indicating whether they should put objects into the same abstract state. A test case is executed by sending the sequences to two objects

of a Class Under Test, checking whether they are in the same state by using the user-supplied equivalence checking facility, EQN, then comparing this result with a tag. This approach allows for substantial automation of many aspects of testing including test-case generation, test driver generation, and test evaluation.

6.2.6 Tool Support

Conducting test preparations and executions is cumbersome. A number of research groups have attempted to create tool support for testing formally specified software. In this approach, the nature of the specification language used is important. Algebraic specification languages provide an easy basis for the creation of automated tools. For other languages, this may not be the case. From the results of this work, it seems that the testing process may be partially automated. Automation is the most beneficial in the case of sequencing the tests, choosing test data, and test result evaluation. However, the automated process should be supervised by a human tester.

Gannon et al. [GMH81] created the automated tool, DAISTS. DAISTS uses the axioms of an algebraic specification to provide an oracle for testing implementations of an ADT. A test case is a tuple of arguments to the left-hand side of an axiom. DAISTS executes a test case by giving it as input to the left-hand side and right-hand side of an axiom, then checks the output by invoking a user-supplied equity function. This tool requires the availability of formal specifications. The implementation is tested against its formal specifications based on user supplied data.

Doong and Frankl [DF91] present their tool, ASTOOT, for testing object oriented classes based on their algebraic specifications. ASTOOT contains a test driver generator, the compiler, and the simplifier. The test generation tool requires the availability of an algebraic specification of an ADT being tested, but the test execution tool can be used when no formal specifications are available. The test cases produced include information, which facilitates the automatic checking of results.

One of the most significant features of Dick and Faivre's work [DF92, DF93] is that it has tool support. A Prolog-based system is used to transform VDM expressions into DNF's (using a VDM grammar and 200 inference rules, and occasional human input). The tool-set also includes a VDM editor and type-checker. Construction of a finite-state automaton and test selection is not yet automated.

A research group in DST (Germany) [HP94] is in the process of creating an automated tool for test class definition. However, they strongly recommend hand analyses of the obtained results to (1) eliminate remaining classes containing predicate parts that cannot be fulfilled, and (2) get new insights into the specification resulting from the different presentation of the specification in the form of test classes.

6.2.7 Representation Mappings

Validation and testing formal specifications requires a representation mapping that relates the states in the implementation to the states in the specification. The implementation and specification languages, however, typically have very different type systems and control models and encourage different representations.

Gannon et al., showed that if the module specification, implementation and representation mappings correspond, the implementation is correct with respect to the specification [GMH87, MBGH89]. That communication is defined as an abstraction function mapping from concrete values to abstract values, and representation function from abstract values to concrete values. Their approach assumes that the specification has sections that intuitively correspond to each implementation operation, and that the relationship between specifications and implementation states is one-to-many.

O'Malley [O94] presents a framework for more general representation mappings. The states of implementation and specification are mapped and compared during execution and/or interpretation. The main assumption is that the framework is not limited to state-based languages, because it can be shown that most languages can be considered to have a relevant state, where the abstract state belongs to the specification, and the concrete/representation state belongs to implementation. The types of analysis and the specification languages determine which aspects of a system must be mapped. Most analyses require both control mapping and data mapping [ROA92], however some may only concentrate on the control structure of the program, requiring only the control mapping.

There are several attributes of specification and implementation which complicate mappings; among them the most useful are paradigm shitt, abstraction level, and homogeneity. *Paradigm shift* is the difference between specification and implementation paradigms. In case the paradigms are similar, the specification and implementation are more similar and the mapping is simpler. *Abstraction level* consists of two

components: the inherit level of the primitives of the formalism, and the capability of adding new domain-specific abstractions to the language. Higher levels of abstraction are less efficient to interpret and execute, but they clarify the specification for the reader. *Homogeneity* between specifications and implementations reflects the tendency to decompose the specification and implementation in similar ways. Homogeneous specifications and implementations also have simpler mappings. Based on these criteria, the complexity, and interrelation between specification and mappings is defined. Diversity between specification and implementation implies a difficulty in creating the mapping, and fewer common features between specification and implementation. Therefore, diverse mappings increase the failure detection capability of the mappings. Nevertheless, they are more difficult to create.

6.3 Dick and Faivre Methodology

In this section, the results of the research done by Dick and Faivre to incorporate formal specifications in the testing process is presented in more detail. Our work was directly and indirectly influenced by their ideas.

Dick and Faivre [DF92, DF93] were working on developing a methodology to verify the correctness of a software against its formal specifications. They narrowed their research to the VDM specification language. Their methodology is based on partition analysis in state based specifications. As a result of their research, it was expected to develop an automatic tool for partition analysis and FSA creation. They present the techniques for automatic partition analysis of model-based specifications with the intent of generating and sequencing test-cases. The main novelty of their approach lies in the use of special DNF for partition analysis, and in using the technique described for extracting an FSA from the specification. The advantages of partition analysis in formal specifications are: (1) the generation of test cases is for internal verification of a product (i.e., is the implementation correct with respect to the specification), (2) the analysis techniques may be useful for external validation (i.e., is the implementation correct with respect to the requirements), (3) the ability to create a FSA from the specification may provide valuable insight into its dynamic aspects.

• **FSA derivation**

In their approach, the formulae of the specification are the relations between states described by operations. These relations are expressed in first-order predicate calculus. The partial relation *spec-OP* between system states was expressed as a set of pairs of states. Dick and Faivre use (1) *pre-OP* which is the expression that characterizes a pre-condition, (2) *post-OP* which characterizes a post-condition, and (3) *inv-State* which characterizes a state invariant. Test domains for individual operations are calculated by reducing $inv-State \wedge pre-Op \Rightarrow inv-State \wedge post-OP$ to a Disjunctive Normal Form (DNF). After the relations are reduced to a DNF, they create a set of disjoint sub-relations (states), each of which is either the before-state or the after-state of at least one of the tests to be performed. Following this, a partition analysis of the system state is done such that each disjunct in the DNF corresponds to each partition. This partitioning permits the construction of a Finite State Automaton from the specification. This in turn can be used to sequence the required tests to avoid redundancy in the testing process and generate test values for validation of the implementation.

• **Test sequencing**

For test sequencing purposes, partition analysis is used to determine cases for which individual operations should be tested. A number of theoretical and practical issues have to be addressed after partition analysis. These issues mainly relate the problems involved in scheduling the sequence of tests to be performed, so as to achieve the right internal state of the system to be tested. To place the system in a desirable state required by the test domain is to provide test-bed functions.

The main advantage of their approach is that partition analysis always gives a finite number of partitions. Consequently, the FSA on the right level of abstraction can be derived. For test sequencing, partition analysis is used to determine cases for which individual operations should be tested.

• **Test execution**

Dick and Faivre developed guidelines for performing the tests; the main purpose of the guidelines is to perform a maximum number of tests in a single run, despite the occurrence of non-determinism and failure. The important fact is that as soon as operations are chained together, possible dependencies become visible which could give rise to possible additional test cases, or the elimination of some suggested by partition analysis of individual operations. The main implication of the approach is

that the automated tool must be fully integrated with the testing environment, i.e. it must provide the facility to solve logical constraints during the execution of test for (1) finding appropriate test sequences, (2) selecting the appropriate test data, (3) verifying the result of the operation, and (4) comparing the state of the physical system with abstract states in the FSA.

The relationship between the abstract values of the specification to the concrete values of the implementation is expressed by a retrieval function. The retrieval function should be implemented as part of the test-bed to expose the state of the system in terms of relating to the specification. Some means of selecting an appropriate concrete value will have to be provided.

- **Tools**

There exists a test generation tool written in SEPIA Prolog, which contains: (1) VDM multi-font editor, (2) VDM type checker, and (3) VDM through Pictures - a tool which allows specifications to be represented as diagrams, with transformations text and visual notation. These tools allow the user to compose a sequence of tests and checks at each stage that the composition is satisfiable. The tool to calculate the FSA by partition analysis will eventually be developed. Reduction to a DNF is performed in Prolog, and VDM knowledge is encoded as a set of about 200 inference rules, which are used in step 4 of the partition analysis. The number of the rules can be extended.

The looseness of the specification does not lead to non-determinism in the FSA. Non-determinism might be resolved by composing constraints during path construction. One of the important limitations of the used specification language is that VDM uses semi-decidable logic, which cannot be fully automated.

Chapter 7

Error Detection and Formal Specifications

This chapter contains the results of analyses for detecting errors using formal specifications. The previous chapter reviewed the work done on testing of an implementation based on its formal specifications. Much of this research does not focus on error detection analysis for several reasons. All of these reasons are related to the specific nature of formal specifications, i.e. the specifications support the software development process, not replace it. Error detection is related to the nature of a software module, and therefore test cases are generally not reusable.

7.1 Limitations and Advantages in Error Detecting

The advantages of using formal specifications in the software development process are more evident in software design. The use of formal specifications may lead to proving the correctness of the design and properties of the software module. The role of formal specifications in software testing was described in Chapter 5. However, it should be noted that error detection based on formal specifications is limited for several reasons. These limitations are due to the characteristics (limitations and advantages) of formal specifications.

Error detection, based on the formal specifications of a software module, is related to the following:

- The formal specification language used

  The type of specification language used is important because the expressive power of each language is different. The Larch family of languages support the explicit statement of exception cases (in LSL layer), and exception handling on the interface layer. VDM, on the other hand, is based on semi-decidable logic, which may cause non-determinism of the specifications. Algebraic specification languages support the existence of executable specifications, but do not provide any guidelines for testing implementation.

- The level of abstraction

  Level of abstraction within formal specifications defines the level of available testing information. In the case of a very abstract specification, only black-box testing can be conducted. If the formal specifications were reified towards implementation then white-box testing can be supported.

- Implementation bias

  Since some of the language constructs may not be expressed formally, implementation bias is of importance when the specification is not related to any particular programming language. It is impossible to formally (explicitly) specify an idea. For example, C++ allows the creation of *deep copy* or *shallow copy* of a variable. A *deep copy* has the same value but different memory location than original source variable, and *shallow copy* has the same value and the same memory location. The formal specification language would have to be able to support these semantics. If it does not, even the most proper formal specification may not carry useful testing information. Hence, the specification may be too abstract in relation to the specified idea.

- The software development paradigm

  Each software development paradigm requires specific aspects of it to be taken into consideration.

- The completeness and correctness of formal specifications

  To compose a reliable source of information used to test implementation, formal specifications have to be correct and complete with respect to the requirements of the software module.

- The type of error

  The extent to which formal specifications support error detection depends on the possibilities of using those specifications to prepare test cases, evaluate test results, and guide the testing process. Some errors cannot be detected if only formal specifications are used as guidelines. These errors are related to the implemented algorithms, proper use of available data structures and procedures, and access to an external environment. In these cases, knowledge of the exact algorithm applied is necessary.

  Formal specifications can help in detecting the proper execution of a sequence of events, by evaluating the expected results. In addition, it can give the exact information about the state in which a system should be in, before an operation is executed.

7.2 Larch/C++ and Testing - Errors

The Larch family of languages offers the unique possibility of separating abstract ideas from the implementation requirements of a specific programming language. The LSL tier, which contains all abstractions, provides a precise definition of an ADT's behaviour. The interface layer provides additional restrictions on the operation invocation and results. Therefore, the error detecting power of a test based on the Larch specification language depends on the analysis of these two layers. In addition, the close relationship between implementation and interface specifications, provides the following advantages:

1. The application of Hoare's logic

   Hoare's logic (pre- and post-conditions) provides information about the input and output domain of variables. The **requires** and **ensures** clauses can be violated to ensure that their operation behaves properly.

2. Explicitly stated boundary conditions

   The **ensures** and **requires** clauses may provide information about the boundaries of these domains. For the methodology proposed in this thesis, the partition domain is used to define states of a FSA. This partition is based on restrictions placed on variables present in both clauses.

3. The existence of a **modifies** clause

   The **modifies** clause contains a list of parameters which can change their values during the interface operation execution. According to the semantics of Larch languages, the value of any parameter which appears in the clause should be explicitly specified in the modifies clause, even if that value does not change in some cases. Therefore, all variables which do not appear in the clause can be tested for preserving their values.

4. The existence of an invariant

   The requirements contained in the **invariant** clause provide information on the cases which should be tested. The invariant should be satisfied before and after any operation is invoked. Specific test cases can be designed to check if the invariant is not violated.

5. Explicit information about exceptions

   Larch/C++ provides exact information about exception cases. The tests to verify proper execution in an exception case can be conducted.

6. Explicitly defined behaviour of an abstract data type

   LSL specifications provide information about the expected behaviour of an ADT. The methodology of this thesis discusses how information is used to build the FSA. In addition, sufficiently complete LSL traits provide detailed and verifiable information on the behaviour of abstract data type. The testing process can be guided to establish the adequacy of an ADT's specification and implementation.

Chapter 8

Creation of Finite State Automaton

In this chapter a methodology is proposed to derive a Finite State Automaton (FSA) from Larch/C++ specifications. The derived FSA will serve a double purpose: (1) to help systematize the testing process, and (2) to compose a base for the creation of test suites.

To design the FSA, a detailed knowledge of the intended behaviour of the software module is necessary. The formal specifications, written in their respective languages, provide that knowledge. The usual question is how to extract the states of the machine and determine the valid transitions.

Most of the work done in the area of testing formally specified software concentrates around the methodology proposed by Dick and Faivre for VDM specifications [DF93] (see also Chapter 6). Their proposed methodology of creating an FSA from the specifications is based on the use of a DNF (Disjunctive Normal Form) [WGS94] to determine valid states and transitions. The methodology presented in this thesis is partially based on Dick and Faivre's approach.

This chapter is organized as follows: Section 8.1 describes Dick and Faivre's methodology of FSA creation; Section 8.2 contains the definitions of newly introduced terms; Section 8.3 discusses the adaptability of the methodology for Larch/C++ specifications; Section 8.4 discusses the role and importance of a class invariant in the FSA derivation process; Section 8.5 presents problems and solutions related to the information contained in LSL traits; Section 8.6 provides information about the proper

analysis of the interface layer; Section 8.7 presents the methodology and algorithm for derivation of the FSA from Larch/C++ specification. Examples on application of the FSA derivation methodology to the simple classes are included in Section 8.8; Section 8.9 discusses the usability of the methodology.

8.1 Methodology when Applied to VDM

A brief overview of Dick and Faivre's methodology, when applied to VDM specifications, is included in Section 6.3 . However, since our research was influenced by their approach to create FSA, their algorithm of FSA creation is presented in more detail in this section.

The main idea of FSA creation is based on DNF analysis of pre- and post-conditions, which are included in the formal specifications of each method in a software module. DNF analysis results in partitions of the input and output space for each individual operation (*sub-operation*). Once the partition analysis of all individual interface operations is performed, valid transitions on the FSA are precisely those of the sub-operations. The states of such an FSA are those in which the pre- or post-conditions of at least one sub-operation are satisfied.

This approach to testing is based on the assumption that everything is sufficiently specified, and satisfaction of a pre-condition guarantees the satisfaction of a post-condition. In addition, if specified, the invariant should always hold. The analysis, was based on the following:

$$inv(pre-state) \wedge pre-condition => inv(after-state) \wedge post-condition$$

This formula is further simplified to a form in accordance with the following rule:

$$A \Rightarrow B \equiv \neg A \vee A \wedge B$$

This expression (transformation rule) was chosen because it gives only the possible input-output relations which are valid.

According to [DF93], the following steps should be taken to derive an FSA:

1. Perform partition analysis (DNF analysis on pre- and post-conditions) on all individual operations and the initial state to obtain the set of sub-operations. From the analysis of each individual operation, the valid transitions will be obtained. The used transformation form identifies only valid transitions.

2. Extract from each sub-operation two sets of constraints, one describing its *before state*, and the other describing its *after state*. This is done by existentially quantifying every variable external to the state in question, and simplifying it.

3. Perform partition analysis, by reducing to a DNF, of the disjunction of the sets of constraints found in Step 2. The resulting partitions will describe disjoint states in which at least one sub-operation creates the state (corresponding to the post-condition) or is executable in that state (corresponding to the pre-condition).

4. Step 1 has provided the transitions of the FSA, and Step 3 has provided its states. The FSA can now be constructed by resolving the constraints of sub-operations against states. A transition labeled `OP` is created from `S1` to `S2` for every sub-operation `OP` and every state `S1` and `S2` satisfying $(S1, S2) \in rel-OP$. The *rel-OP* is the relation on states defined by the constraints on `OP` resulting from partition analysis.

5. If possible simplify the FSA using classical FSA reduction techniques.

6. Use FSA traversal algorithm, which gives canonical coverage of the machine.

This algorithm provided general guidelines for our research. It could not be directly adapted because of the differences between VDM and Larch/C++ formal specification languages. Detailed results of the research and the methodology of creating an FSA from Larch/C++ specifications is presented in the remaining sections.

8.2 Terminology

The purpose of this subsection is to clearly define newly introduced terminology. Much of the terminology is based on the terms defined in Larch/C++ specification language.

- *basic distinguishable domain (sort state)* - the domain of values (state) of the variable of distinguished sort derived from the LSL trait. The basic distinguishable domain describes a particular range of values for which the observable behaviour of the distinguished sort remains the same. The set of basic distinguishable domains, for the same trait, may vary in accordance to a particular purpose of testing; it depends on chosen distinguished sort partitioners.

- *initial sort state* - the basic distinguishable domain of the sort initializer. There can be many initial sort states depending on the number and types of sort initializers. All of them are equally valid from the point of view of the FSA derivation.

- *distinguished sort initializer* - the constructor(s) defined in the LSL trait which create(s) the initial state of the distinguished sort. The distinguished sort initializer appears in the **generated by** clause, and it can be identified by the fact that (1) it resembles definition of a constant, i.e. $new : -> Sort$; (2) it does not include any parameters on the left-hand side other than the ones which compose the sort type, i.e. the field variables in case the sort is modeled as a tuple or union; or (3) it can also be an initialization of the distinguished sort by assigning a value.

- *distinguished sort partitioner* - the inspector(s) defined in the LSL trait which is applied to only the distinguished sort. The distinguished sort partitioner may or may not appear in the **partitioned by** clause. The qualifying inspector usually returns a boolean value, but this is not a rule. For the purpose of testing, it is advisable to choose only those qualifying inspectors which result in a limited number of returned values (states). For example, `StackTrait.lsl` trait (Section 8.8.1), *isEmpty* was chosen to partition the distinguished sort domain because it gives only two partitions: *empty* and $\neg$*empty*. The *top*

inspector was not chosen as a distinguished sort partitioner because it returns only a single element from the set of Integers.

- *alteration* - distinguished sort value modification; it can be a result of addition, subtraction, etc. Alteration of a value of the distinguished sort may or may not change the basic distinguishable domain.

- *distinguished sort alterator* - any operator which changes (alters) the value of the distinguished sort. The change of the value may change the basic distinguishable domain.

- *distinguished sort examiner* - any inspector, excluding distinguished sort partitioners. Examiners do not alter the value of the distinguished sort.

- *input domain* - the basic distinguishable domain(s) for which an LSL operation can be invoked. Those values for which the operation is not defined are specified in the **exemption** clause or, otherwise, they are explicitly stated in the **requires** clause of the interface specification, whenever the LSL operation is used. For example, the purpose of the *push* operation in the `StackTrait.lsl` trait, is to add an element to the `Stack`. This operation can be invoked on `Stack` which is $\neg$*empty* or *empty*.

- *output domain* - the basic distinguishable domain(s) after an LSL operation was invoked. This domain is determined by the type (purpose) of operation. For example, the purpose of the *push* operation in `StackTrait.lsl` trait, is to add an element to the `Stack`. Therefore, after invocation of *push* operation `Stack` is $\neg$*empty*.

- *class variable* - in the interface specification described as *self*. It is modeled by its respective LSL trait (distinguished sort).

- *state variables* - vector of variables which define the state of a class. This vector is contains the following: class variable which appear in **modifies** clause of an interface specification, and any static variables which affect the state. This vector is determined based on an analysis of the interface specification of a class. In addition, the variables which 'compose' distinguished sort, but are not modified, are included in that vector. If any variable is of the composed

distinguished sort type, e.g. union or tuple, it is regarded as a set of variables included in that composed type.

- *state* - *input* and *output domain* to link the information from LSL trait to the interface, when used for FSA derivation.

8.3 Adaptability of Methodology

The fundamental differences between VDM and Larch/C++ formal specification languages have to be taken into account, while assessing the adaptability of the methodology [DF93].

1. VDM has a number of predefined 'data types' to which, according to the methodology, all state variables are reduced. These are: sets, sequences, maps, and basic data types, e.g. integers. The behaviour of these data types is known. This fact can be used to define the set of inference rules, which would assist during the process of determining the valid states for defined operations.

   Larch allows any abstract data type to be defined in the LSL trait. Therefore, the ADT can have any behaviour, and there is no basic universal set of inference rules.

2. VDM is a model-based specification language. It allows simple identification of the state of the variable specified.

   The abstraction defined in the LSL trait does not describe the state of the variable; it only describes its behaviour. Therefore, there has to be an algorithm (methodology) to identify basic states of the variables defined by distinguished sorts.

3. Larch/C++ has two tiers. Each tier models different levels of abstraction. Therefore, the derivation of the states of the FSA is not as straight forward as in VDM. In Larch/C++, the most abstract aspects of a behaviour of the variable is defined in the LSL layer.

   The first tier, the LSL definitions, does not allow for definition of any state of a variable. Hence, LSL is stateless. Analysis of the LSL traits, conducted

according to the guidelines given in Section 8.5, will result in an enriched theory about the sort.

The second tier, interface specification, describes in a less abstract manner, the behaviour of the class. In this respect, the analysis is close to the VDM approach. Therefore, the methodology, as applied to interface layer, uses information directly, while the information carried by LSL layer is used indirectly.

Taking the above into consideration, Dick and Faivre's algorithm for creating a FSA, from Larch/C++ formal specifications, can be applied under the following conditions:

1. The interface specification of the class is complete (see criterion of completeness described in [Uman95]). It means, that the intended behaviour of the class is specified to a known degree; i.e. all incompleteness is intentional.

2. The LSL traits used in the interface specification are sufficiently complete. Sufficient completeness makes it possible to determine the behaviour (input and output domains) of all operations defined in the LSL traits.

3. The class invariant is known. The methodology assumes that if the class invariant exists, it can be extrapolated from the specifications. A tool can be designed, which will determine the class' invariant, and will check the consistency of a global class invariant.

4. All known state variables, which are mentioned in the **modifies** clause have defined domains of their *basic distinguishable states*, and their respective input and output domains.

5. All properties of distinguished sort are known. This knowledge includes also the intended 'under-specification' of a behaviour.

6. Any composed distinguished sort should be simplified in order to correctly determine the set of variables composing the class variable. That is why, in [DF93], it is suggested that all composed data types are simplified, and unfolded to the simple ones.

The following sections deal with the issues mentioned above.

8.4 Class Invariant

The need for invariant analysis follows from one of the principal requirements of our methodology. The invariant is incorporated in the state definition process. It should be satisfied all of the time. Therefore, the following relation (implication) should be true for any interface specification:

$$inv(input - domain) \wedge requires(input - domain) =>$$
$$inv(output - domain) \wedge ensures(output - domain)$$

Since Larch/C++ has two tiers, each of them might contain an invariant. After rigorous analysis, it can be concluded that the class invariant can appear in the following places of the specification:

1. **Class invariant inferred from the LSL trait**

   These are additional restrictions on values of the parameters of distinguished sort. The restrictions can be stated in the **implies clause.** (See example of the `File.lsl` trait, Section 9.2). They can also be embedded in the definitions of the operations.

2. **Class invariant can be explicitly stated in the interface specification**

   The invariant is stated after the keyword **invariant.**

3. **Class invariant can be implicitly stated in the interface specification**

   The invariant may be hidden in the pre- and post-conditions of the interface operations. The theoretical definition for derivation of the invariant from interface specifications are the following:

   Let us assume that each interface specification is defined by a triple $\{p, o, q\}$, where p is the content of its **requires** clause, o is the content of its **modifies** clause, and q is the content of its **ensures** clause. Therefore, the invariant on the pre-conditions of all interface operations is the minimal pre-condition on all of them, while the invariant on the post-conditions of all interface operations is the maximal of all of them. The invariant on the class is then a conjunction of minimal and maximal invariants. The above can be expressed in the following mathematical form:

- for pre-condition

  $\forall i = 1, \ldots, n \; p_i \Rightarrow \bar{\mathrm{p}} \wedge$

  $\bar{\mathrm{p}}$ is minimal in the sense that if

  $\exists \hat{p}$ such that $\forall i = 1, \ldots, n \; p_i \Rightarrow \hat{p}$ then $\hat{p} \Rightarrow \bar{\mathrm{p}}$

- for post-condition

  $\forall i = 1, \ldots, n \; \exists \bar{\mathrm{q}}$ such that $\bar{\mathrm{q}} \Rightarrow q_i \wedge$

  $\bar{\mathrm{q}}$ is maximal in the sense that if

  $\exists \hat{q}$ such that $\forall i = 1, \ldots, n \; \hat{q} \Rightarrow \bar{\mathrm{q}}$ then $\bar{\mathrm{q}} \Rightarrow \hat{q}$

- the class invariant

  $i = \bar{\mathrm{p}} \wedge \bar{\mathrm{q}}$

The class invariant is a composition of all three cases. It is necessary to verify that the class invariants do not contradict each other. This is necessary to avoid an inconsistency in the specifications.

8.5 Derivation of Information from LSL Traits

In the Larch family of formal specification languages, the LSL tier has a special role. In this layer, only the most abstract ideas are defined. Therefore, unlike many other specification languages, Larch allows an unlimited number of data types to be defined. New data types may be defined arbitrarily or may be based on already existing abstractions. Therefore, certain difficulties may be encountered while conducting an automatic analysis of any data type (abstraction) defined in the LSL trait.

The methodology of analysis of the LSL trait, proposed in this thesis, was developed for the purpose of testing. To satisfy the requirements of the FSA derivation from Larch/C++ specifications, the following characteristics of the LSL trait were analyzed:

* LSL describes the intended behaviour of distinguished sort. Therefore the semantic information about all operators, operations, and exception conditions can be extrapolated. This information will help not only in FSA derivation but it will also systematize the process of testing.

* LSL provides additional invariant information. This information is used to determine the states and valid transitions of the FSA. It can be argued, that the LSL trait definition is itself an invariant. However, in some cases there are additional restrictions on the distinguished sort behaviour given in the **implies** clause. For example, the restriction in the `RWFile.lcc` class (Section 9.2) limits the possibility of changing file access code while the file is open.

* LSL provides information about exception and undefined cases of distinguished sort operations. These exceptions, included in the **exemption** clause, bring attention to the under-specification of the distinguished sort, given freedom to the implementation and, subsequently, identify the conditions for which class should be tested.

* LSL provides information about the values of distinguished sort before and after invoking the operation. This information will help to determine valid transitions in the FSA.

* LSL determines the basic distinguishable domains. The interface specification gives information on if and how to partition the domains, additionally.

All operations defined in the LSL trait have pre-defined domains of values for which they can be invoked. Hence, there is pre-defined input and output domain for each operation. To correctly determine all input and output domains for all operations, it is necessary to have a complete LSL trait definition. This manifests itself in fulfilling the rule of sufficient completeness, introduced by Guttag in [GH78]. This rule states that

> *each constructor should be applied to each basic constructor; each inspector should be applied to each basic and non-basic constructor.*

The invocation of an operation which modifies distinguished sort, results in a transition from input to output domains. It is possible that input and output domains will be the same. The invocation of the constructor results in assigning values to the distinguished sort from its output domain, i.e. placing it in *initial sort state*. The invocation of an inspector does not result in a transition; therefore, an input domain is an output domain for the inspector.

8.5.1 Taxonomy of LSL Operations

LSL theory [GH93] divides all operators into the following groups: (1) basic constructors, (2) basic inspectors, (3) non-basic constructors, and (4) inspectors. This classification corresponds to the theory of the Larch family of languages. However, for the purpose of testing, information extracted from the LSL trait has to accommodate different goals. The requirement of the testing methodology is to determine (1) *basic distinguishable domains* of the variable specified by the distinguished sort and (2) *input* and *output domains* of the operations defined in respective LSL trait. Hence, for the purpose of testing, the following taxonomy was created:

1. **distinguished sort initializer** - provides the values of attributes in the initial *state* of the distinguished sort;

2. **distinguished sort partitioner** - provides information on how *distinguishable domains* of the distinguished sort are partitioned;

3. **distinguished sort alterator** - alters the associated value of the distinguished sort. For any distinguished sort alterator *input* and *output domains* should be determined.

4. **distinguished sort examiner** - examines the value of the distinguished sort but it does not alter the distinguished sort. For the distinguished sort examiner, only *input domains* should be determined.

5. **constant** - definition of a constant of a distinguished sort type. There is no *input* and *output domain* associated with a constant.

6. **type converter** - this operator does not have input and output domain associated with the distinguished sort. However, the input and output domains can be defined for these operators.

7. **exemption clause** - determines the *input domains* for which an operation is not defined. This information helps to determine valid *input domains* of the operation.

Hence, to determine the basic distinguishable domains and input and output domains for all operations, the LSL trait has to be first analyzed according to the above taxonomy.

8.5.2 Analysis of LSL Primitives

The predefined LSL theory data structures and their governing rules have to be analyzed for testing. All primitives defined in the LSL layer provide additional information which can be used to determine: (1) the set of state variables, (2) basic distinguishable domains, and (3) additional conditions on the state variables. The primitives in the LSL layer are as follows:

1. **simple type** - data type which contains only one field. Any distinguished sort which is composed of a single field can be regarded as a simple type. In LSL theory there are several predefined 'simple' types, for example Boolean and Integer.

2. **tuples** - fixed-length tuples, similar to records in many languages. For tuples, all inspectors for all fields should be present, in order to be able to determine basic distinguishable domains.

3. **unions** - tagged unions found in many programming languages. For unions, all inspectors for all fields should be present, in order to be able to determine basic distinguishable domains.

4. **constants** - represent the definition of values of the type of a distinguished sort. No modifiers nor inspectors are applied to constants. If used in the interface specification, the constant is instantiated and may provide information on boundary conditions.

5. **enumeration** - finite ordered set of distinct constants and operators that enumerate them. The values of an enumeration may guide the testing process, since they may provide additional conditions on the behaviour of the distinguished sort. For example, in `File.lsl`, the value of the mode of a file determines the execution of an operation.

6. *include* **in LSL** - specifies traits whose theory is incorporated into the theory of the given distinguished sort. Very often, the theory of the given trait enriches the theory of an included trait. In that case, usually, the given trait does not contain a basic constructor, and, subsequently, basic constructors for analysis are taken from the included trait. If the given trait contains a basic constructor, it is added to constructors which are present in the included traits. Several included traits, e.g. Integer, Boolean, should not be analyzed; they are regarded as definitions of basic data types.

7. *assume* **in LSL** - specifies traits whose theory is incorporated into the theory of the given distinguished sort; this theory is not included into traits which assume/include a given trait. However, to derive basic distinguishable domains and input-output domains for individual operations, the same approach is taken as for *included* traits.

8.5.3 Proposed Methodology of LSL Analysis

To extract the needed information from LSL trait the following steps are followed:

1. Identify *distinguished sort initializer*

   The distinguished sort initializer(s) are those basic constructors which result in the creation of the distinguished sort *(initial state)*. There could be a set of distinguished sort initializers, and all of them should be analyzed to define all possible *initial states*. If in the trait there is no qualifying basic constructor, it has to be taken from the *included* traits.

2. Identify *distinguished sort partitioner*

   Distinguished sort partitioners are those inspectors which take only the distinguished sort as a parameter. The ones of interest to us are those which partition the input space into a finite number of sub-partitions. For testing purposes, it is important to have only a limited number of available partitions in order to obtain a meaningful machine.

3. Identify all *distinguished sort alterators*

   From a testing point of view, sort alterators change the state of the distinguished sort. For these alterators, analysis of the available *input* and *output domains* (pre- and post-state) should be done.

4. Identify all *distinguished sort examiners*

   These are all remaining inspectors. They do not introduce any new partition onto the sub-domain. The inspectors do not change the *state* of the distinguished sort; hence, for them, only *input domains* (pre-states) should be defined.

5. Identify all **exceptions**, i.e. not defined operations

   The operations located in the **exemption** clause, are not defined for a particular *input domain* (pre-state); hence, the respective operation cannot have that state in its *input domain* (pre-state).

6. Apply all *distinguished sort partitioners* to all *distinguished sort initializers* to obtain all *basic distinguishable domains (states)*

Consequently, a full set of possible *input* and *output domains* (states) of the distinguished sort should be obtained. On the LSL level, these states cannot be further divided. For this, knowledge about the nature of the interface operation is necessary. Particular values of the parameters will play a role in establishing the output domain of an operation; for example, the union of two sets can result in an empty set if both sets are empty, or not empty if at least one of two sets is not empty.

7. For all *sort alterators*, determine the *input* and *output domains* (pre- and post-states) of their execution

   As a result, the *input* and *output domains* (pre- and post-states) should be obtained. To obtain *input domains*, the definition (signature) of the operation and exempting clause are analyzed. To obtain *output domains*, axioms involving the operation are analyzed.

8. For all *sort examiners*, determine the *input domains* (pre-states) of their execution

   The *examiners* do not change the state of the distinguished sort, so it is sufficient to determine their *input domains* (pre-states). To obtain *input domains*, the definition (signature) of the operation and exempting clause are analyzed.

9. Determine *input domains* (pre-states) for all operations which appear the in **exemption** clause

   Since for the exempted operation it is clearly stated for which domain it is not defined, this step is straight forward.

10. Verify that *input domains* (pre-states) for any operation which appear in **exemption** clause, are not included in the set of *input domains* (pre-states) of that operation

11. For type convertors and constants, *input* and *output domains* do not have to be determined. Type convertors can be applied for any value of the distinguished sort, and does not result in a change of its value

See Section 8.8 for examples of methodology application.

8.6 Derivation of Information from Interface Specifications

Formal specifications must be complete and correct to compose the basis for testing an implementation. Therefore, the specifications which relate to a particular programming language should reflect all of the characteristics of that language.

There are several types of operations which can be defined in the C++ language for a class. In this section, the following types of methods (operations) will be discussed: **public**, **private**, **global** and **static**. The **private** methods in C++ are not 'visible' on the interface level, therefore they do not directly participate in software module reuse and testing. On the other hand, since they are 'visible' for the user, all **public**, **global** and **static** methods should participate in test suites. However, not all of them carry information useful in FSA derivation. Therefore, interface operations should be analyzed and divided according to their use in the following manner:

1. **constructors**

   These operations provide all initial states for a class. They should all be analyzed. In some cases, for simplicity, initial states can be merged, but nonetheless, all initializers should participate in the testing process. The interface specification of a constructor contains the **modifies** clause.

2. **destructor**

   This operation provides additional information on the conditions under which the class can be destroyed. It should be analyzed, and should participate in test suite derivation. The interface specification of a destructor contains the **modifies** clause.

3. **inspectors**

   These operations do not participate in the partition of the states. Nevertheless, they should be analyzed and included in test suites. Inspectors should also be analyzed to determine domains (states of FSA) in which they can be invoked. This analysis may involve a DNF analysis of the pre-condition and post-condition of the interface specification. The interface specification of an inspector does not contain the **modifies** clause.

4. **modifiers**

   Modifiers change the 'active' state of the FSA. They should all be analyzed for testing purposes and should participate in the test suites. The interface specification of a modifier contains the **modifies** clause.

   Some modifiers may be included in the group of inspectors. If they provide a state change it is not detected by the FSA. They usually contain the *type convertors* which are applied to *self.*

5. **global operations/operators**

   They should be included in test suite preparation but may not participate in the creation of the FSA. Interface specification of a global operator does not contain the**modifies** clause.

8.7 Algorithm of Finite State Automaton Derivation from Larch/C++

This section presents the algorithm of FSA derivation from the Larch/C++ specifications. This algorithm is based on a theoretical analysis of both layers in the Larch/C++ specification language. As an input, the algorithm takes the interface specification of the class along with all LSL traits used in it. During the analysis of the **used** LSL traits, all other included and assumed traits may be analyzed. As an output, the algorithm gives the FSA for the class, with all the valid transitions and systematic information which will be used in a testing process. In the analysis, the following the notation of Larch/C++ is used: (^) means state (value) before operation was invoked, and (') means state (value) after operation was invoked. The algorithm proceeds as follows:

1. Determine purpose of testing

   Determine the set of state variables under the test.

2. Analyze all traits placed in the **uses** clause

   The LSL traits, indicated in the **uses** clause, define the data abstractions which are used in the interface specifications. In order to properly identify all data abstractions, the traits have to be identified and analyzed. In some cases however, the LSL traits do not participate in the FSA creation; for example, when defining type conversion.

3. Perform partition analysis for all operations in the chosen LSL traits

   The chosen traits are analyzed according to the methodology presented in Section 8.5. The determined *basic distinguishable domains* and LSL operations with *input* and *output domains* provide information necessary to create the FSA.

4. Establish the invariant on the state variable

   In this phase of analysis, the existence of an explicitly stated invariant is examined. That invariant appears in the **invariant** clause of the interface specifications.

5. Determine the class invariant

   As it was stated in Section 8.4, the class invariant is composed from three basic elements. In this phase of analysis, the implicit invariant is determined from interface specifications of the methods, based on the proposed approach in Section 8.4.

   The class invariant is then derived from all three determined elements.

6. Analyze all interface operations (methods)

   All of the operations from the class interface specification should be analyzed. The order of their analysis is not important.

   The analysis of an interface operation produces a DNF partition of the following expression derived from the specification of the method:

   $$inv(pre-state) \ \wedge \ pre-condition \rightarrow inv(after-state) \ \wedge \ post-condition$$

   The DNF analysis of the expression requires invariant, content of the **requires** and **ensures** clauses, and input/output domains of the LSL operations used in the interface specification of the method, as input.

   The result of this analysis is the set of all states of FSA, where the operation can be invoked or lead to, and all transitions. The transitions should be marked as *source* $\rightarrow$ *destination* where source and destination are names of the states.

7. Determine all states of the FSA

   All states of the FSA determined for individual operations are listed and analyzed, using DNF analysis. These analyses might lead to additional subdivision (partition) of states. The following is a detailed list of steps.

   - List states for all operations (assertions) used in the interface specification
   - Perform DNF analysis on available states

8. Determine all transitions based on the results of previous phase of the analysis, i.e. changes to the states of FSA

   Because the number and type of states might change as a result of the previous step, additional analysis is needed for each operation. This should result in a full set of transitions on all available states for each operation in question.

8.8 Examples to Explain Methodology

This section shows how the proposed methodology is used to extract distinguishable domains of particular data abstractions and input and output domains of defined operations. Examples of several LSL traits, along with their respective interface specification, are used to show the problems and solutions of applying this methodology. These *distinguishable domains*, and conditions for each operation, will eventually be used in automating the derivation of the Finite State Automaton. With the examples are included analyses for FSA derivation for `Istack` and `Screen` classes; for trait `LimitedStack`, only the derivation of distinguished domains, and input/output domains for operations are presented. The analysis for other classes are included in Appendix A.

8.8.1 Stack

Stack is the distinguished sort which models the behaviour of LIFO queue.

StackTrait.lsl Analysis

The LSL trait for a stack, shown in Figure 3, modeis the stack which can contain an unlimited number of elements. This trait fully specifies all operators, i.e. all operations are total functions. According to the methodology, the following steps of analysis were performed:

• sort the operators according to the taxonomy

- sort initializer is *new*
- sort partitioner is *isEmpty*
- sort alterators are *push, pop*
- sort examiner is *top*

StackTrait(*E*, *Stack*) : **trait**
includes
Boolean
introduces
$new :\rightarrow Stack$
$push : Stack, E \rightarrow Stack$
$top : Stack \rightarrow E$
$pop : Stack \rightarrow Stack$
$isEmpty : Stack \rightarrow Bool$
asserts
Stack **generated by** *new*, *push*
Stack **partitioned by** *top*, *pop*, *isEmpty*
$\forall\ s : Stack, e : E$
$top(push(s, e)) == e$
$pop(push(s, e)) == s$
$isEmpty(new)$
$\neg isEmpty(push(s, e))$
implies
converts *top*, *pop*, *isEmpty*
exempting $top(new), pop(new)$

Figure 3: `StackTrait.lsl` definition

- **extract the basic distinguishable domains**

From an analysis of the sort initializer and sort partitioner, it is known that the two basic distinguishable domains (states) of the stack are *empty*, $\neg empty$.

- **perform analysis for all LSL operations**

  - analysis of *push* alterator

    $push(s, e)$:

    From the definition of *push* and exempting clause, we get:
    $s\hat{} = empty \vee s\hat{} = \neg empty$
    From axiom $\neg isEmpty(push(s, e))$, we get:
    $s' = \neg empty$

  - analysis of *pop* alterator

    $pop(s)$:

    From the definition of *pop* and exempting clause, we get:
    $s\hat{} = \neg empty$
    From axiom $pop(push(s, e)) == s$, we get:
    $s' = empty \vee s' = \neg empty$

  - analysis of *top* examiner

    $top(s)$

    From the definition of *top* and exempting clause, we get:
    $s\hat{} = s' \wedge s\hat{} = \neg empty$

  - exemptions:
    - $top(new)$
    - $pop(new)$

    The exemptions should be used in test case preparations, because of the nondeterministic behaviour of some operations in the LSL trait. Since both operations are not defined for *new* stack, the *empty* domain cannot be included in the input domain of *top* and *pop* operations.

- There is no additional restriction on the stack specified in the LSL trait. Hence for FSA derivation, the LSL trait does not give any more information.

FSA Derivation for IntStack Class

The analysis for FSA derivation was done in the following steps:

1. Determine state variables

   Since there is only one **included** trait in the `IntStack` interface definition, and *intStack* is modeled by it, the state variable is only one *intStack* of type `Stack`. In the remaining analysis, for simplicity, *intStack* will be replaced by *self*.

2. Determine invariant

   A class-invariant for `IntStack` does not exist and hence the formula is the following (see Section 8.1):

   $$pre - state \rightarrow post - state \equiv pre - state \wedge post - state$$

3. Determine initial state:

   $self' = empty$

4. Analyze all interface operations:

   (a) Interface Operation *Ipush(i)*

   This operation changes the state of the class.

   requires: any stack, i.e. TRUE

   ensures: $self' = push(self^{\wedge}, i)$

   $pre \rightarrow post \equiv post$

   There is only one 'operation' from LSL trait; this will determine the pre and post states of *self*.

```
class IntStack
{
    uses StackTrait (intStack for Stack, int for E);
public:
    IntStack ()
    {
            modifies self;
            ensures self' = new;
    }
    ¬IntStack ()
    {
            modifies self;
            ensures trashed(self);
    }
    void Ipush (int i)
    {
            modifies self;
            ensures self' = push(self^, i);
    }
    bool IisEmpty ()
    {
            ensures result = isEmpty(self^);
    }
    int Itop ()
    {
            requires ¬isEmpty(self^);
            ensures result = top(self^);
    }
    void Ipop ()
    {
            requires ¬isEmpty(self^);
            modifies self;
            ensures self' = pop(self^);
    }
}
```

Figure 4: Interface specification of `IntStack` class

push(self,i):

$(self\hat{} = empty \vee self\hat{} = \neg empty) \rightarrow self' = \neg empty$

After the analysis, we get:

(1) $self\hat{} = empty \wedge self' = \neg empty$

(2) $self\hat{} = \neg empty \wedge self' = \neg empty$

The $self$ variable can be in two states, $empty \vee \neg empty$, in a pre-state and in one state only, $\neg empty$, in each of post-states. This gives two distinguishable states:

(I) $self = empty$

(II) $self = \neg empty$

Therefore according to the above, the available transitions are

$I \rightarrow II$ and $II \rightarrow II$

(b) Interface Operation *Ipop*

This operation changes the state of the class.

requires: $\neg isEmpty(self\hat{})$

ensures: $self' = pop(self\hat{})$

$pre \rightarrow post \equiv pre \wedge post$

$\neg isEmpty(self\hat{}) \wedge self' = pop(self\hat{})$

There are two 'operations' from the LSL trait. Therefore both are analyzed to determine the pre and post states of *self.*

¬ isEmpty(self):

$self\hat{} = \neg empty$

pop(self):

$self\hat{} = \neg empty \rightarrow (self' = empty \vee self' = \neg empty)$

$self\hat{} = \neg empty \wedge self\hat{} = \neg empty \wedge (self' = empty \vee self' = \neg empty) \equiv$
$self\hat{} = \neg empty \wedge (self' = empty \vee self' = \neg empty)$

After the analysis, we get:

(1) $self\hat{} = \neg empty \wedge self' = empty$

(2) $self\hat{} = \neg empty \wedge self' = \neg empty$

The $self$ variable can be in one pre-state, $\neg empty$, and in two post-states, $empty \vee \neg empty$. That gives two distinguishable states, which for consistency will retain the numbers from the analysis of *Ipush*:

(I) $self = empty$

(II) $self = \neg empty$

Therefore according to the above, the available transitions are:

$II \rightarrow I$ and $II \rightarrow II$

(c) Interface Operation *Itop*

This operation does not change the state of the class.

requires: $\neg isEmpty(self\hat{})$

ensures: $self' = self\hat{} \wedge result = top(self\hat{})$

$pre \rightarrow post \equiv pre \wedge post$

There are two 'operations' from LSL trait. Therefore both are analyzed to determine the pre and post states of *self*.

¬ isEmpty(self):

$self\hat{} = \neg empty$

top(self):

$self\hat{} = \neg empty \rightarrow self' = \neg empty$

After the analysis, we get:

(1) $self\hat{} = \neg empty \wedge self' = \neg empty$

The *self* variable can be in one pre-state, $\neg empty$, only. This one state was marked as (II), for consistency:

(II) $self = \neg empty$

There is only one available transition. However, since it is an inspector, it does not change the state.

$II \rightarrow II$

(d) Interface Operation *IisEmpty*

This operation does not change the state of the class.

requires: any stack, i e. TRUE

ensures: $self' = self^{\wedge} \wedge result = isEmpty(self^{\wedge})$

$pre \rightarrow post \equiv post$

There is only one 'operation' from LSL trait. This will determine the pre and post states of *self.*

isEmpty(self):

$(self^{\wedge} = empty \vee self^{\wedge} = \neg empty) \rightarrow (self' = empty \vee self' = \neg empty)$

After the analysis, we get:

(1) $self^{\wedge} = empty \wedge self' = empty$

(2) $self^{\wedge} = \neg empty \wedge self' = \neg empty$

The $self$ variable can be in two pre-states, $empty \vee \neg empty$. Since it is an inspector , the invocation of the operation does not change the state. This gives following two states:

(I) $self = empty$

(II) $self = \neg empty$

There are two available transitions. However since the operation is an inspector, its invocation does not change of the state.

$I \rightarrow I$ and $II \rightarrow II$

5. Determine all states of the FSA

   - List all states, in the order of analyzed operations:
     (1) *Istack*: (I) *empty*
     (2) *Ipush*: (I) *empty* (II) $\neg$*empty*
     (3) *Ipop*: (I) *empty* (II) $\neg$*empty*
     (4) *Itop*: (II) $\neg$*empty*
     (5) *IisEmpty*: (I) *empty* (II) $\neg$*empty*
   - Perform DNF analysis on the available states

     In the `IntStack` example, the number and type of available *states* do not change. So, the FSA has only two states: *empty*, $\neg$*empty*.

6. Determine all transitions based on the results of the previous operation, i.e. changes to the states of the FSA

   In the current example, since the states did not change, the transitions remain unchanged, and we have:

   Ipush: $I \rightarrow II \wedge II \rightarrow II$
   Ipop: $II \rightarrow I \wedge II \rightarrow II$
   Itop: $II \rightarrow II$
   IisEmpty: $I \rightarrow I \wedge II \rightarrow II$

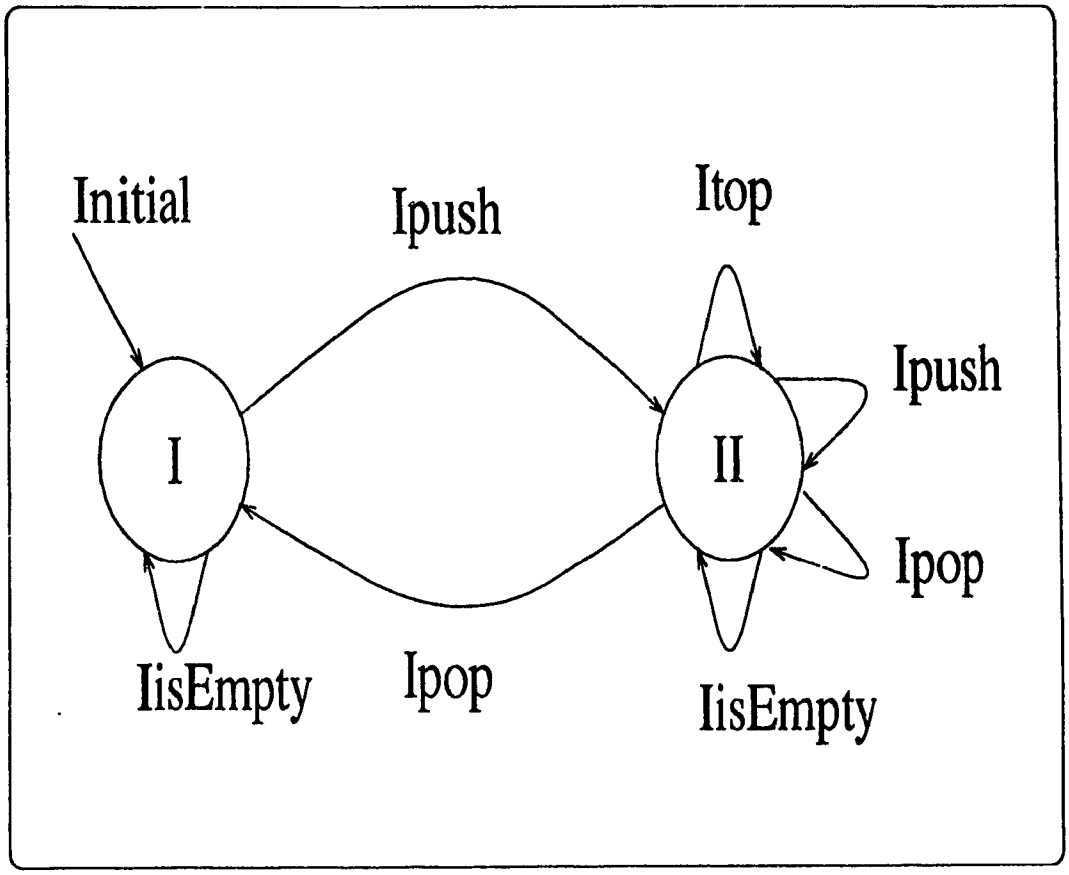

Figure 5: Finite State Automaton for `IntStack` class

8.8.2 LimitedStack

The most common stack used in software systems are those which can contain only a limited number of elements. The limited stack example was chosen to illustrate the analysis of the LSL trait which has more than one distinguished sort partitioners. Although the interface specifications analyses are not included in that example; they resemble the analysis of the `IntStack` class.

LStackTrait.lsl Analysis

The LSL trait, shown in Figure 6, models the behaviour of the stack with a limited number of elements. According to the methodology, the following steps of analysis were performed:

• Sort the operators according to the taxonomy

- sort initializer is *new*
- sort partitioners are *isEmpty, isFull*
- sort alterators are *push, pop*
- sort examiners are *top, size*

• extract basic sort partitions

From the analysis of the sort initializer and sort partitioners, it is known that: the basic distinguishable domains of stack are: **empty, ¬ empty, ¬ full, full;** however, the correspondence among them is as follows:

(I) - empty $\wedge$ ¬ full

(II) - ¬ empty $\wedge$ ¬ full

(III) - ¬ empty $\wedge$ full

The state when stack is empty $\wedge$ full cannot exist. Hence, it is not included into the example.

• perform analysis of all operators

- analysis of *push* alterator

$push(s, e)$:

$(s\hat{} = empty \vee s\hat{} = \neg empty) \wedge s\hat{} = \neg full$

$s' = \neg empty \wedge (s' = full \vee s' = \neg full)$

- analysis of *pop* alterator

  $pop(s)$:

  $s\hat{} = \neg empty \wedge (s\hat{} = full \vee s\hat{} = \neg full)$

  $(s' = empty \vee s' = \neg empty) \wedge s' = \neg full$

- analysis of *size* examiner

  $size(s)$

  $s\hat{} = s' \wedge (s\hat{} = \neg empty \wedge (s\hat{} = full \vee s\hat{} = \neg full) \vee s\hat{} = empty)$

- analysis of *top* examiner

  $top(s)$

  $s\hat{} = s' \wedge s\hat{} = \neg empty \wedge (s\hat{} = full \vee s\hat{} = \neg full)$

- exemptions:
  - $top(new)$
  - $pop(new)$

  The exemptions should be used in test case preparation, because of the non-deterministic behaviour of some operations in the LSL trait. Since both operations are not defined for a *new* stack, the *empty* domain cannot be included in the input domain of *top* and *pop* operations.

- The trait specification is intentionally incomplete. The *push* operation is not defined on the *full* stack. The trait designer intended to leave it unspecified and added restrictions on the interface layer.

LStackTrait(E, *Stack*) : **trait**
includes
Boolean, *Integer*
introduces
new :→ *Stack*
push : *Stack*, E → *Stack*
top : *Stack* → E
pop : *Stack* → *Stack*
isEmpty : *Stack* → *Bool*
isFull : *Stack* → *Bool*
maxEl :→ *Int*
size : *Stack* → *Int*
asserts
Stack **generated by** *new*, *push*
Stack **partitioned by** *top*, *pop*, *isEmpty*, *isFull*
$\forall$ s : *Stack*, e : E
$top(push(s, e)) == e$
$pop(push(s, e)) == s$
$size(new) == 0$
$size(push(s, e)) == size(s) + 1$
$isEmpty(new)$
$\neg isEmpty(push(s, e))$
$\neg isFull(new)$
$isFull(s) == size(s) = maxEl$
$isEmpty(s) \Rightarrow \neg isFull(s)$
$isFull(s) \Rightarrow \neg isEmpty(s)$
implies
converts *top*, *pop*, *isEmpty*, *isFull*
exempting $top(new), pop(new)$

Figure 6: `LStackTrait.lsl` definition

8.8.3 Screen

This example, although called `Screen`, in reality deals only with the cursor on the screen. This follows from the fact that interface specification contains only operation related to the cursor movement.

This example brings to attention the aspect of 'usability' of the interface specification. Here, usability must be understood as the source of information which is needed for a particular implementation of the formally specified distinguished sort. The trait models the behaviour of `Screen`, while on the interface level it is used to model only the behaviour of the cursor. For this reason the analysis of the distinguishable domains (states) of the variable are done only for the cursor, which is part of the screen.

The cursor, and its constructor is not implicitly specified. From the LSL specification, it is known that the cursor is an integer value, which can be changed. The value of the cursor, after a screen is created, is null. It is also known that none of the operations on the cursor change the parameters $(width, height)$ of a screen. From the initial analysis of the operations included in `Screen.lcc` interface specification, Figure 8, it is known that the analysis will be conducted for the cursor.

ScreenTrait : **trait**
  includes *Integer*
  introduces
    $newScreen : Int, Int \rightarrow Scrn$
    $height, width : Scrn \rightarrow Int$
    $setCursor : Scrn, Int \rightarrow Scrn$
    $getCursor : Scrn \rightarrow Int$
  asserts
    $Scrn$ **generated by** $newScreen, setCursor$
    $Scrn$ **partitioned by** $getCursor, height, width$
    $\forall\ s : Scrn, h, w, i : Int$
      $height(newScreen(w, h)) == h$
      $width(newScreen(w, h)) == w$
      $height(setCursor(s, i)) == h$
      $width(setCursor(s, i)) == w$
      $getCursor(newScreen(w, h)) == 0$
      $getCursor(setCursor(s, i)) == i$
  implies
    converts $height, width, getCursor$

Figure 7: `Screen.lsl` definition

Screen.lsl Trait Analysis

• **sort the operators according to the taxonomy**

- sort initializer is *newScreen*
- sort partitioner is *getCursor*
- sort alterators are *setCursor*
- sort examiners are *width, height*

• **extract basic distinguishable domains**

From an analysis of the sort initializer and sort partitioner, the two basic distinguishable domains for cursor are: **cursor = 0, cursor ≠ 0** .

In addition, there is no restriction on the value of cursor, i.e. it can assume any integer value, either positive or negative.

• **perform analysis of all operators**

- analysis of *setCursor* alterator

  $setCursor(s, c)$:

  $cursor^{\wedge} = 0 \vee cursor^{\wedge} \neq 0$

  $cursor' \neq 0 \vee cursor' = 0$

- analysis of *width* examiner

  $width(s)$:

  $cursor^{\wedge} = cursor'$

- analysis of *height* examiner

  $height(s)$:

  $cursor^{\wedge} = cursor'$

- exemptions

  exemption clause is not included in LSL trait definition. Therefor, the additional analysis of the operators and their domains is not necessary.

- There are no additional restrictions on the cursor specified in LSL trait; hence for FSA derivation, LSL does not provide any more information.

```
class Screen
{
    uses ScreenTrait (Screen for Scrn);
    inv ∀i : Int(i = getCursor(self) => i ≥ 0 ∧ i < height(self) * width(self));
public:
    Screen (unsigned w, unsigned h)
    { //
            modifies self;
            ensures self' = newScreen(h, w) ∧ getCursor(self') = 0;
    }
    ¬Screen ()
    { //
            modifies self;
            ensures trashed(self);
    }
    void moveLeft ()
    { //
            requires getCursor > 0;
            modifies self;
            ensures self' = setCursor(self^, getCursor(self^) − 1);
    }
    void moveRight()
    { //
            requires getCursor(self^) < width(self^) * height(self^) − 1;
            modifies self;
            ensures self' = setCursor(self^, getCursor(self^) + 1);
    }
    void moveUp()
    { //
            requires getCursor(self^) ≥ width(self^);
            modifies self;
            ensures self' = setCursor(self^, getCursor(self^) − width(self^));
    }
    void moveDown ()
    { //
            requires getCursor(self^) ≤ (height(self^) − 1) * width(self^);
            modifies self;
            ensures self' = setCursor(self^, getCursor(self^) + width(self^));
    }
}
```

Figure 8: Interface specification of Screen class

FSA Derivation for Screen Class

1. Determine state variables

   uses ScreenTrait

2. Determine invariant

   From the interface specification, the invariant has the following form:

   inv $c \geq 0 \wedge c < w * h \equiv (c > 0 \wedge c < w * h) \vee c = 0$

   As it was previously noted, there are no additional restrictions on the cursor included in LSL trait `Screen.lsl`. The invariant does not contradict any pre- and post-conditions included in **requires** and **ensures** clauses. From the analysis of the `Screen.lsl` trait, it was established that *width*, and *height* of the screen do not change with movement of the cursor; for of simplicity of the analysis, the following can be assumed:

   - **Assumption:** $w = width(self) \wedge h = height(self)$

3. Determine initial state

   - Screen constructor

   pre: TRUE
   post: $(c' = 0 \vee c' > 0 \wedge c' < w * h) \wedge c' = 0 \equiv c' = 0$

   Hence, initial state of the class is: $c = 0$.

4. Analyze all LSL operations

   (a) MoveLeft operation

   pre: $(\hat{c} = 0 \vee \hat{c} > 0 \wedge \hat{c} < w * h) \wedge \hat{c} > 0 \equiv (\hat{c} > 0 \wedge \hat{c} < w * h)$
   $0 \wedge \hat{c} < w * h) \equiv F \vee (\hat{c} > 0 \wedge \hat{c} < w * h) \equiv (\hat{c} > 0 \wedge \hat{c} < w * h)$
   post: $(c' = 0 \vee c' > 0 \wedge c' < w * h) \wedge (c' = \hat{c} - 1) \equiv c' = 0 \vee (c' > 0 \wedge c' <$
   $(w * h - 1))$

   $pre \rightarrow post \equiv \neg pre \vee pre \wedge post$

   $(\hat{c} > 0 \wedge \hat{c} < w * h) \wedge (c' = 0 \vee c' > 0 \wedge c' < (w * h - 1))$

after grouping the pre and post-states we get the following states:

1. $c\hat{} > 0 \wedge c\hat{} < w * h$
2. $c' = 0$
3. $c' > 0 \wedge c' < (w * h - 1)$

After additional partition analysis and removing ($\hat{}$) and ($'$) symbols, we get three states:

(I) $c = 0$

(II) $c > 0 \wedge c < (w * h - 1)$

(III) $c = w * h - 1$

The following transitions are available:

$II \rightarrow I$, $II \rightarrow II$, $III \rightarrow II$

(b) MoveRight operation

pre: $(c\hat{} = 0 \vee c\hat{} > 0 \wedge c\hat{} < w * h) \wedge c\hat{} < (w * h - 1)$

$\equiv (c\hat{} = 0) \vee (c\hat{} > 0 \wedge c\hat{} < (w * h - 1))$

post: $(c' = 0 \vee c' > 0 \wedge c' < w * h) \wedge (c' = c\hat{} + 1) \equiv c' > 0 \wedge c' < w * h$

$pre \rightarrow post \equiv \neg pre \vee pre \wedge post$

$((c\hat{} = 0) \vee (c\hat{} > 0 \wedge c\hat{} < (w * h - 1)) \wedge (c' > 0 \wedge c' < w * h)$

we get the following states:

1. $c\hat{} = 0$
2. $c' > 0 \wedge c' < w * h$
3. $c\hat{} > 0 \wedge c\hat{} < (w * h - 1)$

After additional partition analysis and removing ($\hat{}$) and ($'$) symbols, we get three states:

(I) $c = 0$

(II) $c > 0 \wedge c < (w * h - 1)$

(III) $c = w * h - 1$

The following transitions are available:
$I \rightarrow II$, $II \rightarrow II$, $II \rightarrow III$

(c) MoveDown operation

pre: $(c\hat{} \geq 0 \wedge c\hat{} < w * h \wedge c\hat{} \leq (h-1) * w$
$\equiv c\hat{} \geq 0 \wedge c\hat{} < (h-1) * w$
post: $(c' \geq 0 \wedge c' < w * h) \wedge (c' = c\hat{} + w) \equiv c' \geq w \wedge c' < w * h$

$pre \rightarrow post \equiv \neg pre \vee pre \wedge post$

$c\hat{} \geq 0 \wedge c\hat{} < (h-1) * w \wedge c' \geq w \wedge c' < w * h$

We get the following states:
1. $c\hat{} \geq 0 \wedge c\hat{} \leq (h-1) * w$
2. $c' \geq w \wedge c' < w * h$

After additional partition analysis and removing ($\hat{}$) and ($'$) symbols, we get three states:
(I) $c \geq 0 \wedge c < w$
(II) $c \geq w \wedge c < (h-1) * w$
(III) $c \geq (h-1) * w \wedge c < h * w$
The following transitions are available:
$I \rightarrow II$, $II \rightarrow II$, $II \rightarrow III$

(d) MoveUp operation

pre: $(c\hat{} \geq 0 \wedge c\hat{} < w * h \wedge c\hat{} \geq w$
$\equiv c\hat{} \geq w \wedge c\hat{} < h * w$
post: $(c' \geq 0 \wedge c' < w * h) \wedge (c' = c\hat{} - w) \equiv c' \geq 0 \wedge c' < w * (h-1)$

$pre \rightarrow post \equiv \neg pre \vee pre \wedge post$

$c' \geq w \wedge c' < h * w \wedge c' \geq 0 \wedge c' < w * (h-1)$

We get the following states:
1. $c' \geq w \wedge c' \leq h * w$
2. $c' \geq 0 \wedge c' < w * (h-1)$

After additional partition analysis and removing ($\hat{}$) and ($'$) symbols, we get three states:
(I) $c \geq 0 \wedge c < w$
(II) $c \geq w \wedge c < (h-1) * w$
(III) $c \geq (h-1) * w \wedge c < h * w$

- The following transitions are available:
$II \rightarrow I$, $II \rightarrow II$, $III \rightarrow II$

5. determine all states of FSA

- **List all states**

From *moveLeft* and *moveRight* operations, we have the following states:
(I) $c = 0$
(II) $c > 0 \wedge c < (w * h - 1)$
(III) $c = w * h - 1$

From *moveUp* and *moveDown* operations, we have the following states:
(I) $c = 0$
(II) $c \geq w \wedge c < w * (h-1)$
(III) $c \geq w * (h-1) \wedge c < w * h$

- **Perform additional DNF analysis**

From the conjunction of the above six formula we obtain the following partitions, after additional analysis:
(I) $c = 0$

(II) $c > 0 \wedge c < w$

(III) $c \geq w \wedge c < w * (h - 1)$

(VI) $c \geq w * (h - 1) \wedge c < (w * h - 1)$

(V) $c = w * h - 1$

6. Determine all transitions, based on results of the previous step, i.e. changes to the states of FSA.

   Since the number of distinguished domains (states) changed, the transitions also changed. The result of the analysis is presented below:

   1. *Screen*: I
   2. *moveLeft*: I → I, II → I, III → II, IV → II, V → IV, II → II, III → III, IV → IV
   3. *moveRight*: I → II, II → II, II → III, III → III, III → IV, IV → IV, IV → V, V → V
   4. *moveUp*: I → I, III → I, II → II, III → II, III → III, IV → III, V → III
   5. *moveDown*: I → III, II → III, III → III, III → IV, IV → IV, III → V, V → V

   Some of the above transitions are for the cases when pre-condition (**requires**) clause is not valid. Their validity follows the principal that if the precondition does not hold, nothing changes.

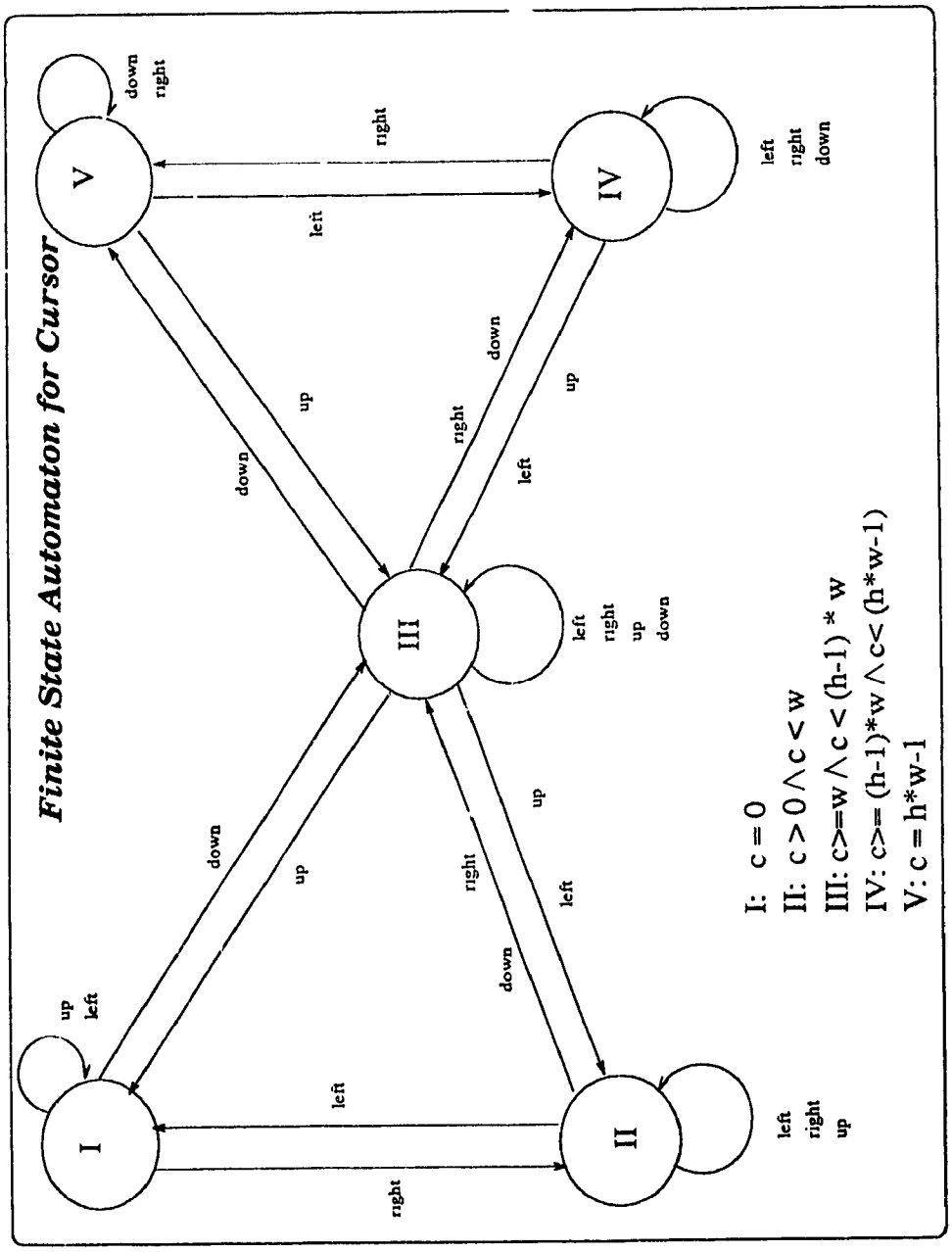

Figure 9: Finite State Automaton for Screen class

8.9 Critique of the Methodology

Formal specifications, due to their abstraction, i.e. without concern of implementation details, provide a clear picture of the software behaviour and properties. The methodology presented in this thesis defines guidelines for conducting a testing process based on formal specifications written in Larch/C++. Application of the methodology results in the creation of a FSA from the formal specifications. In turn, the FSA can be used to construct test cases, then can be used in a black-box fashion to test an implementation. However, this is not the only advantage of the methodology. In addition, the properly analyzed dependencies in a software module, defined by using formal specifications, provide information about necessary conditions to ensure the specified behaviour.

Goodness of the machine

The FSA created by the methodology is only a simplified model of a software system. The form of the created machine (FSA) depends on the following:

1. Abstraction chosen for specifications

   LSL layer of Larch provides the forum to create any abstract data type (ADT) to model the software. Therefore, even though two different formal specifications might model the same system, they would result in different FSAs once they are based on different ADTs. In addition, any non-determinism in a formal specification introduces simplifications or non-determinism in the FSA.

2. Generality of the specifications

   More general specifications allow us to create more general models of the system. Therefore, the created machine will be much more simplified, and the analysis of the formal specifications will provide more general information to test the implementation. On the other hand, the detailed specifications provide information sufficient to test more subtle aspects of the software's behaviour.

3. Chosen set of distinguished sort initializers and partitioners

   Distinguished sort initializers and partitioners determine the basic distinguished domains. These domains compose the basis for the initial form of the FSA.

Role of completeness

The completeness of the formal specifications determines the faithfulness of the FSA. It influences the form of the machine, degree of accuracy of the analysis of the states and transitions, and, hence, the test suites preparation and evaluation. This follows from the fact that incompleteness results in non-determinism. Therefore, the FSA derived from an incomplete specification would be a more general (and subsequently less accurate) model of the system, than that derived from the complete specifications. As it was stated in Section 8.3, there are a number of requirements on the LSL trait specifications and interface specifications to ensure proper FSA derivation.

Extent of use of the FSA for testing

The existence of the FSA modeling the behaviour of a class simplifies the following aspects of the testing process:

1. Systematic approach towards testing

   All aspects of the specified software's behaviour can be derived from the formal specifications. The methodology allows us to systematically document the important state variables, transitions and their conditions. It also allows to determine exceptional cases from the formal specifications.

2. Testing exceptional cases

   The analysis of the formal specifications, according to the methodology, provides a clear definition of exceptional cases.

3. Testing boundary conditions

   The methodology enables us to derive tests cases to test boundary conditions. Recall that many interface specifications directly indicate the values of the variables in the **requires** and **ensures** clauses.

4. Testing minimal required conditions

   An invariant determines the minimal acceptable state of a class in any situation. Therefore, the existence of an invariant can be determined from the formal specifications, based on the methodology.

5. Test oracle

   There is no need to build the test oracle for a software module, since the FSA models the behaviour of the module.

6. Reuse of the test cases

   Another important advantage of our methodology is that any test of the class can be incorporated in the testing process, when it was used to create another software module. In such cases, any time the method from the class is used, the necessary conditions for proper behaviour of the class are known.

   Black-box reuse is based on using the software component based on its specifications, and without modifying its internal structure. Black-box testing is concerned only with the external behaviour of a software component. Therefore, the test cases (and the FSA) for a class can be partially reused while testing its application in a new environment.

On the other hand, some aspects of the software systems cannot be tested by using the proposed methodology. Among them are the following:

1. The degree of faithfulness of the derived FSA depends on several factors: (1) the hidden incompleteness of the specifications, (2) the degree of abstraction of the specifications, and (3) the intentional incompleteness.

2. Not all operations defined in the interface layer can be tested using the FSA. For example, the interface operation which cannot be tested is *applyTokeyAndValue* method in `RWBtreeOnDisk` class.

   It is desirable to guide the process and test specific aspects of the software implementation under the assumption that other 'parts' work well.

3. The resulting FSA, does not always provide enough information to test the specified class.

   The methodology can be applied to different types of classes. In cases where a class has a state-driven behaviour (`Screen`), the machine will unfold it. In the case of a more static class (`RWDate`), the analysis will determine the dependencies, condition and exceptional cases only. In that respect, the application of the methodology is beneficial.

Chapter 9

Validating Correctness of Tools.h++ Library

During the first few months of research on the project "Formal Specifications for Black-Box Reuse", our group developed Larch/C++ specifications for several classes from Tools.h++, a product of Rogue Wave company [rogue]. This work is still pursued by other members of the Black-Box Reuse Research Group. This library of Larch/C++ specifications, along with the implementation of the classes in [ACCUA94] will serve as a test-bed for the various research activities of our group. In this thesis, we use the library [rogue] and the implementation of classes for testing the robustness of the methodology developed so far.

The proceeding chapter is structured as follows: Section 9.1 contains a brief description of the Rogue Wave Tools.h++ library, and its formal specifications. Section 9.2 contains an analysis of `RWFile` class. Section 9.3 contains an analysis of `RWBtreeOnDisk` class. Section 9.4 contains an analysis of `RWHashTable` class. Section 9.5 presents the test cases prepared for class `RWFile` and the results of the conducted tests.

9.1 Rogue Wave Tools.h++ Library

The commercial library for which formal specifications were to be written had to be well structured, complete, and well documented. The Rogue Wave Tools.h++ [rogue] library satisfies these requirements. Tools.h++ consists mostly of a large and

rich set of concrete classes that are usable in isolation and do not depend on other classes for their implementation or semantics. The concrete classes consist of a set of simple classes (such as dates, times, stings), and three different families of collection classes (collection classes based on templates, collection classes that use preprocessor <generic.h> facilities, "Smalltalk-like" classes for heterogeneous collection). The library also includes a set of abstract data types (ADTs), and the corresponding specializing classes that provide a framework for persistence, localization, and other issues. All collection classes have a corresponding iterator. Multiple iterators can be active on the same collection at the same time. Version 6 of Tools.h++ introduces very powerful facilities for internationalization and localization.

In addition, the fact that the Rogue Wave Tools.h++ library is used by several software developers provides an opportunity to: (1) introduce formal specifications of the library to the programmers, (2) test the implementation of the library against the formal class interface specifications, and (3) use their specifications in the software design process, when the whole library is fully formally specified.

During the first phase of the research, several classes from the Tool.h++ library were formally specified. Formally specified classes can be now used to validate their implementations. To check the adaptability of the methodology of testing software based on its implementation, several formally specified classes from Tools.h++ library were tested. The results of tests show how writing formal specifications benefits the user, and how the test can be conducted, i.e. what errors were 'uncovered' by preparing tests based on formal specifications.

9.2 Analysis of RWFile Class

The class `RWFile` [rogue][page 53-1] encapsulates binary file operations, using the Standard C stream library. Because this class is intended to encapsulate binary operations, it is important that it is opened using a binary mode. This is particularly important under MS-DOS; otherwise bytes that happen to match a newline will be expanded to carriage return or line feed.

9.2.1 File.lsl Trait Analysis

A detailed analysis of the `File.lsl` trait can be found in [ACCUA91]. In this subsection, only the information added for the purpose of testing based on the formal specifications of the `RWFile` class is presented.

File : **trait**

includes *rwString(Byte, Data)*

File **tuple of** *name* : *Name*, *data* : *Data*, *mode* : *MODE*

OpenFile **tuple of** *file* : *File*, *data* : *Data*, *mode* : *MODE*, *fpointer* : *Int*

MODE **enumeration of** *READ*, *WRITE*, *READ_WRITE*

readeffect **tuple of** *ofile* : *OpenFile*, *reddata* : *Data*

introduces

create : *Name*, *MODE* → *File*

open : *File*, *MODE* → *OpenFile*

flush : *OpenFile* → *OpenFile*

error : *OpenFile* → *Bool*

read : *OpenFile*, *Int*, *Int* → *readeffect*

write : *OpenFile*, *Data*, *Int* → *OpenFile*

openMode : *OpenFile* → *MODE*

createMode : *File* → *MODE*

isEmpty : *OpenFile* → *Bool*

isOpen : *File* → *Bool*

isEof : *OpenFile* → *Bool*

isBof : *OpenFile* → *Bool*

asserts

∀ *f* : *File*, *opf*, *opf*$_1$: *OpenFile*, *mode*, *m* : *MODE*, *nm* : *Name*, *i*, *p* : *Int*, *dat* : *Data*

create(*nm*, *m*) == [*nm*, *empty*, *m*]

open(*f*, *READ*) == [*f*, *f.data*, *READ*, 1]

open(*f*, *READ_WRITE*) == [*f*, *f.data*, *READ_WRITE*, 1]

open(*f*, *WRITE*) == [*f*, *f.data*, *READ_WRITE*, *len*(*f.data*)]

flush(*opf*) ⇒ *opf.data* = *opf.file.data*

read(*opf*, *i*, *p*).*reddata* == *prefix*(*removePrefix*(*opf.data*, *p*), *i*)

read(*opf*, *i*, *p*).*ofile* == [*opf.file*, *opf.data*, *opf.mode*, *p* + *i*]

write(*opf*, *dat*, *p*) == [*opf.file*, *prefix*(*opf.data*, *p*) || *dat* ||
removePrefix(*opf.data*, *p* + *len*(*dat*)), *opf.mode*, *p* + *len*(*dat*)]

$openMode(opf) == opf.mode$

$createMode(f) == f.mode$

$isEmpty(opf) == len(opf.data) = 0$

$opf.File = f == isOpen(f)$

$isEof(opf) == len(opf.data) = opf.fpointer + 1$

$isBof(opf) == opf.fpointer = 1$

$read(opf).OpenFile.mode = opf.mode$

$write(opf).mode = opf.mode$

$flush(opf).mode = opf.mode$

$isBof(create(nm, m)) == isEof(create(nm, m))$

implies $\forall\ opf, opf_1 : OpenFile, dat : Data, i, p : Int$

$read(write(opf, dat, p), len(dat), p).reddata == dat$

converts *create*, *open*

exempting $\forall\ nm : Name, f : File, dat : Data, p : Int$

$write(open(f, READ), dat, p)$,

$open(create(nm, READ), WRITE), open(create(nm, READ), READ\_WRITE)$

Figure 10: `File.lsl` definition

Analysis

1. *OpenFile* is modeled as a tuple with the following four fields: *file*, *data*, *mode*, *fpointer*. The *file* field is also a tuple with three fields: *name*, *data*, *mode*. Therefore, according to the methodology proposed in this thesis, these tuples should be simplified to the set of variables, each representing a field in a tuple. As a result, *OpenFile* has six fields: *file.name*, *file.data*, *file.mode*, *data*, *mode*, *fpointer*. In addition, to fulfill the requirement of sufficient completeness of the LSL trait, the inspectors and modifiers should be applied to each of these variables.

2. *MODE* data type is an enumeration, therefore it should be regarded as a number of cases for which the particular value of the type is assigned.

3. *readeffect* is a tuple with two fields: *ofile*, *reddata*. Therefore, according to the methodology, the tuple should be simplified to the set of variables, each representing a field in a tuple. In addition, to fulfill the requirement o sufficient

completeness of LSL trait, the inspectors and modifiers should be applied to each of these variables.

4. After analyzing of all attributes of the *OpenFile* sort, a decision was made to use only some of them to partition the domain (state variables). The remaining attributes will compose global state variables, since they do not change while the operators are applied, or after distinguished sort was created.

   - *file.name* - value of this attribute does not change. Hence, it does not participate in domain partition analysis.
   - *file.data* - even though this value may change, the state of disk copy of the memory content is not important, from the point of view of this analysis.
   - *file.mode* - value of this attribute determines access to the file on disk. This value is important when opening the file.
   - *data* - memory copy of the content of the file on disk. The value of this attribute is important to determine domain partition for the distinguished sort.
   - *mode* - access mode to the content of the file, while it is open. The value of this attribute is important to determine the set of operations on the distinguished sort.
   - *fpointer* - position of the pointer in open file. The value of this attribute is important to (1) determine the set of operations on the distinguished sort, and (2) partition the domain.

   Therefore, the set of distinguished sort attributes, which participate in partitioning the domain are: *OpenFile.fpointer*, *OpenFile.data*, *OpenFile.file.mode*, and *OpenFile.mode*.

5. The `RWFile` class provides error checking. To model this, the *error* operator was added to the LSL trait. The value of an *error* is important to determine a partition. However, when the value of an *error* becomes TRUE, none of the operations of the distinguished sort can be invoked.

6. Distinguished sort initializer

   This trait has two sort initializers: *open, create*. However, their nature and importance are related to the structure of *OpenFile*. The *create* sort initializer, in `File.lsl` trait, is only used to relate the trait operations to the creation of the file on disk. None of the other operators for a file on disk manipulations (e.g., delete, move, copy) is specified in `File.lsl` trait. The reason is that these operations are not used in the interface specification of the class `RWFile.lcc`. However, the only relation between the created and opened file is determined by the value of *file.mode* when the file is opened with given *OpenFile.mode*. These dependencies are given in the **exemption** clause. In addition, once the distinguished sort is initialized with the specific *OpenFile.mode*, this mode cannot be changed while file is opened.

7. Distinguished sort partitioner

   The inspectors are: *error, isEmpty, isOpen, isEof, isBof, openMode, createMode*. However, only the *isEmpty, isOpen, isEof, isBof, error* can be considered as distinguished sort partitioners.

8. Basic distinguished domains

   After applying the basic sort partitioners to the distinguished sort initializer, we get the following partitions:
   $isEmpty : empty, \neg empty$
   $isOpen : open, \neg open$
   $isEof : eof, \neg eof$
   $isBof : bof, \neg bof$
   $error : error, \neg error$

   After additional analysis, we get the following partitions:

   - $\neg open \wedge (empty \vee \neg empty)$
   - $open \wedge empty \wedge bof \wedge \neg error$
   - $open \wedge \neg empty \wedge bof \wedge \neg error$
   - $open \wedge \neg empty \wedge eof \wedge \neg error$

- $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$
- $open \wedge error$

9. For sort initializer, the following conditions apply:

   (1) $OpenFile.file.mode = READ \ \wedge \ OpenFile.mode = READ$
   (2) $OpenFile.file.mode = WRITE \wedge$
   $OpenFile.mode = (READ \vee WRITE \vee READ\_WRITE)$
   (3) $OpenFile.file.mode = READ\_WRITE \wedge$
   $OpenFile.mode = (READ \vee WRITE \vee READ\_WRITE)$

10. The input and output domains have to be specified for the following sort alterators: *flush, read* and *write* according to value of $OpenFile.mode$.

    (a) $OpenFile.mode = READ$

    - Operation *read*

    input: $open \wedge empty \wedge bof \wedge \neg error$
    output: $open \wedge empty \wedge bof \wedge \neg error \ \vee error$

    input: $open \wedge \neg empty \wedge bof \wedge \neg error$
    output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \ \vee error$

    input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$
    output: $open \wedge empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \ \vee error$

    - Operation *write*

    This operation cannot be applied because the access to the file is $READ$.

    - Operation *flush*

    This operation cannot be applied because the access to the file is $READ$.

    (b) $OpenFile.mode = WRITE$

    - Operation *read*

input: $open \wedge empty \wedge bof \wedge \neg error$
output: $open \wedge empty \wedge bof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$
output: $open \wedge \neg empty \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

• Operation *write*

input: $open \wedge empty \wedge bof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

• Operation *flush*

input: $open \wedge empty \wedge bof \wedge \neg error$
output: $open \wedge empty \wedge bof \wedge \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$
output: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$

output: $open \wedge \neg empty \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

(c) $OpenFile.mode = READ\_WRITE$

• Operation *read*

input: $open \wedge empty \wedge bof \wedge \neg error$

output: $open \wedge empty \wedge bof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$

output: $open \wedge \neg empty \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

• Operation *write*

input: $open \wedge empty \wedge bof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

• Operation *flush*

input: $open \wedge empty \wedge bof \wedge \neg error$

output: $open \wedge empty \wedge bof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge eof \wedge \neg error \vee error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$

output: $open \wedge \neg empty \wedge \neg bof \wedge (\neg eof \vee eof) \wedge \neg error \vee error$

(d) The input domains have to be specified for following examiners: *openMode*, *createMode*.

• Operation *createMode*

input: $open \wedge empty \wedge bof \wedge \neg error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$

• Operation *openMode*

input: $open \wedge empty \wedge bof \wedge \neg error$

input: $open \wedge \neg empty \wedge bof \wedge \neg error$

input: $open \wedge \neg empty \wedge eof \wedge \neg error$

input: $open \wedge \neg empty \wedge \neg bof \wedge \neg eof \wedge \neg error$

The results of analysis of `File.lsl` trait can be represented in a graphical form, Figure 11. This representation visualizes different aspects of distinguished sort behaviour. The graph has a specific notation: lines and arrows represent transitions as a result of an operation; rectangles encircle the partitions for which a certain attribute value does not change. The operations are not shown. Since it is assumed that their presence on a graph is not necessary; it is self explanatory.

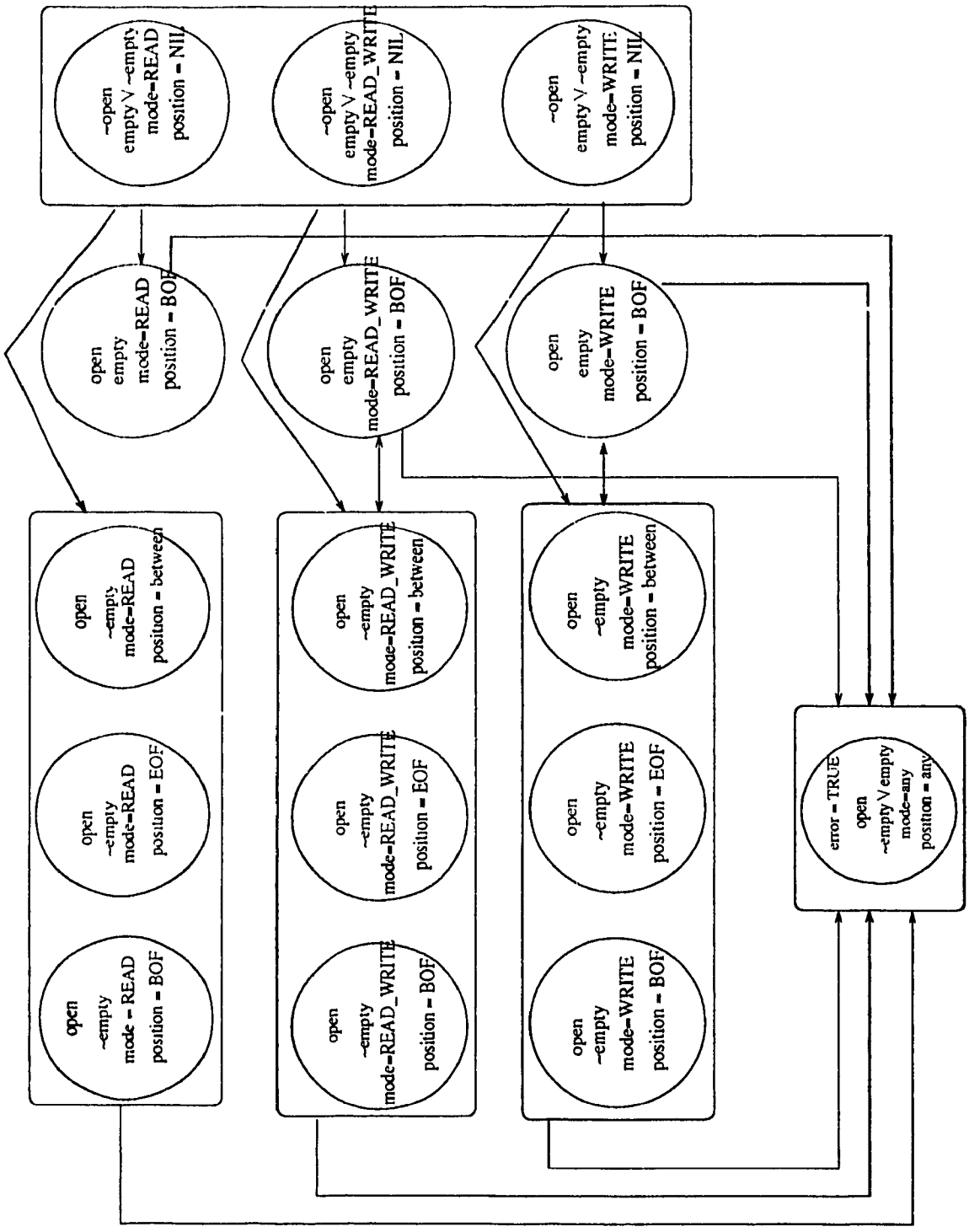

Figure 11: All Valid and not valid states for `File` trait

9.2.2 Interface Specification Analysis

The full interface specification in Larch/C++ for the `RWFile` class from Rogue Wave Tools.h++ library [rogue] is included in [ACCUA91]. For the purposes of this example, the specifications were modified. The main modifications resulted in not including some operations into the example. This applies especially to the *read* and *write* interface operations for which only a representative case was chosen. Several interface operations were omitted.

The interface specifications, included below were changed, and are presented in a form, which allows for efficient DNF analysis.

class RWFile

typedef unsigned size_t;

typedef char *String;

uses File(RWFile **for** OpenFile,String **for** Name, String **for** MODE),

Types(char) ;

public:

1. RWFile - constructor

   RWFile(const char* filename, const char* mode=0)

   pre: TRUE

   post: $(mode = 0 \wedge f.name = filename \wedge$

   $\tilde{}error(open(f, READ\_WRITE)) \wedge self' = open(f, READ\_WRITE))$

   $\vee$

   $(mode = 0 \wedge f.name = filename \wedge$

   $error(open(f, READ\_WRITE)) \wedge self' =$

   $open(create(filename, READ\_WRITE), READ\_WRITE))$

   $\vee$

   $(mode <> 0 \wedge self' = open(create(filename, mode), mode))$;

2. CurOffset - inspector

   long CurOffset()

   pre: TRUE

   post: $self' = self \wedge result = self\hat{}.fpointer$;

3. EOF - inspector
   RWBoolean Eof()
   pre: TRUE
   post: $self' = self\hat{} \wedge result = isEof(self\hat{})$;

4. Erase - modifier
   RWBoolean Erase()
   pre: TRUE
   modifies $self$;
   post: $self'.data = empty \wedge self'.fpointer = 1 \wedge result = \tilde{}error(self')$;

5. Error - inspector
   RWBoolean Error()
   pre: TRUE
   post: $self' = self\hat{} \wedge result = error(self\hat{})$;

6. Flush - modifier
   RWBoolean Flush()
   pre:TRUE
   modifies $self.file.data$;
   post: $result = \tilde{}error(flush(self\hat{}))$;

7. GetName - inspector
   const char* GetName()
   pre: TRUE
   post: $self' = self\hat{} \wedge result' = self\hat{}.file.name$;

8. IsEmpty - inspector
   RWBoolean IsEmpty()
   pre: TRUE
   post: $self' = self\hat{} \wedge result = isEmpty(self\hat{})$;

9. isValid - inspector
   RWBoolean isValid() const
   pre: TRUE
   post: $self' = self\hat{} \wedge result = isOpen(self\hat{})$;

10. Read - modifier

    RWBoolean Read(char& c)

    pre: $len(self\hat{}.data) - self\hat{}.fpointer \geq len(toByte(c\hat{}))$;

    modifies $self.fpointer, c$;

    post: $result = \tilde{}error(read(self\hat{}, len(toByte(c\hat{})), self\hat{}.fpointer).self')$;

11. SeekTo - modifier

    RWBoolean SeekTo(long offset)

    pre: TRUE

    modifies $self.fpointer$;

    post: $result = \tilde{}error(self') \wedge self'.fpointer = offset \wedge self'.data = self\hat{}.data$;

12. SeekToBegin - modifier

    RWBoolean SeekToBegin()

    pre: TRUE

    modifies $self.fpointer$;

    post: $result = \tilde{}error(self') \wedge self'.fpointer = 1 \wedge self'.data = self\hat{}.data$;

13. SeekToEnd - modifier

    RWBoolean SeekToEnd()

    pre: TRUE

    modifies $self.fpointer$;

    post: $result = \tilde{}error(self') \wedge self'.fpointer = len(self\hat{}.data) + 1$

    $\wedge\ self'.data = self\hat{}.data$;

14. Write - modifier

    RWBoolean Write(char i)

    pre: $self\hat{}.mode = WRITE \vee self\hat{}.mode = READ\_WRITE$;

    modifies $self$;

    post: $result = \tilde{}error(self')\wedge$

    $self' = write(self\hat{}, toByte(i), self\hat{}.fpointer)$;

Analysis

1. Class interface specification uses two LSL traits; they are `File.lsl` and `Types.lsl`.

The results of the analysis of the `File.lsl` trait are in a Section 9.2.1. `Types.lsl` trait is used to model distinguished sort conversion only. Therefore in this analysis this trait is not important.

2. Invariant

   The invariant for the class is not stated in the **invariant** clause. Additional analysis of the interface specifications did not result in finding the invariant. Hence, the only restriction on the class behaviour is stated in the **exemption** clause in `File.lsl`. It requires that file which has access mode *READ* cannot be opened with mode *WRITE* or *READ_WRITE*.

3. There is only one constructor of the class; however, its form shows that it might have two initial states. After DNF analysis of the interface specification, one initial state is when the file is created and it is empty, and a second initial state is when existing file is opened (and it is empty or not). The first case is a result of two circumstances: (1) the value of the mode parameter requires creation of the file, and (2) the file does not exists or the error occured while opening the file.

4. There are a number of inspectors in the class specifications: *CurrOffset, EOF, Error, GetName, IsEmpty, isValid.* After analysis, it can be concluded that for any value of *OpenFile.mode* these inspectors can be applied in any, but error states.

5. There are a number of modifiers in the class specifications: *Erase, Flush, Read, SeekTo, SeekToBegin, SeekToEnd, Write.* When *OpenFile.mode* = *READ* *Erase, Flush, Write* cannot be applied in any state.

6. After determining the input and output domains for every individual operation, all operations were grouped together (their pre and post-states), and additional DNF analysis was performed. It did not result in any additional partitioning of the input/output space.

7. Two different state machines were created. One, Figure 12, is used to test the file when *open.Mode* is *READ*. The other, Figure 13, is used to test the file when *open.Mode* is *WRITE* or *READ_WRITE*

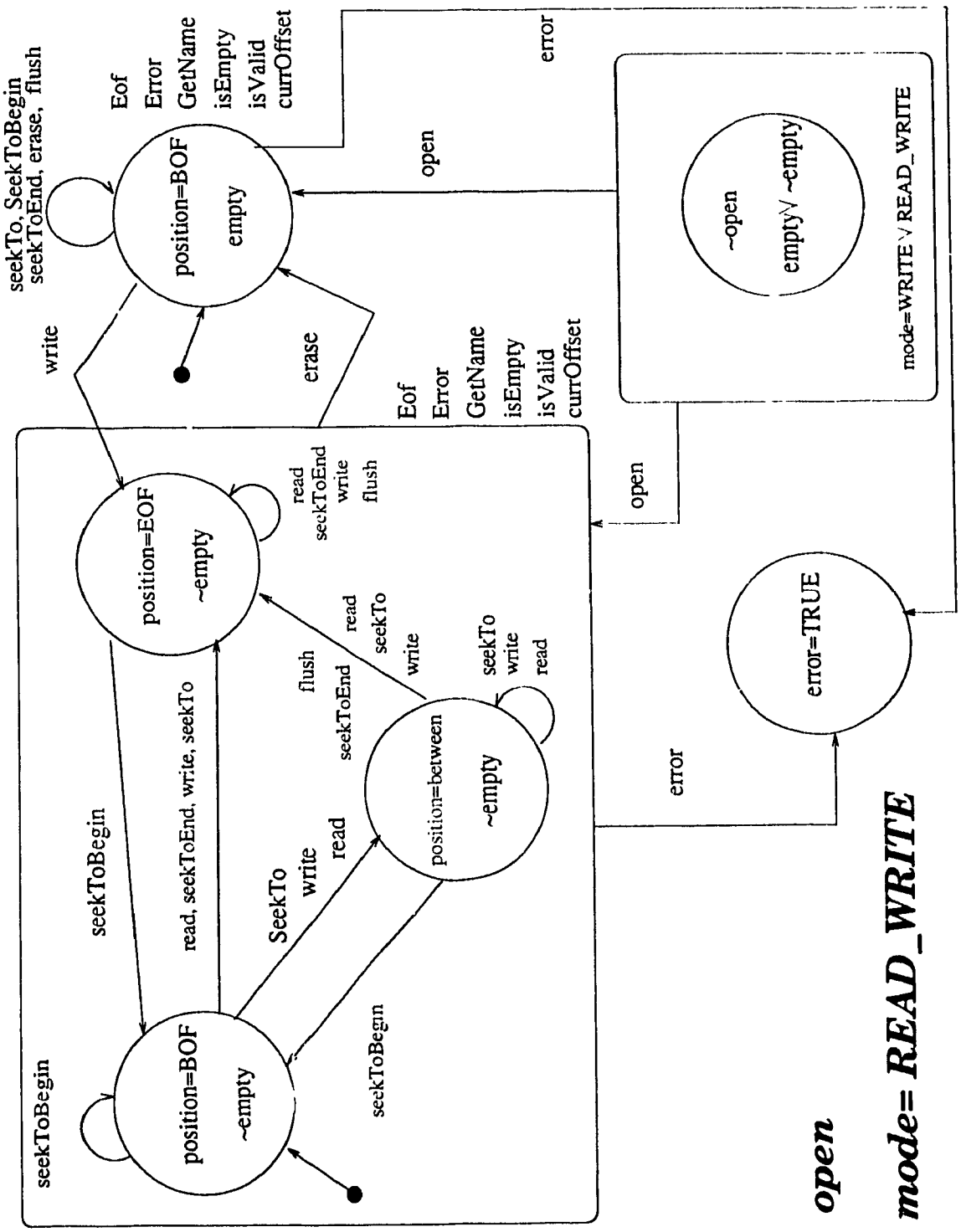

Figure 12: FSA for `RWFile` class, open for READ_WRITE or WRITE_ONLY

Figure 13 FSA for RWFile class, open for READ_ONLY

9.3 Analysis of BtreeOnDisk Class

As a second example of the application of our methodology the class `RWBTreeOnDisk` [rogue] was chosen. The reasons behind that choice are as follows

1. Analysis of the formal specifications of this class finds the the existence of several initial states. The existence of these states and values of the variables are not clearly visible from informal specifications.

2. Interface specifications **uses** two LSL traits. One distinguished sort `BtreeOnDisk.lsl` is a tuple which has an element which is also a tuple. The other, `BtreeAux.lsl`, does not provide any distinguished sort but keys and partitioners.

In addition, by choosing `RWBtreeOnDisk` class we hoped to illustrate some aspects of LSL analysis. For example, `BtreeOnDisk` is defined by seven attributes, but only one determines its *basic distinguishable domains*.

Class Description

Class `RWBTreeOnDisk` [rogue, page 33-1] represents an ordered collection of associations of keys and values. The ordering is determined by comparing keys using an external function. The user can set this function. Duplicate keys are not allowed. Given a key, the corresponding value can be found.

Comments

- The text in the library description [rogue] does not give any btree theory. In contrast, we specify behavior of btree object in the LSL trait. It is necessary to understand the sufficiency of the class specification when the user selects data structures for a project.

- Method *applyToKeyAndValue* traverses btree in the order of its key values, applying the user specified function with the key and value as arguments. Btree theory does not have any mechanism to visit the tree nodes in order. To specify the ordering of keys, we introduced the concept of priority queue. Converting a Btree structure to its priority queue and traversing priority queue nodes explains to the user how the function works.

- Relations of *BtreeOnDisk* to the file where it resides are expressed using *FileManager* theory: the object that knows how the file space is allocated, the beginning and end of file, and how to match Btree keys and values stored in the file.

9.3.1 Btree.lsl Trait Analysis

Btree : **trait**

includes *TotalOrder*(*Key*), *Set*(*Key*, *Node*), *Set*(*Node*, *Btree*), *Set*(*Node*, *SetOfNodes*), *Integer*

introduces

btree : *Int*, *Int* → *Btree*
order : *Btree* → *Int*
fullness : *Btree* → *Int*
son : *Key*, *Btree* → *Node*
son : *Node*, *Btree* → *Node*
leaf : *Node*, *Btree* → *Bool*
hasClosure : *Key*, *Node* → *Bool*
Closure : *Key*, *Node* → *Key*
onestepSerch : *Key*, *Node*, *Btree* → *SetOfNodes*
serchPath : *Key*, *Node*, *Btree* → *SetOfNodes*
height : *Btree* → *Int*
root : *Node*, *Btree* → *Bool*
add : *Key*, *Btree* → *Btree*
remove : *Key*, *Btree* → *Btree*
value : *Key*, *Btree* → *Value*
_ ∈ _ : *Key*, *Btree* → *Bool*
numItems : *Btree* → *Int*

asserts

Btree **generated by** *btree*, *add*
∀ $n, n_1, n_2, r, r_1, fail$: *Node*,
k, k_1, k_2 : *Key*, *bt* : *Btree*, *h*, *f*, *ord* : *Int*
$k \in n \wedge n \in bt == k \in bt$
order(*btree*(*ord*, *f*)) == *ord*
fullness(*btree*(*ord*, *f*)) == *f*

$numItems(btree(ord, f)) == 0$

$numItems(add(k, bt)) ==$ **if** $k \in bt$ **then** $numItems(bt)$ **else**

$\quad numItems(bt) + 1$

$n \in bt \Rightarrow size(n) \geq fullness(bt)$

$n \in bt \Rightarrow size(n) \leq order(bt)$

$(k \in n \wedge n \in bt) \Rightarrow (son(k, bt) \neq n \vee leaf(n, bt))$

$(son(k, bt) = n \wedge son(k, bt) = n_1) \Rightarrow n = n_1$

$(son(k, bt) = n \wedge son(k_1, bt) = n) \Rightarrow k = k_1$

$k_1 \in son(k, bt) \Rightarrow k_1 > k$

$son(n, bt) \neq n \vee leaf(n, bt)$

$(son(n, bt) = n_2 \wedge son(n_1, bt) = n_2) \Rightarrow n = n_1$

$(k \in n \wedge k_1 \in son(n, bt)) \Rightarrow k_1 \geq k$

$leaf(n, bt) == son(n, bt) = n$

$leaf(n, bt) == k \in n \Rightarrow son(k, bt) = n$

$hasClosure(k, n) == \neg(k_1 \in n \wedge k_1 < k)$

$Closure(k, n) = k_1 == (k_1 \in n \wedge k_1 > k \wedge$

$\quad ((k_2 \in n \wedge k_2 < k) \Rightarrow k_1 < k_2))$

$onestepSerch(k, n, btree(ord, f)) == \{\}$

$onestepSerch(k, n, bt) = \{n_1\} ==$

$\quad (hasClosure(k, n) \wedge son(Closure(k, n), bt) = n_1) \vee$

$\quad (\neg hasClosure(k, n) \wedge son(n, bt) = n_1)$

$serchPath(k, n, btree(ord, f)) == \{\}$

$serchPath(k, n, bt) ==$ **if** $(leaf(n, bt) \wedge k\neg \in n)$

$\quad$ **then** $\{fail\}$ **else**

$\quad$ **if** $k \in son(Closure(k, n), bt) \vee k \in son(n, bt)$ **then**

$\quad onestepSerch(k, n, bt)$ **else**

$\quad onestepSerch(k, n, bt) \cup ($ **if** $hasClosure(k, n)$ **then**

$\quad serchPath(k, son(Closure(k, n), bt), bt)$ **else**

$$searchPath(k, son(n, bt), bt))$$

$$height(btree(ord, f)) == 0$$
$$height(bt) = h == size(searchPath(k, r, bt)) = h \wedge$$
$$h \geq size(searchPath(k_1, r_1, bt)) \wedge k \in bt \wedge k_1 \in bt \wedge r \in$$
$$bt \wedge r_1 \in bt$$

$$root(\{\}, btree(ord, f))$$
$$root(n, bt) == n \neq son(k, bt) \wedge n \neq son(n_1, bt)$$
$$(k \in n \wedge n \in bt) == fail\neg \in searchPath(k, n_1, bt) \wedge$$
$$root(n_1, bt)$$
$$k \in add(k_1, bt) == k = k_1 \vee k \in bt$$
$$\neg(k \in remove(k, bt))$$
$$remove(k, btree(ord, f)) == btree(ord, f)$$

implies

$$\forall\ k : Key, n : Node, bt : Btree$$
$$son(k, bt) = son(n, bt) \Rightarrow leaf(n, bt)$$

Figure 14: `Btree.lsl` definition

Analysis

1. *Btree* is a distinguished sort which has two attributes: *order* and *fullness*. *Order* of btree defines the maximum number of keys in a node. *Fullness* of btree defines the minimum number of keys in each node. The values of these attributes are constant during the existence of distinguished sort's instantiation. However, *Btree* is also a set of nodes, and each node is a set of keys. Therefore, the distinguished sort is a complicated model. The creation of the *Btree* results in the creation of a set with one node, which is an empty set of keys. By adding/deleting an element (key) to *Btree*, the number of nodes in the tree may change to preserve the fullness requirement.

2. The structure of *Key* is not explicitly defined. *Key* indirectly stores the value of the element in the *Btree*. However, this characteristic of *Btree* is not fully

defined. There is only one operation which is related to the value of the element stored in the tree; this operation is called *value*.

3. Distinguished sort initializer

   Trait `Btree.lsl` contains only one sort initializer - *btree*. The sort *btree* creates *Btree*, with the given values of the constant parameters *fulness* and *order*. The height of the empty *Btree* = 0.

4. Distinguished sort partitioner

   In the LSL trait definition, none of the inspectors qualify to be an initial sort partitioner; since the nature of *Btree* is a set of sets. Therefore, the inspectors from `Set.lsl` trait definition should be analyzed. Only *isEmpty* qualifies, since it gives a finite (two) number of domains.

5. Basic distinguished domains

   After applying the sort partitioner (*isEmpty*) to the sort initializer (*btree*), the following basic distinguished domains were obtained: *empty* and $\neg$*empty*.

6. The input and output domains have to be specified for the following sort alterators:

   - Operation *add*

   input: *empty* $\vee$ $\neg$*empty*

   output. $\neg$*empty*

   - Operation *remove*

   input: $\neg$*empty*

   output: *empty* $\vee$ $\neg$*empty*

7. The input domains have to be specified for the following sort examiners:

   - Operation *order*

   input: *empty* $\vee$ $\neg$*empty*

   - Operation *fullness*

   input: *empty* $\vee$ $\neg$*empty*

   - Operation *son*

   input: *empty* $\vee$ $\neg$*empty*

- Operation *leaf*

input: $empty \vee \neg empty$

- Operation *hasClosure*

input: $empty \vee \neg empty$

- Operation *Closure*

input: $empty \vee \neg empty$

- Operation *onestepSerch*

input: $empty \vee \neg empty$

- Operation *serchPath*

input: $empty \vee \neg empty$

- Operation *height*

input: $empty \vee \neg empty$

- Operation *root*

input: $empty \vee \neg empty$

- Operation *value*

input: $\neg empty$

- Operation $\_\_ \in \_\_$

input: $\neg empty$

- Operation *numItems*

input: $\neg empty$

8. Invariant

None

9.3.2 BtreeAux.lsl Trait Analysis

BtreeAux : **trait**

includes *Btree*, *PriorityQueue*(*Key* **for** *E*, *PriorityQueue* **for** *C*)

introduces

$\_\_\{\_\_\} : Fun, Key \rightarrow Key$

$toBtree : PriorityQueue \rightarrow Btree$

$apply : Btree, Fun \rightarrow Btree$

$\_\_\{\_\_,\_\_\} : comparisonFun, Key, Key \rightarrow Int$

$comparisonfun : Btree \rightarrow comparisonFun$

asserts

$\forall\ k, k_1 : Key, bt : Btree, f : Fun, pq : PriorityQueue,$

$comp : comparisonFun$

$k \in toBtree(pq) == k \in pq$

$f\{k\} == k$

$apply(toBtree(add(k, pq)), f) = add(k, apply(toBtree(pq), f)) ==$

$head(pq) = k$

$comp\{k, k_1\} < 0 == k < k_1$

$comp\{k, k_1\} > 0 == k > k_1$

$comp\{k, k_1\} = 0 == k = k_1$

Figure 15: `BtreeAux.lsl` definition

Analysis

1. Distinguished sort initializer

   None: this trait only adds operations to `Btree.lsl`

2. Distinguished sort partitioner

   None

3. Invariant

   None

4. The input domains have to specified for the following sort examiners:

   - Operation _{_}

   input: $empty \vee \neg empty$

   - Operation *toBtree* - type conversion

   input: $empty \vee \neg empty$

   - Operation *apply*

   input: $empty \vee \neg empty$

   - Operation _{_,_}

   input: $empty \vee \neg empty$

   - Operation *comparisonfun*

   input: $empty \vee \neg empty$

9.3.3 FileManager.lsl Trait Analysis

FileManager : **trait**
  includes *File*
  introduces
    $filemanager : File \rightarrow FileManager$
    $file : FileManager \rightarrow File$
    $allocate : FileManager, Int \rightarrow Int$
    $start : FileManager \rightarrow Int$
    $size : FileManager, Int \rightarrow Int$
  asserts
    $\forall f : FileManager, fl : File, nbytes : Int$
      $file(filemanager(fl)) == fl$
      $size(f, allocate(f, nbytes)) == nbytes$
      $allocate(f, nbytes) \leq start(f)$

Figure 16: `FileManager.lsl` definition

Analysis

1. `FileManager.lsl` trait enhances the `File.lsl` trait with operations required for file on disk access. This trait does not have an initializer. Therefore, an analysis of `File.lsl` trait should be performed. And only those `BtreeOnDisk` partitions which are important should be taken into consideration.

2. The input domains have to be specified for the following sort examiners:

   - Operation *allocate*

   input: $empty \vee \neg empty$

   - Operation *start*

   input: $empty \vee \neg empty$

   - Operation *size*

   input: $empty \vee \neg empty$

   - Operation *filemanager* - type convertor

   input: $empty \vee \neg empty$

   - Operation *file* - type convertor

   input: $empty \vee \neg empty$

9.3.4 BreeOnDisk.lsl Trait Analysis

BtreeOnDisk : **trait**

includes *String*(*Char*, *Str*), *Btree*(*String* **for** *Key*), *FileManager*, *Integer*

Smode **enumeration of** *v6Style*, *v5Style*

Char **enumeration of** *null*, *aChar*

introduces

BtreeOnDisk : *FileManager*, *Int*, *Int*, *Bool*, *Smode*, *Int*, *Btree* → *BtreeOnDisk*

keyTostr : *Key* → *Str*

dataTokey : *Data* → *Key*

dataToNode : *Data* → *Node*

btree : *BtreeOnDisk* → *Btree*

file : *Btree* → *File*

$valueOffset : Key, FileManager \rightarrow Int$

asserts

BtreeOnDisk **generated by** *BtreeOnDisk*

$\forall\ BT : BtreeOnDisk, f : FileManager, NumBuf : Int, keylen : Int,$

$IgnoreNull : Bool, smode : Smode, start : Int, btr : Btree, k : Key$

$BT = BtreeOnDisk(f, NumBuf, keylen, IgnoreNull, smode, start, btr)$

$\wedge k \in btr ==$

$len(keyTostr(k)) \leq keylen \wedge (null \in keyTostr(k) \Rightarrow IgnoreNull) \wedge$

$(\neg IgnoreNull \Rightarrow \neg(null \in keyTostr(k))) \wedge$

$root(dataToNode(read(open(file(f), READ), start(f),$

$size(f, start(f))).reddata), btr) \wedge start = start(f)$

$btree(BtreeOnDisk(f, NumBuf, keylen, IgnoreNull, smode, start, btr))$

$== btr$

Figure 17: `BtreeOnDisk.lsl` definition

Analysis

1. *BtreeOnDisk* has seven attributes: *FileManager, numberOfBuffers, keylen, IgnoreNull, Smode, start, btree.* They contain the following information:

   • *FileManager* establishes a link to operations on the disk file containing btree; this value does not change once the link is established.

   • *numberOfBuffers* contains an integer value, which specifies the number of buffers in memory to store a copy of btree; this value does not change during any operation;

   • *keylen* defines the length of the key and does not change during any operation;

   • *IgnoreNull* defines the type of string of characters. If it is set to TRUE, the null character is ignored, otherwise it is not ignored. This value does not change during any operation;

   • *Smode* defines the type of mode; its value belongs to the enumeration, and it cannot be changed by any operation;

   • *start* number defines where in the file on disk the content of btree is stored; this cannot be changed by any of the operations;

   • *btree* is a dynamic structure, and its content changes.

Therefore the following parameters compose global values: *FileManager*, *numberOfBuffers*, *keylen*, *IgnoreNull*, *Smode*, *start*. However, the *btree* attribute of *BtreeOnDisk* is the only one which defines basic distinguishable states.

2. Values of other attributes are not restricted by any constraints. Therefore, they may remain as global attributes.

3. Distinguished sort initializer

   BtreeOnDisk is a sort initializer. However, no specific sort partitioner for that initializer is present. After analyzing all of the attributes of *BtreeOnDisk*, the only mutable attribute is *btree*. Therefore, the basic distinguished domains are determined by this attribute.

4. Distinguished sort partitioner

   There are none. Partitions are the same as for `Btree.lsl` distinguished sort.

5. Distinguished sort examiner

   $valueOffset$ is an examiner, and it can be applied to whole domain of *BtreeOnDisk*.

6. All remaining operators *KeyTostr*, *dataTokey*, *dataToNode*, *btree*, *file* are type convertors. They do not change the *btree*.
   input: $empty \vee \neg empty$

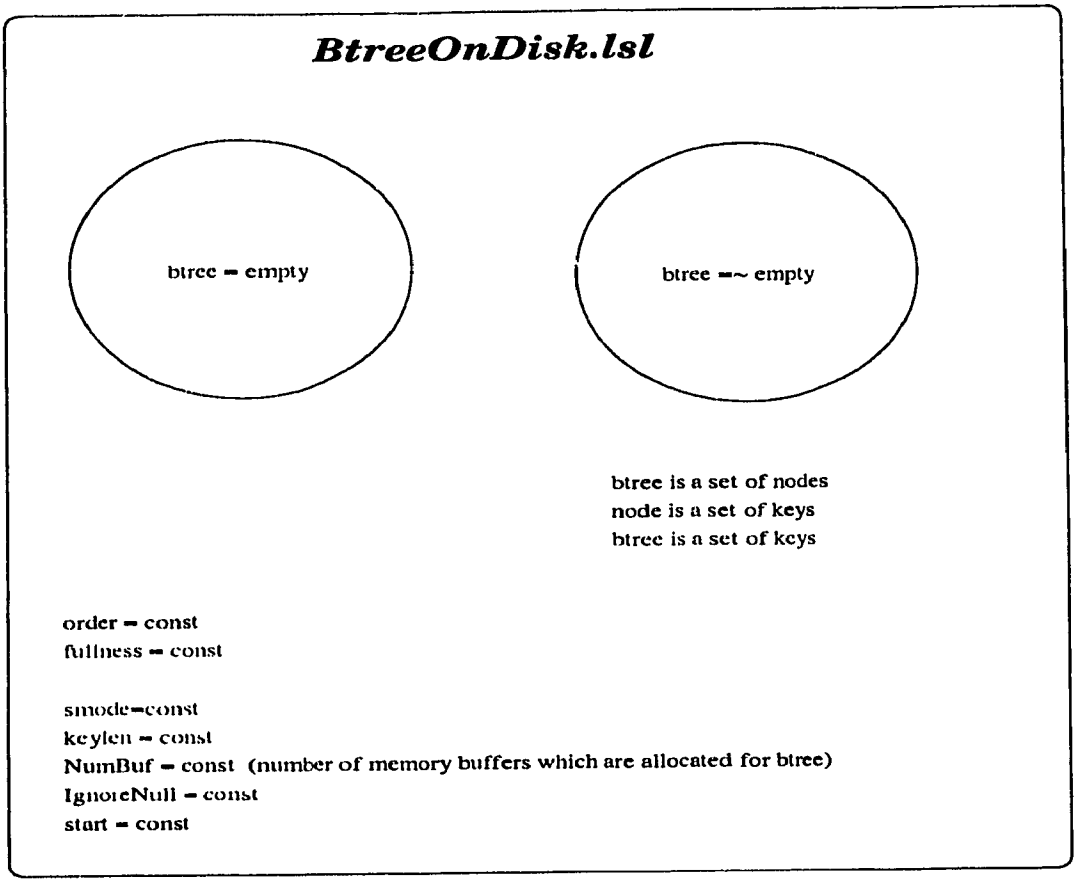

Figure 18: Valid states for `Btree` trait

9.3.5 Interface Specification Analysis

The full interface specifications in Larch/C++ for `RWBtreeOnDisk` class from Rogue Wave Tools.h++ library is included in [ACCUA94]. The interface specification included below is presented in a form which allows efficient DNF analysis.

class RWBtreeonDisk
uses BtreeOnDisk(size_t **for** Int, RWStoredValue **for** Value,
RWBoolean **for** bool, RWoffset **for** Int),
BtreeAux(RWdiskTreeCompare **for** comparisonFun);
public:

1. RWBtreeOnDisk - constructor
RWBtreeOnDisk(RWFileManager& f, unsigned nbuf=10,
createMode omode = autoCreate, unsigned keylen = 16,
RWBoolean ignorenull = FALSE, rwoffset start = RWNIL,
styleMode smode = v6Style, unsigned halfOrder = 10,
unsigned minFill = 10)
pre: true
modifies $self$;
post:

(a) $omode = autoCreate \wedge$
$start = RWNIL \wedge start(f) \neq RWNIL \wedge$
$self' = BtreeOnDisk(f, nbuf, keylen, ignoreNull, smode, start(f),$
$btree(minFill, halfOrder))$

(b) $\vee$
$omode = autoCreate \wedge$
$start = RWNILL \wedge start(f) = RWNIL \wedge$
$self' = BtreeOnDisk(f, nbuf, keylen, ignoreNull, smode, start,$
$btree(minFill, halfOrder)) \wedge height(btrec(self')) = 0$

(c) $\vee$
$omode = autoCreate \wedge$
$start \neq RWNILL \wedge$
$self' = BtreeOnDisk(f, nbuf, keylen, ignoreNull, smode, start,$
$btree(minFill, halfOrder)) \wedge$

$height(btree(self')) = 0$

(d) $\vee$

$omode \neq autoCreate \wedge$

$self' = BtreeOnDisk(f, nbuf, keylen, ignoreNull, smode, start,$

$btree(minFill, halfOrder)) \wedge$

$height(btree(self')) = 0$

2. apply

void applyTokeyAndValue((*ap)(const char* c, RWStoredValue, void* x))

pre: TRUE

modifies $self$;

post: $self' = apply(btree(self^), (*ap))$;

3. clear

void clear()

pre: TRUE

modifies $self$;

post: $height(btree(self')) = 0$

4. contains

RWBoolean contains(const char* ky) const

pre: TRUE

post: $self' = self^ \wedge result = ky \in btree(self^)$;

5. entries

size_t entries()

pre: TRUE

post: $self' = self^ \wedge result = numItems(btree(self^))$;

6. findValue

RWStoredValue findValue(const char* ky)

pre: TRUE

post: $(*ky)^ \in btree(self^) \wedge result = value((*ky)^) \vee$

$(*ky)^ \notin btree(self^) \wedge result = RWNILL$;

7. height

int height()

pre: TRUE

post: $self' = self^{\wedge} \wedge result = height(btree(self^{\wedge}))$;

8. insertKeyandValue

int insertKeyandValue(const char* ky, RWStoredValue v)

pre: TRUE

modifies $self$;

post: $result = TRUE \wedge \neg error(file(self')) \wedge$
$btree(self') = add(btree(self^{\wedge}), (*ky)^{\wedge}) \wedge v = value((*ky)^{\wedge}, btree(self^{\wedge}))$
$\vee$
$result = FALSE \wedge error(file(self')) \wedge$
$btree(self') = add(btree(self^{\wedge}), (*ky)^{\wedge}) \wedge v = value((*ky)^{\wedge}, btree(self^{\wedge}))$

9. isEmpty

RWBoolean isEmpty() const

pre: TRUE

post: $self' = self^{\wedge} \wedge result = (numItems(self^{\wedge}) = 0)$;

10. remove

void remove(const char* ky)

pre: TRUE

modifies $self$;

post: $self' = remove((*ky)^{\wedge}, btree(self^{\wedge}))$;

11. rootLocation

RWoffset rootLocation()const

pre: TRUE

post: $self' = self^{\wedge} \wedge$

$root(dataToNode(read(file(self^{\wedge}), result, size(file(self^{\wedge}), result))), btree(self^{\wedge}))$;

12. setComparison

RWdiskTreeCompare setComparison(RWdiskTreeCompare fun)

pre: TRUE

post: $self' = self^{\wedge} \wedge comparisonfun(btree(self^{\wedge})) = fun \wedge result = fun$;

Interface Analysis

1. The class interface specification uses two LSL traits: `BtreeOnDisk` and `BtreeAux`.

   The results of the analysis of `BtreeOnDisk.lsl` trait are in Section 9.3.4, and for trait `BtreeAux.lsl` in Section 9.3.2.

2. Invariant - None

3. Constructor

   There is only one constructor for the `RWBtreeOnDisk` class. However, it has two initial states which depend on the values of the parameters. When the file containing Btree exists and is not empty, and is open without error, *btree* in memory is not empty. Otherwise, the file on disk is created, and it contains empty btree.

4. There are a number of inspectors in the class specifications: *contains, entries, findValue, height, isEmpty, rootLocation.* After DNF analysis of each of these inspectors, it can be concluded that they can be applied in any state.

5. There are a number of modifiers in the class specifications: *remove, insertKeyand-Value, clear.* After DNF analysis of each of these modifiers, it can be concluded that they can be applied in any state.

6. There are also a number of other operations, which perform particular functions, and cannot be classified as constructors, destructors, inspectors, or modifiers. They are *apply* and *setComparison.* However, from the point of view of this analysis, they act as inspectors. They can be applied in any state. Nevertheless, for testing the results of their invocation, they should be evaluated separately.

7. After determining the input and output domains for every individual operation, all operations were grouped together (their pre and post states), and additional DNF analyses were performed. They did not result in additional partitioning of the input/output space.

8. Based on results of the analysis, the following FSA was created, as shown in Figure 19 and Figure 20. The structure of the machine is a sufficient guideline for test suit derivation.

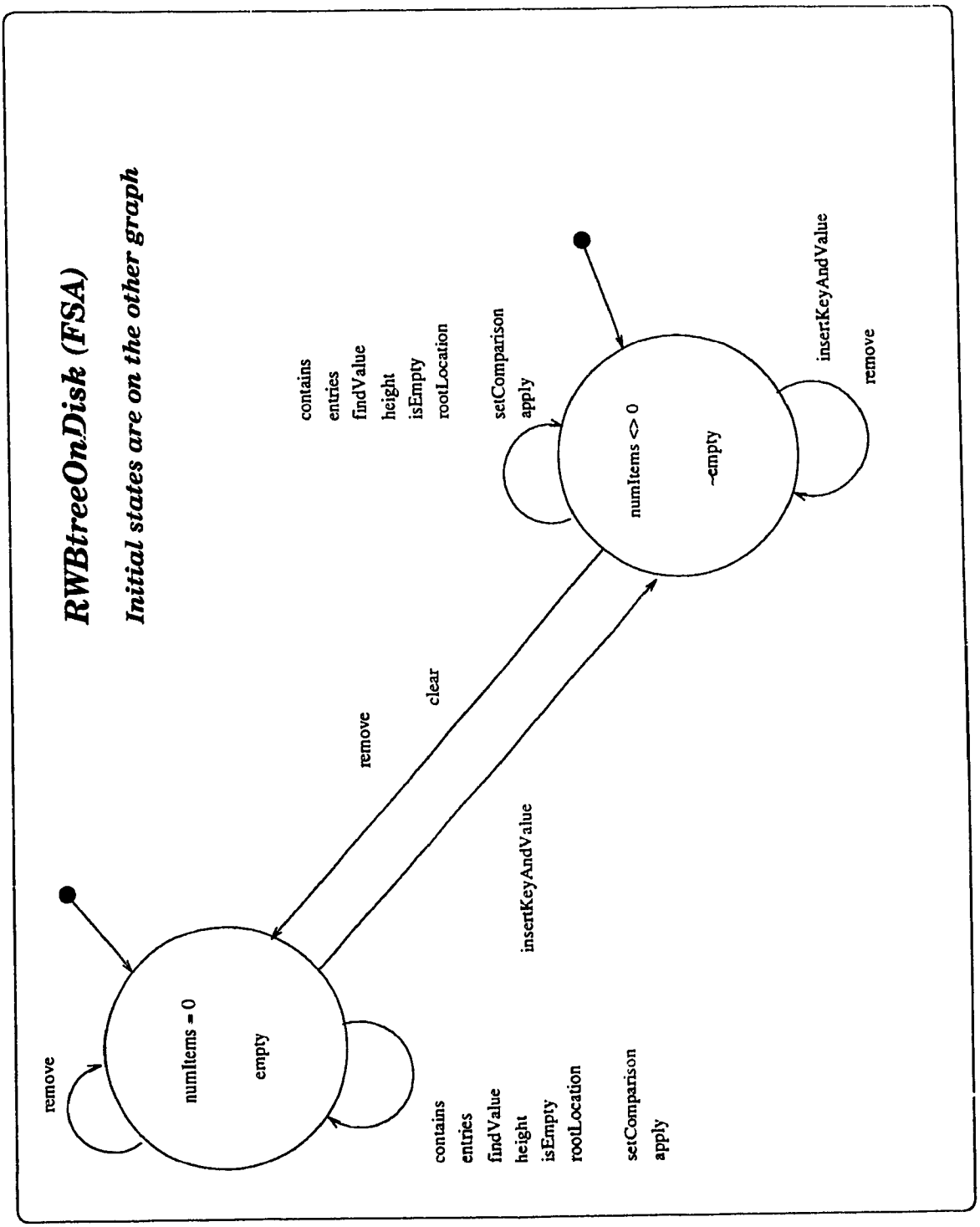

Figure 19: Finite State Automaton for `RWBtreeOnDisk` class

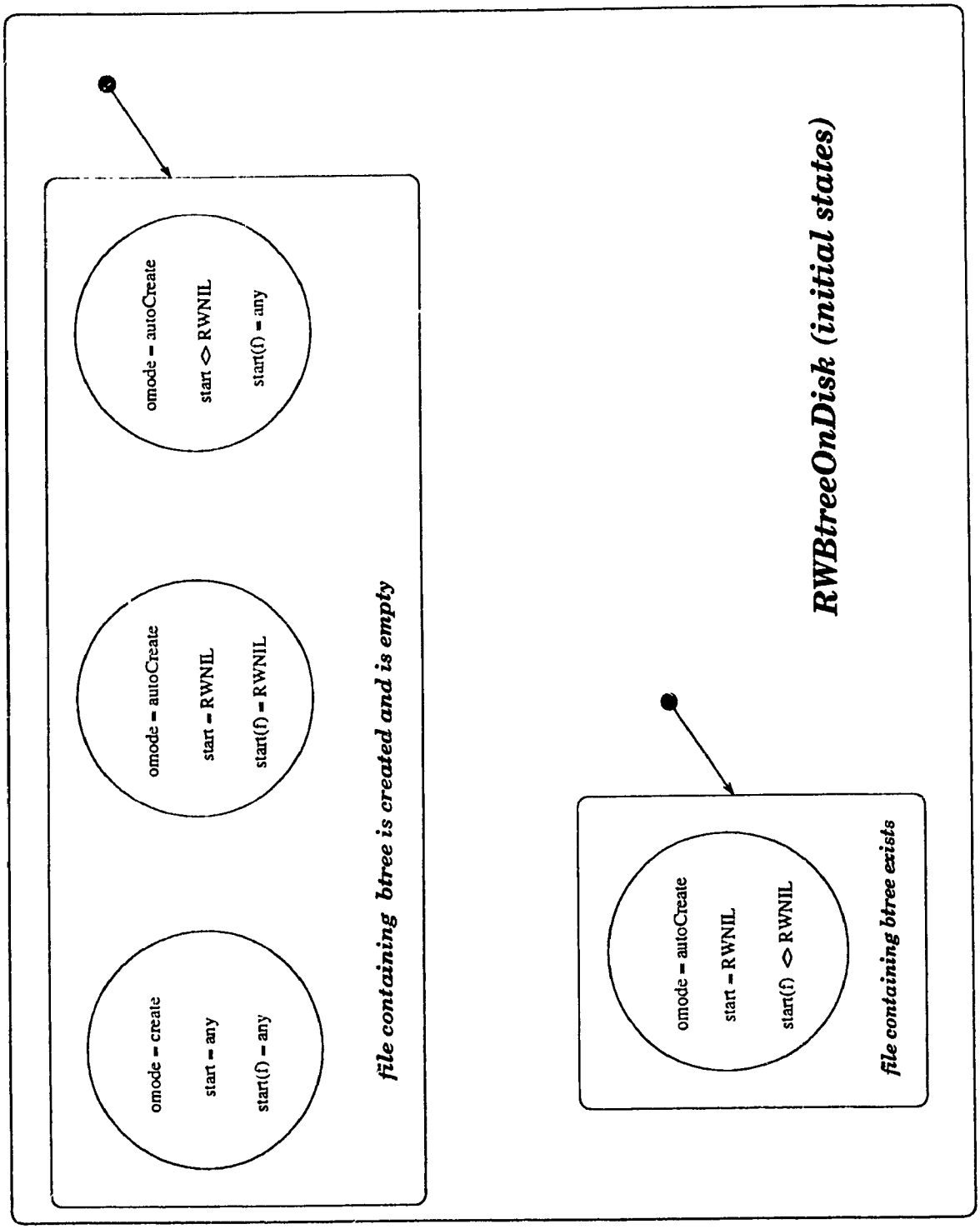

Figure 20: FSA for `RWBtreeOnDisk` class: Initial States

9.4 Analysis of RWHashTable Class

As a third example of the application of our methodology, the class `RWHashTable` [rogue] was chosen. The reasons behind this choice are the following:

1. There is an invariant present in the **invariant** clause of the interface specification.

2. The interface specification uses two LSL traits: `ClassID.lsl` does not provide any information related to domain partitions; the second trait, `HashTable` contains all information used to determine the *basic distinguishable domains*.

3. The *basic distinguishable domains* were analyzed, and one of the attribute of the domain was dismissed (from `Array.lsl` trait).

In addition, by choosing `RWHashTable` class, we hoped to illustrate some aspects of LSL analysis. For example, `HashTable` does not contain any generators and inspectors.

Class Description

Class `RWHashTable` [rogue][page 57-1] is a simple hash table for objects inheriting from `RWCollectable`. The class inherits from `RWCollection`, which in turn, inherits from `RWCollectable` base class. Duplicate objects are allowed in the hash table. To find any object that matches a key, the key's virtual function hash() is first called to determine the bucket in which the object occurs. The bucket is then searched linearly by calling the virtual function *isEqual()* for each candidate with the key as the argument. The first object to return a TRUE value is the returned object.

9.4.1 HashTable_V.lsl Trait Analysis

HashTable_V(E, HT) : **trait**

includes *Bag*(E, B), *array*(B, HT), *Std_Eq*(*equal*, E), *Std_Eq*(*equalHT*, HT)

introduces

DefaultCapacity :→ *Int*

toCollection : $HT \rightarrow B$

isEmpty : $HT \rightarrow$ *Bool*

hash : $E \rightarrow$ *Int*

$\_\_ \in \_\_ : E, HT \rightarrow Bool$

$insert : E, HT \rightarrow HT$

$remove : E, HT \rightarrow HT$

$size : HT \rightarrow Int$

$apply : HT, F \rightarrow HT$

$\_\_\{\_\_\} : F, E \rightarrow E$

$occurencesOF : E, HT \rightarrow Int$

$resize : Int, HT \rightarrow HT$

asserts

HT **generated by** $create, insert$

HT **partitioned by** $isEmpty, \in$

$\forall\ e, e_1 : E, ht, ht_1 : HT, i, j : Int, b : B, f : F$

$(e \in ht) \Rightarrow (e \in toCollection(ht))$

$(e \in toCollection(ht)) \Rightarrow (e \in ht)$

$isEmpty(create(i))$

$\neg isEmpty(insert(e, ht))$

$(legalIndex(ht, i) \wedge e \in insert(e, ht)[i] \wedge hash(e) = hash(e_1)) \Rightarrow$
$e_1 \in insert(e_1, ht)[i]$

$(legalIndex(ht, i) \Rightarrow \neg(e\neg \in insert(e, ht)[i]))$

$(legalIndex(ht, i) \wedge (e \in insert(e, ht)[i])) \Rightarrow$
$(\neg(e \in insert(e, ht)[j] \wedge legalIndex(ht, j)) \vee i = j)$

$e \in (insert(e_1, ht)) == equal(e, e_1) \vee e \in ht$

$remove(e, insert(e_1, ht)) ==$ **if** $equal(e, e_1)$ **then** ht **else**
$insert(e_1, remove(e, ht))$

$apply(insert(e, ht), f) == insert(f\{e\}, apply(ht, f))$

$equal(f\{e\}, e) \wedge (hash(e) = hash(f\{e\}))$

$size(create(i)) = 0$

$size(insert(e, ht)) == size(ht) + 1$

$occurencesOF(e, insert(e_1, ht)) ==$ **if** $equal(e, e_1)$ **then**
$occurencesOF(e, ht) + 1$ **else** $occurencesOF(e, ht)$
$occurencesOF(e, create(i)) == 0$

$maxIndex(resize(i, ht)) == i - 1$
$resize(i, insert(e, ht)) == insert(e, resize(i, ht))$
$resize(j, create(i)) = create(j)$

implies

converts *apply*, *remove*

exempting $\forall\ f : F, i : Int, e : E$
$apply(create(i), f), remove(e, create(i))$

Figure 21: `HashTable_V.lsl` definition

Analysis

1. `HashTable_V` is an array. Each element of the array is a bag of elements. Therefore, the same elements may appear in a hash table more than once. The array has a predefined maximum capacity. However, the maximum capacity can be changed.

2. Distinguished sort initializer

   The distinguished sort initializer is not specified in the trait. The sort initializer comes from the `Array.lsl` trait.

   From analysis of the `Array.lsl` (see Appendix A), trait it follows that *create* is a distinguished sort initializer.

   The `Bag.lsl` trait (see Appendix A) analysis does not provide any additional information for hash table analysis.

3. Distinguished sort partitioner

   The `HashTable_V` has one distinguished sort partitioner: *isEmpty*.

4. Basic distinguished domains

   After applying the distinguished sort partitioner to sort initializer, we get the following partitions:

   (I) $legal \wedge empty$
   (II) $legal \wedge \neg empty$
   (III) $\neg legal \wedge (empty \vee \neg empty)$

   The sub-domain (III) is not legal for any of the operations from the trait. Therefore, *legal* will not be taken into consideration while determining the input and output domains for the operations. Hence, there are only two subdomains: $empty \vee \neg empty$.

5. The input and output domains have to specified for the following sort alterators *insert, remove, apply, resize*, _{_}

   • Operation *insert*

   input: $empty \vee \neg empty$
   output: $\neg empty$

   • Operation *remove*

   input: $\neg empty$
   output: $empty \vee \neg empty$

   • Operation *resize*

   input: $empty \vee \neg empty$
   output: $empty \vee \neg empty$

   • Operation *apply*

   input: $\neg empty$
   output: $\neg empty$

   • Operation _{_}

   input: $empty \vee \neg empty$
   output: $empty \vee \neg empty$

6. The input domains have to specified for the sort examiners *hash*, _ ∈ _, *size*, *occurencesOf*.

- Operation *hash*

input: $empty \vee \neg empty$

- Operation $\_\_ \in \_\_$

input: $empty \vee \neg empty$

- Operation *size*

input: $empty \vee \neg empty$

- Operation *occurencesOf*

input: $empty \vee \neg empty$

7. There is one sort type converter: *toCollection*

   - Operation *toCollection*

   input: $empty \vee \neg empty$

8. There is one constant definition: *DefaultCapacity*

9. The **exempting** clause contains two operations, *apply* and *remove*. These operations are not defined for the empty hash table.

   - Operation *apply*
   - Operation *remove*

9.4.2 HashTable.lsl Trait Analysis

HashTable(*E*, *HT*) : **trait**

includes *HashTable_V*(*Obj_E*, *HT*), *IsA*

introduces

Num_Obj_in_bucket : *Obj_E*, *B* → *Int*

asserts

∀ *ht* : *HT*, *e*, *e*$_1$: *Obj_E*, *i*, *j* : *Int*

legalIndex(*create*(*j*), *i*) ⇒ *Num_Obj_in_bucket*(*e*, *create*(*j*)[*i*]) = 0

Num_Obj_in_bucket(*e*, *insert*(*e*$_1$, *ht*)[*i*]) == **if** *e* = *e*$_1$ **then**

Num_Obj_in_bucket(*e*, *ht*[*i*]) + 1 **else** *Num_Obj_in_bucket*(*e*, *ht*[*i*])

IsA(*ht*) = *_RWHASHTABLE*

Figure 22: `HashTable.lsl` definition

Analysis

1. The `HashTable` trait includes `HashTable_V` trait. It adds one inspector.

2. Distinguished sort initializer

   Since the trait does not introduce any distinguished sort initializers, the distinguished sort initializer is the same as for `HashTable_V`.

3. Distinguished sort partitioner

   Since this trait does not introduce any distinguished sort partitioners, the distinguished sort partitioner is the same as for `HashTable_V`.

4. Basic distinguished domains

   Basic distinguished domains are the same as for `HashTable_V`.

5. The input domains have to be specified for the following sort examiner:

   • Operation *Num_Obj_in_bucket*

   input: *empty* ∨ ¬*empty*

9.4.3 Interface Specification Analysis

The full interface specification in Larch/C++ for the `RWHashTable` class from Rogue Wave Tools.h++ library [rogue] is included in [ACCUA94]. For this analysis, the specifications were modified. Main modifications resulted in not including some of the operations into the example.

The interface specifications, included below, were changed and are presented in a form that allows efficient DNF analysis.

class RWHashTable : virtual public RWCollection
uses HashTable(RWHashTable **for** HT, RWBoolean **for** Bool),
ClassID(RWClassID);
simulates RWCollection **by** toCollection;
invariant $\forall\ (e : E, e1 : E, i, j : int)((e \in self[i]) =>$
$(\neg(e1 \in self[j] \wedge identical(e, e1) \wedge j \neq i) \wedge$
$legalIndex(self, i) \wedge legalIndex(self, j))$);
extern void ap(RWCollectable*, void*);

.

public:

1. RWHashTable - constructor
   RWHashTable(unsigned $N = DefaultCapacity$)
   pre: TRUE
   modifies $self$;
   post: self' = create(N);

2. RWHashTable - copy constructor
   RWHashTable(const RWHashTable& t)
   pre: TRUE
   modifies $self$;
   post: $self' = create(maxIndex(t) + 1) \wedge$
   $\forall i : int, e : E\ (legalIndex(self', i) =>$
   $(Num\_Obj\_in\_bucket(e, t[i]) = Num\_Obj\_in\_bucket(e, self'))$);

3. clear - modifier

virtual void clear()

pre: TRUE

modifies $self$;

post: $isEmpty(self')$;

4. entries inspector

   virtual unsigned entries() const

   pre: TRUE

   post: $self' = self\hat{} \wedge result = size(self\hat{})$;

5. find - inspector

   virtual RWCollectable* find(const RWCollectable* e1) const

   pre: $*e1 \neq nil$;

   post: $self' = self\hat{} \wedge (e \in self' \wedge e = (e1*)\hat{} \wedge (result*) = e)$

   $\vee(e \in self' \wedge e \neq (e1*)\hat{} \wedge result' = nil)$;

6. insert - modifier

   virtual RWCollectable* insert(RWCollectable* a)

   pre: $*a \neq nil$

   modifies $self$;

   post: $(result = a \wedge self' = insert(self\hat{}, a)) \vee (result = nil \wedge self' = self\hat{})$;

7. isEmpty - inspector

   virtual RWBoolean isEmpty() const

   pre: TRUE

   post: $self' = self\hat{} \wedge result = isEmpty(self')$;

8. isEqual - inspector

   virtual RWBoolean isEqual(const RWCollectable* a) const

   pre: $*a \neq nil$

   post: $self' = self\hat{} \wedge result = (self\hat{} = (a*)\hat{})$;

9. occurencesOf - inspector

   virtual unsigned occurencesOF(const RWCollectable* a) const

   pre: $*a \neq nil$

   post: $self' = self\hat{} \wedge result = occurencesOF((a*)\hat{}, self\hat{})$;

10. remove - modifier

    virtual RWCollectable* remove(const RWCollectable* a)

    pre: $*a \neq nil$

    modifies $self$;

    post: $self' = remove(self\hat{},(a*)\hat{})$;

Analysis

1. Class interface specification **uses** two LSL traits `HashTable_V` and `HashTable`.

   The results of the analyses of both traits are in Section 9.4.1 and 9.4.2.

2. Invariant

   None of the LSL traits introduces any additional restrictions on the defined ADT. There is also no 'hidden' invariant in the **requires** and **ensures** clauses. However, there is an invariant specified in the **invariant** clause of the interface specifications. It states the following: if two elements are identical, they are hashed at the same location, and the reference is legal.

3. Class constructor

   The class `RWHashTable` has two constructors. One of them creates the empty hash table. The other creates the hash table containing a copy of another hash table. Therefore, there are two initial states of the FSA.

4. There are several inspectors in the class interface specifications: *entries, find, isEmpty, isEqual, occurencesOf.* After DNF analysis of each inspector, it can be concluded that they can be applied in any state of the FSA.

5. There are several modifiers in the class interface specifications: *clear, insert, remove.* After DNF analysis of each modifier, it can be concluded that they can be applied in any state of the FSA.

6. After determining the input and output domains for every individual operation, all operations were grouped together (their pre- and post-states), and additional DNF analysis was performed. It did not result in any additional partitioning of the input/output space. Hence, the *input* and *output* domains for all interface operations remain unchanged.

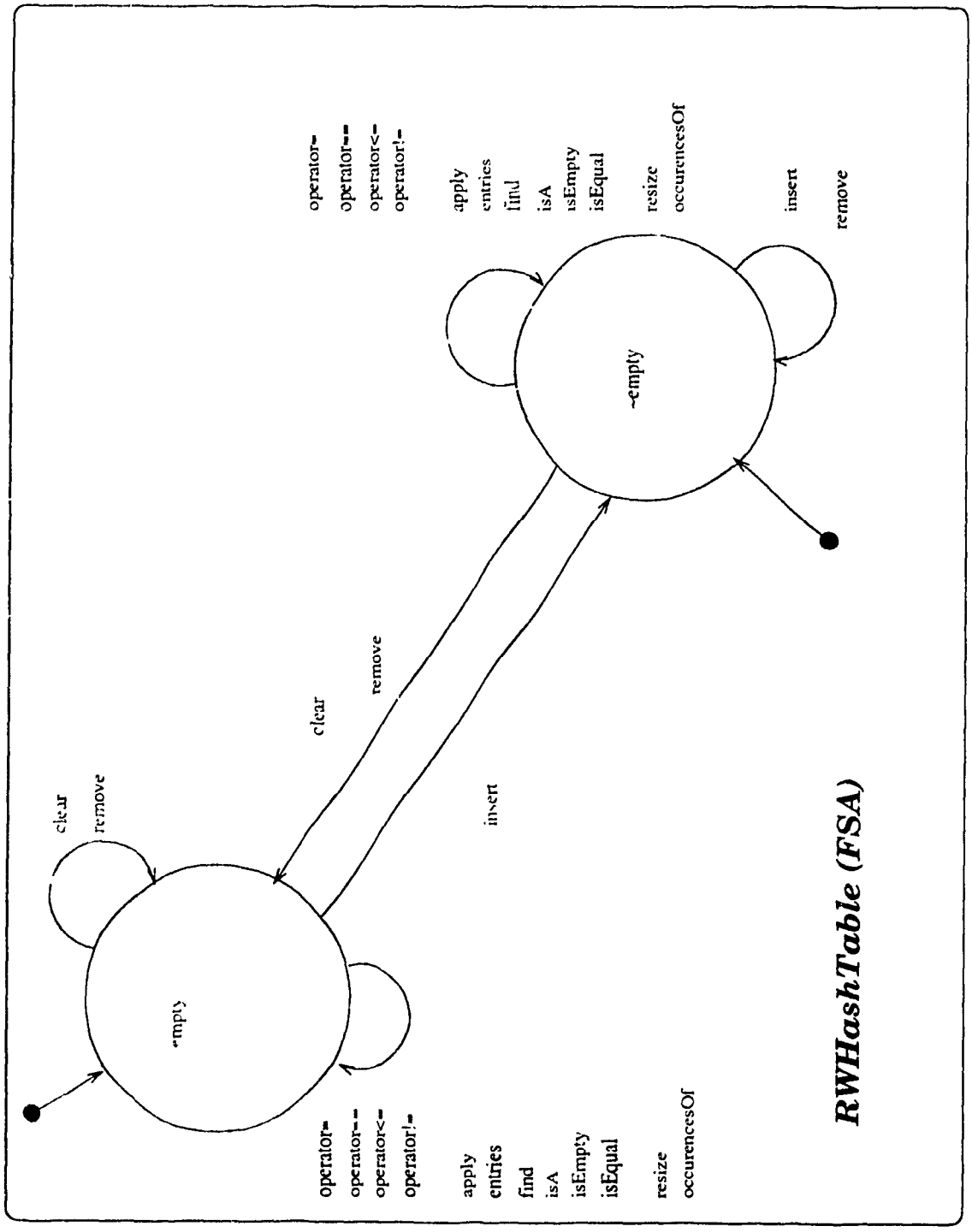

Figure 23: FSA for `RWHashTable` class

9.5 Sample Tests for RWFile Class

Test cases are intended to determine whether a given implementation satisfies all properties required by the specification. Many test selection methods have been developed for the case of the specification of the system being tested, given in the form of a Finite State Machine (FSM). The best known methods are called Transition Tour [NT81], W-method [Chow78], Distinguishing Sequence Method [Gon70], and Unique-Input-Output (UIO) method [SD88]. The test suites derived by each of the above methods will detect any output error of the implementation; that is, if the implementation follows the FSM specification.
In general, the following factors have an influence on the test selection and test value:

1. Test suite length and coverage

   In general, the elimination of a test from the test suite reduces its coverage, and, inversely, the addition of a test will increase the coverage, if a suitable test was selected.

2. Justification of theoretical assumptions

   The proof of error detection in the test-selection methods is based on certain assumptions which are not necessarily satisfied in practice. Among them are: (1) a limited number of states in the implementation; it is usually assumed that the number of states in the implementation is not larger than the number of states in the specification; and (2) incompleteness and special input/output interactions; it is quite common to encounter FSM specifications which include "don't care" entries for certain inputs in certain states.

3. The case of non-deterministic implementations and/or specifications

   When the implementation is non-deterministic, it is impossible to have any guarantee for error detection. If the specification is non-deterministic, the tests are not necessarily repeatable; that is for a given sequence of input, there may be different resulting output sequences, depending on the internal choices of the implementation/specification.

The research goals of this thesis do not include test suite derivation. So, in this section, we only illustrate how the created FSA can be used to test the implementation, without providing a detailed methodology of test suite derivation. Additional

research is required to explore the full possibility of the test suite derivation from any FSA among the methodology proposed in this thesis.

To verify the usefulness of our methodology and the derived FSA, several test cases were created to test the `RWFile` class implementation. The `RWFile` class was chosen over the other two classes presented in this thesis, because of the complexity of the created FSA and the form of the vector of the state variables. As a result of the analysis (Section 9.2), two different FSAs were obtained. The FSA in Figure 12 will serve as a test suite derivation basis for files which are opened in *READ_WRITE* mode. The second FSA in Figure 13 will serve as a test suite derivation basis for the files which are open in *READ_ONLY* mode. The vector of state variables is composed of the following variables: *OpenFile. fpointer, OpenFile.data, OpenFile.mode, error.*

During testing of the `RWFile` class, the following aspects of the class will be tested: proper behaviour when accessing the file in *READ_ONLY* mode and *READ_WRITE* mode, execution of the *Erase* method, and execution of *SeekTo* method. The listings of the programs, and a printout of the results are included in Appendix B.

♣ Test 1

Purpose of the test: To verify the correct behaviour of the class after invocation of *Erase* method in a file opened with *READ_WRITE* mode.

To test the class, the following sequence of methods was executed: *RWFile*, *isValid*, *Write*, *SeekToBegin*, *Read*, *Erase*, *IsEmpty*.

Both *Read* and *Write* operations were executed n-times. The *Erase* operation should put the file into the *empty* state. The last method in the sequence is an inspector, *IsEmpty*. It returns *TRUE* if the file on disk (and its memory copy) is empty. Since the preceding operation (*Erase*) has an expression in the post-condition $self'.data = empty$, then the expected result should be TRUE.

Result: After running the test, the value returned by *IsEmpty* was *TRUE*. Hence, the class shows the expected behaviour.

♣ **Test 2**

Purpose of the test: To verify the correct behaviour of the class after invocation of *SeekTo* method in a file opened with *READ_WRITE* mode.

To test the class, the following sequence of methods was executed: *RWFile*, *isValid*, *Write*, *SeekToBegin*, *Read*, *SeekTo*(n), *Read*.

Operation *Write* was executed n-times, and the following values were written to the empty file $\{3,4,5,6,7,8,9,10,11,12\}$. After that, the pointer was set to point to the first element in the file. Operation *Read* was executed once, and correctly returned the first element in the file. *SeekTo*(n) had to place the pointer on the $n-th$ byte from the beginning of the file. The expected result was the value of the read element of a file, which is the (n/k1)th element from the beginning of the file. For a variable type *Integer*, k1 = 4.

Result: The *Write* method was invoked 10 times, and hence the number of elements in a file after writing was 10. After *SeekToBegin*, one element was read (first element in the file). *ScekTo* set the file pointer at the 12th byte, hence the 4th value in the file. Therefore, the class shows the expected behaviour.

♣ **Test 3**

Purpose of the test: To verify the correct behaviour of the class after invocation of *SeekTo* and *Write* methods in a file opened with *READ_WRITE* mode.

To test the class, the following sequence of methods was executed: *RWFile*, *isValid*, *IsEmpty*, *Read*, *seekTo*, *Write*, *Read*.

The file is open in the default mode (*READ_WRITE*). The file is not empty. After all elements of the file are read, the operation *SeekTo*(n) is executed. It places the file pointer at the n-th position from the beginning of the file. The value is written to the file, and after *SeekToBegin*, the entire content of the file is read. The value of n is 12. Therefore, the 4th element of the file should be replaced.

Result: The result of the test shows that the value of the 4th element has changed. Hence, the class shows the expected behaviour.

♣ **Test 4**

Purpose of the test: To verify the correct behaviour of the class after invocation of *Write* method in a file opened with *READ_ONLY* mode.

To test the class, the following sequence of methods was executed: *RWFile*, *isValid*, *IsEmpty*, *Read*, *SeekToBegin*, *Write*

The file is open in *READ_ONLY* mode, and it is not empty. The first element of the file is read. After *SeekToBegin*, the operation *Write* is invoked. Since the access mode to the file is *READ_ONLY*, the operation should result in an error; the corresponding message is "The write NON-Successful".

Result: After executing the test the message "The write NON-Successful" was obtained. Hence, the class shows the expected behaviour.

9.6 Concluding Remarks

The commercial library of C++ classes, Rogue Wave Tools.h++ [rogue], serves as a test-bed for our proposed methodology. Our methodology to create FSA from formal specification allows one to (1) document each class with specifically stated behavioural dependencies within a class, (2) provide test suites for a class, and (3) reuse the results of a class' analysis during the preparation of tests for software modules which use that class. This last remark is important, since the object-oriented class should always be re-tested in a new environment.

Rogue Wave Tools.h++ library classes were properly implemented. Therefore, our tests did not detect any errors in the implementation of these classes. Equally important, formal specifications and the created FSA will benefit potential users of the library during their software system development.

Chapter 10

Concluding Remarks and Future Research

10.1 Summary

Software reuse is one of the main advantages of the object-oriented methodology. It allows one to create a better software in a short time, based on the existing classes. However, to support reuse, the behaviour of the class has to be fully specified. In addition, reusable modules have to be throughly tested to be reliable.

Formal specification languages support the development of formal specifications of any software module. Based on formal specifications, completeness and correctness analysis of software can be conducted. The formal specification can support the testing process by providing a systematic approach towards testing; test suites can be developed from the specifications, and the expected results of testing can be evaluated.

In this thesis, it was argued that the formal specifications can support software testing. In particular, the research concentrated on the use of formal specifications written in Larch/C++ for testing implementations of object-oriented software modules. As a result, the methodology of test suite preparation based on formal specifications written in Larch/C++, is presented. The main attention was concentrated on FSA derivation. Then, the test cases can be derived using known algorithms to test the class.

The analysis of problems related to testing object-oriented software gave an insight into the expected properties of the specifications of a software module. One of the

most popular approaches to testing object-oriented software is the creation of a Finite State Automaton (FSA). Therefore, the research was geared towards the creation of the FSA.

In addition, research on the use of formal specifications for implementation testing was conducted. The results of this work are included in Chapters 5, 6, and 7. The analyses show that the broad range and variety of formal specification languages do not support a unified approach towards their use in testing.

To support the claim that a proposed methodology is valid, the methodology was applied to several object-oriented classes of academic interest, and classes chosen from the commercial library Rogue Wave Tools.h++ [rogue].

In summary, the contributions of this thesis are:

- Identification of critical issues involved with the testing of object oriented software.
- Identification of critical issues involved with the testing of an implementation based on its formal specifications.
- Analysis of the Larch LSL layer for the purpose of identifying conditions for FSA creation.
- A methodology for the creation of FSA based on Larch/C++ specification of a software module.
- Illustrations of how the implementation of the class can be tested based on its formal specifications written in Larch/C++.

Analysis of human input to test suite preparation

Formal specifications provide detailed and unambiguous information about software behaviour. However, the quite high level of abstraction result in simplification of any model. Since some knowledge of the expected behaviour of a specified class can only come from the designer, the assistance of the tester/designer in test preparation and execution process is necessary. In addition, some knowledge of possible erroneous

behaviour may be not accounted for in the specified model. For LSL traits, knowledge of the behaviour of an ADT may come only from the specifier. This is true when the LSL trait is not complete. Therefore, the system specifier or test designer interactively should guide the testing process.

10.2 Future Work

As it was stated in Section 2, the work in this thesis does not address all aspects of testing object-oriented software modules. This thesis concentrates only on testing single classes, based on the complete Larch/C++ class specifications. The remaining problems related to reuse of the test suites to test inherited or derived classes, and perform the cluster or system layer testing, are still open for research. What is obvious is the fact that both layers of software testing will be based on the initial methodology for derivation of a FSA from formal specifications. However, issues specific to testing each layer remain open for research.

An important future goal is to create a tool based on the methodology discussed in the thesis. Some hints on the design and implementation of the tool are discussed in the next section.

10.3 An Overview of the Tool Based on the Proposed Methodology

In previous chapters, a methodology was developed to test the implementation of a C++ class using formal specifications written Larch/C++. The proposed methodology of Finite State Automaton creation from Larch/C++ formal specifications can be implemented to assist the software tester during the testing process. The tool, called TSD (Testing Suite Derivation), can be designed to accommodate the requirements of the method, and to assist the tester during testing activities. The tool would provide an environment to interactively utilize the existing methodology of FSA derivation and formal specifications to conduct test preparation and execution. This section proposes an Analysis Design document for the tool.

After analysis of the methodology it can be concluded that the following aspects of it can be automated:

1. Analysis of LSL traits

   In order to conduct the analysis of an LSL trait, sufficient completeness of the trait is assumed. Therefore, the process of identifying the basic distinguishable domains and the input and output domains for individual operators can be automated.

   The input to the system consists of a set of LSL traits.

   The output from the system consists of (1) the resulting subdivision of the LSL operators, (2) the basic distinguishable domains, and (3) the input and output domains for individual operations. In addition, any of the restrictions on the abstract data types should be specified to compose the part of the invariant for the analyzed class.

   The outcome of the LSL trait analysis may be part of the library of information which can be reused for any class and trait which uses its respective abstract data type.

2. Analysis of class interface layer

   During the analysis of a class interface layer, each individual method in the interface specification should be analyzed to identify the LSL operators which are included in the **requires** and **ensures** clauses. The identification would require a Larch/C++ parser.

   The input to the system is (1) an interface specification of a class, and (2) the results of an analysis of all **used** LSL traits.

   The output consists of (1) a vector of state variables, (2) a list of all states of the FSA, (3) a list of all interface operations with valid transitions, and (4) a list of all exceptional cases obtained.

3. Invariant identification

   Our methodology requires that a class invariant is defined. As stated in Section 8.4, the class invariant is composed of three parts: (1) additional restrictions

from LSL the trait, (2) information included in the **invariant** clause in the interface specification, and (3) the invariant hidden in pre- and post-conditions for individual class method specifications. The identification and analysis of all three components of the class' invariant can be automated. However, human assistance in assessing the results will be required.

The input to this system is (1) the interface specification, and (2) all **used** LSL traits; and the output is the invariant in a DNF form.

4. DNF analysis for the individual methods and a whole class

   The test derivation methodology is based on the Disjunctive Normal Form (DNF) analysis of the predicates. After conducting DNF analysis for each individual method, the machine is assembled. During the assembly process, DNF analysis will be done, if necessary, to determine additional states. Later, the results can be used to conduct additional analyses of the individual methods, when the number of initial partitions has changed the transitions. Nevertheless, the results of DNF analysis should be evaluated by the tester.

   The input to this system are the interface operations, and the output is in the form of a DNF specifying pre- and post-states of each variable.

5. Test suite and test result derivation

   Based on the constructed FSA, registered restrictions, and exceptional cases, test suite and test result derivation can also be automated.

Not all steps of the methodology can be automated, and human assistance is required; see Section 10.3.3. The first draft of an Architectural Design of a tool is presented below. It will be an interactive tool using the X-Windows system, which would enable the tester (user) to prepare and conduct test cases based on the formal specifications of a class.

10.3.1 Subsystems Decomposition

The tool's architecture is decomposed into independent interacting sub-systems, based on the analysis presented in the previous section. The subsystems are shown in Figure 24. The system is a composition of many co-operating systems. Each of them can use

its own window(s) to run a selected process. The architecture of the TSD is presented in Figure 25.

This approach was chosen to support a high independence of individual sub-systems. To a certain degree, these sub-systems can operate without other systems being active. This allows us to extend each system according to needs without changing its basic functionalities. In the following sub-section, only basic functionalities of the tool are presented and discussed. This is done to describe only their intended functionality.

According to the conducted analysis, the tool would be composed from the following systems:

- **TSD**

The main system, which co-ordinates all activities related to FSA derivation, and test suite generation.

- **FSA Generator**

This system is responsible for all actions related to the creation of the FSA from formal specifications. It is composed of several other systems, which can operate independently. It also uses **Larch/C++ Parser** and **DNF Analyzer** to support the analysis.

  - **LSL Analyzer**

    This sub-system is responsible for handling all operations related to the analysis of a given LSL trait. It provides the following services:

    1. displays the contents of an LSL trait
    2. displays all qualifying basic constructors
    3. displays all qualifying basic inspectors
    4. displays exemptions
    5. provides access to any *included* and *assumed* trait. It may use already processed information, if any is acceptable.
    6. displays the taxonomy of operations in the LSL trait

7. conducts analyses of input/output domains for LSL trait operations

- **Invariant Analyzer**

  This sub-system uses information from LSL traits and interface specifications to determine a global class invariant. This invariant is composed from three elements:

  1. the invariant stated in the **invariant** clause of interface specifications
  2. the invariant extracted from interface specifications
  3. additional restrictions from LSL trait specifications

  As input, this sub-system receives the following: (1) the **invariant** clause in the interface specification or **TRUE** otherwise, (2) additional restrictions on the distinguished sort or **TRUE** otherwise, and (3) interface specifications for a class, (4) a list of the used LSL traits, and (5) the name of a class under test.

  The output consists of the following: (1) individual elements which compose the global invariant, and (2) the global invariant in a DNF form.

- **Interface Analyzer**

  This sub-system is responsible for all operations related to analyzing interface specifications for a given class.

  The input consists of the following: (1) interface specification, (2) analyzed LSL traits, and (3) the class invariant.

  The output consists of a set of tuples containing the following information: (1) a name of the class, (2) the class invariant in DNF form, (3) the set of class variables, (4) the name of method, (5) the list of pre-states (in DNF form), (6) the list of post-states (in DNF form), and (7) the list of transitions with the applying conditions.

- **Test Pool**

  - **LSL Pool**

    This sub-system is responsible for storing and retrieving information about LSL traits analysis. The output consist of: (1) a list of distinguished sort

initializers, (2) a list of distinguished sort partitioners, (3) basic distinguished domains, (4) operations with specified input and output domains, (5) exception cases for operations with input domains, and (6) additional restrictions on the distinguished sort.

- **Interface Pool**

  This sub-system is responsible for storing and retrieving information about class interface specification analysis. The output consist of: (1) a vector of state variables, (2) a class invariant, (3) a list of used LSL traits, (4) a list of global/static functions, (5) a list of global/static variables, and (6) a list of operations with allowable transitions, prohibited transitions, and (3) exception cases.

- **FSA Pool**

  This sub-system maintains all information about created FSA for object-oriented classes. This information should be stored in the file to allow future access. During process execution, the information about the FSA can be displayed in (1) textual form, and (2) graphical form with marked states and transitions.

- **Class Library**

This system supports all operations related to choosing the class for analysis, and viewing formal specifications. It could be enhanced to edit formal specifications and record the changes.

- **LSL Library**

  This sub-system supports all operations related to viewing LSL traits.

- **LCC Library**

  This sub-system supports all operations related to viewing interface specifications for a class (LCC layer).

- **DNF Analyzer**

This system transforms logical expressions it into their Disjunctive Normal Form (DNF). It displays the initial form of the expression along with its DNF form.

- **Test Generator**

This system generates test cases based on the structure of the FSA and additional information obtained from the analysis of interface specifications of the class. This system assists the user in the test derivation process, records the decisions, and stores information obtained during test execution.

- **Larch/C++ Parser**

This system is responsible for recognition of LSL and LCC primitives, along with their associated traits. It analyzes the formal specifications of a given class, and the returns needed for analysis information.

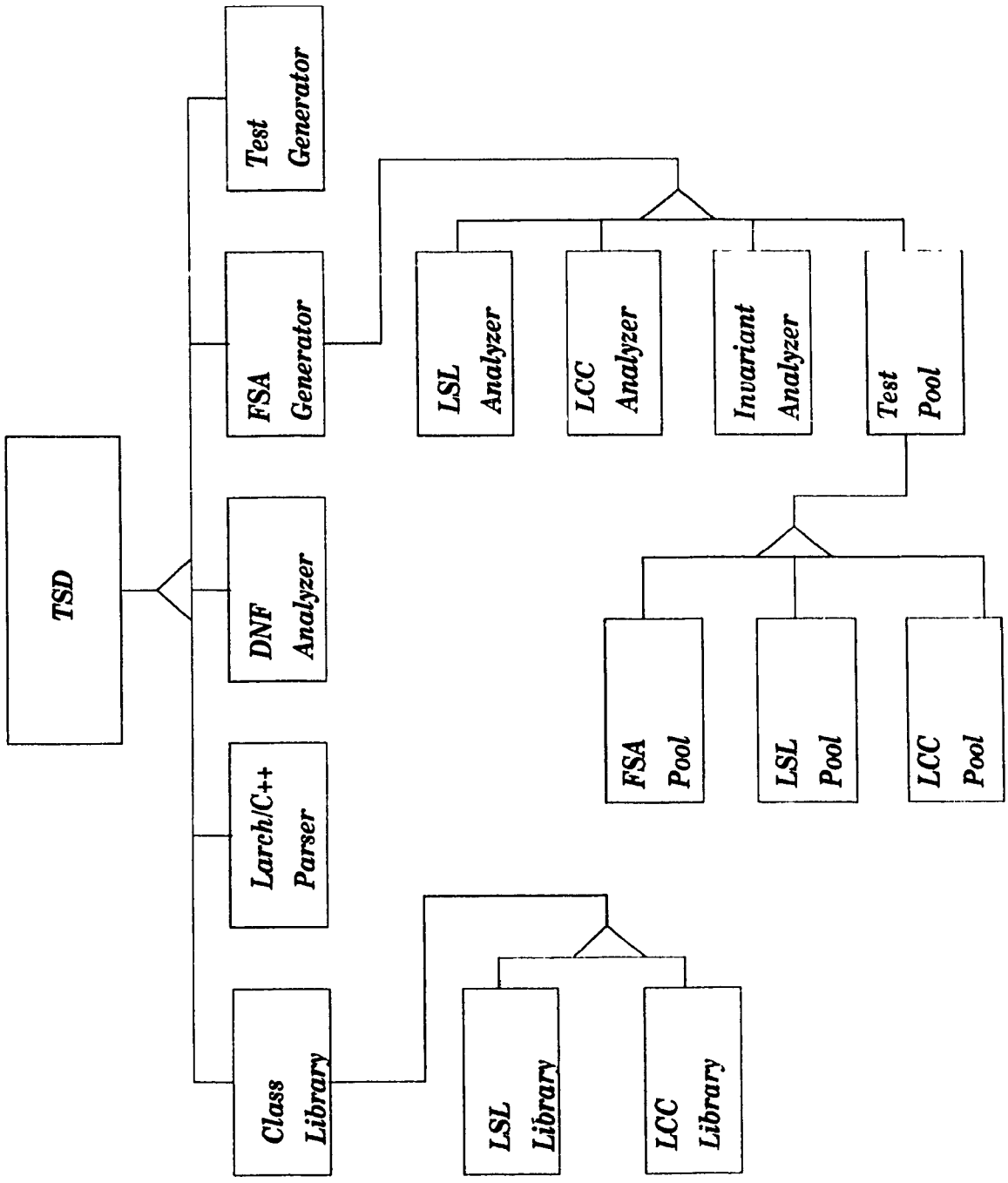

Figure 24: DST: System decomposition

Figure 25: DST: Architectural Design

10.3.2 Data Dictionary

The following is the data dictionary. It contains the definitions of systems and their functionalities.

Module	*Methods*	*Description*
TSD		This system coordinates the work of a whole system. It is composed of the following sub-systems: **Class Library, FSA Generator Larch/C++ Parser, DNF Analyzer, Test Generator**
	Select	Selects the class for analysis
	Derive	Derives FSA from the specifications
	Generate	Generates test cases based on FSA
	New	Restarts the analysis
	Save	Saves results of analysis to the file on disk
	Retrieve	Retrieves results of analysis from the file on disk
DNF Analyzer		This system performs DNF analysis of a given logical expression
	View Expression	displays the expression which is to be analyzed.
	View Result	Displays the results of analysis
	Analyze	Invokes the analysis process.
Parser Larch/C++		This system conducts the parsing of LSL and LCC traits to detect LSL operations and other specification's primitives
FSA Generator		This system coordinates all activities related to FSA creation. It is composed of the following sub-systems: **LSL Analyzer, Interface Analyzer, Invariant Analyzer, Test Pool**

Table 1: DST: Data Dictionary

Module	*Methods*	*Description*
Test Pool		This system contains all information necessary to create a FSA. It is composed of: **LSL Pool, LCC Pool, FSA Pool**
FSA Pool		This system contains information about the created Finite State Automaton
	Store	Stores created FSA into file on disk
	Retrieve	Retrieves information about FSA from file on disk
	Display	Displays FSA as: (1) graph, and (2) text
LSL Pool		This system contains the information about the analyzed LSL traits
	Select	Selects a LSL trait form the list
	Display LSL	Displays contents of any *included/* or *assumed* LSL trait.
	Display Taxonomy	Creates and displays the taxonomy of operators in the analyzed/selected trait
	Display Basic Domains	Displays information about the *distinguished sort initializer*, *distinguished sort partitioners*, and *basic distinguished domains*
	Store	Stores information about analysis into file on disk
	Retrieve	Retrieves information about analysis of LSL trait in a file on disk
LCC Pool		This system contains information about the analyzed LCC class interfaces
	Retrieve	Retrieves information from the file on disk, with information obtained on the analyzed LCC specifications
	Store	Stores information about LCC specification analysis in a file on disk
	Display	Displays information about the analyzed LCC specifications; this information includes: *requires* and *ensures* clauses, class variables, etc.

Table 2: DST: Data Dictionary cont.

Module	*Methods*	*Description*
LSL Analyzer		This system is responsible for all operations related to LSL trait analysis.
	Display	Displays the results of analysis
	Analyze	Invokes of a series of procedures which result in analysis of the LSL trait
	Select	Selects of the LSL trait for analysis
	Store	Stores information about the analyzed LSL trait in the designated area
	Retrieve	Retrieves information about analyzed LSL trait from a designated area
LCC Analyzer		This system is responsible for all operations related to LCC layer analysis.
	Display	Displays results of analysis
	Analyze	Invokes series of procedures which result in the analysis of an interface specification
	Store	Stores information about the results of an analysis in a designated area
	Retrieve	Retrieves information about interface analysis from a designated area.
Invariant Analyzer		This system is responsible for all operations related to class invariant analysis.
	Display	Displays all three components of a class invariant
	Calculate	Calculates class invariant from three components
	Extract	Extracts an invariant from the interface specifications
Test Generator		This system assists the tester during test case generation.
	Retrieve	Retrieves test cases from a file on disk. The test cases are prepared for a given FSA
	Store	Stores test cases prepared for a given FSA in a file on disk
	Generate	Generates test cases for a given FSA

Table 3: DST: Data Dictionary cont.

Module	*Methods*	*Description*
Class Library		This system is responsible for maintaining information about class specifications. It is composed of : **LSL Library, LCC Library**
	Select	Selects a class from the list
	LSL Select	Activates the LSL Library
	LCC Select	Activates the LCC Library
LSL Library		This system is responsible for maintaining access to the library of LSL traits.
	Select	Selects the LSL trait from a list
	Display	Displays selected LSL trait in a separate window
LCC Library		This system is responsible for maintaining access to the library of LCC interface specifications.
	Select	Selects interface specification from a list
	Display	Displays interface specification for a selected class

Table 4: DST: Data Dictionary cont.

10.3.3 Human Assistance

The TSD tool requires extensive human assistance during specification analysis, FSA creation, and test suite derivation, for the following reasons:

1. **LSL Analysis**

   (a) The choice of the *distinguished sort initializer(s)* and *distinguished sort partitioner(s)* cannot be fully automated. All qualifying inspectors (see Section 8.5) have to be analyzed in order to choose those which will provide the most satisfactory information.

   (b) The LSL trait may not have any (or a sufficient amount of) distinguished sort generators. In such cases, *included/assumed* traces have to be analyzed by a human. This process cannot be fully automated, because the understanding of the specifications is provided by a human.

(c) From the signature of an operation, the type of operation may not be obvious. The tester has to accept the proposed taxonomy of all operators (see Section 8.5), even though the operations can be classified according to the proposed criteria.

(d) The LSL trait may not be sufficiently complete. In such cases, *input/output domain* analysis of all operations have to be verified by the tester.

(e) Any intentional incompleteness is determined by the tester. Therefore, this part of analysis (*input/output domains* determination) will also have to be done by a human.

(f) The existence and importance of restrictions on distinguished sort, in many cases, may be determine only by a human.

2. **LCC Analysis**

Interface specifications are abstract representation of a method's behaviour. Therefore, certain information can come only from the specifier.

(a) Final definitions of state variables

Not all variables, which are specified during interface analysis, may compose the set of state variables. This may be because some of qualifying variables remain constant in all states of an FSA.

(b) Analysis of **used** traits

Not all traits carry relevant information for machine derivation. Some of them may be there to define data types of a method's parameters.

(c) Acceptance of the proposed taxonomy of interface methods

Some methods may qualify into other groups, because they use *type converters*. Therefore, they should be assigned to the relevant group, in order to properly define their *input/output states*. In such cases, methods which resemble **modifiers** are considered **inspectors**.

(d) Simplification of the **requires** and **ensures** expressions

The methodology requires that all variables which are not part of the state variables set are existentially quantified.

3. **Invariant Analysis**

An invariant can be composed of three different parts (Section 8.4). The analysis of its final form (*class invariant*) should be supervised by the tester. The following should be analyzed with human assistance:

(a) simplification of an invariant expression, and

(b) interpretation of additional restrictions on the distinguished sort.

4. **Test Generator**

During the test generation process, the final choice of the test suite, based on FSA and exceptional conditions, should done by the tester.

Bibliography

[Abra93] R. F. Abraham, "Black-Box Testing Using Module Interface Specifications", *CLR Report N0. 267*, Faculty of Engineering, McMaster University, April 1993.

[ACCUA94] V.S. Alagar, P. Colagrosso, A. Celer, I. Umansky and R. Achuthan, *"Formal Specifications for Effective Black-Box Reuse. Phase I Progress Report"*, Department of Computer Science, Concordia University, 1994.

[AHKN90] J. Arkko, V. Hirvisalo, J. Kuusela and E. Nuutila, "Supporting Testing of specifications and implementations", *EUROMICRO Journal*, 30(1-5), pp. 297-302, August 1990.

[BCFG86] L. Bouge, N. Choquet, L. Fribourg and M.-C. Gaudel, "Test sets generation from algebraic specifications using logic programming", *Journal of Systems and Software*, 6(4), pp. 343-360, November 1986.

[Bei90] B. Beizer, *"Software Testing Techniques"*, Van Nostrand Reinhold. New York, second edition, 1990.

[BF93] J.P. Bowen and M. Franzle, "Developing Correct Systems", in *Proceedings of Fifth Euromicro Workshop on Real-Time Systems*, June 1993.

[BGM91] G. Bernot, M.-C. Gaudel and B. Marre, "Software Testing based on formal specifications: A theory and a tool", *Software Engineering Journal*, 6(6), pp. 387-405, November 1991.

[BH94a] J.P. Bowen and M.G. Hinchey, *"Then Commandments of Formal Methods"*, URL: http//www.comlab.ox.ac.uk/oucl/people/jonathan.bowen.html

[BH94b] J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods", *PRG-TR-7-94*, Oxford University, Computing Laboratory, 1994.

[Bin94] R.V. Binder, "Design for Testability in Object-Oriented Systems", *Communications of ACM*, September 1994.

[Boo86] G. Booch, "Object-Oriented Development", *IEEE Transactions on Software Engineering*, SE-12, 2, pp. 211-221. 1986.

[Chow78] T.S. Chow, "Testing Software Design Modeled by Finite-State Machines", *IEEE Transactions on SE*, May 19978.

[CL90] M. Cline and D. Lea, "Using annotated C++", in *Proceedings of the 1990 C++ At Work Conference*, 1990.

[CM90] T.J. Cheatham and L. Mellinger, "Testing Object-Oriented Software Systems", in *Proceedings of the 1990 Computer Science Conference*, pp. 161-165, 1990.

[Col93] P. Colagrosso, *"Formal Specification of C++ Class Interface for Software Reuse"*, Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1993.

[CW93] E. Cusack and C. Wezeman, "Deriving tests for objects specified in Z", In JP Bowen, JE Nicholls (eds), *Z User Workshop*, Springer-Verlag, 1993.

[DF92] J.Dick and A. Faivre, "Automatic partition analysis of VDM specifications", *TR RAD/DMA/92027*, Research and Advanced Development, Bull System Products, BULL CA, 1992.

[DF93] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications", *In Woodcock JCP, Larsen PG (eds), FME'93 Industrial Strenght Formal Methods, Lecture Notes in Computer Science 670*, pp. 286-284, Springer-Verlag, 1993.

[DF91] R. Doong and P.G. Frankl, "Case Studies on Testing Object-Oriented Programs", in *Proceedings of The Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pp. 165-177, October 1991.

[DO91] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation", *IEEE Transaction on SE*, vol.17, September 1991.

[FBKAG91] S. Fujiwara, G. Bochmann et. all, "Test Selection Based on Finite State Models", *IEEE Transactions on SE*, vol.17, June 1991.

[Fie89] S.P. Fiedler, "Object-Oriented Unit Testing", *Hewlett-Packard Journal*, vol. 40, pp. 69-74, April 1989.

[GCG90] C.P. Gerrard, D. Coleman and R.M. Gallimore, "Formal specification and Design Time Testing", *IEEE Transactions on SE*, vol.16, January 1990.

[GG93] S.V. Garland and J.V. Guttag, *"A Guide to LP, The Larch Prover"*, Massachusetts Institute of Technology, March 1993.

[GH178] J.V. Guttag and J.J. Horning, "The algebraic specification of abstract data types", *Acta Informatica*, vol. 10, pp.27-52, 1978.

[GH86] J.V. Guttag and J.J. Horning, "Report on Larch Shared Language", *Science of Computer Programming*, vol.6, pp.103-134, 1986.

[GHW85] J.V. Guttag, J.J. Horning and J.M. Wing, " The Larch family of specification languages", *IEEE Software*, vol.2, pp. 24-36, September 1985.

[GH93] J.V. Guttag and J.J. Horning, "Larch: Languages and Tool for Formal Specifications", Springer-Verlag, 1993.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet, "Data-abstaretion implemenattion, specification, and testing", *ACM Transactions on Programming Languages and Systems*, 3(3), pp. 211-223, July 1981.

[Gon70] G. Gonenc, "A method for the design of fault-detection experiments", *IEEE Transactions on Software Engineering*, June 1970.

[GW88] J.A. Goguen and T. Winkler, "Introducing OBJ", *TR SRI-CSL-889*, SRI International, 1988.

[H88] PAV Hall, "Towards testing with respect to formal specifications", in *Second IEE/BCE Conference on Software Engineering 88*, pp. 159-163, IEE, 1988.

[H91] P.A.V. Hall, "Relationship between specifications and testing", *Information and Software Technology*, 33(1), pp. 47-52, 1991.

[Har87] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8, pp. 231-274, 1987.

[Hay86] I.J. Hayes, "Specification directed module testing", *IEEE Transactions on Software Engineering*, 12(1), pp. 124-133, January 1986.

[HLN+90] D.Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive-systems", *IEEE Transactions on Software Engineering*, April 1990.

[Ho89] D. Hoffman, "A CASE study in module testing", in *Proceedings IEEE Conference on Software Maintenance*, Los Alamitos CA, pp. 100-105, 1989.

[HoJ] D M. Hoffman and G. Jones, "Module State Machines", *Technical Report*, Department of Computer Science, University of Victoria, 1992.

[HP94] H.M. Horcher and J. Peleska, "The role of formal specifications in software testing", in *Proceedings of FME'94*, 1994.

[HT90] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence", *IEEE Transactions on SE*, vol.16, December 1990.

[HS91] D M. Hoffman and Paul Strooper, "Automated Module Testing in Prolog", *IEEE Transactions on Software Engineering*, September 1991.

[HS93] D. Hoffman and P. Strooper, "A Case Study in Class Testing", in *Proceedings of CASCON'93 IBM Toronto laboratory*, October, 1993.

[HSnz93] D. Hoffman and P. Strooper. "Graph-based Class Testing", *Presented at The Australian S.E. Conference, Christ Church, New Zealand*, September, 1993.

[Jon90] C.B. Jones, *"Systematic Software Development Using VDM"*, Prentice-Hall International, second edition, 1990.

[Jon91] K.D. Jones, "Lm3: A Larch interface language for modula-3. A definition and introduction", *TR 72*, Digital Equipment Corporation System Research Center, 1991.

[L92] G. Laycock, "Formal specification and testing: A case study", *Journal of Software Testing, Verification and Reliability*, 2(1), pp. 7-23, 1992.

[LC92] G.T. Leavens and Y. Cheon, "Preliminary design of Larch/C++, in U. Martin and J. Wing (ed), in *Proceedings of the First International Workshop on Larch*". Springer-Verlag, 1992.

[LC94] G.T. Leavens and Y. Cheon, *"Larch/C++ Reference Manual"*, Department of Computer Science, Iowa State University, November 1994.

[LDB94] G. Luo, A. Das and G. v Bochmann, "Software testing based on SDL specifications with Save", *IEEE Transactions on SE*, vol.20, January 1994.

[Luc91] D. Luckham, *" Programming With Specifications: An Introduction to Anna, A Language for Specifying ADA Programs"*, Springer-Verlag, 1991.

[Lus94] F. Lustman, "Specifying Transaction-Based Information Systems with Regular Expressions", *IEEE Transactions on SE*, vol.20, March 1994.

[MBGH] H. Mills, V. Basili, J. Gannon, and R. Hamlet, "Mathematical principles for a first course in software engineering, *IEEE Transactions on Software Engineering*, May 1989.

[McD94] R. McDaniel, "The Effect of Polymorphism, Dynamic Binding, and Object State on Testing the Interactions of Classes", em Technical Report, Department of Computer Science, Clemson University, 1994

[MGD93] J.D. McGregor and D.M Dyer, "Selecting Functional Test Cases for an object-oriented software", in *Proceedings of the Pacific Northwest Software Quality Conference*, 1993.

[McG] J.D. McGregor, "Functional Testing of Classes", *Technical Report*, Department of Computer Science, Clemson University, Canada.

[McGH91] J.D. McGregor and M.J. Harrold, "Hierarchical Incremental Testing", *Technical Report TR91-111*, Department of Computer Science, Clemson University, 1991.

[McGHK92] J.D. McGregor, M.J. Harrold and K. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", in *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[McG2] J.D. McGregor, "Constructing Functional Test Cases Using Incrementally Derived State Machines", *Technical Report*, Department of Computer Science, Clemson University, Canada.

[McGK94] J.D. McGregor and T.D. Korson, "Integrating Object-Oriented Testing and Development Process", *Communications of ACM*, September 1994.

[Mey88] B. Meyer, *"Object-Oriented Software Constructions"*, Prentice-Hall, 1988.

[MTW91] G.C. Murphy, P. Townsend and P.S. Wong, "Experiences with Cluster and Class Testing", *Communications of ACM*, September 1991.

[Nich90] R.A. Nicholl, "Unreachable States in Model-Oriented Specifications", *IEEE Transactions on SE*, vol.16, April 1990.

[NP92] G. Nota and G. Pacini, "Querying of Executable Software Specifications", *IEEE Transactions on SE*, vol. 18, August 1992.

[NT81] S. Naito and M. Tsunoyama, " Fault detection for sequential machines by transition-tours", in *Proceedings FTCS (Fault Tolerant Computer Systems)*, 1981

[O94] O. O'Malley, "Representation mappings for validation and testing", *Technical Report*, Information and Computer Science Department, University of California, Irvine, 1991.

[Parn93] D.L. Parnas, "Predicate Logic for Software Engineering", *IEEE Transactions on SE, vol.19*, September 1993.

[PaW89] D.L. Parnas and Y. Wang, "The Trace Assertion Method of Module Interface Specification", *TR 89-261*, Telecommunications Research Institute of Ontario (TRIO), Queen's University, 1989.

[PF90] D.H. Pitt and D. Freestone, "The Derivation of Conformance Tests from LOTOS Specifications", *IEEE Transactions on SE*, vol.16, December 1990.

[PW93] D.L. Parnas and Y. Wang, "Simulating the Behaviour of Software Modules by Trace Rewriting", *IEEE 15th International Conference on SE*, 1993.

[PZ93] A. Parrish and S. H. Zweben, "Clarifying Some Fundamental Concepts in Software Testing", *IEEE Transactions on SE*, vol.19, July 1993.

[rogue] Rogue Wave, "*Tools.h++ Class Library v6.0*" ,Rogue Wave Software, 1993.

[ROA92] D.J. Richardson, S.L. Aha and T.O. O'Malley, "Specification-based test oracles for reactive systems", in *Proceedings of the 14th International Conference on Software Engineering*, pp. 105-118, May 1992.

[ROT89] D.J. Richardson, O. O'Malley and C. Tittle, "Approaches to specification-based testing", in *Proceedings of ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pp. 86 96, published as ACM SIGSOFT Software Engineering Notes 14(8), 1989.

[SC] P. Stocks and D. Carrington, "Test Template Framework: A specification-based testing case study", in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pp.11-18, 1993.

[SC'91] P. Stocks and D. Carrington, "Deriving software test cases from formal specifications", in *6th Australian Software Engineering Conference*, pp. 327-340, July 1991.

[SC93] P. Stocks and D. Carrington, "Test Templates: A Specification based Testing F amework", *IEEE 15th International Conference on SE*, pp. 405 414, 1993.

[SC94] P. Stock and D. Carrington, "A tale of two paradigms: Formal methods and software testing", *Technical Report No.94-4*, The University of Queensland, Australia, 1994.

[SD88] K.K. Sabnani and A.T. Dahbura, "A protocol testing procedure", *Computer Networks and ISDN Systems*, vol. 15, no. 4, pp. 285-297, 1988.

[SNH93] E. V. Sorensen, J. Nordahl and N. H. Hansen, "From CSP Models to Markov Models", *IEEE Transactions on SE*, vol.19, June 1993.

[Spi89] J.M. Spivey, *"The Z Notation: A Reference Manual"*, Prentice HAll, New York, 1989.

[Str86] B. Stroustrup, *"The C++ Programming Language*, Addison Wesley, 1986.

[SR91] M.D. Smith and D.J. Robinson, "A Framework for Testing Object-Oriented Programs", 1991.

[Tai9] K-C Tai, "Predicate-Based Test Generation for Computer Programs", *IEEE 15th International Conference on SE*, 1993.

[TY92] S. Tyszberowicz and A. Yehudai, "OBSERV-A Prototyping Language and Environment", *ACM Transactions on SE and Methodology*, vol.1, July 1992.

[Uman95] I. Umansky, *"Completeness of Formal Specifications of C++ Classes Intended for Black-Box Reuse"*, Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1995.

[WF93] E. Weyuker and P. Frankl, "A Formal Analysis of Fault-Detecting Ability of Testing Methods", *IEEE Transactions on SE*, vol. 19, March 1993.

[WJ91] E. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies", *IEEE Transactions on SE*, vol.17, July 1991.

[WGS94] E. Weyuker, T. Goradia and A. Singh, "Automatically Generating Test Data from Boolean Specification", *IEEE Transaction on SE*, vol.20, May 1994.

[Wil91] A. Wills, "Capsules and Types in Fresco: Program Verification in Smalltalk", in *Proceedings of ECOOP'91, LNCS 512*, 1991.

[Wing90] J M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer*, 1990.

[Woit92a] D.M. Woit, "Realistic Expectations of Random Testing", *CRL Report No.246*, Faculty of Engineering, McMaster University, 1992.

[Woit92b] D M. Woit, "An Analysis of Black-Box Testing Techniques", *CRL Report No..245*, Faculty of Engineering, McMaster University, May 1992.

[Woit93] D M. Woit, "Estimating Software Reliability with Hypothesis Testing", *CRL Report No. 263*, Faculty of Engineering, McMaster University, March 1993.

Appendix A

Analysis of Selected LSL Traits

This Appendix contains analyses of several LSL traits done according to our proposed methodology.

A.1 SetBasic.lsl Trait

The basic behaviour of a set is modeled by the `SetBasics.lsl` trait, Figure 26.

Analysis

- **sort the operations according to the taxonomy**

* sort initializer is $\{\}$
* sort partitioner is *isEmpty*
* sort alterator *insert*
* sort examiner is $\in$

- **extract the basic distinguishable domains**

From the analysis of the sort initializer and partitioner, it is known that that distinguished sort has two *basic distinguishable domains*: $\{\}$, $\neg\{\}$. In order to simplify the used notation and analysis, *empty* will be used instead of $\{\}$, to describe the domain of the variable of type `SetBasics`.

- **perform analysis of all LSL operations**

  - analysis of $insert(e, s)$ alterator

$s^{\wedge} = empty \vee s^{\wedge} = \neg empty$

$s' = \neg empty$

- analysis of $e \in s$ examiner

  $s^{\wedge} = empty \vee s^{\wedge} = \neg empty$

  $s^{\wedge} = s'$

- exemptions: there are no exemptions and additional restrictions on the distinguished sort.

SetBasics(E, C) : **trait**
introduces
$\{\} :\rightarrow C$
$insert : E, C \rightarrow C$
$\_\_ \in \_\_ : E, C \dashrightarrow Bool$
$isEmpty : C \rightarrow Bool$
asserts
C **generated by** $\{\}, insert$
C **partitioned by** $\in, isEmpty$
$\forall\ s : C, e, e_1, e_2 : E$
$\neg(e \in \{\})$
$e_1 \in insert(e_2, s) == e_1 = e_2 \vee e_1 \in s$
implies
$InsertGenerated(\{\}$ **for** $empty)$
$\forall\ e, e_1, e_2 : E, s : C$
$insert(e, s) \neq \{\}$
$insert(e, insert(e, s)) == insert(e, s)$
$insert(e_1, insert(e_2, s)) ==$
$insert(e_2, insert(e_1, s))$
$isEmpty(\{\})$
$\neg isEmpty(insert(e, s))$
converts $\in$

Figure 26: `BasicSet.lsl` definition

A.2 Set.lsl Trait

The `Set.lsl` trait, Figure 27, extends the formal specification of the `SetBasics` by adding several operations.

Set(E, C) : **trait**

includes

SetBasics, *Integer*,

DerivedOrders$(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq,$

$\subset$ **for** $<, \supset$ **for** $>)$

introduces

$\_\_\neg \in \_\_ : E, C \rightarrow Bool$

$delete : E, C \rightarrow C$

$\{\_\} : E \rightarrow C$

$\_\_ \cup \_\_, \_\_ \cap \_\_, \_\_ - \_\_ : C, C \rightarrow C$

$size : C \rightarrow Int$

asserts

$\forall\ e, e_1, e_2 : E, s, s_1, s_2 : C$

$e\neg \in s == \neg(e \in s)$

$\{e\} == insert(e, \{\})$

$e_1 \in delete(e_2, s) == e_1 \neq e_2 \wedge e_1 \in s$

$e \in (s_1 \cup s_2) == e \in s_1 \vee e \in s_2$

$e \in (s_1 \cap s_2) == e \in s_1 \wedge e \in s_2$

$e \in (s_1 - s_2) == e \in s_1 \wedge e\neg \in s_2$

$size(\{\}) == 0$

$size(insert(e, s)) ==$

if $e\neg \in s$ **then** $size(s) + 1$ **else** $size(s)$

$s_1 \subseteq s_2 == s_1 - s_2 = \{\}$

implies

AbelianMonoid$(\cup$ **for** $\circ, \{\}$ **for** $unit, C$ **for** $T)$,

$AC(\cap, C)$, *JoinOp*$(\cup, \{\}$ **for** $empty)$,

MemberOp$(\{\}$ **for** $empty)$,

PartialOrder$(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq,$

$\subset$ **for** $<, \supset$ **for** $>)$

C **generated by** $\{\}, \{\_\}, \cup$

$\forall\ e : E, s, s_1, s_2 : C$

$s_1 \subseteq s_2 \Rightarrow (e \in s_1 \Rightarrow e \in s_2)$

$size(s) \geq 0$

converts

$\in, \neg \in, \{\_\_\}, delete, size, \cup, \cap, - : C, C - C,$

$\subseteq, \supseteq, \subset, \supset$

exempting

$remove\{e, \{\}\}$

Figure 27: `Set.lsl` definition

Analysis

- **sort the operators according to the taxonomy**

* sort initializer $\{\_\_\}$. However, this trait includes `SetBasics` which provides an additional sort initializer. This sort initializer is $\{\}$

* sort partitioner is none. However, from included the`SetBasics` trait, the *isEmpty* sort partitioner can be applied.

* sort alterators are *delete*, $\cup$, $\cap$, $-$

* sort examiner are *size*, $\neg \in$

- **extract the basic distinguishable domains**

From the analysis of the distinguished sort initializer and partitioner, it is known that that two *basic distinguishable domains* are: $\{\}$, $\neg\{\}$. In order to simplify the used notation and analysis, *empty* will be used instead of $\{\}$ to describe the domain of the variable of type `Set`.

- **perform analysis of all LSL operations**

  - analysis of $delete(e, s)$ alterator

    $s\hat{} = \neg empty$

    $s' = \neg empty \vee s' = empty$

  - analysis of s1 $\cup$ s2, s1$\cap$ s2, s1 - s2 alterators

    $s\hat{} = \neg empty \vee s\hat{} = empty$

    $s' = \neg empty \vee s' = empty$

- analysis of $size(s)$ examiner

  $s\hat{} = s'$

  $s\hat{} = \neg empty \vee s\hat{} = empty$

- analysis of $\neg \in$ examiner

  $s\hat{} = s'$

  $s\hat{} = \neg empty \vee s\hat{} = empty$

- exemptions:

  - $delete(e, \{\})$

A.3 BagBasics.lsl Trait

`BagBasics.lsl`, Figure 28, is a distinguished sort which provides operations to create a bag and count the number of identical elements in it.

BagBasics(E, C) : **trait**
  includes *Integer*
  introduces
    $\{\} :\rightarrow C$
    insert : $E, C \rightarrow C$
    count : $E, C \rightarrow$ *Int*
  asserts
    C **generated by** $\{\}$, *insert*
    C **partitioned by** *count*
    $\forall\ b : C, e, e_1, e_2 : E$
      count$(e, \{\}) == 0$
      count$(e_1,$ *insert*$(e_2, b)) ==$
        count$(e_1, b) +$ (**if** $e_1 = e_2$ **then** 1 **else** 0)
  implies
    InsertGenerated($\{\}$ **for** *empty*)
    $\forall\ e : E, b : C$
      insert$(e, b) \neq \{\}$
      count$(e, b) \geq 0$
    converts *count*

Figure 28: `BagBasics.lsl` definition

Analysis

- **sort the operators according to the taxonomy**
  - sort initializer is $\{\}$
  - sort partitioner is none
  - sort alterators is *insert*
  - sort examiner is *count*

• **extract basic sort partitions**

From the analysis of the basic constructor and inspector, it is known that there is no distinguished sort partitioner. Therefore, further analysis cannot be conducted.

A.4 Bag.lsl Trait

`Bag.lsl`, Figure 29, is a distinguished sort which contains elements with the same value. It extends the formal definition of `BagBasics.lsl` by adding new operations.

Bag(*E*, *C*) : **trait**
includes
BagBasics,
DerivedOrders(*C*, ⊆ **for** ≤, ⊇ **for** ≥,
⊂ **for** <, ⊃ **for** >)
introduces
$delete : E, C \rightarrow C$
$\{\_\} : E \rightarrow C$
$\_\_ \in \_\_, \_\_^{\neg} \in \_\_ : E, C \rightarrow Bool$
$size : C \rightarrow Int$
$\_\_ \cup \_\_, \_\_ - \_\_ : C, C \rightarrow C$
$isEmpty : C \rightarrow Bool$
asserts
$\forall\ e, e_1, e_2 : E, b, b_1, b_2 : C$
$count(e_1, delete(e_2, b)) ==$
if $e_1 = e_2$ **then** $max(0, count(e_1, b) - 1)$
else $count(e_1, b)$
$\{e\} == insert(e, \{\})$
$e \in b == count(e, b) > 0$
$e^{\neg} \in b == count(e, b) = 0$
$size(\{\}) == 0$
$size(insert(e, b)) == size(b) + 1$
$count(e, b_1 \cup b_2) ==$
$count(e, b_1) + count(e, b_2)$
$count(e, b_1 - b_2) ==$

$max(0, count(e, b_1) - count(e, b_2))$

$b_1 \subseteq b_2 == b_1 - b_2 = \{\}$

$isEmpty(b_1) == b_1 = \{\}$

implies

$AbelianMonoid(\cup$ **for** $\circ, \{\}$ **for** $unit, C$ **for** $T)$,

$JoinOp(\cup, \{\}$ **for** $empty)$,

$MemberOp(\{\}$ **for** $empty)$,

$PartialOrder(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq$,

$\subset$ **for** $<, \supset$ **for** $>)$

$\forall\ e, e_1, e_2 : E, b, b_1, b_2 : C$

$insert(e, b) \neq \{\}$

$count(e, b) \geq 0$

$count(e, b) \leq size(b)$

$b_1 \subseteq b_2 \Rightarrow count(e, b_1) \leq count(e, b_2)$

converts $count, \in, \neg \in, \{\_\}, \cup, - : C, C \rightarrow C$,

$delete, size, \subseteq, \supseteq, \subset, \supset$

Figure 29: `Bag.lsl` definition

Analysis

• sort the operators according to the taxonomy

* sort initializer is $\{\_\}$. In addition, from the `BagBasics.lsl` trait the $\{\}$ constructor should be taken, in order to perform the analysis.

* sort partitioner is *isEmpty*

* sort alterator is *delete*,

* sort examiners are *size*, $\{\_\}$, $\_ \in \_$, $\_ \neg \in \_$

• extract the basic distinguishable domains

From the analysis of the distinguished sort initializer and partitioner, it is known that there are two *basic distinguishable domains*: $\{\}$, $\neg\{\}$. To simplify the used notation and analysis, *empty* will be used instead of $\{\}$ to describe the domain of the variable of type Bag.

- **perform analysis of all LSL operations**
  - analysis of $delete(e, s)$ alterator

    $s\hat{} = \neg empty$

    $s' = \neg empty \vee s\hat{} = empty$
  - analysis of s1 ∪ s2, s1∩ s2, s1 - s2 alterators

    $s\hat{} = \neg empty \vee s\hat{} = empty$

    $s' = \neg empty \vee s' = empty$
  - analysis of $size(s)$ examiner

    $s\hat{} = s'$

    $s\hat{} = \neg empty \vee s\hat{} = empty$
  - analysis of $\neg \in$ examiner

    $s\hat{} = s'$

    $s\hat{} = \neg empty \vee s\hat{} = empty$
  - exemptions:

    - $delete(e, \{\})$

A.5 Array.lsl Trait

`Array.lsl`, Figure 30, is the distinguished sort which models the behaviour of a table. The array can contain a limited number of elements of the same type.

Array(*Elem*, *Arr*) : **trait**

includes *int*

introduces

$create : int \rightarrow Arr$

$assign : Arr, int, Elem \rightarrow Arr$

$\_\_[\_\_] : Arr, int \rightarrow Elem$

$maxIndex : Arr \rightarrow int$

$legalIndex : Arr, int \rightarrow Bool$

asserts

Arr **generated by** *create*, *assign*

Arr **partitioned by** $\_\_[\_\_]$

$\forall\ a : Arr, i, j, n : int, e : Elem$

$assign(a, i, e)[j] == \textbf{if } i = j \textbf{ then } e \textbf{ else } a[j]$

$maxIndex(assign(a, i, e)) == maxIndex(a)$

$maxIndex(create(n)) == n - 1$

$legalIndex(a, i) == (0 \leq i) \wedge (i \leq maxIndex(a))$

implies $\forall\ a : Arr, i, j : int, e_1, e_2 : Elem$

$assign(assign(a, i, e_1), i, e_2) == assign(a, i, e_2)$

$i \neq j \rightarrow assign(assign(a, i, e_1), j, e_2) = assign(assign(a, j, e_2), i, e_1)$

Figure 30: `Array.lsl` definition

Analysis

- **sort the operators according to the taxonomy**

* sort initializer is *create*
* sort partitioner is *legalIndex*
* sort alterator is *assign*
* sort examiners are _[_], *maxIndex*

- **extract basic sort partitions**

From the analysis of the distinguished sort initializer and partitioner, it is known that two *basic distinguishable domains* of an array are: *legal*, *illegal*

- **perform analysis for all operators**

  - analysis of $assign(a, i, e)$ alterator

    $s\hat{} = legal$

    $s' = legal$

  - analysis of _[_] examiner

    $s\hat{} = legal$

    $s\hat{} = s'$

  - analysis of $maxIndex(a)$ examiner

    $s\hat{} = s' \wedge s\hat{} = legal$

  - There is no additional restriction on an array specified in LSL trait.

A.6 Deque.lsl Trait

`Deque.lsl`, Figure 31, is a distinguished sort which models the behaviour of a double ended queue.

Deque(E,C) : **trait**

% Double ended queue operators

includes *Integer*

introduces

$empty : \rightarrow C$

$\_\_ \dashv \_\_ : E, C \rightarrow C$

$\_\_ \vdash \_\_ : C, E \rightarrow C$

$count : E, C \rightarrow Int$

$\_\_ \in \_\_ : E, C \rightarrow Bool$

$head, last : C \rightarrow E$

$tail, init : C \rightarrow C$

$len : C \rightarrow Int$

$isEmpty : C \rightarrow Bool$

asserts

C **generated by** $empty, \vdash$

$\forall\ e, e_1, e_2 : E, d : C$

$count(e, empty) == 0$

$count(e, e_1 \dashv d) ==$

$count(e, d) + (\ \textbf{if}\ e = e_1\ \textbf{then}\ 1\ \textbf{else}\ 0)$

$e \in d == count(e, d) > 0$

$e \dashv empty == empty \vdash e$

$(e_1 \dashv d) \vdash e_2 == e_1 \dashv (d \vdash e_2)$

$head(e \dashv d) == e$

$last(d \vdash e) == e$

$tail(e \dashv d) == d$

$init(d \vdash e) == d$

$len(empty) == 0$

$len(d \vdash e) == len(d) + 1$

$isEmpty(d) == d = empty$

implies

Stack(*head* **for** *top*, *tail* **for** *pop*,

$\dashv$ **for** *push*, *len* **for** *size*),

Queue ($\dashv$ **for** *append*, *last* **for** *head*,

init **for** *tail*)

C **generated by** *empty*, $\dashv$

C **partitioned by** *len*, *head*, *tail*

C **partitioned by** *len*, *last*, *init*

$\forall\, d : C$

$d \neq empty$

$\Rightarrow (head(d) \dashv tail(d) = d$

$\wedge init(d) \vdash last(d) = d)$

converts *head*, *last*, *tail*, *init*, *len*

exempting *head*(*empty*), *last*(*empty*),

tail(*empty*), *init*(*empty*)

Figure 31: `Deque.lsl` definition

Analysis

- **sort the operations according to taxonomy**

  - sort initializer is *empty*
  - sort partitioner is *isEmpty*
  - sort alterators are $-|$, $|-$, *head*, *last*, *tail*, *init*
  - sort examiners are *count*, $\in$, *len*

- **extract basic distinguishable domains**

From the analysis of the distinguished sort initializer and partitioner, it is known that two *basic distinguishable domains* are: *empty*, $\neg$*empty*.

- **perform analysis for all operations**
  - analysis of $c - |s$ alterator

    $s\hat{} = empty \vee s\hat{} = \neg empty$

    $s' = \neg empty$
  - analysis of $s| - c$ alterator

    $s\hat{} = empty \vee s\hat{} = \neg empty$

    $s' = \neg empty$
  - analysis of *head(s)* alterator

    $s\hat{} = \neg empty$

    $s' = \neg empty \vee s' = empty$
  - analysis of *last(s)* alterator

    $s\hat{} = \neg empty$

    $s' = \neg empty \vee s' = empty$
  - analysis of *tail(s)* alterator

    $s\hat{} = \neg empty$

    $s' = \neg empty \vee s' = empty$
  - analysis of *init(s)* alterator

    $s\hat{} = \neg empty$

    $s' = \neg empty \vee s' = empty$
  - analysis of *count(e,s)* examiner

    $s\hat{} = \neg empty \vee s\hat{} = \neg empty$

    $s\hat{} = s'$
  - analysis of *len(s)* examiner

    $s\hat{} = \neg empty \vee s\hat{} = \neg empty$

    $s\hat{} = s'$
  - analysis of $\in (e, s)$ examiner

    $s\hat{} = \neg empty \vee s\hat{} = \neg empty$

    $s\hat{} = s'$

- exemptions:
  - *head(empty)*
  - *last(empty)*
  - *tail(empty)*
  - *init(empty)*

Appendix B

RWFile Class Tests

This Appendix contains test programs and test result outputs which were conducted for RWFile class.

- **Test One**

```
//--------------------------------------------------------------------------
//                          FIRST TEST
//      RWFile, Write, SeekToBegin, Read, Erase, Read, IsEmpty
//      Expected result: TRUE
//--------------------------------------------------------------------------
//
#include <rw/rwfile.h>
#include <rw/rstream.h>

main()
{
 RWFile f1("test1.dat");
 int k1,j; n1 = 3;

 cout << "--------------------" << endl;
 cout << "Start of FIRST test" << endl;
 cout << "Operations: RWFile,Write,SeekToBegin,Read,Erase,Read,IsEmpty";
 cout << endl << "--------------------" << endl;
 cout << " is valid test1.dat: " << f1.isValid() << endl;
 for (k1=0; k1 < 10; k1++)
   { n1++; f1.Write (n1); cout << "n1 = " << n1 << endl;}
```

```
cout << "File is written" << endl;
cout << "----- SeekToBegin -----" << endl;
f1.SeekToBegin();
cout << "----- reading -----" << endl;
j = 0;
while (f1.Eof() == 0)
{
 j++;
 if (f1.Read(n1) == 1)
   cout << "read(n1) = " <<  n1 << endl;
}
cout << " total number read: " << --j << endl;
//  ------ adding clear and read ---
cout << "----- clearing the file -----" << endl;
cout << " erase was sucessfull? : " << f1.Erase() << endl;
cout << "----- reading from the cleared file -----" << endl;
if (f1.Read(n1) == 0)
 cout << "error after reading from cleared file: " << f1.Error() << endl;
else
 cout << " read value: " << n1 << endl;
cout << " is it eof : " << f1.Eof() << endl;
cout << "-------------" << endl;
cout << " The End of a test:   is empty ? : " << f1.IsEmpty() << endl;
return(0);
}
//
//--------------------  END OF FIRST TEST ------------------------------
```

Results

```
--------------------
 Start of FIRST test
 Operations: RWFile, Write, SeekToBegin, Read, Erase, Read, IsEmpty
--------------------
 is valid test1.dat: 1
```

```
n1 = 4
n1 = 5
n1 = 6
n1 = 7
n1 = 8
n1 = 9
n1 = 10
n1 = 11
n1 = 12
n1 = 13
File is written
----- SeekToBegin -----
----- reading -----
read(n1) = 4
read(n1) = 5
read(n1) = 6
read(n1) = 7
read(n1) = 8
read(n1) = 9
read(n1) = 10
read(n1) = 11
read(n1) = 12
read(n1) = 13
 total number read: 10
----- clearing the file -----
 erase was sucessfull? : 1
----- reading from the cleared file -----
 error after reading from cleared file: 0
 is it eof : 1
-------------
 The End of a test:   is empty ? : 1
```

- Test Two

```
//-------------------------------------------------------------------------
//           SECOND  TEST
//      RWFile, Write, SeekToBegin, Read, SeekTo, Read
//      Result: Eof = FALSE
//-------------------------------------------------------------------------
//
#include <rw/rwfile.h>
#include <rw/rstream.h>

main()
{
 RWFile f1("test2.dat");
 int k1, j; int n1 = 3;

 cout << "--------------------" << endl;
 cout << " Start of SECOND test" << endl;
 cout << " Operations: RWFile, Write, SeekToBegin, Read, SeekTo, Read"<<endl;
 cout << "--------------------" << endl;
 cout << " is valid test2.dat: " << f1.isValid() << endl;
 for (k1=0; k1 < 10; k1++)
   { n1++;
     if (f1.Write (n1) == 1)
      { cout << "n1 = " << n1 << " error: " << f1.Error();
        cout<< " k1 = " << k1 << endl; }
   }
 cout << "File is written" << endl;
 cout << "------ SeekToBegin ------" << endl;
 cout << "seek.to.begin : " << f1.SeekToBegin() << endl;
 cout << "----reading-----" << endl;
 j = 0; n1 = -1;
 if (f1.Read(n1) == 1)
   cout << "first element read(n1) = " <<  n1 << endl;
 //  ------ adding seekTo and read ---
 cout << "------ seekTo ------ " << endl;
```

```
long l1 = 12;  // the lenght of one integer is 4 bytes
cout << "n = " << l1 << "    seekTo (n) : " << f1.SeekTo(l1) << endl;
cout << "------ reading from the file ------" << endl;
cout << " is it eof : " << f1.Eof() << endl << "-------------" << endl;
n1 = -1;
if (f1.Read(n1) == 1)
  cout << "**** read(n1) = " <<  n1 << endl;
cout << " error after reading (after SeekTo): " << f1.Error() << endl;
cout << " ----------------------------- " << endl;
cout << " End of Test:   is it eof : " << f1.Eof() << endl;
cout << "-------------" << endl;
return(0);
}
//
//---------------- END OF SECOND TEST ------------------------
```

Results

```
--------------------
 Start of SECOND test
 Operations: RWFile, Write, SeekToBegin, Read, SeekTo, Read
--------------------
 is valid test2.dat: 1
n1 = 4 error: 0 k1 = 0
n1 = 5 error: 0 k1 = 1
n1 = 6 error: 0 k1 = 2
n1 = 7 error: 0 k1 = 3
n1 = 8 error: 0 k1 = 4
n1 = 9 error: 0 k1 = 5
n1 = 10 error: 0 k1 = 6
n1 = 11 error: 0 k1 = 7
n1 = 12 error: 0 k1 = 8
n1 = 13 error: 0 k1 = 9
File is written
------ SeekToBegin ------
```

```
seek.to.begin : 1
----reading-----
first element read(n1) = 4
------ seekTo ------
n = 12    seekTo (n) : 1
------ reading from the file ------
 is it eof : 0
-------------
**** read(n1) = 7
 error after reading (after SeekTo): 0
 ------------------------------
 End of Test:   is it eof : 0
-------------
```

- Test Three

```
//--------------------------------------------------------------------------
//                        THIRD TEST
//       RWFile, isValid, IsEmpty, Read, SeekToBegin, Write, Read
//       Expected Result: WRITING NON-Successful
//--------------------------------------------------------------------------
//
#include <rw/rwfile.h>
#include <rw/rstream.h>

main()
{
 RWFile f1("test3.dat");
 int i, j; int n1 = 2;

 cout << "--------------------" << endl;
 cout << "Start of THIRD test" << endl;
 cout << "Operations: RWFile, isValid, IsEmpty, Read, SeekTo, Write, Read";
 cout << endl << "--------------------" << endl;
 cout << "isValid = " <<  f1.isValid() << endl;
 cout << "Error = " << f1.Error() << endl;
```

```
 cout << "IsEmpty = " <<  f1.IsEmpty() << endl;
 while (!f1.Eof() )
   if (f1.Read(i) == 1)
    {
     cout << " read from the file: i=" << i;
     cout << " Error = " << f1.Error() << endl;
    }
 cout << "------ SeekTo ------" << endl;
 f1.SeekTo(12);
 cout << "----- writing -----" << endl;
 j = 50;
 if ( f1.Write(j) == 1)
   cout << " WRITING Succesful!!!!" << endl;
 else
   cout << " WRITING NON-Succesful  !!!!" << endl;
 cout << "------ Reading all elements ------" << endl;
 f1.SeekToBegin();
 j = 0;
 while (f1.Eof() == 0)
 { j++;
   if (f1.Read(n1) == 1)
     cout << "read(" << j << ") = " <<  n1 << endl;
 }
 cout << " total number read: " << --j << endl;
 return(0);
}
//
//---------------- END OF THIRD TEST ------------------------
```

Results

```
--------------------
 Start of THIRD test
 Operations: RWFile, isValid, IsEmpty, Read, SeekTo, Write, Read
--------------------
isValid = 1
```

```
Error = 0
IsEmpty = 0
 read from the file: i=4 Error = 0
 read from the file: i=5 Error = 0
 read from the file: i=6 Error = 0
 read from the file: i=7 Error = 0
 read from the file: i=8 Error = 0
 read from the file: i=9 Error = 0
 read from the file: i=10 Error = 0
 read from the file: i=11 Error = 0
 read from the file: i=12 Error = 0
 read from the file: i=13 Error = 0
------ SeekTo ------
----- writing -----
 WRITING Succesful!!!!
------ Reading all elements ------
read(1) = 4
read(2) = 5
read(3) = 6
read(4) = 50
read(5) = 8
read(6) = 9
read(7) = 10
read(8) = 11
read(9) = 12
read(10) = 13
 total number read: 10
```

- **Test Four**

```
//--------------------------------------------------------------------------
//                        FOURTH TEST
//     RWFile, isValid, IsEmpty, Read, SeekToBegin, Write
//     Expected Result: fourth element in file has changed
//--------------------------------------------------------------------------
//
#include <rw/rwfile.h>
```

```
#include <rw/rstream.h>

main()
{
 RWFile f1("test4.dat", "rb");
 int i, j;

 cout << "--------------------" << endl;
 cout <<  Start of FOURTH test" << endl;
 cout << "Operations: RWFile, isValid, IsEmpty, Read, SeekToBegin, Write",
 cout << "--------------------" << endl;
 cout << "isValid = " <<  f1.isValid() << endl;
 cout << "IsEmpty = " <<  f1.IsEmpty() << endl;
 if (!f1.Eof() )
  {
   if (f1.Read(i) == 1)
    { cout << " Error = " << f1.Error() << endl;
      cout << " Read from the file: i=" << i << endl;}
    }
 else { cout << "File is EMPTY" << endl;}
 cout << "------ SeekToBegin ------" << endl;
 f1.SeekToBegin();
 cout << "----- writing -----" << endl; j = 0;
 j = 100;
 if ( f1.Write(j) == 1)
   cout << " WRITING Succesful!!!!!" << endl;
 else
 cout << " WRITING NON-Succesful  !!!!" << endl;
 return(0);
}
//
//---------------- END OF FOURTH TEST ------------------------
```

Results

```
--------------------
 Start of FOURTH test
 Operations: RWFile, isValid, IsEmpty, Read, SeekToBegin, Write
--------------------
isValid = 1
IsEmpty = 0
 Error = 0
 Read from the file: i=100
------ SeekToBegin ------
----- writing -----
 WRITING NON-Succesful  !!!!
```