CONSTRUCTING BLACK-BOX TEST SUITES FOR SYSTEMS SPECIFIED IN LARCH/C++

ANTONIOS PROTOPSALTOU

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

JUNE 1996

ISBN 0-612-18426-9

Abstract

Constructing Black-Box Test Suites for Systems Specified in Larch/C++

Antonios Protopsaltou

The selection of appropriate test-cases, in the testing process of an object-oriented software component, is an essential and critical issue, for it provides the confidence to assert that a certain software component is correct. A number of methods for the selection of a test suite, based on specifications, have been developed over the years. Informal specifications are usually ambiguous and incomplete. Formal specifications provide precise definitions of functionalities and properties of a system using mathematical abstractions and semantics. These precise definitions supply useful information that may be exploited in the testing of an implementation. This thesis presents a new test suite selection method based on Larch/C++ formal specifications. The method converts Larch/C++ specifications into a finite state machine (FSM) by partitioning function input spaces and reducing them to disjunctive normal forms (DNF). Given a FSM, a technique for the generation of test-cases is also developed. The technique provides test coverage equivalent to "switch coverage" and in parallel it tries to minimize the size of the suite. Finally, the design of a tool that implements the technique is also presented in this thesis.

Acknowledgments

I wish to express my deepest gratitude to my supervisor Dr.V.S. Alagar. His academic intuition was fundamental to the successful completion of this work. I would like to thank him for the moral support, patience, encouragement and the good advices he always gave me. Without him this work would not be possible.

I would like to thank professor Gary Leavens from Iowa State University for many lengthy discussions on Larch/C++ and sub-typing. Also, I would like to thank Mr. Jeremy Dick, from B-Core UK, for many discussions about testing and VDM.

I would also like to thank the Computer Science system administrators, Stan Swiertz, Michael Assels and Paul Gill, for the help and the Unix tips that they provided me.

My good friends George Karabotsos, Dharmalingum Muthiayen, and Thanassis Michailidis have always been there for me, and I thank them from the bottom of my heart.

I would like to give special thanks to five people that stood by me for my entire stay in Canada. The Katergaris family, my aunt, my uncle, and my cousins, helped me go through every difficulty.

I wish to thank my family in Greece for their continuous love and support. I would like to thank them for believing in me, and making me understand that nothing is unachievable.

Last but never least, I wish to thank Dimitra Tzima for her love, sacrifices, moral support, and patience for all the years that I have been away.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Software development process has gone through many changes since the first software system was developed fifty years ago. In the past few years, software engineers have been successfully practicing the principles of *software reuse*. Software development has been shown to benefit from *software reuse* of well tested software components.

Testing is an important part of a software development. However, it does not get the attention it deserves from researchers and developers. The main goal of testing is to detect faults in a software system. Dijkstra [21] states that the only thing testing can tell us is that the system has failed. It cannot tell us that the system is correct. This is true for all testing methods that are currently in use.

Formal methods promise high product confidence through mathematical proof of system correctness. The use of formal methods aims in replacing the large amount of effort spent in error detection, by effort spent in the construction of "error free" systems.

The dependability of a software component relies on its testing and its formal verification. Both testing and formal verification provide the foundation for a scientifically based engineering approach to software development. Even though this might be true, the software engineering industry is reluctant to move in that direction because formal methods require abstract mathematical knowledge that many employees do not possess.

Object-oriented methods are considered to be a great advance in the pursuit of quality software because of their foundation on the principle of *reusability*. However, a very substantial amount of effort has to be devoted to the task of testing object-oriented components. The use of inheritance and dynamic binding require that the software component should be tested more extensively than other non-object-oriented components.

This thesis addresses only one part of a much broader research on black-box reuse of object-oriented software: constructing adequate black-box test suites for implementations based on Larch/C++ specifications. This research is sponsored by BNR-NSERC under a Collaborative Research and Development Grant to Dr. V.S. Alagar. This research is currently on its second phase [3, 4]. The first phase was completed by members of the previous group [1, 17, 10, 60].

The organization of the remainder of this thesis is as follows. Chapter 2 presents concepts of object-orientation, formal methods, functional testing, and the scope of the thesis. A brief overview and the latest improvements on Larch/C++ are given in Chapter 3. Chapter 4 describes a methodology that converts Larch/C++ specifications to a finite state machine, along with its limitations. Chapter 5, proposes solutions to the limitations of the methodology in Chapter 4. A test suite derivation technique is proposed in Chapter 6. Chapter 7 presents some issues related to sub-typing in Larch/C++ and investigates the relationship between the finite state machines of a super-type and its sub-type. Chapter 8 summarizes the work presented in this thesis and identifies some future work. Appendix A contains the design of a tool that implements the technique described in Chapter 6.

Chapter 2

Basic Concepts in Functional Class Testing of Object-Oriented Software

In the past decade, object-oriented software development has been considered the most important software development method. Object-orientation is a specific way of structuring applications. To apply this technology successfully in software development, however, different ways of thinking about programs must be adopted. The most common characteristic of object-oriented programming and design is the extensive use of abstractions. An abstraction is a high level description or a model of a detailed or complex concept. In this Chapter, the basic concepts of object-oriented programming will be introduced. Also, comments on how formal methods are useful in the software development process and a description of fundamental concepts in testing of software systems will be given. Finally the scope of this thesis will be defined.

2.1 The Object-Oriented Paradigm

Object-orientation can be seen as a kind of technique of organizing a system in terms of objects. An object is a named entity that combines a data structure with its associated operations:

object = unique identifier + data + operations

An object can therefore be regarded as a state machine, with an internal state that remembers the effect of operations. Because the state can be manipulated only by those operations exported in the object's interface, the details of its internal implementation are protected from external views (with exception in special cases). This guarantees that an object may interact with other objects only at a desired level of abstraction. The concept of an object, therefore, unites the principles of data abstraction and information hiding into a single concept. The foundation of the whole approach is called *encapsulation*. This means that the properties of data and the functions for the manipulation of this data are combined into an object. The objects communicate through messages which invoke the functions within these objects; these functions are usually called methods.

An *object* is the basic building block in an object-oriented software development. It models an entity. It is composed of a set of attributes and a set of operations. The data constitutes the information in the object (the state of the object), and the methods, which are analogous to procedures and functions in non-object-oriented languages, manipulate the data. In most applications, there are many objects of a certain type. An object has an identity that can be used to uniquely identify and distinguish it from other objects of the same type. In C++, the data and the methods for objects of the same type are defined in a data type called *class*. In object-oriented programming, each *object* is an instance of a particular class. In C++, the data items are called data members, and the methods are called member functions.

The object-oriented paradigm can be characterized by the following principles:

- **Encapsulation.**

  Encapsulation can be thought of as a form of information hiding. It is the technique by which an object's data is packaged together with its corresponding methods. This data can only be accessed through the message interface for a particular object.

- **Message Passing**

  Message Passing is the principle of one object O_1 sending a signal to another

object O_2 to request one of O_2's services. This service is going to be carried out by one of the O_2's methods.

- **Inheritance**
  Inheritance is the mechanism that allows reuse of the behavior of an existing class in the definition of a new class.

- **Polymorphism**
  Polymorphism is a word of Greek origin that means "having multiple forms". It is the mechanism that allows the same message to be interpreted differently by objects of different classes.

Object-Orientation has the capability of reducing the application maintenance load because of modularity and low coupling among object-oriented constructs provided by encapsulation. A key concept in object orientation is that a new application can be built from existing modules. If those modules are well tested, the new application will be built with a large proportion of high quality code, and the maintenance load will be reduced. Furthermore, the use of already existing modules facilitate the reuse of code and model.

2.2 Formal Specifications and Software Development

One of the main goals of Software Engineering is the production of correct software. The development process of a large and complex system necessitates the gathering and management of a great amount of information and details about the system.

In Software Engineering, the term "correctness" does not have a precise definition. Most of the definitions agree that a software system is correct if it follows its specification. Therefore, the software development process heavily relies upon the correctness of the software specifications. One of the ways that *validation* of a software system may be achieved is by testing the existence of the properties, stated by the specification, on the implementation of the system.

Since informal documentation can be vague and ambiguous, the use of formal specifications has gained much popularity during the past decade. Formal specification of a system denotes a precise description of the essential properties of the objects in the system to be developed. This precision is accomplished through mathematical notations and semantics which are used to guarantee that a software system will exhibit a certain behavior, which becomes observable when the system is interacting with the users or other client software components.

The two important tools to cope with the enormous complexity of software systems, are decomposition and abstraction. In this thesis, we are going to deal with algebraic and model based specifications in Larch/C++. Using algebraic specifications, a system is specified as a collection of abstract data types, each of which includes a specification of all the operations, syntax and semantics with type signatures and algebraic axioms, included in the abstract data type. With Larch/C++, developers specify the functionality of class implementations using an interface specification, which uses Hoare logic of pre- and post-conditions (the *requires* and *ensures* clauses).

2.3 Functional Class Testing

Class testing is the first formal activity performed in the object-oriented software life cycle. It occurs during the implementation phase soon after a class is coded. Class tests are designed to test individual classes, and they form the basis upon which the system tests are built. Since classes and class tests are fundamental entities, class testing is critical to ensuring the final quality of a system.

Specifications are of great importance in testing, for they determine what the software ought to do, and must necessarily form the basis for the testing of the functionality of the system. This kind of testing is called *functional*, or *black-box*, or *specification-based*, or *behavioral* testing. The goal of *functional* testing is to find discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification. Therefore, *functional* testing provides confidence in the correctness of a program. With

testing, the only way to guarantee a program's correctness is to execute it on all possible inputs. In practice, such an exhaustive testing may be impossible. Therefore, systematic testing techniques generate a representative set of test-cases to provide coverage of the program according to some selected criteria. *Black-box* testing treats the software as a box whose internal structure is unknown, but for which the required behavior is known from the functional specification.

The first activity, in the generation process of specification-based functional test-cases, is the partitioning of the input space of the function under test, into equivalence classes. Subsequently, test-cases may be selected from each equivalence class of the partition. The key concept to partitioning a function's input space is that the function should exhibit the same functional behavior for all elements within an equivalence class.

Functional test-cases attempt to find errors in the following categories:

- initialization and termination errors
- incorrect function results
- incorrect or missing functions
- interface errors

2.4 Scope of Thesis

Based on the foundation work of Celer [10] we extend and improve the methodology for generating a finite state machine from Larch/C++ specifications. In this thesis, a method for generating the test suites from a finite state machine is given. The adequacy of the test-cases generated by our method is formally proved. We also discuss the relationship between the finite state machines due to inheritance relationship between classes. Finally, we present the design of a tool which generates minimal canonical test suites; the tool has also been implemented.

Chapter 3

Recent Improvements to Larch/C++

3.1 Introduction - A quick overview

Larch/C++ is a **Larch**-style interface specification language tailored to the C++ programming language. Its main objective is the formal specification of C++ program modules. Larch provides a two-tiered approach to specification:

- In the first tier, the semantics of a program module written in a particular programming language is described. This tier is called Larch Interface Language (LIL). LIL is not one particular specification language but a family of interface specification languages. Each specification language in the LIL family is designed for a specific programming language. Since programming languages differ in how components communicate across the interface, it is easier to be precise about communication when the interface specification language reflects the programming language. LIL specifications specify the interface of a procedure and its behavior. To formally specify the behavior of a function, Larch provides some clauses that assist the specifier to describe the states on which the function is defined, what the function is allowed to change (*modifies* clause), the restrictions on the states and the arguments with which the client is allowed to call the function (*requires* clause), and the constraints on the function's behavior, when the function is called properly, which relate the pre-state and the post-state of the function (*ensures* clause). The pre-condition, expressed in the

requires clause, and the post-condition, expressed in the *ensures* clause, are expressed in predicate logic, using logical assertions that contain terms of which formal meaning is specified in the second tier of Larch (LSL tier).

- In the second tier, auxiliary specifications are written to provide semantics for the primitive terms used in the *requires* and *ensures* clause of the LIL specifications. These specifications are mathematical abstractions that are state independent. These abstractions are called traits and are written in the style of an equational algebraic specification using the Larch Shared Language (LSL). These specifications, written in LSL, are programming language independent and they can be used by any member of the LIL family.

Some of the Larch languages that have been developed are LCL (Larch C)[28, 11], LM3 (Larch Modula-3)[34], Larch Smalltalk [14], Larch/C++ [40, 41]. This Chapter summarizes the recent additions and improvements to Larch/C++ specification language. See [40] for a complete analysis of the extensions.

3.2 Improvements on Larch/C++

Larch/C++ is still in the process of development. Not all parts have been implemented. Currently only the parser is available, some parts of which are still in the development stage. Many versions and upgrades have been created with numerous changes in the syntax and semantics of the language. The Larch/C++ manual [40] is a very fast changing document describing the language as the changes take place. In this section a summary of the most important new features is given. These new features have been added recently to the language and have been discussed in detail in the Larch/C++ reference manual [40].

3.2.1 Composite Sorts

Composite sort names have been added only recently to Larch/C++. In the absence of any documentation, several users of the Larch/C++ community were consulted. The consensus is that composite sort names are to be treated exactly as atomic sort names except that when it comes to renaming. The rule is that the rename binding is propagated through the components. For example, if *Loc[E]* is a composite sort

and E is an atomic sort, with the renaming E to F, the bindings

e:→ E
f:→ Loc[E]

become

e:→ F
f:→ Loc[F]

Composite sorts are not parameterized sorts: '*Loc[E]*' is a sort name that has '*Loc*' and '*E*' as components: '*E*' does not have any special status. In particular E is not a parameter.

Composite sort names are used in order to be able to distinguish between T values and T objects. In the past, an object of type T was denoted by *Obj_T*. One problem with this approach is that while renaming T, the specifier had to rename *Obj_T*. With the composite sort names, as shown in the example above, the renaming of the object sort (*Loc[E]* in the example) is done implicitly. Also, *Obj_T* did not have any fixed semantics. The specifier had to explicitly specify the semantics of an object in an LSL trait. The new version of Larch/C++ has provided a few LSL traits that generally specify the behavior of objects (mutable, immutable, typed and untyped), and the semantics of states [42].

3.2.2 The Formal Model of Objects, Values, and States

According to the designers of C++, "an object is a region of storage." All objects have an address, or location. They store values, which in the context of a specification are called *abstract values*. Figure 1 illustrates the concept of objects and values by showing the Larch/C++ model of a list object in a certain state. A state associates each object with an abstract value. In this example the object *My_List* is associated with the abstract value *append(19, append(7, append(89, empty)))*. Objects in Larch/C++ are modeled using several traits. The main trait is **TypedObj**, which handles the translation between typed objects and values and the untyped

objects and values used in the trait **State**. Objects can be either mutable or constant (immutable). Mutable objects are modeled by sorts with names of the form *Obj[T]*, which is the sort of an object containing an abstract value of sort *T*. The

My_list : append(19, append(7, append(89, empty)))

Figure 1: Larch/C++ Model of a List Object

trait **MutableObj** gives the formal model of mutable objects by adding the capability of mutation to the trait **TypedObj**. Constant objects are modeled by sorts with names of the form *ConstObj[T]*, which is the sort of a constant object containing abstract values of sort *T*. The trait **ConstObj** gives the formal model of constant objects. A state is a mapping from objects to values. During the course of execution, a program creates objects and binds values to objects. A state captures the set of objects that exist at a particular time and their bindings. The trait **State** gives the formal model of states used by Larch/C++. It defines a state (sort State) as a mapping between untyped objects (sort Obj) and abstract values (sort Val).

3.2.3 Declarations and Declarators

C++ provides numerous kinds of declarators for every possible declaration. Larch/C++ has incorporated these declarators both in syntax and semantics. In Larch/C++, a declaration is needed to specify that the C++ module that implements the specification must have the same declaration, and to give information about the declared construct's type, and other attributes. Although Larch/C++ syntax for the declarators tries to be identical to the C++ one, there are some minor differences which were created purposely in Larch/C++ in order to resolve some parsing ambiguities in the C++ grammar.

In a declaration a declarator defines a single object, function or type, along with its name. The semantics of each declarator is defined using LSL traits which are built in Larch/C++. For instance when declaring an integer global variable, Larch/C++ implicitly uses the `int` LSL trait. Using the following operators a declarator may refine an object's type : * (**pointer**), ::* (**pointer to member**), & (**reference**), [ ] (**array**), ( ) (**function**). A variable declared globally, or a formal parameter passed to function

using one of these declarators, has a particular sort. **Pointers** have the **Ptr** sort generator as part of the sort name of the term. **References** use the **Obj** sort generator, and **arrays** use the **Arr** sort generator. **ConstObj[cpp_function]** is the sort of a **non-member function** and **ConstObj[cpp_member_function]** is the sort of a **member function**. The semantics of these sorts are described using the LSL traits discussed previously. In Tables 1 and 2 there is a summary of the sorts that global variables, and formal parameters take when declared with the various declarators:

Declaration	**Sort of x (x is global)**
T x	Obj[T]
const T x	ConstObj[T]
T & x	Obj[T]
const T & x	ConstObj[T]
T & const x	Obj[T]
T * x	Obj[Ptr[Obj[T]]]
const T * x	Obj[Ptr[ConstObj[T]]]
T * const x	ConstObj[Ptr[Obj[T]]]
T x[3]	Arr[Obj[T]]
const T x[3]	Arr[ConstObj[T]]
IntList x	ConstObj[IntList]
int x(int i)	ConstObj[cpp_function]

Table 1: Sorts of Global Variables

Tables 1, 2 assume that IntList is a structure of the following type:

```
struct IntList{
int val;
IntList *next;
}
```

In these tables a term x of sort $Ptr[Obj[T]]$ is a pointer that points to an object that contains an abstract value of sort T. To obtain the object that the pointer points to, the operator $*$ must be used. Therefore, $*x$ would be of sort $Obj[T]$. A term x of sort $Arr[Obj[T]]$ is an array of objects that contain abstract values of sort T. To obtain any of these objects, the operator [ ] and the integer index of the particular object are used. A *structure* or a *union* declared globally is an object. Since in

C++ parameters are passed by value (except for reference parameters), a *structure* or a *union* passed as a parameter to a function is not an object but simply a tuple of fields. That is the reason why in the first table the sort of the global variable of type *IntList* is *ConstObj*[*IntList*] and in the second table the sort of the formal parameter of type *IntList* is *Val*[*IntList*].

Declaration	Sort of x (x is formal parameter)
T x	T
const T x	T
T & x	Obj[T]
const T & x	ConstObj[T]
T & const x	ConstObj[T]
T * x	Ptr[Obj[T]]
const T * x	Ptr[ConstObj[T]]
T * const x	Ptr[Obj[T]]
T x[]	Ptr[Obj[T]]
const T x[]	Ptr[ConstObj[T]]
IntList x	Val[IntList]

Table 2: Sorts of Formal Parameters

3.2.4 State Functions

An object can be in an infinite number of states throughout its entire life. Not all states are visible to the client of a class interface. In particular, only a very limited number of states are visible. The states that are not visible to a client are called *internal object states*. Therefore, the states that are of particular interest to the class interface are:

- the *pre-state*, which maps objects to their values just before the function body is run, but after parameter passing,
- the *post-state*, which maps objects to their values at the point of returning from the call (or signaling an exception), but before the function's parameters are out of scope.

To obtain an object's abstract value in a particular state (provided that the object is assigned in that state), a *state function* must be used. There are four state functions

in Larch/C++ :

- **\pre** or ^ : it obtains the abstract value of an object in the pre-state.
- **\post** or ' : it obtains the abstract value of an object in the post-state.
- **\any** : it obtains the abstract value of an object in any arbitrary state. This state function is usually used when the object is immutable (not modifiable) and therefore, its abstract value is the same in both pre-state and post-state.
- **\obj** : it is used to explicitly refer to an object itself, instead of its abstract value. It is only used for emphasis.

The state functions can only be applied to terms that denote objects and their sort is either $Obj[T]$ or $ConstObj[T]$ for some type T. The sort of any object of type T that has been applied one of the first three state functions (**\pre**, **\post**, **\any**) is the same as that of the object but without the leading *Obj* or *ConstObj* sort generator. When the **\any** state function is applied to an object, the sort of the term associated with the state function is the same as the sort of the object itself. For example, if the sort of x is $Obj[int]$ then the sort of x' is *int* and the sort of $x\backslash any$ is $Obj[int]$.

3.2.5 New Features in Function Specifications

Several new clauses have been added lately to the syntax and semantics of Larch/C++.

- The **constructs clause** is an equivalent of the *modifies* clause. Larch/C++ provides this clause for the added convenience of the reader/specifier. This clause is used in constructor functions in order to express that an object is not only modified but there is memory allocated for it, and its attributes are initialized.
- The **trashes clause** is used for any function that trashes objects. In Larch/C++ the trashing of an object is done whenever the object was assigned in the pre-state and not assigned in the post-state, or when the object was allocated in the pre-state and not allocated in the post-state. The trashes clause lists a set of objects that may be trashed from the function.

- The **claims clause**, as in LCL, contains a predicate which does not affect the meaning of a function specification, but rather describes redundant properties which can be checked by a theorem prover. The following example illustrates the use of the *trashes* and *claims* clauses.

```
void dec_ref(char *cp, int & ref_count)
{
   requires allocated(cp, pre) ∧ assigned(ref_count, pre)∧
            ref_count^ >= 1;
   modifies ref_count;
   trashes * cp;
   ensures ref_count' = ref_count^ − 1∧
      (if ref_count' = 0 then trashed(*cp)
       else ¬isTrashed(*cp, pre, post));
   claims ref_count' > 0 ⇒ ¬isTrashed(*cp, pre, post);
}
```

- The **let clause** can appear in any function specification. It can be used in order to abbreviate expressions that will be used many times in the function specification (*requires*, *ensures*, *example* clauses). The following example illustrates the use of this clause.

```
imports BankAccount;
void transfer(BankAccount& source, BankAccount& sink, long int cts)
{
   let amt:Q be dollars(cts),
      presrc:Q be source^, presink:Q be sink^,
      oldsrc:Q be presrc.credit, oldsink:Q be presink.credit;
   requires source! = sink ∧ assigned(source, pre) ∧ assigned(sink, pre)
      ∧oldsrc >= amt ∧ amt >= 0;
   modifies source, sink;
   ensures sink' = set_credit(presink, oldsink + amt)
```

$\wedge source' = set\_credit(presrc, oldsrc - amt);$

}

- The **example clause** can be used to give the reader/specifier a concrete example of the function behavior. Examples do not change the meaning of a specification.

The new keywords that have been recently added in the language are the following:

- The keyword **allocated** can be used in a predicate (*requires*, *ensures* clauses) in order to specify that an object is allocated at a certain state. An object can exist without being allocated.

- The keyword **assigned** can be used in a predicate (*requires*, *ensures* clauses) in order to specify that an object has a well-defined value [11]. In other words an object is assigned if its value is initialized.

- The keyword **fresh** can only appear within an *ensures* clause predicate and it is used to specify that an object was not allocated in the pre-state, but it is allocated in the post-state. The following example illustrates the use of fresh in function specifications.

```
typedef int *ratl;
ratl make_ratl(int n, int d)
{
    requires d > 0;
    ensures assigned(result, post) ∧ size(locs(result)) = 2
      ∧(result[0])' = n ∧ (result[1])' = d
      ∧fresh(result[0], result[1]);
}
```

- The keyword **unchanged** is used within predicates when there is a need to express that a modifiable object is indeed not modified.

- The keyword **reach** can be used with an object to denote the set of all objects reachable from that object.

- The keyword **liberally** can be used in an *ensures* or *claims* clause when the predicate gives a *partial-correctness* specification. In any *partial-correctness* or *total-correctness* specification [21], if the pre-condition is true, and if the function terminates normally, then the post-condition must be true. In *partial-correctness* specifications normal termination is not guaranteed, even when the pre-condition is true. Therefore a specification that does not use the keyword *liberally* is a *total correctness* specification and a specification that uses the keyword *liberally* is a *partial-correctness* specification.

Following are some more new constructs recently built in Larch/C++.

- In C++, a **default value** can be given for a formal argument of a function. This means that when calling the function without supplying a value for one of its particular formal arguments, the argument takes the specified default value. This is also the case in Larch/C++. The syntax is identical.

- Often functions are specified using different **cases**. Larch/C++ provides the specifier with the ability to specify a function using many cases. For every case, the specifier is able to specify a set of different *requires*, *modifies*, *trashes*, *ensures*, *let*, and *claims* clauses. The predicates in the requires clauses should be exclusively disjoint.

- In C++, the interface of a function can declare what **exceptions** the function can throw. The same is also true in Larch/C++ which specifies a function that throws an exception by considering its result to be either the normal result or an exception result. The following example illustrates the specification of exceptions, and the use of cases in a function specification.

```
imports Overflow;
void inc2(int& i) throw(Overflow)
{
    requires assigned(i, pre) ∧ i^ + 2 <= INT_MAX;
    modifies i;
    ensures result = theVoid ∧ i' = i^ + 2;
    requires assigned(i, pre) ∧ i^ + 2 > INT_MAX;
    ensures thrown(Overflow) = theException;
```

}

3.2.6 New Features in Class Specifications

- C++ implicitly defines several member functions for the user if they are not explicitly defined. These are default constructor, copy constructor, destructor, and assignment operator, provided that they are not explicitly declared. Larch/C++ implicitly provides an appropriate specification for these implicitly defined functions. These are called **implicit specifications**.

- Larch/C++ provides the **simulation clause** whereby a user may specify the relationship between super-type and sub-type. As will be seen in Chapter 7, a specification of the super-type might be expressed in a different mathematical domain. The use of the *simulation* function gives valid meaning to a super-type's specifications in the context of a sub-type. The behavior of the simulation function is specified using an LSL trait.

- Larch/C++ allows the specifier to specify **history constraints** on the values that an object may take. It specifies a relationship between each pair of visible states ordered in time. An object may remain immutable through its entire life. As an example, in a *Person* object with fields *Name* and *Age*, the *Age* field may only be increasing. These cases can be specified using the **history constraint clause**. A history constraint is a syntactic sugar in Larch/C++. The same behavior can be specified in the predicate of every member and friend function's postcondition, except for the constructors.

- Larch/C++ provides an **invariant clause** which allows the user to specify an invariant property that must be true during the entire life time of an object. There are two equivalent ways of interpreting invariants. The first is that an invariant is conjoined with the pre and post-condition of each member function in the specification. The second is that the invariant is considered to be true in all visible states. Within an implementation of a member function, an invariant may be temporally violated. This is acceptable, since any intermediate state of the class variable is invisible to clients.

- Larch/C++ can also specify **friendship relationships.** A friendship specification records information that may be needed in the implementation phase of a class. As in C++, friendship grants access to private interface of a class to some or all member functions of a class.

The following is a class interface specification that includes many of the features mentioned in this section.

```
class Person
{
uses PersonTrait, cpp_string: // age interpreted as number of years old
invariant len(self\any.name) > 0 ∧ self\any.age >= 0;
constraint self^.age <= self'.age: // age can only increase
public:
Person(const char *moniker. int years)
{
    requires nullTerminated(moniker. pre) ∧ lengthToNull(moniker, pre) > 0
        ∧years >= 0;
    modifies self;
    ensures self'.name = uptoNull(moniker, pre) ∧ self'.age = years;
}
virtual ¬Person()
{
    ensures true;
}
virtual void change_name(const char *moniker)
{
    requires nullTerminated(moniker, pre) ∧ lengthToNull(moniker, pre) > 0;
    modifies self;
    ensures self'.name = uptoNull(moniker, pre)
        ∧self'.age = self^.age;
}
virtual char * name() const
{
```

```
    ensures nullTerminated(result, post) ∧ fresh(objectsToNull(result, post))
        ∧uptoNull(result, post) = self\any.name;
}
virtual make_year_older()
{
    requires self^.age < INT_MAX;
    modifies self;
    ensures self' = set_age(self^, self^.age + 1);
    example self^.age = 29 ∧ self'.age = 30;
    claims self'.name = self^.name;
}
virtual int years_old() const
{
    ensures result = self\any.age;
}
};
```

Chapter 4

A Functional State-Based Testing Methodology

A lot of research has been conducted around the subjects of *fu· ·tional* and *state-based* testing techniques [16. 35, 19. 20. 31, 32, 59. 45, 46, 47, 10]. The research presented in this thesis is a continuation of the one initiated by Celer in [10]. Celer proposed a Finite State Machine derivation methodology, given Larch/C++ interface specifications, by partitioning the input space of all functions in the interface. In this Chapter, we summarize that methodology, and present its limitations. In the next Chapter we propose ways to overcome these limitations. We first state the definitions followed by the methodology and then its limitations.

4.1 Primitive Definitions

Distinguished Sort : It is the sort that is generated (all its values) by a set of generators (ie. all operators in the *generated by clause*). It is also called *type of interest* or *data sort.*

Generator : A set of generators is the set of operators whose range is the `distinguished sort` and are those which produce all the abstract values of the `distinguished sort`.

Extension : An extension is an operator whose range is the `distinguished sort` but that does not belong to the set of `generators`.

Observer : An observer is an operator whose domain includes the `distinguished sort` and whose range is some other sort (excluding the `distinguished sort`).

Observable Behavior of a Sort : Let the set of `observers` in an LSL trait be denoted by *Obs*. The set of different abstract values that the `distinguished sort` may take is denoted by *Abs*. The set abstract values that can be returned from all observers is denoted by *ObsRet*. The observable behavior of the distinguished sort in an LSL trait is a function Φ from *Obs*, *Abs* to *ObsRet* ($\Phi : Obs \times Abs \rightarrow ObsRet$)

Basic Distinguishable Domain: It is a set of abstract values of the `distinguished sort` D, $D \subseteq Abs$, such that for any observer in *Obs*, the `observable behavior` remains the same. This set of abstract values is produced by the `generators` of the trait. Two `basic distinguishable domains` are always disjoint. The union of all possible `basic distinguishable domains` is the set *Abs* of abstract values that can be produced by the set of `generators` of the trait.

Distinguished Sort Initializer: It is any of the `generators` of the trait which appear on the *generated by* clause. Its signature resembles $New :\rightarrow Sort$. It doesn't have any arguments (besides the ones that compose the sort type or convert some other sort to the distinguished one).

Initial Sort State: It is the `basic distinguishable domain` in which the abstract value of a `distinguished sort initializer` belongs to.

Distinguished Sort Partitioner : It is any qualifying observer defined in an LSL trait which is applied only to the `distinguished sort`. A "qualifying observer" is any `observer` that evaluates in a limited number of abstract values. For example, in the stack trait the *isEmpty* operation is a `distinguished sort partitioner` because it may only evaluate to two different abstract values (*true*, and *false*), and not the *top* operation which evaluates in the range of integers.

Alteration : It is the modification of the `distinguished sort` abstract value. Alteration of an abstract value may or may not change the basic `distinguishable`

`domain` of the `distinguished sort`.

Distinguished Sort Alterator : It is any `generator`, excluding the `distinguished initializers`, or any extension operator. An alterator is assigned to a term, whose sort is the `basic distinguishable sort`. It modifies the abstract value of that term and therefore, it may or may not change the `basic distinguishable domain` of the term.

Distinguished Sort Examiner : It is any `observer`, excluding the `distinguished sort partitioners`. `Examiners` do not change the abstract value of the `distinguished sort` and therefore they do not change the `basic distinguishable domain` either.

Input Domain : It is the set of `basic distinguishable domains` for which an LSL operation is defined (pre condition for LSL operation). Those abstract values for which the operation is not claimed to be defined are specified in the *exemption clause*.

Output Domain :The set of `basic distinguishable domains` to which an LSL operation can lead the distinguished sort to (post condition for LSL operation).

Class Variable : It is described in the interface specification as *self* and is modeled by its respective LSL trait (`distinguished sort`).

State Variable : It is a vector of variables which define the state of the class. This vector contains the following: the `class variable`, and the global variables in the interface specification. In addition, the variables which "compose" (the word compose refers to a sort type which is a tuple or a union) the `distinguished sort`, but are not modified, are included in that vector. If any variable is of the composed `distinguished sort` (e.g. union, tuple), it consists of a set of variables.

4.2 LSL Analysis

This stage of the methodology assumes that there is a given LSL trait that abstractly models a certain data type. The result of this step will provide the required `input`

and `output domains` of all the LSL operations used in the Interface Specification. These will be used in the LCC Analysis which is described in the next section. The LSL Analysis has the following steps:

Step 1: Identify all `distinguished sort initializers, partitioners, alterators, examiners`.

Step 2: Apply all `distinguished sort partitioners` to all `generators` to obtain all `basic distinguishable domains`. For every `partitioner`, the axioms that define it and the corresponding *exemption* clause assertion are examined. By doing this, a number of different `sort domains` are determined where each `partitioner` has consistent behavior. If all `partitioners` behave consistently in the same domains then, these domains are the `basic distinguishable domains`.

Step 3: For all `sort alterators` determine the `input` and `output domains`. To obtain the input domains the signature and the exempting clauses are examined. To obtain the output domains, axioms involving the operation are examined.

Step 4: For all `sort examiners`, determine the input domains. To achieve that, the signature of the operation and the exempting clause are examined.

4.3 LCC Analysis

This stage of the methodology assumes that, for all LSL traits appearing in the *uses* clause of the class interface specification, the LSL Analysis is complete. The LCC Analysis has the following steps:

Step 1: Determine the set of variables under the test.

Step 2: Apply LSL analysis to all traits that appear in the *uses* clause as described in the previous Section.

Step 3: Establish an invariant on the state variable. This invariant appears in

the invariant clause of the interface specifications.

Step 4: Determine the class invariant. It is composed of three parts as described in section 8.4 of [10].

Step 5: For all member function specifications extract all relevant parts (pre/post-conditions, invariants) for the composition of the *Spec_OP*:

$$Spec\_OP = Pre \wedge Post \wedge Inv$$

Step 6: Transform *Spec_OP* into DNF to obtain disjoint sub-relations.

Step 7: Simplify each sub-relation, possibly splitting it into further sub-relations, by using semantics of *First Order Logic*.

Step 8: 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.

Step 9: Perform partition analysis, by reduction to DNF, of the disjunction of the sets of constraints found in Step 8. The resulting partitions will describe disjoint states in which at least one sub-operation either creates the state (corresponding to the post-condition), or is executable in that state (corresponding to the pre-condition).

4.4 FSM Derivation

Step 1: Steps 5, 6, 7 of LCC analysis have provided the transitions of the FSM, and Step 9 has provided its states. The FSM can now be constructed by resolving the constraints of sub-operations against states. A transition labeled OP is created from state S1 to state S2 for every sub-operation OP and every state S1 and S2 satisfying (S1, S2) $\in$ rel-OP where rel-OP is the relation on states defined by the constraints on OP resulting from partition analysis.

Step 2: If possible, simplify the FSM using classical FSM reduction techniques.

4.5 A Simple Example

In this section the FSM construction methodology is illustrated with a simple example. The example of a Larch/C++ specification of the class Queue is used to present this methodology.

4.5.1 LSL Analysis of Queue LSL Trait

Following is the LSL specification of an abstract queue. This trait is taken from the LSL handbook [28]. The specification is straight forward. There are eight operators that are defined. The *empty* operator specifies an empty queue. The *append* operator adds an element at the end of the queue while the *tail* operator removes an element from the front of the queue. The *isEmpty* boolean operator evaluates to whether the queue is empty or not. The *len* operator evaluates to the length of the queue. The *count* operator evaluates to the number of times that a specific item exists inside the queue. Finally the *head* operator evaluates to the first item in the queue.

```
Queue(E, C) : trait
    % FIFO operators
    includes Integer
    introduces
        empty :→ C
        append : E, C → C
        count : E, C → Int
        __ ∈ __ : E, C → Bool
        head : C → E
        tail : C → C
        len : C → Int
        isEmpty : C → Bool
    asserts
        C generated by empty, append
        ∀ q : C, e, e1 : E
```

$count(e, empty) == 0$

$count(e, append(e_1, q)) ==$

$\quad count(e, q) + (\ \textbf{if}\ e = e_1\ \textbf{then}\ 1\ \textbf{else}\ 0)$

$e \in q == count(e, q) > 0$

$head(append(e, q)) ==$

$\quad \textbf{if}\ q = empty\ \textbf{then}\ e\ \textbf{else}\ head(q)$

$tail(append(e, q)) ==$

$\quad \textbf{if}\ q = empty\ \textbf{then}\ empty$

$\quad \textbf{else}\ append(e, tail(q))$

$len(empty) == 0$

$len(append(e, q)) == len(q) + 1$

$isEmpty(q) == q = empty$

implies

$Container(append\ \textbf{for}\ insert)$

$C\ \textbf{partitioned by}\ head, tail, isEmpty$

$\forall\ q : C$

$\quad len(q) \geq 0$

converts $head, tail, len$

$\quad \textbf{exempting}\ head(empty), tail(empty)$

Operator Type	Operator Name
Initializers	empty
Partitioners	isEmpty
Alterators	append, tail
Examiners	count, $\in$, head, len

Table 3: Classification of Queue Trait Operations

We can now proceed with the analysis of the above LSL trait. Table 3 gives the classification of the operators in the Queue trait. The classification is easy to obtain except for the `Partitioner` where one out of the three operators in the *partitioned by* clause had to be chosen. The choice was made according to the number of returned values that these operators have. Both the *head* and the *tail* operators can evaluate to an infinite amount of abstract values. The *isEmpty* operator can evaluate to *true*

or *false*.

Next, the partitioner is applied to the initializer. This is done by the last axiom of the trait which states that *isEmpty(q)* can be either true or false, depending on whether q has the same abstract value as the one evaluated from the initializer *empty*. Therefore, a queue can be either empty or non_empty which implies that there are two basic distinguishable domains, *d_empty* and *d_non_empty*.

Using these two basic distinguishable domains, the input and output domains of the operations are determined:

- **len** operator:
  input domain : $q \in d\_empty \vee q \in d\_non\_empty$
  output domain : Not needed.
  output domain is not needed because *len* is not an alterator.

- **tail** operator:
  input domain : $q \in d\_non\_empty$
  output domain : $q \in d\_empty \vee q \in d\_non\_empty$

- **head** operator:
  input domain : $q \in d\_non\_empty$
  output domain : Not needed.
  output domain is not needed because *head* is not an alterator.

- **count** operator :
  input domain : $q \in d\_empty \vee q \in d\_non\_empty$
  output domain : Not needed
  output domain is not needed because *count* is not an alterator.

- $\in$ operator :
  input domain : $q \in d\_empty \vee q \in d\_non\_empty$
  output domain : Not needed
  output domain is not needed because $\in$ is not an alterator.

- **append** operator :
  input domain : $q \in d\_empty \vee q \in d\_non\_empty$

output domain : $q \in d\_non\_empty$

The output domain can be derived with a small proof using the axioms in the assertions clause.

4.5.2 LCC Analysis of Queue Class Specification

Given the results of the LSL analysis, the LCC analysis is ready to be illustrated . Following is the Larch/C++ specification of an integer *Queue* class, which contains a constructor, a destructor, and a few member functions. As member functions it contains : an *Enqueue* function, which appends an integer object at the end of the queue, a *Dequeue* function, which removes the integer object from the front of the queue and returns it, a *Length* function which returns the length of the queue without modifying the queue object, and a *Find* function which searches the queue for a particular integer and returns true or false according to whether the integer was found in the queue or not.

```
uses Queue(Queue for C, int for E):
class Queue
{
Queue()
    {
        contructs self:
        ensures isEmpty(self');
    }
~Queue()
    {
        trashes self;
        ensures trashed(self);
    }
void Enqueue(int a)
    {
        modifies self;
        ensures self' = append(self^, a);
```

```
    }
int Dequeue()
    {
        modifies self;
        ensures result = head(self^) ∧ self' = tail(self^);
    }
bool Find(int a)
    {
        ensures result = a ∈ self\any;
    }
int Length()
    {
        ensures result = len(self\any);
    }
};
```

LCC Analysis of member functions

From the above specification, it is clear that there is no invariant clause.

- **Queue()** (Constructor)
  pre-condition: $true$
  post-condition: $isEmpty(self')$
  Spec_OP: $self' \in d\_empty$
  simplified: $self' \in d\_empty$
  DNF: $self' \in d\_empty$

- **Enqueue(a)**
  pre-condition: $true$
  post-condition: $self' = append(self^{\wedge}, a)$
  Spec_OP: $(self^{\wedge} \in d\_empty \vee self^{\wedge} \in d\_non\_empty) \wedge self' \in d\_non\_empty$
  simplified: $(self^{\wedge} \in d\_empty \vee self^{\wedge} \in d\_non\_empty) \wedge self' \in d\_non\_empty$
  DNF: $(self^{\wedge} \in d\_empty \wedge self' \in d\_non\_empty) \vee (self^{\wedge} \in d\_non\_empty \wedge self' \in d\_non\_empty)$

- **Dequeue()**

  pre-condition: $true$

  post-condition: $self' = tail(self) \wedge result = head(self)$

  Spec_OP: $self \in d\_non\_empty \wedge (self' \in d\_empty \vee self' \in d\_non\_empty) \wedge self \in d\_non\_empty$

  simplified: $self \in d\_non\_empty \wedge (self' \in d\_empty \vee self' \in d\_non\_empty)$

  DNF: $(self \in d\_non\_empty \wedge self' \in d\_empty) \vee (self \in d\_non\_empty \wedge self' \in d\_non\_empty)$

- **Find(a)**

  pre-condition: $true$

  post-condition: $result = a \in (self \backslash any)$

  Spec_OP: $(self \backslash any \in d\_empty \vee self \backslash any \in d\_non\_empty)$

  simplified:$(self \backslash any \in d\_empty \vee self \backslash any \in d\_non\_empty)$

  DNF: $(self \backslash any \in d\_empty \vee self \backslash any \in d\_non\_empty)$

- **Length(a)**

  pre-condition: $true$

  post-condition: $result = len(self \backslash any)$

  Spec_OP: $(self \backslash any \in d\_empty \vee self \backslash any \in d\_non\_empty)$

  simplified:$(self \backslash any \in d\_empty \vee self \backslash any \in d\_non\_empty)$

  DNF: $(self \backslash any \in d\_empty \vee self \backslash any \in d\_non\_empty)$

4.5.3 FSM Derivation

Now that both the LSL and LCC analysis are complete, the construction of the finite state machine may begin. First let us determine the transitions that are going to be contained in the FSM. Table 4 gives information about any transition that can be derived from the Larch/C++ specification of the Queue class.

From Table 4 it is obvious that the derived state machine can have two states. Table 5 summarizes the two states along with their constraints. For the sake of simplicity the states have been given a number for name.

Using Table 5 we can construct a state machine that represents the *Queue* class specification. This state machine is shown in Figure 2.

Transition	Label	pre-state	post-state
1	Queue	$true$	$self \in d\_empty$
2	Enqueue	$self \in d\_empty$	$self \in d\_non\_empty$
3	Enqueue	$self \in d\_non\_empty$	$self \in d\_non\_empty$
4	Dequeue	$self \in d\_non\_empty$	$self \in d\_empty$
5	Dequeue	$self \in d\_non\_empty$	$self \in d\_non\_empty$
6	Find	$self \in d\_empty$	$self \in d\_empty$
7	Find	$self \in d\_non\_empty$	$self \in d\_non\_empty$
8	Length	$self \in d\_empty$	$self \in d\_empty$
9	Length	$self \in d\_non\_empty$	$self \in d\_non\_empty$

Table 4: Queue Class Transitions

State	Constraints
1	$self \in d\_empty$
2	$self \in d\_non\_empty$

Table 5: Queue Class States

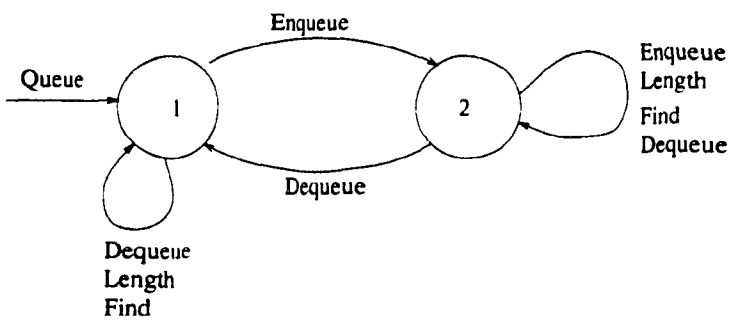


Figure 2: The Queue Class State Machine

4.6 Limitations of Methodology

In the previous section, the methodology, proposed in [10] was illustrated with an example specification of a *Queue* class. Although the methodology in theory satisfies

many testing objectives as described in [10], the example shows that there are some issues that have not been taken into consideration. In this Section, these issues will be addressed, while in the next Chapter a solution will be proposed.

4.6.1 Non-Determinism

As already seen in the previous sections, the methodology derives a finite state machine where the transitions are labeled with class function names, and the states contain constraints specific to the class variable. In the simple *Queue* example that was illustrated in the previous section, a finite state machine was derived by analyzing Larch/C++ specifications. This finite state machine is non-deterministic. The non-determinism is clearly noticeable in state *2*, where there are two departing transitions with label *Dequeue*.

The finite state machine that is derived from formal specifications is used in the test-case generation and test-case execution processes. During the test-case generation process, test-cases will be selected. At the stage of the test-case execution, the validity of the system will be checked against the constraints of the current state in the finite state machine. This is going to be done by selecting paths in the finite state machine. When non-determinism appears in a finite state machine, the issue of selecting a test sequence from a finite state machine becomes a lot more complex. The non-determinism in a finite state machine that is derived from a class specification affects the test-case generation process. For instance, using the FSM in Figure 2 the tester will not be able to decide if the following test sequence should lead to state *1* or *2*:

```
Queue q;
q.Enqueue(6);
a = q.Dequeue();
```

Therefore, the tester would not be able to claim that the above test sequence was tested successfully or not, because there is no way to resolve the non-determinism and test for the validity of one of the two state constraints. Of course, in this simple example the tester can guess which state the test sequence should lead to. Guessing

is an absolutely not reliable method to resolve non-determinism in a finite state machine that is used for testing. Also, in more complicated cases with more complex specifications and greater number of member functions, the derived finite state machines would be more complex and the tester would not be able to guess in order to resolve non-determinism.

The issue of non-determinism in the derived finite state machine is not only present in this methodology, but is also in the methodology developed in [20, 19] which motivated Celer's work. Non-determinism in this case was to be expected, since the finite state machine is a finite abstraction of the infinite number of states that the class variable is allowed to be in, by the specification. Often the non-determinism is a result of a loose specification and may not be resolved until the implementation has been completed. The issue of resolving non-determinism is going to investigated in this thesis.

4.6.2 Consideration of Class Destructor

One other issue that is to be investigated is the *Destructor* of a class. A *Destructor* is a major part of a C++ class and its specification. It should be part of the test sequences that are derived for the testing of a particular class. In the *Queue* example that was shown in the previous section, the *Queue* destructor was not used to derive the finite state machine. Consequently, the FSM contains neither a state that a destructor would lead to, nor a transition with the name *~Queue*.

4.6.3 Considering the New Larch/C++ Features

As seen in the last Chapter, Larch/C++ has gone through many changes since this methodology was developed. It is therefore expected that many of the new constructs and features that have been added in the language, are not taken into consideration when deriving a finite state machine. The derived finite state machine should reflect the formal specification of the class as it is documented in Larch/C++, in order for the entire behavior of the class to be tested and not only partial. In this thesis, ways to involve all the current features of Larch/C++ in the finite state machine construction will be investigated.

4.6.4 Relation Between Sub-type and Super-type FSMs

As already noted, one of the most important characteristics of C++, and of all object-oriented languages, is inheritance. Celer's work did not address this issue. In the next Chapter, we will see that Larch/C++ provides a mechanism that enables *inheritance of specifications.* This mechanism allows the user to specify behavioral sub-types. In this thesis, the relationship between finite state machines of a sub-type and of a super-type will be examined.

4.6.5 Test-case Generation

The methodology provides a finite state machine that expresses the formal specification of class. Given such a state machine, the final goal is to derive test sequences that will be executed in order to test the implementation of a class against its specification. The work described in [10] does not contain a technique or algorithm that, given such a state machine, will produce a set of test sequences, execute them and check the constraints of the states that the class variable passes through. In this thesis, a test-case generation technique will be investigated.

Chapter 5

FSM Derivation: A Revised Technique

In this Chapter, the weaknesses of the FSM generation methodology are studied in detail and some solutions for the strengthening and enhancing of that technique are proposed.

5.1 Non-Determinism and its Consequences

In Chapter 4, a technique that uses formal specifications, in order to produce a finite state machine was studied. The primary goal for translating formal specifications into a finite state machine was to automate the test-case generating process. Since the finite state machine is a finite abstraction of the infinite number of states that the class variable is allowed to be in, the finite state machine contains non-determinism. The presence of non-determinism in a finite state machine makes the test-case generation process rather complicated and not accurate.

5.1.1 Complications in State Constraint Selection

During the test-case generation process, the tester selects representative test sequences which are going to be used for the testing of the associated class implementation. While selecting the test sequences, the tester also selects the state constraints that the class variable should satisfy after the execution of a test sequence.

When non-determinism appears in a finite state machine, the selection of a state constraint for a particular test-case is impossible and therefore the testing of the class variable cannot be completed. Let us illustrate this with the *queue* example seen in Chapter 4. Using the finite state machine that was created from the formal specification of the class *Queue*, the following test sequence may be generated:

```
{
int a, b;
Queue q;
    q.Enqueue(a);
    b = q.Dequeue();
}
```

In order to test the class variable with the above test sequence, state constraints for every operation in the test sequence must be selected . These constraints are going to be the constraints of the states that each operation leads to. Therefore, after executing the constructor of the *Queue* object, the constraints of state 1 should be checked ($self \in d\_empty$). After sending the *Enqueue* message to the *Queue* object q, the constraints of state 2 should be checked ($self \in d\_non\_empty$).

After sending the *Dequeue* message to the *Queue* object q, there are two transitions with the label **Dequeue** that have state 2 as source state, but different destination states. Consequently, it is not possible to decide the state that the operation leads to. Therefore, it is obvious that the finite state machine does not reflect the specification of the class very accurately. The reason for this inaccuracy, is the non-determinism that appears in the derived finite state machine.

5.1.2 Legality of a Test-Case

Another issue that results from the existence of non-determinism in a finite state machine is the construction of *illegal* test-cases. As it will be seen in the next Chapter, an *illegal* test-case is a test-case that is not faithful to the class specification.

Let a finite state machine M_1 contain non-determinism in state "A" with the

following transitions t_1, t_2, ... t_n; these transitions correspond to the same member function "F".

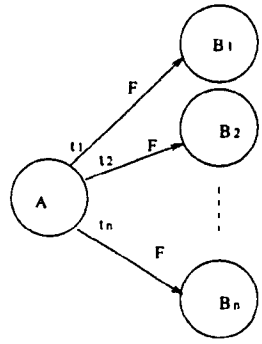


Figure 3: Non-determinism

t_1 has the state "B_1" as destination, while t_2 has the state "B_2",... ,and t_n has the state "B_n" (see Figure 3). In the context of finite state machines, the transitions t_1, t_2 ... t_n may be followed at any time, given that the current state is "A". In the context of black-box and state-based testing, only a *legal* transition may be traversed at a specific time. t_1, t_2 ... t_n may correspond to the same member function, but they all refer to a different sub-state of the class variable according to the class specification. In particular, the sets of sub-states that t_1, t_2, ... t_n are allowed to be traversed are mutually exclusive. A transition t_1 is *legal* when the constraints of the sub-state of the class variable, that t_1 refers to, are satisfied. Otherwise, the transition is *illegal*. Therefore, only one of the transitions that create the non-determinism may be traversed at a time. These sets of sub-states can be derived from the class specification. A test-case becomes *illegal* when the test-case contains an *illegal* transition.

Let us illustrate this with the *Queue* example from Chapter 4. In state 2 of the finite state machine, there is non-determinism on the two transitions that are labeled *Dequeue* (see Figure 2). In the test-case generating process we can derive the following test-case with two different state constraints for the last operation:

```
{
int a, b;
Queue q;
    q.Enqueue(a);
    b = q.Dequeue(); }
```

The first test-case requires the class variable to be tested against the state constraints of state 1 while the second test-case requires the class variable to be tested against the state constraints of state 2. According to the class specification, the second test-case is *illegal* because when the queue object, that contains only one item, is sent a *Dequeue* message, the object would never arrive at a non-empty state. Therefore, the second test-case is not faithful to the class specification. This does not mean that the transition *Dequeue*, that has source state 2, and destination state 2, is always unfaithful to the specification. For other test-cases, the transition might be *legal*. For example, in the following test-case, we can derive two different state constraints for the last operation.

```
{
int a, c, b;
Queue q;
    q.Enqueue(a);
    q.Enqueue(c);
    b = q.Dequeue();
}
```

The non-determinism appears again in state 2 with the same two transitions. This time, the transition that leads to state 1 (empty state) is *illegal*, while the transition that leads to state 2 (non-empty state) is *legal*.

5.1.3 Eliminating the Non-Determinism

In the above two sections, the problems that result from using a finite state machine, that contains non-determinism, to generate test-cases were analyzed. In this section a solution is given in order that the tester may be able to generate *legal* test-cases. That is, only test-cases that would test a class implementation correctly would be generated. Since the cause of the above mentioned problems is the existence of non-determinism in a finite state machine, a valid solution to these problems, would be its elimination. Next is a presentation of the proposed solution which extends the finite state machine, that is constructed from Larch/C++ specification, to a *Conditioned Finite State Machine*. Let us first introduce the notion of a *Conditioned Finite State Machine*.

Definition: A *Conditioned Finite State Machine* consists of a six-tuple (S, Σ, B, δ, i, F), where:

- S is the finite set of states.
- Σ is the machine's alphabet.
- B is a set of Boolean expressions.
- δ is a function (called the transition function) $\delta : S \times \Sigma \times B \rightarrow S$ such that, $\delta(p, x, c) = q$ if and only if the machine can move from state p to state q while reading the symbol x and the boolean condition c is true.
- i (an element of S) is the initial state.
- F (a subset of S) is the set of accept states.

Statement: With a *Conditioned Finite State Machine*, as defined above, since δ is a function, it follows that any two distinct transitions that start from the same state s_1 with the same input symbol x_1 must have different boolean conditions c_1 and c_2.

As seen in the above definition, a *Conditioned Finite State Machine* is a finite state machine with conditions on every transition. In the previous section, it was seen that, whenever we have non-determinism, there is atmost one transition that can be followed, depending on the sub-state of the class variable. The condition associated with each transition is meant to question the class variable. Therefore, when generating a test-case, a transition may be followed, if and only if the condition associated with the transition is satisfied.

We now demonstrate how the conflict of the previous *Queue* example is resolved using a conditioned finite state machine instead of a plain finite state machine. Figure 4 shows an FSM for the *Queue* class, along with a condition for the transition that causes the non-determinism.

Using this state machine, the following test-case does not derive two different state constraints. One of them is characterized *illegal* from the condition on the transitions.

We now show how the original methodology is modified in order to be able to create a conditioned FSM instead of a plain FSM. The steps to deriving a *conditioned finite state machine* would be the same as the ones described in Chapter 4 but with some changes:

After creating the finite state machine, according to the steps in Chapter 4, if the machine contains non-determinism on a particular set of transitions related to a particular member function, a further analysis of the function specification should be done in order to derive necessary conditions for the set of transitions that cause the non-determinism. This analysis should be done on the set of transitions that is causing the non-determinism, using the constraints of the pre and post states of a particular transition along with the defined LSL axioms. Since these conditions are supposed to be questioning the class variable, they should contain `observers` defined on the `distinguished sort`.

Note: Transitions that do not cause non-determinism are not required to have

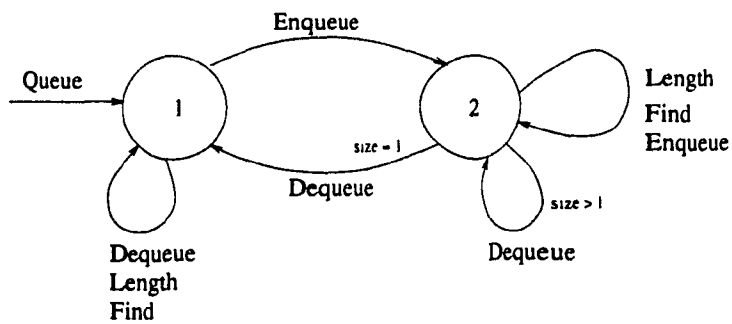


Figure 4: A Conditioned FSM for the Queue Class

conditions. In this case, the value *true* is assigned for the particular transition.

Let us illustrate the method with the *Queue* example from Chapter 4. By observing the finite state machine it is clear that the machine contains non-determinism at state 2 on the member function Dequeue. First analyze the transition that leads from state 2 to state 1. For this transition the following predicates can be derived from the specification :

$$self \in d\_non\_empty \wedge self' \in d\_empty$$

$$self \in d\_non\_empty \Rightarrow self = append(x, \epsilon)$$

$$self' \in d\_empty \Rightarrow self' = empty$$

$self' = tail(self)$

$empty = tail(append(x, e)) == x = empty$

$len(self) = len(append(empty, e)) = len(empty) + 1 = 0 + 1 = 1$

Now analyze the transition that leads to state 2 from state 2. Using the specification the following predicates hold from this transition :

$self \in d\_non\_empty \wedge self' \in d\_non\_empty$

$self \in d\_non\_empty \Rightarrow self = append(x, e)$

$self' \in d\_non\_empty \Rightarrow self' = append(y, a)$

$self' = tail(self)$

$len(y) \geq 0$

$append(y, a) = tail(append(x, e)) == x = append(y, a)$

$len(self) = len(append(append(y, a), e)) = len(append(y, a)) + 1 =$
$len(y) + 2 \geq 2$

These two conditions are mutually exclusive and their union results to the set of all possible values that the length of a queue may take in state 2. Therefore, in order for the first transition to be traversed, $len(self) = 1$ has to be to true, and in order for the second transition to be traversed, $len(self) > 1$ has to be true.

Sometimes, the class specification is loose or incomplete. In these cases, the derivation of necessary conditions for every transition may not be possible. The only solution to eliminate non-determinism, is to complete the specification according to the criterion of sufficient completeness that was derived by Umansky and Colagrosso in [60, 17].

5.2 FSM Expressing Class Behavior

Looking closely to the state constraints, it can be concluded that the methodology enables us only to check constraints that deal with whether or not the *Queue* is empty. It does not enable the tester to test the *FIFO* (First In First Out) property of a queue.

For instance, when executing the following test sequence:

```
int c;
Queue a;
a.Enqueue(6);
a.Enqueue(7);
c = a.Dequeue();
```

it would be expected for the object *c* to contain the value *6*. According to the information that the FSM provides, the value *6* may be an incorrect one. The object *c* may actually contain any value and the class variable would not be empty. The tester would detect no error in the implementation because the FSM does not provide any constraints that will make the tester check if the returned value of the *Dequeue* function is correct, or if the object *a* actually contains two elements (*6* and *7*) after the second invocation of the *Enqueue* function.

The methodology succeeds to provide the tester with valid test sequences of functions for which the class variable would arrive at a valid state after executing them. The derived FSM cannot be expected to fully express the specified class behavior. The result of a function is specified in the function specification. Therefore, when a particular class implementation is being tested with test-cases, constructed using a finite state machine, the result of a certain function is checked against the predicate given in the function specification.

5.3 Involvement of Class Destructor

A destructor in *Object-Oriented* programming plays a very important role in the implementation of a class. In C++, a destructor's primary responsibility is to deallocate objects that the constructor, or other member functions, may have dynamically allocated through the object's life time. When a constructor is used to create an object, the program undertakes the responsibility of tracking that object until it expires. At that time, the program automatically calls the destructor of that object. Since such dynamic allocations, that are done from a constructor or any other member functions,

are usually invisible to the clients, the deallocations are also usually of no concern to clients. Thus a specification of a destructor, merely states that calling the destructor modifies nothing and terminates.

Therefore, it is clear that since a destructor is of no concern to the client of a class and since it is called automatically, there is no reason for it to be part of a test-case, and therefore there is no need for it to be part of the derived finite state machine.

In order for the derived finite state machine to be complete, as far as the specification of a class is concerned, we are going to propose a simple way to include the destructor of a class in the derived state machine. The destructor of a class is called automatically by the client program as soon as the object expires. An object is valid only in the block of code that it has been constructed. When the control of the client program gets past the end of the block, the object expires. Therefore, a destructor can be called at any time during the object's life time. This implies that, from any state in the derived finite state machine a destructor transition may be traversed which will lead to a destruction state that will contain no state constraints. Figure 5 shows the queue example with the optional (dotted) destructor transitions and states.

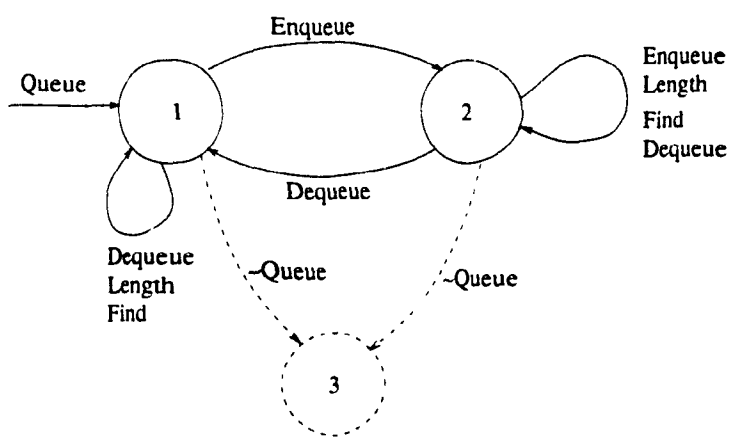


Figure 5: The Queue Class State Machine with Destructor Transitions

5.4 FSM Derivation and the New Larch/C++ Constructs

As seen in Chapter 3, Larch/C++ has evolved since the original methodology was developed. Many constructs have been added and some of the already existing ones have been modified. In this Chapter, the methodology will be modified accordingly, in order for the derived finite state machine to reflect the exact behavior of the class variable that is specified in Larch/C++.

5.4.1 State Functions

The state functions are not new in Larch/C++. The `\pre` and `\post` functions existed from the initial construction of the language and they are also part of the original methodology. The new state functions that has been recently added to Larch/C++ are the `\any` and `\obj`.

The `\any` state function is used whenever an object is immutable and the specifier wants to state something about the object but in no particular state (pre and/or post). Therefore, when stating something about an object using the `\any` state function, it is equivalent to stating the same thing for both pre and post states of the object. Consequently, every application of the `\any` state function to *self* in a certain expression, will be replaced by the conjunction of two occurrences of the same expression where in the first occurrence the `\any` function will be replaced by `\pre` and in the second occurrence it will be replaced by `\post`

In following example, there is a predicate that states that a queue object (*self*) is empty in any state.

$self\backslash any = empty$

By applying the above mentioned rule, the following equivalent predicate may be derived:

$self\backslash pre = empty \wedge self\backslash post = empty$ or

$self = empty \wedge self' = empty$

5.4.2 Modifies Clause

The *modifies* clause is not new to Larch/C++ either. This clause has always been part of any Larch specification language. It was not taken into consideration in the original methodology. In many cases this would not cause any harm. But in some other cases it would cause a lot of trouble in deriving state constraints.

In particular, function specifications that state that a certain object may be modified (in our case *self*) have special assertions that specify how this object is modified. In cases where the *modifies* clause does not list all the objects visible by the client and the function, Larch/C++ implicitly states that these objects remain immutable after the execution of the function terminates. Therefore, if *self* is omitted from the *modifies* clause, then Larch/++ implicitly states:

$$unchanged(self)$$

or as will be seen in the next sub-section,

$$self = self'$$

Definition: We define *Modifies_Predicate* to be:

$$Modifies\_Predicate = \begin{cases} self = self' & \text{if self is omitted from the modifies clause} \\ true & \text{otherwise} \end{cases}$$

Therefore, the *Spec_OP* is modified as follows:

$$Spec\_OP = Pre \wedge Post \wedge Inv \wedge Modifies\_Predicate$$

5.4.3 Larch/C++ Keywords

Let Clause:

A specification using a *let clause* is syntactic sugar for a specification written without it. The meaning of such specification is given by textually replacing the defined variables with their definitions. This "de-sugaring" is required before analyzing the interface specification.

Case Analysis:

The *case analysis* is also syntactic sugar. Given a specification S_1 that uses n cases:

requires Pre_1;
modifies O_1;
ensures $Post_1$;

requires Pre_2;
modifies O_2;
ensures $Post_2$;
.....
.....
requires Pre_n;
modifies O_n;
ensures $Post_n$;

where Pre_i, $Post_i$, and O_i, $1 \leq i \leq n$, are the pre , post conditions and the set of objects that may be modified by the ith case. An equivalent "de-sugared" specification S_2, can be obtained as follows:

requires $Pre_1 \vee Pre_2 \vee ... \vee Pre_n$;
modifies $O_1 \cup O_2 \cup ... \cup O_n$;
ensures $Post_1 \wedge Post_2 \wedge ... \wedge Post_n$;

This "de-sugaring" is required before analyzing the interface specification.

Liberal Specifications:

Liberal specifications add looseness to the specification of a class. This looseness contributes to the existence of non-determinism in the finite state machine. As noted earlier the problems that non-determinism brings, can be avoided if we replace the FSM with a conditioned FSM. The non-determinism that is resulting from the looseness of a specification cannot be avoided with a conditioned FSM. Therefore, the following assumption will be made: the user does not specify a class with partial correctness specifications.

Claims Clause:

A *claim* is valid for an entire function specification if it is proved. To prove a *claim*, the truth of the following expression has to be established:

$$(Pre \wedge Post \wedge Inv \wedge MP \wedge TP \wedge Hist) \Rightarrow claim$$

where Pre, $Post$, are the pre and post conditions, Inv is the invariant of the class,

Hist is the specified history constraint for the class. Finally, TP and MP, are the predicates that code the *trashes clause* and *modifies clause* respectively [40]. Given that a claim is already proved, it may assist in the partition analysis that is performed during the LCC analysis, by defining the Spec_OP as:

$$Spec\_OP = (Pre \wedge Post \wedge Inv \wedge Modifies\_Predicate) \Rightarrow claim$$

Unchanged:

Informally, *unchanged* asserts that its argument objects are not mutated. Since, the only visible states, for a client, are the pre and post states, then *unchanged(x)* may be "de-sugared" with $x^{\wedge} = x'$, which means that the value of the object in the post state is the same as the value of the object in the pre state.

Allocated and Assigned:

Allocated is used to explicitly state that an object has allocated memory for. *Assigned* is used to explicitly state that an object has memory allocated for, and it is also initialized with a *well defined value* [11]. These two primitives are going to be used as part of any Spec_OP that is derived for any member function specification. Consequently, any state constraint may contain the terms *allocated* and *assigned*, and the tester will have to verify their correctness.

Reach:

Since *reach(x)* denotes the set of all objects that are directly reachable from x, it is necessary for the Larch/C++ user to explicitly state which objects are directly contained in each sort of abstract value. This could be accomplished, by specifying the trait function `contained_objects` [40]. In this way, *reach(x)* can be replaced by the set of objects directly reachable from x, whenever used in a function specification.

5.4.4 Implicit Functions

Since C++ defines implicit functions and Larch/C++ implicitly specifies them, the FSM deriving methodology should consider these function specifications when constructing the machine.

As already discussed in Chapter 3, these functions are: the *default constructor*, the

copy constructor, the *destructor*, and the *assignment operator*. Following, we show the specification that is implicitly generated for the above mentioned member functions, for a class T, by Larch/C++.

```
T()
    {
        constructs self;
        ensures true;
        claims assigned(self, post);
    }

T(const T& arg)
    {
        requires assigned(arg, any);
        constructs self;
        ensures self' = arg\any;
        claims assigned(self, post);
    }

~T()
    {
        ensures true;
    }

T& operator = (const T& from)
    {
        requires assigned(from, any);
        modifies self;
        ensures result = self ∧ self' = from\any;
        claims assigned(result, post);
    }
```

These member function specifications should also be included in the LCC analysis when deriving the finite state machine for a certain class that the specifier did not

explicitly specify.

5.4.5 History Constraints

The history constraints are not part of the original methodology because they were introduced recently to Larch/C++. This clause states a particular property that the class implementation should preserve. As already stated in Chapter 3, it is syntactic sugar for Larch/C++, as the same behavior can be specified in the *ensures* clauses of every function of the specified class. Therefore, the Spec_OP expression takes the following form:

$$Spec\_OP = (Pre \wedge Post \wedge Inv \wedge Hist \wedge Modifies\_Predicate) \Rightarrow claim$$

5.5 Multiple Constructors

The methodology as presented in [10], creates a finite state machine with the initial state being the state that the class constructor leads to. In most cases, a class may have more than one constructor. Each one may lead the class variable to another state. This would mean that the derived finite state machine would have more than one initial state which is inconsistent with the definition of a finite state machine.

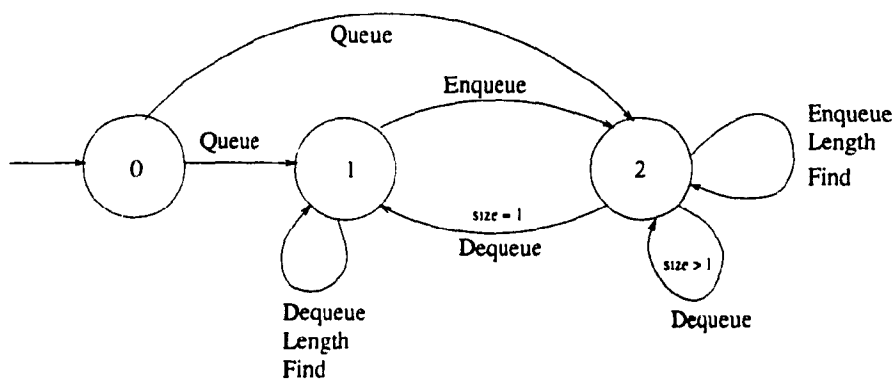


Figure 6: The Queue Class State Machine with Multiple Constructors

A simple solution to this problem would be to employ an additional state with no state constraints. This state would be the only initial state in the machine. The initial transition that would lead to this state would not be associated with any member functions. This state would be the source state of all the constructor associated transitions. Consequently, for every state machine, with n states, derived by the

original methodology, another machine with the same structure and $n + 1$ states may be derived using this new approach. All transitions remain the same except for constructors which they now require a source state. The conditioned FSM that is derived for the Queue class is shown in Figure 6.

Chapter 6

Test-Case Generation

We have seen so far how to create a finite state machine from Larch/C++ specifications. In this Chapter we first study some of the previous methods used in order to generate a test suite from a finite state machine, and we argue the point that these methods are not suitable to derive a test suite in our methodology. Next, we develop a new algorithm to derive test sequences using a finite state machine, and we comment on the test suite adequacy.

6.1 Previous Work on Test-Case Generation

The frequent use of finite state machines for the specification of software has led to much research in constructing test suites from such specifications. A test suite should be able to "cover" the entire specification of a software system.

Many methods have been developed in the context of communication protocol testing. The goal of the test-cases derived from these methods is to validate whether the properties stated by the protocol specification are satisfied by the given protocol implementation. The most popular methods are the "Automata Theoretic" [16] developed by Chow, the "Distinguishing Sequence" [26] developed by Gonenc, the "Unique Input Output" (UIO) [54] developed by Sabmani and Dahbura, and the "Partial W-method" [35] developed by Fujiwara et al.

Although these techniques achieve acceptable test coverage for protocol testing.

they don't seem to be applicable in the context of object-oriented class testing. The protocol methods have a few basic requirements which restrict the wide use of any FSM. In particular. the above mentioned protocol techniques require a deterministic and minimal finite state machine. In addition, they require every transition in the FSM to output a value which is used to construct the test suite.

In the technique proposed in this thesis, the resolution of non-determinism is achieved with the notion of the *conditioned finite state machine*. A transition may be followed only when its associated condition is satisfied. The same does not hold for the above mentioned protocol methods since a transition may be followed at any time.

The "automata theoretic" and the "partial W-method" assume the existence of a correctly implemented function *Reset*. A *Reset* function is a function which when invoked forces the system to safely return to the initial state. Even though a *Reset* function may be correctly implemented. it is not required by our method.

In addition, most of the protocol techniques mentioned earlier assume a minimal finite state machine as input. LSL and LCC analysis cannot guarantee the construction of a minimal finite state machine. This issue depends entirely on the class specification.

Finally, a part of every protocol method, mentioned above, is the existence of a set of output responses (including the null output) that are emitted when a transition is followed. Each transition may emit only one output. These output responses are used to derive the characterization set of the FSM. The characterization set [24] consists of input sequences that can distinguish between the behaviors of every pair of states in a minimal automaton. The behavior of a pair of states is the output response of a transition that connects this pair. In the method proposed in this thesis, a transition corresponds to a member function invocation. Each member function specification contains the specification of its result which is analogous to the notion of output responses. A function result is specified using abstract mathematical domain. This means that a property of the result is known by the specification and not the actual result. The abstract value that specifies the function result may correspond to a set of actual results and not only one, as it is required the by the above mentioned protocol

testing techniques.

For the above stated reasons, we are going to propose a new test sequencing technique that will apply to object-oriented class testing and it will require as input a finite state machine that is constructed with LSL and LCC analysis.

6.2 Test Cases versus Legal Traces

In the original methodology that was proposed in [10], Larch/C++ specifications were translated into a non-deterministic finite state machine. Let us recall the definition of a non-deterministic finite state machine:

Definition: A non-deterministic finite state machine consists of a quintuple (S, Σ,ρ, i, F), where:

- S is the finite set of states.
- Σ is the machine's alphabet.
- ρ is the Cartesian product $S \times \Sigma \times S$
- i (an element of S) is the initial state.
- F (a subset of S) is the set of accept states.

Definition: A *Test Sequence* generated from a formal specification is a sequence of member function invocations. Such a sequence must always begin with the invocation of a class constructor. A class constructor may not appear in a test sequence in a place other than the beginning of the sequence.

A *test sequence* corresponds to the notion of *legal trace* described in [17]. The set of all *traces* of a class C is the set of all permutations, of any length, of the member functions in C, with two restrictions: the first member function should be a class constructor, and the rest of the permutation does not contain any constructors of class C. Therefore, a *trace* of a class C represents a sequence of messages that may be sent to an object of the class C, starting at the initial state. A *legal trace* is a *trace* which is faithful to the class specification. A *legal trace* ensures the correct usage of the class' methods. The set of *legal traces* is a subset of the set of all *Traces* of a class.

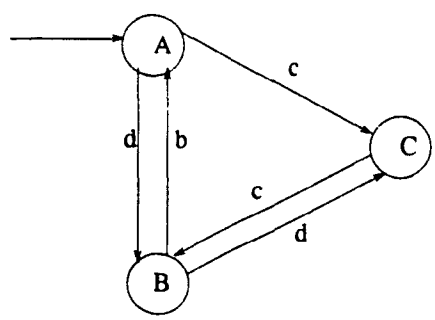
Any *trace* that is not *legal* is called *illegal*. Therefore:

- $LegalTr(C) \subset Traces(C)$
- $IllegalTr(C) \subset Traces(C)$
- $LegalTr(C) \cup IllegalTr(C) = Traces(C)$
- $LegalTr(C) \cap IllegalTr(C) = \emptyset$

From the above we can conclude that the notion of a *legal trace* is equivalent to the notion of a *test sequence*.

In a finite state machine derived from a class specification, a path originating from the initial state corresponds to a *legal trace* of that class. Since the finite state machine is faithful to the class specification, the set of all *legal traces*, $LegalTr(C)$, may be produced by the finite state machine. In the case where the finite state machine contains non-determinism, a subset of the set of *illegal traces* may also be produced.

In the remaining part of this thesis, a *test sequence* or a *trace* will be denoted by a string of transition labels separated by semicolons. Figure 7, is an example of a finite state machine, and three arbitrary *test sequences*. In this example, there is a state



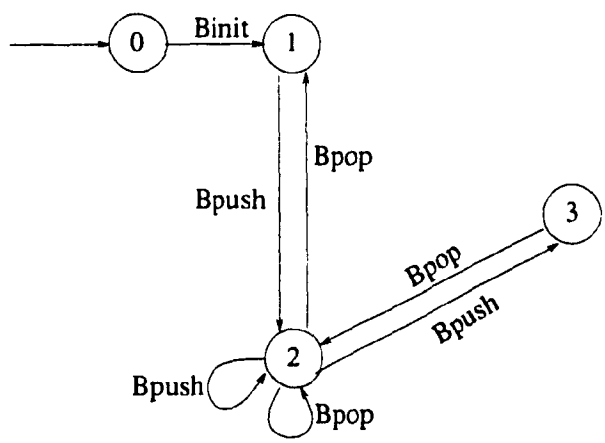
Paths
d;d;c;b;c
c;c;b
c;c;d;c;b;d;d

Figure 7: A Few Simple Test-Cases

machine with three states, namely A, B, C. State A is the initial state. Usually, the

notation of an accept state in a finite state machine, is different from the notation of non-accept states. In this methodology, any state can be an accept state. Therefore, there is no need of adopting the accept state notation. Furthermore, we see that the labels of the transitions are simply letters. In practice, the labels of the transitions are names of interface operations. Therefore, in the following bounded stack example, (Figure 8) the sequence [*Binit; Bpush; Bpush; Bpop; Bpush; Bpop*] is a *trace* and therefore, a *test sequence*.

The non-determinism that may appear in a finite state machine, may affect the test-case generation process. Since a finite state machine, that allows non-determinism, is able to produce both *legal* and *illegal traces*, the test-case generator would be confused when generating test-cases by traversing paths on the finite state machine. For instance, the test-case [*Binit; Bpush; Bpop*] is unable to be characterized as *legal* or *illegal*, since there are two paths in the finite state machine that can be followed and produce this test-case. Since the specification of the class states that a *Bpop* message sent to a stack object with one element, would lead the class variable to an empty state, the test-case that leads to state 1 (empty state) is *legal*, while the test case that leads to state 2 (not empty and not full) is illegal. Therefore, using a non-deterministic finite state machine for the test-case generation provides a set of test-cases that are both *legal* and *illegal*. As already seen in Chapter 5, the



[Binit; Bpush; Bpush; Bpop; Bpush; Bpop]

Figure 8: FSM of a Bounded Stack

revised methodology translates Larch/C++ specifications into a *conditioned finite*

state machine. A path originating from the initial state of the *conditioned finite state machine* corresponds to a *legal trace.* Since the problem of the non-determinism is resolved with the conditioned transitions, the machine does not produce *illegal traces.* Therefore, any trace generated from a *conditioned finite state machine* corresponds to a *legal trace.* Conversely, the entire set of *legal traces* for a class C, can be produced by the *conditioned finite state machine* of C.

6.3 Satisfaction of State Constraints

In the previous section, it was observed that to produce any test-case (*legal trace*), it is sufficient to translate the formal specification of a class into a *conditioned finite state machine.* In this section, a criterion will be introduced which will assist the tester in classifying the test-cases into categories.

Definition: Given two test-cases S_1 and S_2, the *is_prefix* relation is defined by the following LSL trait:

PrefixTrait : **trait**
  includes $String(E, S)$
  introduces
    $is\_prefix : S, S \rightarrow Bool$
  asserts
    $\forall\ s_1, s_2 : S, e, f : E$
      $is\_prefix(empty, s_1)$
      $\neg is\_prefix(s_1, empty)$
      $is\_prefix(e \dashv s_1, f \dashv s_2) == e = f \wedge is\_prefix(s_1, s_2)$

Definition: Let p be a test-case that can be generated by a *conditioned finite state machine* M. Define $\tilde{p}$ to be the set of all test-cases that have p as prefix:

$$\tilde{p} = \{q | is\_prefix(p, q)\}.$$

p is called the *generator* of the set $\tilde{p}$.

Statement: If a test-case p is tested correct, with respect to its state constraints,

then the remaining of the test-cases in the $\tilde{p}$ set, can be tested without re-checking the state constraints of the generator test-case p.

Therefore, the testing process of any of the test-cases of the $\tilde{p}$ set, consists of only executing the generator test case p and then executing the rest of the sequence, while checking the state constraints of every state that the class variable arrives at.

Statement: If a test-case p fails to place the class variable in a specific state, then the entire $\tilde{p}$ set will also fail.

The testing process is very time consuming. One of the goals of this research, is to reduce the time spent on testing without sacrificing the quality of testing. The above two statements can save a great deal of effort and time to the tester.

6.4 Proposing a Test-Case Generating Methodology

In this Section, we make all the necessary definitions that will assist us in proposing a new methodology for generating sufficient test-cases. The goal of this new methodology is to reduce the amount of testing without sacrificing the testing quality.

As stated in previous sections, a test-case (*legal trace*) may be generated from a *conditioned finite state machine* by traversing a path that originates from the initial state. To be able to create all the necessary test-cases, a special kind of acyclic graph called *TCG graph* (Test Case Generation graph) will be defined. This graph has many similarities with a general tree. Let us recall the definition of a general tree:

Definition: A general tree is a set of nodes that is either empty or has a designated node, called root, from which zero or more subtrees (disjoint set of nodes) descend Each subtree itself satisfies the definition of a tree.

Let us now introduce the nodes of the *TCG graph*:
Definition: Given a *conditioned finite state machine* $M_1 = (S, \Sigma_1, B_1, \delta_1, i, F)$, a

TCG quadruple $t_1 =< s_1, d_1, Op_1, c_1 >$, is defined to represent a transition in M_1, where $s_1, d_1 \in S$, $Op_1 \in \Sigma_1$, and $c_1 \in B_1$. s_1 and d_1 are the source and destination states of transition t_1. c_1 is the boolean condition which must be satisfied in order for t_1 to be traversed. Op_1 is the member function which is associated with the transition t_1.

Statement: A test sequence can now be thought as a sequence of *TCG quadruples*. In any sequence of *TCG quadruples*, for any two consecutive quadruples T_1 and T_2, where

$T_1 =< s_1, Op_1, c_1, d_1 >$ and

$T_2 =< s_2, Op_2, c_2, d_2 >$, we have:

$d_1 = s_2$.

Here, the source state of the second quadruple should be equal to the destination state of the first quadruple. That is, the state constraints of d_1 should match the state constraints of s_2.

Definition: A node that consists of a *TCG quadruple* t_1 and a set of values q_1, is called a *TCG node*. The set q_1, contains the values of the variables in the boolean condition of t_1.

Definition: A *TCG graph* (Test Case Generation graph) is an acyclic graph which is derived from a *conditioned finite state machine* and behaves as a general tree. The nodes of the graph, are the above defined *TCG nodes*. There are three kinds of *TCG nodes*: the *root*, the *leaf* and the *non-leaf* nodes. The *root* node, which is on the top of the tree, may have many children and no parent. The last node of a branch is called a *leaf* node if and only if, that node exists twice in the path from the root down to this node (once at the end of the path and once at some point between the root and the end of the path). A *leaf* node may have only one parent and no children. A *non-leaf* node is any node that is not a *leaf*. It may have one parent and many children.

Statement: A *TCG graph* path is *complete* when it terminates with a *leaf* node. A *TCG graph* is *complete* when all paths from the root are *complete*.

A *TCG graph* is pictorially represented by a general tree. Figure 9 shows an example of a *TCG graph* which is not *complete*. It is clear that the path a-b-a is

complete (since "a" appears twice in the path) and therefore the third node is a *leaf* node. The path a-b-b is complete (since "b" appears twice in the path) and therefore the third node is a *leaf node*. The path a-c-b is not complete (because "b" appears only once in the path) and therefore all nodes of that path are *non-leaf*. Also a *TCG graph* represents the total set of test-cases that can be derived from the respective *conditioned finite state machine*. Each path represents a particular set of test-cases. For instance, the a-b path represents the set that is generated from the test case [a;b]. In order to generate the rest of the test cases that exist in that set, only the path a-b and all its descendants have to be followed.

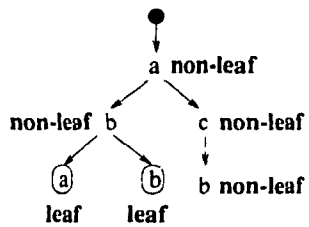


Figure 9: A TCG Graph

A *TCG graph*, as seen already, seems to be a general tree with finite *height*. Since any finite state machine may have loops, the corresponding *TCG graph* will have infinite *height*. This is expected since the testing process could be endless and therefore the set of possible test-cases is infinite. Many of the test-cases in that set, as it will be seen in the next section, are redundant. Our methodology eliminates redundant test-cases. That is the purpose of incorporating the notion of *leaf* nodes in a *TCG graph*.

A *leaf* node is the last node in any *complete TCG graph* path. The purpose of having *leaf* nodes is to reduce the height of the infinite-height *TCG graph*. For instance, when having a machine like the one in Figure 10, the actual graph would

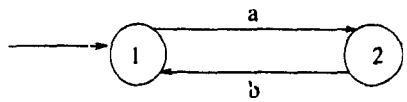


Figure 10: A FSM with a Loop

look like the one in Figure 11. In that figure, part of the actual tree is showed, since the actual tree has an infinite height. Therefore, a *TCG graph* with no *leaf* nodes

produces an infinite amount of test sequences. We can also note that the first few test sequences that will be produced from the tree would be:

1. [a]
2. [a; b]
3. [a; b; a]
4. [a; b; a; b]
5. ....

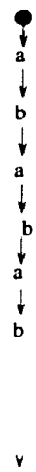


Figure 11: Infinite Test Case Tree

The sequence [a] will test the transition that has state "1" as source, state "2" as destination, input "a", and no boolean condition. Given that the first test sequence was tested successfully, the sequence [a; b] will test the transition that has state "2" as source, state "1" as destination, input "b", and no boolean condition. Given that the first two test sequences were tested successfully, the sequence [a; b; a] tests the transition that has state "1" as source, state "2" as destination, input "a", and no boolean condition. It is now obvious that the third sequence has identical testing objectives as the first one. Also, further analysis of this problem, shows that the fourth sequence has the same testing objectives as the second sequence, etc. By studying carefully the tree (Figure 11), it can be noted that the pattern "a; b" is repeated an infinite number of times. This is perfectly logical since the state machine contains a

loop with these two transitions. Therefore, we can conclude that of all the infinite number of test sequences that are generated by the actual tree, only two sequences are sufficient in order to test the class without reducing the reliability of testing.

Statement: The purpose of a *leaf* node in a *TCG graph* is to reduce the height of the tree by eliminating the repetition of certain parts of it.

As previously seen in the definition of the *TCG graph*, a *complete path* ends with a *leaf node* if and only if the last node of the path exists twice in the path. This means that whenever the path is traversed, the *leaf node* can be considered as a branch to the other occurrence of the node in the same path. In this way, the *TCG graph* is a tree that contains all the possible paths that can be traversed in a finite state machine, and its height is finite.

Basically, a *canonical* test-case is any sequence of operations derived from the respective FSM that does not contain the same transition more than once. Therefore, a *canonical* test-case may be produced by following any path in the FSM that does not contain a repetitive sequence of transitions.

Definition: We define a *Canonical* test-case to be a test-case which is generated by following any path on the *TCG graph*, starting from the *root* node and finishing at any *non-leaf* node without treating any *leaf* nodes as branches.

Definition: We define a *Valid* test-case to be a test-case which is generated by following any path on the *TCG graph*, starting from the *root* node and finishing at any *non-leaf* node with or without treating the *leaf* nodes as branches.

Therefore, given a *TCG graph*, the set of *Canonical* test-cases, *Can*, is a subset of the set of *Valid* test cases *Val*: Can $\subset$ Val. Furthermore, from the previous definition of *Valid* test-cases and the definition of a *leaf* node, we can conclude that the set of *Valid* test-cases *Val* is infinite.

During the process of constructing a *TCG graph*, many paths have to be followed in the finite state machine. The root node is always the first transition label in

the path. If there is more than one different starting transition label in the set of paths (say n number of starting labels), then n *TCG graphs* will be constructed. The following rules should be followed in order to construct a *TCG graph*

1. While constructing each path, visit states of the state machine by following a particular transition.

2. At any state in the FSM, create a child node for every outgoing link of the state, unless the same node already occurs in the path traversed from the *root* node to the current node. In that case, terminate the path by creating the child node as a *leaf* node. (see Figure 12).

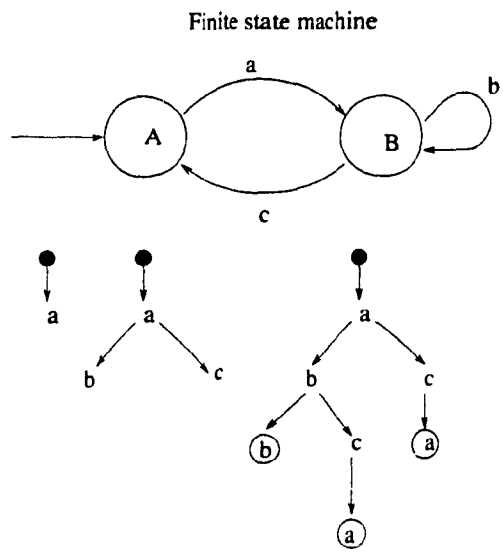


Figure 12: TCG derivation

A *TCG graph* has the property that, given any path in the graph, there is no two identical *non-leaf* nodes. Figure 12, shows an example of a state machine and its corresponding *TCG graph*.

Now that all the basic required definitions have been introduced, here is a proposal of a new test-case generating methodology. This methodology tries to minimize the number of test-cases that are required for testing a C++ class implementation without affecting the quality of testing. The method has the following steps:

1. **Step 1.** From a state machine M_1, create the set of *TCG quadruples* contained in M_1.

2. **Step 2.** Given the set of *TCG quadruples*, create the necessary *TCG graphs* by traversing many paths on M_1, until all the required *TCG graphs* are *complete*.

3. **Step 3.** Given all *TCG graphs* that were created in step 2, create all the *Canonical* test-cases by traversing the *TCG graphs* without treating the *leaf* nodes as branches.

6.5 Sufficiency in Testing Canonical Test Cases

The overall goal of testing is to provide confidence in the correctness of a program. The only way to guarantee absolute correctness of a program is to execute it on all possible test-cases, which is usually impossible. In the previous section the notion of a *TCG graph* was introduced. The two sets of test-cases, *Canonical* and *Valid*, that can be generated from it were also defined. It was also noted that the set of *Valid* test-cases, Val, is infinite, where the set of *Canonical* test-cases, Can, is finite, and that $Can \subset Val$. Furthermore, it was claimed that any test-cases that belong to Val but not to Can, are redundant. In this section, this claim will be proved.

Theorem: (Adequacy) Let Can denote the canonical test-cases generated by the FSM F constructed from the specification of a class C. Testing the implementation of class C using test-cases in Can is equivalent to testing the implementation of class C using all the test-cases in Val.

Proof:

Our hypothesis is that the implementation of class C has been tested for all test-cases from Can and the behavior of class C is correct. Let us choose α, any test-case, from $Val - Can$ (in other words, the set of test-cases that we want to prove redundant). We construct a set $\{x_1, x_2, ...x_k\}$, $k > 1$, of canonical test-cases corresponding to α. This will establish that testing with α is equivalent to testing with $\{x_1, x_2, ...x_k\}$.

Since α is *valid* and not *canonical*, we can write $\alpha = x_1\beta$, where x_1 is *canonical*, β is the rest of the test-case α after removing x_1, and $|\beta| \geq 1$. So, the leaf node v of x_1 can be used to generate β. This means that a particular transition has been

reached for the second time in the same path. Since this *leaf* node v also corresponds to a unique *non-leaf* node w in the same path, the unique prefix y of the path terminating at this *non-leaf* node w, satisfies the following properties:

- $y = prefix(x_1)$
- $|y| < |x_1|$
- $y \neq x_1$
- $Post(y) = Post(x_1)$
- $Post(y) = Pre(\beta)$
- $y\beta \in Val$

Let $\alpha_1 = y\beta$. Consequently, testing the implementation of class C with α is equivalent to testing it with x_1, followed by testing it with $y\beta$ (or just α_1). Due to the above properties, α_1 has a unique canonical prefix on the tree. Moreover, $|\alpha_1| < |\alpha|$. Hence, by repeating the above process on α_1 and its successively obtainable strings of smaller lengths, a set $\{x_1, x_2, ...x_k\}$ of canonical test-cases will be constructed. The process terminates due to the finiteness of $|\alpha|$.

6.6 Axiomatization of Black-Box Test-Case Adequacy

In the past, some research has been conducted in the context of test-case adequacy. The most important work is due to Weyuker [61]. In this work, Weyuker developed a general axiomatic theory of test data adequacy. In this section, it will be seen whether the test-case generating methodology that was proposed in this Chapter, agrees with that theory.

A program is adequately tested if it has been covered according to the selected criteria. These criteria are dependent on the form of testing that is performed. As we already know, there are two forms of testing: specification based (black-box) and program based (white-box). It has been already seen that the goal of black-box testing

is to determine whe her the program meets its specification. The goal of white-box testing is to inspect the source code as opposed to its specification.

Claim: Let the specification of a class implementation I_1 to be S_1. Let also the derived conditioned FSM to be M_1 and the associated TCG graph T_1. The set of *canonical* test-cases C_1 that are derived from T_1, is adequate to black-box test I_1.

1. **Applicability:** *"For every program, there exists a finite adequate test set."*
   As stated in a previous section, a TCG graph has finite depth. Therefore, the set C_1 of canonical test-cases is finite. Due to the *adequacy* theorem, C_1 is adequate to test I_1.

2. **Non-Exhaustive Applicability:** *"There is a program P and test set T such that P is adequately tested by T, and T is not an exhaustive test set."*
   In the previous section, it has been proved that the set of *canonical* test-cases is equivalent to the set of *Valid* test-cases. The set of *Valid* test-cases is exhaustive, while the set of *canonical* test-cases is finite and non-exhaustive.

3. **Monotonicity:** *"If T is adequate for P, and $T \subseteq T'$ then T' is adequate for P."*
   Let T be a set of test-cases such that $Can \subseteq T$. Testing a class implementation with T ensures that the class is being tested with at least the set Can. Therefore, since Can is adequate, T is adequate also.

4. **Inadequate Empty Set:** *"The empty set is not adequate for any program."*
   If the set Can can be empty, that would mean that the TCG graph would have no nodes (root, leaf, non-leaf). Furthermore, it implies that none of the states, of the associated FSM, are reachable from the initial state which implies that the CUT (class under test) does not provide any constructors. This will not occur in a C++ class. Therefore, the set of *canonical* test-cases cannot be empty and consequently, the empty set is inadequate to test any program.

5. **Antiextensionality:** *"There are programs P and Q such that $P \equiv Q$, T is adequate for P, but T is not adequate for Q."*

This axiom is not true in the case of a specification-based testing. If two programs P and Q have the same specification, then any black-box test set T that is adequate for P is also adequate for Q.

6. **General Multiple Change:** *"There are programs P and Q which are the same shape, and a test set T such that T is adequate for P, but T is not adequate for Q."*

   This axiom is only applicable to white-box testing since the notion of two programs having the same shape is only defined for the internal structure of a program implementation [61].

7. **Antidecomposition:** *"There exists a program P and a component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q."*

   This axiom is not applicable for class testing. It is intended for integration testing.

8. **Anticomposition:** *"There exist programs P and Q such that T is adequate for P and $P(T)$ is adequate for Q, but T is not adequate for $P;Q$."*

   This axiom is not applicable for class testing. It is intended for integration testing.

Therefore, the methodology that was proposed in this Chapter, constructs a test suite which satisfies four of the eight axioms proposed by Weyuker. The fifth axiom, was stated for white-box testing and it was modified for the sake of black-box testing. The sixth axiom is only applicable to white-box testing, while the last two are only applicable for integration testing. The satisfaction of the above axioms by the proposed methodology gives better confidence to the test suite produced by a *TCG graph*.

Although the test-cases derived from a *TCG graph* provide confidence in testing a class implementation, that does not imply that the testing process should be terminated. Formal specifications cannot express every requirement of the system. J.Wing in [63] states that a specifier should first decide what aspect of the system should specified, and then take a specification approach. For instance, a specification that is indented only for additional documentation would not be sufficient to be used for

constructing an adequate test suite. Therefore, a system should be tested also against its design and its implementation.

6.7 More Examples

In previous Chapters we studied how to derive a finite state machine given formal specifications written in Larch/C++. In this section we are going to illustrate this new test suite generating method with a few examples : the Stack, and the Bounded Stack classes.

6.7.1 Example 1: Stack

Stack LSL Analysis

$StackTrait(E, Stack)$: **trait**

includes $Integer$

introduces

$new :\rightarrow Stack$

$push : Stack, E \rightarrow Stack$

$pop : Stack \rightarrow Stack$

$isEmpty : Stack \rightarrow Bool$

$size : Stack \rightarrow Int$

$top : Stack \rightarrow E$

asserts

$Stack$ **generated by** $new, push$

$Stack$ **partitioned by** $top, pop, isEmpty$

$\forall s : Stack, e : E$

$top(push(s, e)) == e$

$pop(push(s, e)) == s$

$size(new) == 0$

$size(push(s, e)) == size(s) + 1$

$size(pop(s)) == size(s) - 1$

$isEmpty(new)$

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

$isEmpty(s) == size(s) = 0$

implies

converts $top, pop, isEmpty$

exempting $top(new), pop(new)$

Operator Type	Operator Name
Initializers	new
Partitioners	isEmpty
Alterators	push, pop
Examiners	size, top

Table 6: Classification of StackTrait Operations

- **size** operator:

  input domain : $q \in d\_empty \vee q \in d\_non\_empty$

  output domain : Not needed.

  output domain is not needed because *size* is not an alterator.

- **pop** operator:

  input domain : $q \in d\_non\_empty$

  output domain : $q \in d\_empty \vee q \in d\_non\_empty$

- **top** operator:

  input domain : $q \in d\_non\_empty$

  output domain : Not needed.

  output domain is not needed because *top* is not an alterator.

- **push** operator :

  input domain : $q \in d\_empty \vee q \in d\_non\_empty$

  output domain : $q \in d\_non\_empty$

  The output domain can be derived with a small proof using the axioms in the *asserts* clause.

Stack LCC Analysis

```
class Stack
{
imports StackTrait(int, Stack);
   Stack()
   {
      constructs self;
      ensures self' = new;
   }
   virtual void Push(int a)
   {
      modifies self;
      ensures self' = push(self^, a);
   }
   virtual int Pop()
   {
      requires !isEmpty(self^);
      modifies self;
      ensures self' = pop(self^) ∧ result = top(self^);
   }
   };
```

- **Stack()** (Constructor)
  pre-condition: $true$
  post-condition: $self' = new$
  Spec_OP: $self' \in d\_empty$
  simplified: $self' \in d\_empty$
  DNF: $self' \in d\_empty$

- **Push(a)**
  pre-condition: $true$
  post-condition: $self' = push(self^{\wedge}, a)$
  Spec_OP: $(self^{\wedge} \in d\_empty \vee self^{\wedge} \in d\_non\_empty) \wedge self' \in d\_non\_empty$

simplified:$(self^{`} \in d\_empty \vee self^{`} \in d\_non\_empty) \wedge self' \in d\_non\_empty$
DNF: $(self^{`} \in d\_empty \wedge self' \in d\_non\_empty) \vee (self^{`} \in d\_non\_empty \wedge self' \in d\_non\_empty)$

- **Pop()**
  pre-condition: $!isEmpty(self^{`})$
  post-condition: $self' = pop(self^{`}) \wedge result = top(self^{`})$
  Spec_OP: $self^{`} \in d\_non\_empty \wedge (self' \in d\_empty \vee self' \in d\_non\_empty) \wedge self^{`} \in d\_non\_empty$
  simplified: $self^{`} \in d\_non\_empty \wedge (self' \in d\_empty \vee self' \in d\_non\_empty)$
  DNF: $(self^{`} \in d\_non\_empty \wedge self' \in d\_empty) \vee (self^{`} \in d\_non\_empty \wedge self' \in d\_non\_empty)$

Stack FSM Derivation

After completing the LSL and LCC analysis the finite state machine can be derived. Figure 13 shows the derived conditioned finite state machine.

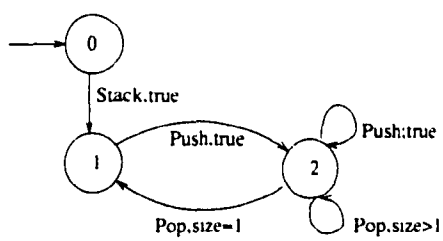


Figure 13: Conditioned FSM for Stack

Test-Case Generation

Let us now follow the proposed methodology.

Step 1. Derive all *TCG quadruples* from the finite state machine. For every transition in the original state machine that departs from state s_i ($i \geq 0 \wedge i \leq num\_trans$), arrives at state d_i, has label op_i, and boolean condition c_i, create a *TCG quadruple* $< s_i, op_i, c_i, d_i >$. For our ease of understanding this method we are going to denote every quadruple with a unique name.

Init_1 :$\rightarrow < 0, 1, true, Stack >$

Push_1 :→< 1, 2, *true*, *Push* >
Push_2 :→< 2, 2, *true*, *Push* >
Pop_2 :→< 2, 2, *size* > 1, *Pop* >
Pop_1 :→< 2, 1, *size* = 1, *Pop* >

Step 2. Given all the above *TCG quadruples*, the test-case derivation algorithm is now applied. The *TCG graph* will be constructed by completing the left most incomplete path at a time.

- Current state : 0.
- Observing the finite state machine, there is only one starting label. Therefore, only one *TCG graph* will be constructed with a root node "Init_1". Note that no variable value exists in that *TCG node* label since the boolean condition is *true*. Current state : 1.
- Since there is only one outgoing link from state 1, create a child to the current node called "Push_1". Current state : 2.

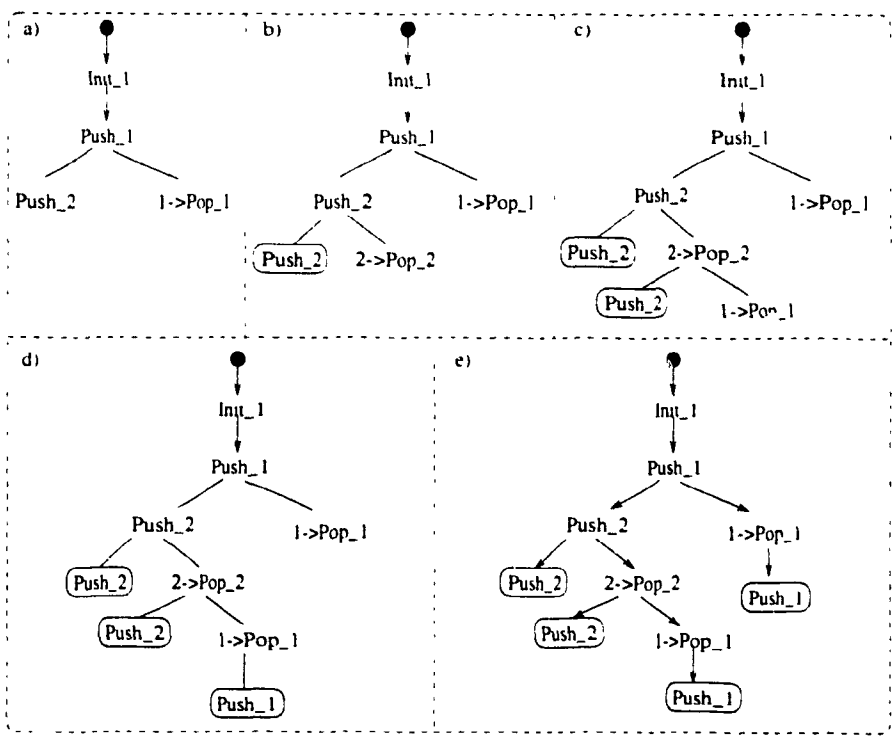


Figure 14: TCG Derivation for Stack

- In this case there are three outgoing links from state 2. Therefore, create three children called : "Push_2", "1→ Pop_1" (Pop operation when the size is equal to 1), and "1→Pop_2". Since the size of the stack at this point is: $size = 1$ the transition "Pop_2" is impossible to be followed. Therefore we create only the two children "Push_2" and "1→ Pop_1" (see Figure 14a). Continuing with "Push_2", the current state is :2.

- Again there are three outgoing links from this state. Therefore, create three children nodes: "Push_2", "2→Pop_1", and "2→Pop_2". Since at this stage the size of the stack is 2 the transition "2→Pop_1" is impossible to be followed. Therefore, create only the two children "Push_2" and "1→Pop_1". By traversing the path from the "Push_2" node up to the root of the graph it can be seen that the node "Push_2" appears twice. Therefore, this node is a *leaf* node (see Figure 14b). The leftmost incomplete path to be traversed is the [Init; Push_1; Push_2; Pop_2]. Current state:2

- "2→Pop_2" has three children :"Push_2", "1→ Pop_1", and "1→Pop_2", from which "1→Pop_2" is impossible to follow. Therefore the graph becomes as in Figure 14c. It is easy to see that the first child ("Push_2") is a *leaf* node and that the second ("1→Pop_1") is a *non-leaf* node. The leftmost incomplete path to be traversed is the [Init; Push_1; Push_2; Pop_2; Pop_1]. Current state: 1.

- "1→Pop_1" has only one child : "Push_1" which is clearly a *leaf* node (see Figure 14d). The leftmost incomplete path to be traversed is the [Init; Push_1; Pop_1]. Current state: 2.

- "1→Pop_1" has only one child : "Push_1" which is clearly a *leaf* node (see Figure 14e). All paths of the *TCG graph* are complete.

Step 3. The last step of the procedure is to traverse the *TCG graph* in every possible way. Following, are the test sequences derived from the *TCG graph*.

1. [Init_1]

2. [Init_1; Push_1]

3. [Init_1; Push_1; Push_2]

4. [Init_1; Push_1; Push_2; Pop_2]

5. [Init_1; Push_1; Push_2; Pop_2; Pop_1]

6. [Init_1; Push_1; Pop_1]

The above six test-cases are the *canonical* test-cases that are derived from the *TCG graph*. Testing the class implementation with the last two test-cases, cover the testing objectives of the first four test-cases. Therefore, the *minimal canonical test-cases* are :

- [Init_1; Push_1; Push_2; Pop_2; Pop_1]
- [Init_1; Push_1; Pop_1]

The following is a demonstration of the redundancy of any non-canonical test-case. Let the *Valid* test-case α to be :

α = [Init_1; Push_1; Pop_1; Push_1; Push_2]

Given that the above six *canonical* test-cases have been tested successfully, it will be shown that α is redundant. Let x_1 be the sixth canonical test-case and

β = [Push_1; Push_2] and

y = [Init_1] Then the following properties hold:

- $y = prefix(x_1)$
- $|y| < |x_1|$
- $y \neq x_1$
- $Post(y) = Post(x_1)$
- $Post(y) = Pre(\beta)$
- $y\beta \in Val$

Let, $\alpha_1 = y\beta$, then $|\alpha_1| < |\alpha|$ Consequently, testing the class implementation with α is equivalent to testing with the set of test-cases $\{x_1, \alpha_1\}$. Now we repeat the same process on α_1. Since α_1 is identical to the third *canonical test-case*, testing the class implementation with α is equivalent to testing with the third and sixth *canonical* test-cases.

6.7.2 Example 2: Bounded Stack

Bounded Stack LSL Analysis

$BStackTrait(E, BStack)$: **trait**

includes *Integer*

introduces

$new :\rightarrow BStack$

$push : BStack, E \rightarrow BStack$

$pop : BStack \rightarrow BStack$

$isempty : BStack \rightarrow Bool$

$isfull : BStack \rightarrow Bool$

$maxEl :\rightarrow Int$

$size : BStack \rightarrow Int$

$top : BStack \rightarrow E$

asserts

$BStack$ **generated by** $new, push$

$BStack$ **partitioned by** $top, pop, isempty, isfull$

$\forall\ s : BStack, e : E$

$maxEl > 0$

$top(push(s, e)) == e$

$pop(push(s, e)) == s$

$size(new) == 0$

$size(push(s, e)) == size(s) + 1$

$size(pop(s)) == size(s) - 1$

$isempty(new)$

$\neg isempty(push(s, e))$

$\neg isfull(new)$

$\neg isfull(pop(s))$

$isfull(s) == size(s) = maxEl$

$isempty(s) == size(s) = 0$

$isfull(s) \Rightarrow \neg isempty(s)$

$isempty(s) \Rightarrow \neg isfull(s)$

implies

converts $top, pop, isempty, isfull$

exempting $top(new), pop(new)$

Operator Type	Operator Name
Initializers	new
Partitioners	isempty
Alterators	push, pop
Examiners	size, top
Constants	maxEl

Table 7: Classification of BStackTrait Operations

- **size** operator:
  input domain : $q \in d\_empty \vee q \in d\_non\_empty/full \vee q \in d\_full$
  output domain : Not needed.
  output domain is not needed because *size* is not an alterator.

- **pop** operator:
  input domain : $q \in d\_non\_empty/full \vee q \in d\_full$
  output domain : $q \in d\_empty \vee q \in d\_non\_empty/full$

- **top** operator:
  input domain : $q \in d\_non\_empty/full \vee q \in d\_full$
  output domain : Not needed.
  output domain is not needed because *top* is not an alterator.

- **push** operator :
  input domain : $q \in d\_empty \vee q \in d\_non\_empty/full$
  output domain : $q \in d\_non\_empty/full \vee q \in d\_full$
  The output domain can be derived with a small proof using the axioms in the assertions clause.

Bounded Stack LCC Analysis

class BoundedStack

```
{
uses BStackTrait(int, BoundedStack);
    BoundedStack()
    {
        constructs self;
        ensures self' = new;
    }
    void Push(int a)
    {
        requires !isfull(self^);
        modifies self;
        ensures self' = push(self^, a);
    }
    int Pop()
    {
        requires !isEmpty(self^);
        modifies self;
        ensures self' = pop(self^) ∧ result = top(self^);
    }
};
```

- **BoundedStack()** (Constructor)

  pre-condition: $true$

  post-condition: $self' = new$

  Spec_OP. $self' \in d\_empty$

  simplified:$self' \in d\_empty$

  DNF: $self' \in d\_empty$

- **Push(a)**

  pre-condition: $!isfull(self^{\wedge})$

  post-condition: $self' = push(self^{\wedge}, a)$

  Spec_OP: $(self^{\wedge} \in d\_empty \vee self^{\wedge} \in d\_non\_empty/full) \wedge (self' \in d\_non\_empty/full \vee self' \in d\_full)$

  simplified:$(self^{\wedge} \in d\_empty \vee self^{\wedge} \in d\_non\_empty/full) \wedge (self' \in d\_non\_empty/full \vee$

$self' \in d\_full)$

DNF: $(self \in d\_empty \wedge self' \in d\_non\_empty/full) \vee (self \in d\_non\_empty \wedge self' \in d\_non\_empty/full) \vee (self \in d\_empty \wedge self' \in d\_full) \vee (self \in d\_non\_empty/full \wedge self' \in d\_full)$

- **Pop()**

  pre-condition: $!isempty(self)$

  post-condition: $self' = pop(self) \wedge result = top(self)$

  Spec_OP: $(self \in d\_non\_empty/full \vee self \in d\_full) \wedge (self' \in d\_empty \vee self' \in d\_non\_empty/full) \wedge self \in d\_non\_empty$

  simplified: $(self \in d\_non\_empty/full \vee self \in d\_full) \wedge (self' \in d\_empty \vee self' \in d\_non\_empty/full) \wedge self \in d\_non\_empty$

  DNF: $(self \in d\_non\_empty/full \wedge self' \in d\_empty) \vee (self \in d\_non\_empty/full \wedge self' \in d\_non\_empty/full) \vee (self \in d\_full \wedge self' \in d\_non\_empty/full) \vee (self \in d\_full \wedge self' \in d\_empty)$

Bounded Stack FSM Derivation

After completing the LSL and LCC analysis, the finite state machine can be derived. Figure 15 shows the derived conditioned finite state machine.

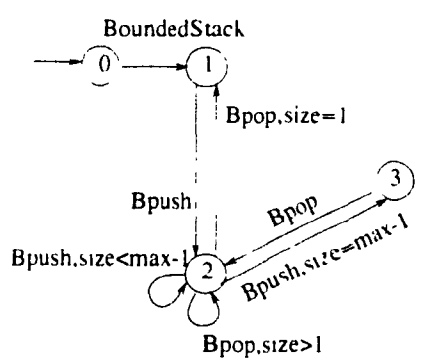


Figure 15: Conditioned FSM for Bounded Stack

Test-Case Generation

Given the bounded stack class specification that was converted into a finite state machine, the proposed methodology is used in order to derive the necessary test-cases. As in the previous example, the steps of the proposed method are followed. Since

the state machine represents a bounded stack, the maximum number of elements, $MaxEl$, that can exist in the stack will be given a value. For the sake of this example four will be the given value (the maximum size of the stack is 4).

Step 1. This step produces the following *TCG quadruples*:

Binit_1 :→< 0, 1, *true*, *BoundedStack* >
Bpush_1 :→< 1, 2, *true*, *Bpush* >
Bpush_2 :→< 2, 2, *size* < *max* − 1, *Bpush* >
Bpop_2 :→< 2, 2, *size* > 1, *Bpop* >
Bpop_1 :→< 2, 1, *size* = 1, *Bpop* >
Bpush_3 :→< 2, 3, *size* = *max* − 1, *Bpush* >
Bpop_3 :→< 3, 2, *true*, *Bpop* >

Step 2. Following the same procedure as in the first example, we derive the following *TCG graph* (Figure 16).

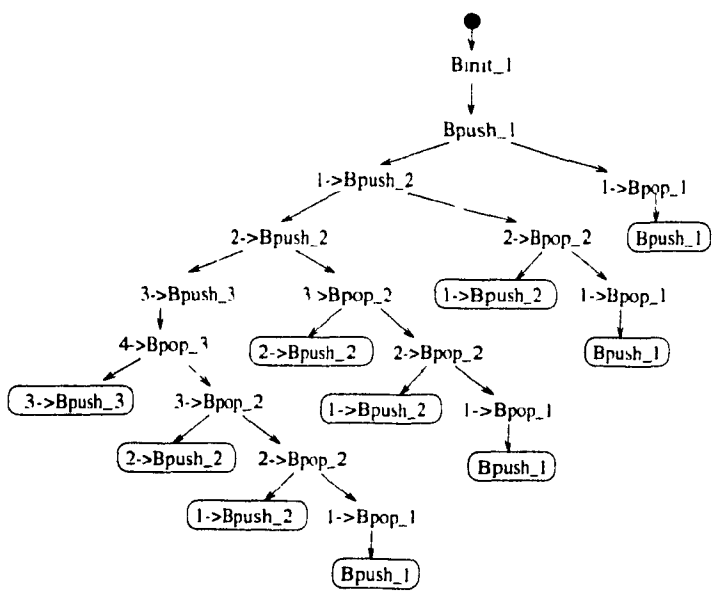


Figure 16: TCG graph for Bounded Stack

Step 3. Traversing the above derived *TCG graph*, the following set of test sequences is obtained:

1. [Binit_1]
2. [Binit_1; Bpush_1]
3. [Binit_1; Bpush_1; Bpop_1]
4. [Binit_1; Bpush_1; Bpush_2]
5. [Binit_1; Bpush_1; Bpush_2; Bpop_2]
6. [Binit_1; Bpush_1; Bpush_2; Bpop_2; Bpop_1]
7. [Binit_1; Bpush_1; Bpush_2; Bpush_2]
8. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpop_2]
9. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpop_2; Bpop_2]
10. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpop_2; Bpop_2; Bpop_1]
11. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpush_3]
12. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpush_3; Bpop_3]
13. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpush_3; Bpop_3; Bpop_2]
14. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpush_3; Bpop_3; Bpop_2; Bpop_2]
15. [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpush_3; Bpop_3; Bpop_2; Bpop_2; Bpop_1]

The above fifteen test-cases are the *canonical* test-cases that are derived from the *TCG graph*. The testing objectives of all the above test-cases are covered by the following set of *minimal canonical test-cases*:

- [Binit_1; Bpush_1; Bpop_1]
- [Binit_1; Bpush_1; Bpush_2; Bpop_2; Bpop_1]
- [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpop_2; Bpop_2; Bpop_1]
- [Binit_1; Bpush_1; Bpush_2; Bpush_2; Bpush_3; Bpop_3; Bpop_2; Bpop_2. Bpop_1]

6.8 Test Coverage

Test effectiveness usually depends on the coverage strategy that was used to construct the test suite [49]. The most commonly used coverage strategies are the *state*, *transition*, *switch* and *exhaustive coverage*. Each strategy provides some degree of error detection and carries some cost based on the size of the suite that must be constructed. Figure 17 illustrates the hierarchy of these strategies according to the degree of error detection and size of the repective test suite. Strategies at the top exhibit higher degree of error detection, while strategies at the bottom require test suites of smaller sizes.

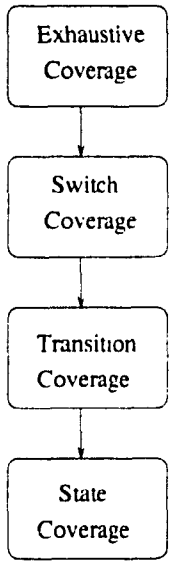


Figure 17: Test Cover Strategy Hierarchy

Let us briefly describe the different types of errors that can be detected from from the above stated coverage strategies:

- missing state
- extra state
- missing transition
- corruption

Almost all of the above errors are self explanatory. `Corruption` is the error that occurs when a particular function executes correctly but it has a side effect that modifies the state of the object in such a way that some subsequent operations perform incorrectly. Les us now briefly describe each strategy:

State Coverage: The goal of this strategy is to place the object under test in every state in the FSM at least once. A test suite constructed following this strategy will be able to detect the following types of errors:
`Any missing states` will be detected since every state is scheduled to be visited by the test suite. Also, some `extra states`, some `errors in transitions`, and some `instances of corruption` may be detected but there is no systematic coverage of these errors.

Transition Coverage: The goal of this strategy is to traverse every transition in the FSM at least once. If an FSM is not disconnected, traversing all transitions implies the *state coverage* strategy. A test suite constructed following this strategy will be able to detect the following types of errors:
`Any missing states` and `any missing transitions` will be detected since every transition (and every state) is scheduled to be traversed by the test suite at least once. Also, an improved detection of some `extra states`, and some `instances of corruption` may be performed but there is no systematic coverage of these errors.

The FSM construction process described in Chapters 4 and 5, ensures that an FSM cannot be disconnected. A test suite generated with the *TCG* technique satisfies *transition coverage* because it traverses every transition in the finite state machine. Since *transition coverage* implies *state coverage* it also satisfies *state coverage*.

Switch Coverage: The goal of this strategy is to traverse all possible sequences of transitions in the FSM at least once. The sequence of transitions may have a predefined length. For instance, when the sequence length is 3, the coverage is called 3-switch. 1-switch is equivalent to *transition coverage*. Any n-switch coverage with $n > 1$ implies the *transition coverage* strategy. A test suite constructed following this strategy will be able to detect the following types of errors:
`Any missing states` or `transitions` will be detected since the strategy is at least

equivalent to the *transition coverage*. Also, a much more improved detection of `extra states` and `instances of corruption` can be performed. Even though this strategy provides a systematic coverage of these errors, it is not able to produce a test suite that will detect all faults.

The *TCG graph* construction and traversal ensures *switch coverage*. The degree of the *switch cover* is not constant. It depends on the *TCG graph* that was derived from the corresponding finite state machine. The depth of the *TCG graph* would be the degree of the *switch cover*. Therefore, the test suite for the *BoundedStack* example is a *9-switch cover* and the test suite for the *Stack* example is *5-switch cover*.

Exhaustive Coverage: The goal of this strategy is to traverse every possible path in a finite state machine. This strategy becomes an infinite process when there is at least one loop of transitions in the FSM. Therefore, it should be able to detect all errors. It is the most expensive strategy and due to the infinite number of test-cases it is infeasible.

A test suite generated by a *TCG graph* is not an exhaustive test suite. The *adequacy* theorem states that any test-case that is not part of the test suite is redundant and therefore, the test suite is "equivalent" to an exhaustive test suite.

6.9 Test-Case Execution

Dick and Faivre [20], describe a methodology that derives a finite state machine from VDM specifications. This methodology is similar to the state machine derivation from Larch/C++ specifications methodology. Additionally to the technique, they provide a few guidelines that can be used during the execution of the test-cases in order to detect the maximum amount of faults with one test-case. Modifying these guidelines, an efficient test-case execution process is obtained.

Figure 18 shows a flowchart which describes the guidelines in a form of an algorithm:

1. Construct the *TCG* and produce the *canonical test-cases*.

2. For each test-case choose test data values which satisfy the constraints. That is, the values of the arguments of the member functions should satisfy the function pre-condition.

3. Perform the first (next) test. and check the result against the specification. That is. the result of the invoked member function should satisfy the function post-condition.

   (a) If the test is successful. then

      i. If the system is in the expected state for the currently selected sequence. continue with the next operation in the test sequence. That is. if the state of the system satisfies the constraints of the expected state in the FSM. continue with the next member function invocation in the same test sequence.

      ii. If the system state does not satisfy the constraints of the expected state of the FSM, the test has failed and therefore. the process continues from step 3 for another test-case.

   (b) Otherwise. if the test is unsuccessful. check the after-state of the system

      i. If the system is in the correct state, continue with the next operation in the test sequence

      ii. If the system is not in the correct state. then the test has failed completely. Continue with another test sequence.

4. Repeat for another test sequence.

Remark: We assume that a specification is complete according to the completeness criteria [17. 60]. Hence. even if a test is unsuccessful. the system cannot enter into a state which is not in the FSM.

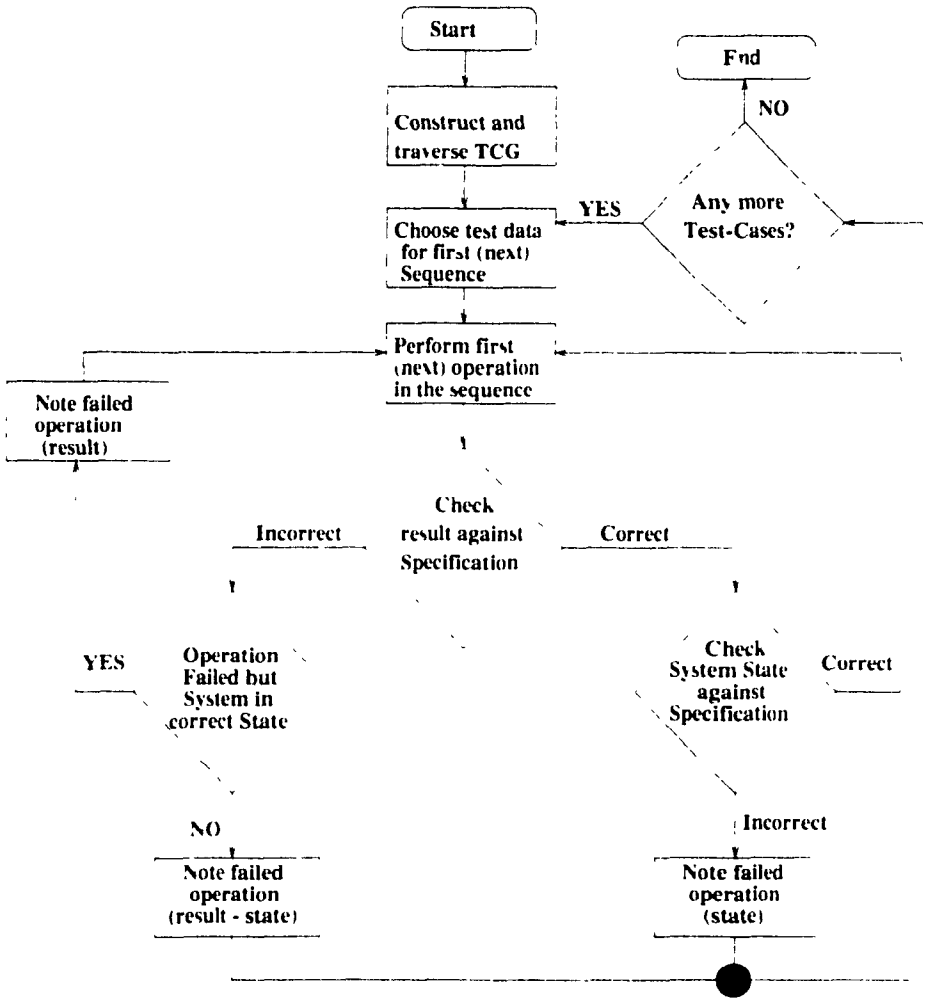


Figure 18: Test Case Execution Flowchart

Chapter 7

Supertype - Subtype Relationship of Finite State Machines

One of the most important characteristics of C++ and of all Object-Oriented languages is the *Inheritance* of classes. Larch/C++ follows C++ very closely and provides mechanisms of specifying inherited class interfaces. As already shown in a previous Chapter, Larch/C++ allows the specifier to reuse the base class specifications. In particular, a derived class (subclass) inherits its base class' specifications by inheriting the data member declarations and virtual member function specifications from the base class (superclass). *Specification inheritance* is the mechanism, provided by Larch/C++, which allows the user to specify a derived class by adding additional properties without re-specifying all properties stated in the base class specification. In this Chapter, we discuss how the relationship between sub-type and super-type specifications affects the relationship between FSMs derived from the specifications of the sub-type and super-type classes.

7.1 Informal Definition of Behavioral Sub-typing

A substantial amount of work has been conducted lately in the field of *Behavioral Sub-typing*. The most important ones are due to Liskov and Wing [44, 43], America [5], and Leavens [38, 18]. Although all of them are very close to each other, they also contain semantic differences. Larch/C++ follows Leavens' approach.

Let two Larch/C++ class interface specifications be called I_1 and I_2. I_1 is the specification of the C++ class C_1, and I_2 the specification of the C++ class C_2. Also, let m be a member function in C_1 which is inherited by C_2. Its interface specification in I_1 would be:

requires Pre_{super}:

modifies $ModList_{super}$:

ensures $Post_{super}$:

where Pre_{super}, $Post_{super}$ are assertions using terms defined in Tr_1 LSL trait, and $ModList_{super}$ is the list of objects that may be modified by m. Also the inherited function's interface specification in I_2 would be:

requires Pre_{sub}:

modifies $ModList_{sub}$:

ensures $Post_{sub}$:

where Pre_{sub}, $Post_{sub}$ are assertions using terms defined in Tr_2 LSL trait, and $ModList_{sub}$ is the list of objects that may be modified by m. In order for C_2 to be a behavioral sub-type of C_1 the following implications must be true:

1. $Pre_{super} \Rightarrow Pre_{sub}$

2. $Post_{sub} \Rightarrow Post_{super}$

These requirements ensure that there may be used an object of class C_2 in place of an object of class C_1. Therefore, when a message m is sent to an object of class C_2, using it as an object of class C_1, and given that Pre_{super} holds, by implication (1) we can conclude that Pre_{sub} will also hold. After the execution and termination of the method m, it follows that the post-condition $Post_{sub}$ will be true, and by implication (2) we can conclude that $Post_{super}$ also holds as required by the specification I_1 for an object of class C_1.

Although these requirements seem, at first, to be a satisfiable and an adequate definition of a behavioral sub-type, they fail to take into account that I_1 and I_2 might be expressed using different *distinguished sorts* and different *abstract values*. This creates a problem when abstract values of the sub-type are used in place of abstract values of the super-type. America [5] proposes a solution to this problem by requiring a mapping function that maps the abstract values of the sub-type, to the abstract

values of the super-type. This mapping function o America calls the *coercion function*:

$$o : S \rightarrow T$$

where S is the set of abstract values of the sub-type, and T is the set of abstract values of the super-type.

In the context of Larch/C++, Leavens uses the notion of the *coercion function* and names it *simulation function*. The *simulation function* should have the property that, it commutes with all the trait functions that take the super-type as argument. Technically, this property makes the *simulation function* a homomorphism on the abstract values and hence, is called the *homomorphism property*. The *homomorphism property* allows Larch/C++ to define all the trait functions that take as argument abstract values of the super-type, making these trait functions applicable and meaningful to the abstract values of the sub-type. Larch/C++ provides a special clause for the simulation function. This clause is called the *simulation clause* and it allows the specifier to state the *simulation function* with which the sub-type's abstract values are mapped to the super-type's abstract values. It also allows the specifier to state whether the derived class is a *weak* or a *strong behavioral sub-type*.

In a *strong behavioral sub-type*, the history constraint of the super-type must be satisfied by all of the sub-type's member functions. That means that a *strong behavioral sub-type* inherits its super-type's history constraint. In a *weak behavioral sub-type*, only the virtual member functions that are inherited from the super-type must satisfy the super-type's history constraint. The use of *weak behavioral sub-typing* in Larch/C++, is currently under research. In the next section, the formal definition of specification inheritance will be shown as defined in [18].

7.2 Formal Semantics for Specification Inheritance

As seen in the previous section, Larch/C++ creators provide the *simulation clause* in order to assist the user in specifying inherited class interfaces without having to re-specify inherited functions. In this section the formal semantics of specification inheritance and the two types of behavioral sub-typing, as defined in [18], will be

introduced.

Behavioral sub-typing in Larch/C++, is defined using syntactic and semantic constraints. The former are used to ensure that an expression of a sub-type can be used in place of an expression of its super-type without any type error. The latter are used to ensure that when a sub-type object is used in place of a super-type object, no unexpected behavior is produced.

Let us now see how specification inheritance is formally defined in Larch/C++ by its creators. The notation used below is as follows. The set of all super-types of a class S is given by $Sups(S)$. The set of all member functions of a class T is given by $meths(T)$. $added\_I_S$, $added\_pre_S(m)$, and $added\_post_S(m)$ are the predicates that are added in the specification of the sub-type, for invariant of S, and the pre and post conditions of a method m in S respectively.

Specification Inheritance: Let S be a strong behavioral sub-type of all the classes in $Sups(S)$, and $c_{U\_V} : U \leq V$, be the specified family of simulation functions. The completed specification of S is:
Invariant:

$$I_S = added\_I_S \wedge \left(\bigwedge_{T \in Sups(S)} I_T(c_{S\_T}) \right)$$

Pre Condition: For all virtual member functions m of S
$pre_S(m, self, \vec{x}) = added\_pre_S(m, self, \vec{x})$

$$\vee \left(\bigvee_{\substack{T \in Sups(S), \\ m \in meths(T)}} pre_T(m, c_{S\_T}(self), \vec{x}) \right)$$

Post Condition: For all common virtual member functions m of S

$$
\begin{aligned}
& post_S(m, self, self', \vec{x}, \vec{x}', result') \\
& = (added\_pre_S(m, self, \vec{x}) \Rightarrow added\_post_S(m, self, self', \vec{x}, \vec{x}' result')) \\
& \wedge \left(\begin{array}{ll} & pre_T(m, c_{S \to T}(self), \vec{x}) \\ \bigwedge & \Rightarrow (post_T(m, c_{S \to T}(self), c_{S \to T}(self'), \\ T \in Sups(S), & \vec{x}, \vec{x}', c_{V_r \to V_r}(result'))) \\ m \in meths(T) & \end{array} \right)
\end{aligned}
$$

The above formal definition of specification inheritance applies only to virtual member functions of public subclasses. In cases where non-public subclasses or non-virtual member functions are specified, specification inheritance cannot be applied [40].

7.3 Super-type/Sub-type FSM Relationship

In the previous two sections we studied the work that has been done regarding *specification inheritance* in the context of Larch/C++. In order to specify the behavior of the sub-type, a *simulation function* is used, which maps abstract values of the sub-type to the abstract values of the super-type, and reuse the specification of the super-type. In this section, the reused specification of a sub-type will be used in order to derive a finite state machine for it. Finally, the relationship between the super-type's state machine and the sub-type's state machine will also be studied.

In the process of LSL analysis, not many modifications have to take place from the original methodology. As was studied in Chapter 3, the *simulation function* is an LSL operation. The only addition would be the new kind of operation in the classification of operations. The *simulator* is the *simulation function* which takes as arguments the sort of the sub-type class and evaluates into the sort of the super-type class. For this operation, the input domains are determined from the signature of the operation and the exempting clause. Also, the mapping of the sub-type domains to the super-type domains is determined from the axioms that define the operation. Using this mapping, a constraint can be derived that will be used in LCC analysis. Since this constraint is derived through the *simulation function*, we will call it the *simulation constraint*.

For example, let an object S that takes abstract values from $A_1 \cup A_2$, and the following mapping be derived from the simulation function TO_Sup:

$$A_1 \rightarrow B_1$$

$$A_2 \rightarrow B_2$$

The *simulation constraint* would be:

$$(S \in A_1 \wedge TO\_Sup(S) \in B_1) \vee$$

$$(S \in A_2 \wedge TO\_Sup(S) \in B_2)$$

The *simulation constraint* will be used in the LCC analysis to eliminate all impossible cases, of the $Spec\_OP$ (in DNF).

When specifying a derived class, a lot of Larch/C++ syntactic sugar is used for the ease of the specification process. To obtain the actual specification of the derived class, the specification must be "de-sugared" as shown in Larch/C++ manual [40]. This "de-sugaring" is motivated from the definition of specification inheritance that was studied in the previous section. For instance, in a case where a sub-type S has only one public super-type T, and a virtual member function M of which signature and behavior are defined in the super-type's specification, M is inherited in the sub-type S. To determine the specification of the member function M both specifications of M in S and T are considered. The "de-sugaring" of the pre-condition, is done by taking the disjunction of the pre-condition of M in S, Pre_S, and the pre-condition of M in T, Pre_T: $Pre_S \vee Pre_T$. The "de-sugaring" of the post-condition, is done by taking the conjunction of the following two implications :

$$Pre_S \Rightarrow Post_S$$

$$Pre_T \Rightarrow Post_T$$

After reducing the above into DNF the following expression is obtained:

$$(Pre_T \wedge Post_T \wedge \neg Pre_S) \vee (Pre_T \wedge Post_T \wedge Post_S) \vee$$

$$(Pre_S \wedge Post_S \wedge \neg Pre_T) \vee (Pre_S \wedge Post_S \wedge Post_T)$$

Pre_T and $Post_T$ are expressed with terms in the context of the super-type T with the assistance of the simulation function. The sub-type's derived FSM will have states that are refinements of the states of the super-type's FSM. Since for any inherited member function, the pre-condition is a disjunction of the super-type's pre-condition and the sub-type's added pre-condition, the "de-sugared" pre-condition will ensure that the pre-states of all member functions in the super-type are present in the sub-type's FSM because the super-type's pre-condition is weakened. In other words, the

FSM of the sub-type is a hierarchical FSM.

Furthermore, the post-condition of any sub-type's member function is the super-type's post-condition but strengthened with the added post-condition. This makes any post-state in the super-type's FSM to get decomposed in a number of sub-states in the context of the sub-type's FSM. These sub-states have as state constraints the assertions that are added to the sub-type's member function post-condition. An important property of a sub-type object is that at any state, it may behave as a super-type object. To maintain this property preserved in the derived FSM, the following rules for creating an FSM are applicable:

1. For any transition in the super-type's FSM, a corresponding transition should appear in the sub-type's FSM.

2. For any state in the super-type's FSM, a corresponding state should appear in the sub-type's FSM.

3. For any case where a new state is to be added to the super-type's FSM in order to derive an FSM for the sub-type, this new state must be a sub-state of one of the super-type's states.

In the process of LCC analysis, not many modifications have to be done from the original methodology. All interface member functions have to be analyzed by extracting the Spec_OP from the pre-condition, the post-condition, the invariant and the history constraint of the "de-sugared" interface specification. In the case of the sub-type, the Spec_OP would contain *basic distinguishable domains* of the sub-type, as well as of the super-type. *Spec_OP* terms that relate to a *basic distinguishable domain* of the super-type specify constraints of the particular state in the context of the super-type. *Spec_OP* terms that relate to a *basic distinguishable domain* of the sub-type specify constraints of the particular state in the context of the sub-type. If for a particular member function, in the context of the sub-type, it is obtained that the pre or post-state must satisfy a constraint of the form $C_T \wedge C_S$, where C_T and C_S are predicates containing terms that take abstract values from the super-type's and sub-type's set of *basic distinguishable domains* respectively, then the derived FSM must include a state with C_T as a state constraint. This state must contain a sub-state with C_S as a state constraint. The derived transition for that member function

would depart from, or arrive at, the sub-state C_5 and not only from the state C_7. After the derivation of the *Spec_OP* for a particular function and the conversion to DNF, the mapping, derived from the *simulation function* in the LSL analysis, will be used in order to eliminate impossible cases in the *Spec_OP*. In the following section, the LSL and LCC analysis of a subtype will be illustrated.

7.4 Examples

Let us consider, once again, the stack example. In a previous Chapter, the two finite state machines were created from the two classes *Stack* and *BoundedStack*. These two classes were not inheriting each other. In this section, the two classes are redefined so that one inherits the other. The class *BoundedStack* is a behavioral sub-type of *Stack*. The two finite state machines will be derived, and their relationship will be shown. The interface specification of the class Stack is the same, while the BoundedStack uses the BSub_BStackTrait LSL trait defined below:

BSub_BStackTrait(E, BStack) : **trait**

includes *Integer, StackTrait(E, BStack), StackTrait*

introduces

$TOStack : BStack \rightarrow Stack$

$full :\rightarrow BStack$

$isfull : BStack \rightarrow Bool$

$maxEl :\rightarrow Int$

asserts

$\forall\ bs : BStack, e : E$

$maxEl > 0$

$TOStack(new) == new$

$TOStack(full) == push(TOStack(pop(full)), top(full))$

$TOStack(push(bs, e)) == push(TOStack(bs), e)$

$TOStack(pop(bs)) == pop(TOStack(bs))$

$size(full) == maxEl$

$\neg isfull(new)$

$\neg isfull(pop(bs))$

$isfull(full)$

$isfull(bs) == size(bs) = maxEl$

$isfull(bs) \Rightarrow \neg isEmpty(bs)$

$isEmpty(bs) \Rightarrow \neg isfull(bs)$

implies

converts *full*

exempting

\ forall e:E

$push(full, e)$

Let us now proceed with the LSL analysis of the above trait. Table 8 shows the classification of the operations that take as arguments or evaluate the *distinguished sort BStack*. Next, the partitioners are applied on the initializers. By the axioms, it

Operator Type	Operator Name
Initializers	new, full
Partitioners	isEmpty, isfull
Simulators	TOStack
Alterators	push, pop
Examiners	size, top
Constants	maxEl

Table 8: Classification of BSub_BStackTrait Operations

is concluded that a bounded stack can be *empty* but not *full*, not *empty* and not *full*, *full* and not *empty*. A bounded stack can never be *empty* and *full* because maxEl > 0. Therefore, there are three basic distinguishable domains: *d_emp*, *d_n_emp_n_full*, and *d_full*.

- **push** operator:
  input domain : $s \in d\_emp \vee s \in d\_n\_emp\_n\_full$
  output domain : $s \in d\_n\_emp\_n\_full \vee s \in d\_full$

- **pop** operator:
  input domain : $s \in d\_n\_emp\_n\_full \vee s \in d\_full$
  output domain : $s \in d\_emp \vee s \in d\_n\_emp\_n\_full$

- **size** operator:

  input domain : $s \in d\_emp \vee s \in d\_n\_emp\_n\_full \vee s \in d\_full$

  output domain : Not needed.

  output domain is not needed because *size* is not an alterator.

- **top** operator:

  input domain : $s \in d\_emp \vee s \in d\_n\_emp\_n\_full \vee s \in d\_full$

  output domain : Not needed.

  output domain is not needed because *top* is not an alterator.

- **TOStack** operator:

  input domain : $s \in d\_emp \vee s \in d\_n\_emp\_n\_full \vee s \in d\_full$

  output domain : Not needed.

  output domain is not needed because *TOStack* is a simulator.

  domain mapping : $d\_emp \rightarrow d\_emp$

  $d\_n\_emp\_n\_full \rightarrow d\_n\_emp$

  $d\_full \rightarrow d\_n\_emp$

The mapping derived from the simulator shows what states and sub-states are going to exist in the FSM. In this case, the FSM will contain two states (just like the super-type). The first one will contain one sub-state, while the second will contain two. Following is the *simulation constraint*:

$$(TOStack(s) \in d\_n\_emp \wedge s \in d\_full) \vee$$
$$(TOStack(s) \in d\_n\_emp \wedge s \in d\_n\_emp\_n\_full) \vee$$
$$(TOStack(s) \in d\_emp \wedge s \in d\_emp)$$

The interface specification of the BoundedStack class, which is a behavioral sub-type of the class Stack, is shown next.

imports Stack:

class BoundedStack: **public** Stack

{

uses BSub_BStackTrait(int, BoundedStack):

simulates Stack **by** TOStack:

```
    BoundedStack()
    {
        constructs self;
        ensures self' = new;
    }
    void BPush(int a)
    {
        requires !isfull(self);
        modifies self;
        ensures self' = push(self, a);
    }
    int BPop()
    {
        requires !isEmpty(self);
        modifies self;
        ensures self' = pop(self) ∧ result = top(self);
    }
};
```

At a first glance, the specification of the class is not very different. A closer look shows that the *simulates clause* makes all the difference. The above specification is not the complete specification of the class **BoundedStack** but only the "added specification". Previously, the specification showed that **BoundedStack** was a stand-alone class. This time it is inheriting from the **Stack** class specification which follows.

```
class Stack
{
imports StackTrait(int, Stack);
    Stack()
    {
        constructs self;
        ensures self' = new;
```

```
}
virtual void Push(int a)
{
    modifies self;
    ensures self' = push(self^, a);
}
virtual int Pop()
{
    requires !isEmpty(self^);
    modifies self;
    ensures self' = pop(self^) ∧ result = top(self^);
}
};
```

In order to proceed with the LCC analysis, the specification of the BoundedStack class must be "de-sugared" using the definition of specification inheritance. Note that only virtual member functions in public subclasses can have specification inheritance. The rest of the functions simply keep the given specification.

- **BoundedStack()** (Constructor)

  $Pre_S : true$

  $Post_S : isEmpty(self')$

  DNF: $self' \in d\_empty$

- **Push(a)** (virtual member function)

  $Pre_T : true$

  $Post_T : TOStack(self') = push(TOStack(self^), a)$

  $added\_Pre_S : !isfull(self^)$

  $added\_Post_S : self' = push(self^, a)$

  $Post_S : (true \Rightarrow TOStack(self') = push(TOStack(self^), a)) \wedge$
  $(!isfull(self^) \Rightarrow self' = push(self^, a))$

  DNF: $(TOStack(self') \in d\_n\_emp \wedge TOStack(self^) \in d\_emp \wedge$
  $self' \in d\_n\_emp\_n\_full) \vee$
  $(TOStack(self') \in d\_n\_emp \wedge TOStack(self^) \in d\_emp \wedge$
  $self' \in d\_full) \vee$

$(TOStack(self') \in d\_n\_emp \wedge TOStack(self) \in d\_n\_emp\_n\_full \wedge$
$\quad self' \in d\_n\_emp\_n\_full) \vee$
$(TOStack(self') \in d\_n\_emp \wedge TOStack(self) \in d\_n\_emp\_n\_full \wedge$
$\quad self' \in d\_full)$

- **Pop()** (virtual member function)

$Pre_T : !isEmpty(TOStack(self))$
$Post_T : TOStack(self') = pop(TOStack(self))$
$added\_Pre_S : !isEmpty(self)$
$added\_Post_S : self' = pop(self)$
$Post_S : (!isEmpty(TOStack(self)) \Rightarrow TOStack(self') = pop(TOStack(self))) \wedge$
$\quad (!isEmpty(self) \Rightarrow self' = pop(self))$
DNF: $(TOStack(self) \in d\_n\_emp \wedge TOStack(self') \in d\_emp \wedge$
$\quad self \in d\_full \wedge self' \in d\_emp) \vee$
$(TOStack(self) \in d\_n\_emp \wedge TOStack(self') \in d\_emp \wedge$
$\quad self \in d\_n\_emp\_n\_full \wedge self' \in d\_emp) \vee$
$(TOStack(self) \in d\_n\_emp \wedge TOStack(self') \in d\_n\_emp \wedge$
$\quad self \in d\_n\_emp\_n\_full \wedge self' \in d\_n\_emp\_n\_full) \vee$
$(TOStack(self) \in d\_n\_emp \wedge TOStack(self') \in d\_n\_emp \wedge$
$\quad self \in d\_full \wedge self' \in d\_n\_emp\_n\_full)$

In the above expressions, only the final form of the *Spec_OP* in DNF is given. All eliminations, due to the *simulation constraint*, are omitted.

As seen previously, the predicates that contain the simulation function *TOStack* refer to the behavior of the object when it is acting as a super-type object. Any predicates that do not map the *self* object to the super-type refer to the behavior of the object when it is acting as a sub-type. In order to construct the state machine for the BoundedStack the above DNF expressions are used in the same way as used in simple non-derived class. The only difference is that both behaviors of the super-type and sub-type are used to construct the state machine. Furthermore, the state machine of the sub-type should be the same as the state machine of the super-type when eliminating any nested states and transitions. Figure 19 shows the state machine for the subclass BoundedStack.

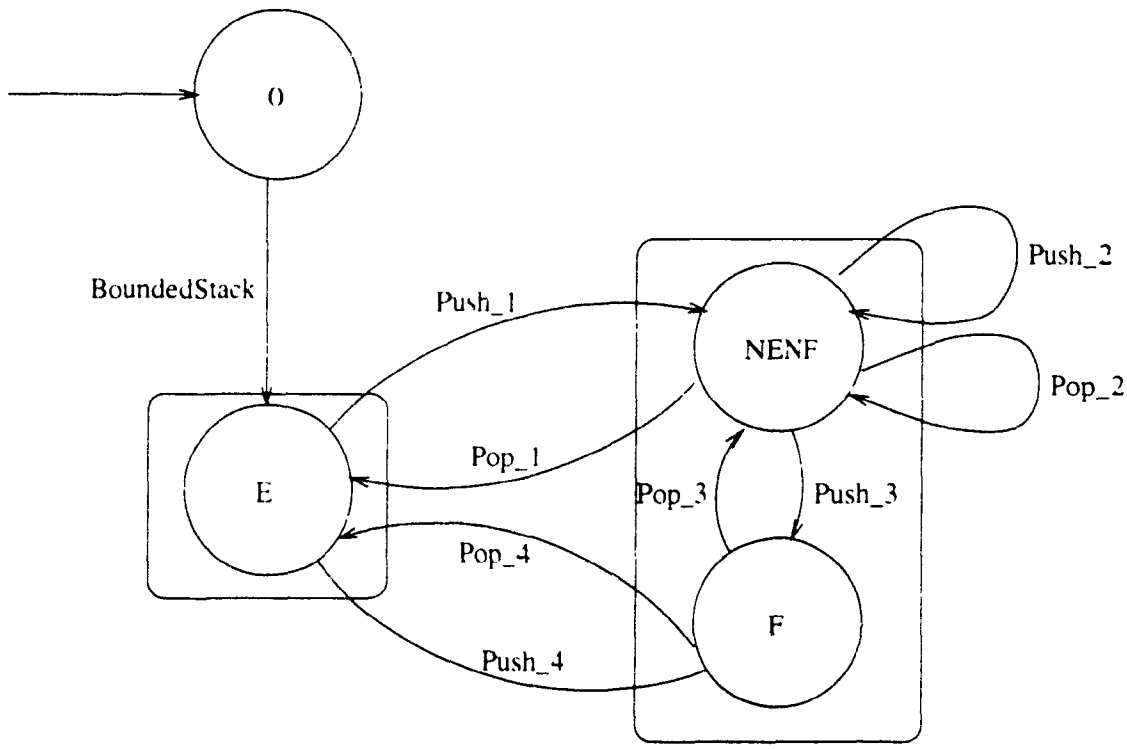


Figure 19: The Hierarchical FSM of a Subtype

Chapter 8

Conclusions

8.1 Summary

The overall goal of *black-box* object-oriented class testing is to assure the proper functioning of a class' implementation. The only way to guarantee a class' correctness is to execute it for all possible inputs or conduct a formal verification. Formal verification requires a formal semantics of the language and is quite expensive to conduct. Exhaustive testing is usually impossible since a class may have an infinite amount of possible inputs.

In this thesis we developed a methodology that, given a Larch/C++ specifications, generates a minimal and adequate test suite for testing a class implementation. The research in this thesis is a continuation of the methodology developed in [10], which constructs a FSM from Larch/C++ specifications.

Since Larch/C++ is still under development, the methodology developed in [10] did not capture all its new features. These new features were outlined and a revision of the methodology was proposed in order to capture them. Additionally, since many protocol test suite generation techniques do not satisfy the requirements of the FSM constructing methodology, a new technique was developed. Following, is the summary of contributions made by this thesis:

- All the new features and improvements of Larch/C++ have been illustrated.
- The FSM generating methodology is updated to resolve limitations and also

make it support the new features of Larch/C++ specification language.

- A new test suite generation technique has been proposed. for object-oriented class testing. This method generates a minimal number of test-cases without sacrificing the power of testing.
- The test suite generated from the FSM is shown to be adequate and to meet all the axioms relevant to black-box testing.
- The relationship between the FSMs of a sub-type and its super-type. in the context of test-case generation has been identified.

8.2 Future Work

This thesis does not address how the relationship between the FSMs of a sub-type and a super-type may be used for test-case reuse. Also. issues related to fault detection may be used to correct and debug a class implementation are not investigated either. All these issues are still open for research.

A very essential future research objective is to design a tool that implements the first part of the methodology. Celer [10] has presented a preliminary design of the tool. This tool will be linked with the test-case generation tool developed as part of this thesis The final aim is to automate as much as possible the test suite selection and execution and use it in the context of reusing class libraries [1. 3. 4].

Bibliography

[1] V.S.Alagar, P. Colagrosso, R. Achuthan, A.Celer, I.Umansky. *"Formal Specifications for Effective Black-Box Reuse - Phase I Progress Report"*. Department of Computer Science, Concordia University, Montreal Canada, September 1994.

[2] V.S.Alagar, P. Colagrosso, R. Achuthan, I.Umansky. *" Evaluating the Completeness of C++ Class Interface Specifications for Software Reuse"*, Technical Report, Department of Computer Science, Concordia University, Montreal Canada, 1995.

[3] V.S.Alagar, P. Colagrosso, A.Loukas, S.Narayanan, A.Protopsaltou. *"Formal Specifications for Effective Black-Box Reuse - Revised Phase I Progress Report"*, Department of Computer Science, Concordia University, Montreal Canada, February 1996.

[4] V.S.Alagar, P. Colagrosso, A.Loukas, S.Narayanan, A.Protopsaltou. *"Formal Specifications for Effective Black-Box Reuse - Final Report"*, Department of Computer Science, Concordia University, Montreal Canada, February 1996.

[5] P. America. *"Designing an Object-Oriented Programming Language with Behavioral Subtyping"* Lecture Notes in Computer Science 489, pp 60-90, Springer-Verlag, New York, N.Y., 1991

[6] Thomas R.Arnold and William A. Fuson. *"Testing in a perfect World"*, Communications of the ACM, September 1994, Vol 37, No 9.

[7] B.Beizer. *"Software Testing Techniques"*, Van Nostrand Reinhold, New York, second edition, 1990.

[8] J.P. Bowen, M.G. Hinchey. *"Seven More Myths of Formal Methods"*, Oxford University, Computing Laboratory, Technical Report PRG-TR-7-94, June 1994.

[9] J.P. Bowen. M.G. Hinchey. *"Ten Commandments of Formal Methods"*. Computing Laboratory. Oxford University.

[10] A.Celei. *"Role of Formal Specification in Black-Box Testing of Object-Oriented Software"*. Master of Computer Science. Department of Computer Science. Concordia University. Montreal Canada. 1995.

[11] P.Chalin. *On the Language Design and Semantic Foundation of LSL, a Larch/C Interface Specification Language*. Ph.D. Thesis. Department of Computer Science. Concordia University. 1995.

[12] Betty P. Chao. Donna M.Smith. *"Applying Software Testing Practices to an Object-Oriented Software Development"*. OOPSLA'93. Addendum to the Proceedings.

[13] Y.Cheon. G.T. Leavens. *"A Gentle Introduction to Larch/Smalltalk Specification Browsers"*. Department of Computer Science. Iowa State University. TR #94-01. January 1994.

[14] Y. Cheon. G.T. Leavens. *"The Larch/Smalltalk Interface Specification Language"*. ACM Transactions on Software Engineering and Methodology. 3(3):221-253. July 1994.

[15] Y.Cheon. G. Leavens. *"A Quick Overview of Larch/C++."*. Journal of Object-Oriented Programming. 7(6):39-49. October 1994.

[16] T.S.Chow. *"Testing Software Design Modeled by Finite-State Machines"*. IEEE Transactions on Software Engineering. May 1978.

[17] P. Colagrosso. *"Formal Specification of C++ Class Interfaces for Software Reuse"*. Master of Computer Science. Department of Computer Science. Concordia University. Montreal Canada. 1993.

[18] Krishna Kishore Dhara. Gary T. Leavens. *"Forcing Behavioral Subtyping Through Specification Inheritance"*. TR #95-20a. Aulowa State University. August 1995.

[19] J.Dick, A.Faivre, *"Automatic Partition Analysis of VDM Specifications"*, TR RAD/DMA/92027, Research and Advanced Development, Bull Systems Products, 1992.

[20] J.Dick, A.Faivre, *"Automating the Generation and Sequencing of Test Cases from Model-Based Specifications"*, FME'93 Industrial Strength Formal Methods, Lecture Notes in Computer Science 670, pp.286-284, Springer-Verlag, 1993.

[21] E.Dijkstra, *"A Discipline of Programming"*, Prentice-Hall, 1976.

[22] S.P.Fiedler, *"Object-Oriented Unit Testing"*, Hewlett-Packard Journal, pp 69-74, April 1989.

[23] S.V.Garland, J.V.Guttag, *"A Guide to LP, The Larch Prover"*, MIT, March 1993.

[24] A.Gill, *"Introduction to the Theory of of Finite-State Machines"*, New York: McGraw-Hill, 1962.

[25] J.A.Goguen, T.Winkler, *"Introducing OBJ"*, TR SRI-CSL-889, SRI International, 1988.

[26] G.Gonenc, *"A Method for the Design of Fault-Detection experiments"*, IEEE Trans. Comput., vol C-19, pp. 551-558, June 1970.

[27] J.Guttag J. Horning, *"Report on Larch Shared Language"*, Science of Computer Programming, vol 6, pp.103-134, 1986.

[28] J.Guttag J. Horning, *"Larch: Languages and Tools for Formal Specification"*, Springer-Verlag, 1993.

[29] P A V Hall, *"Relationship Between Specifications and Testing"*, Information and Software Technology, vol 33 no 1 January/February 1991.

[30] Mary Jean Harrold, John D.McGregor and Kevin J.Fitzpatrick, *"Incremental Testing of Object-Oriented Class Structures"*, Proceedings of the 14th International Conference on Software Engineering, ACM, 1992, pp. 68-80.

[31] Daniel Hoffman, Paul Strooper, *"A Case Study In Class Testing"*, Proceedings of CASCON '93, pp. 472-482, IBM Toronto, October 1993.

[32] Daniel Hoffman, Paul Strooper, *"Graph-based Class Testing"*, The Australian Computer Journal, Vol 26, No. 4, Nov 1994, pp.158-163.

[33] C.B.Jones, *"Systematic Software Development Using VDM"*, Prentice Hall International, second edition, 1990.

[34] K.D.Jones, *"LM3: A Larch Interface Language for Modula-3, A Definition and Introduction"*, TR 72, Digital Equipment Corporation System Research Center, 1991.

[35] S.Fujiwara et al, *"Test Selection Based on Finite State Models"*, IEEE Transactions on Software Engineering, vol 17, June 1991.

[36] Shekhar Kirani, W.T.Tsai, *"Specification and Verification of Object-Oriented Programs"*, Technical report, Computer Science Department, University of Minnesota.

[37] David Kung, Jerry Gao et al, *"Developing an Object-Oriented Software Testing and Maintenance Environment"*, Communications of the ACM, Vol 38, No 10, October 1995, pp.75-87.

[38] G.T. Leavens, *"Modular Verification of Object-Oriented Programs with Subtypes"*, Department of Computer Science, Iowa State University, TR 90-09, July 1990.

[39] G.T.Leavens, Y.Cheon, *"Preliminary Design of Larch/C++"* Proceedings of the First International Workshop on Larch, Springer-Verlag, 1992.

[40] G.T.Leavens, *"Larch/C++ Reference Manual"*, Draft, $Revision 4.1$, December 24 1995.

[41] G.T.Leavens, *"An Overview of Larch/C++ . Behavioral Specifications for C++ Modules"*, Department of Computer Science, Iowa State University.

[42] R.A. Lerner *"Specifying Objects of Concurrent Systems"*, School of Computer Science, Carnegie Mellon University, CMU-CS-91-131, May 1991.

[43] B. Liskov, J. Wing, *"A Behavioral Notion of Subtyping"*, ACM Transactions on Programming Languages and Systems, 16(6): 1811-1841, November 1994.

[44] B. Liskov, J. Wing. "*Specifications and their Use in Defining Subtypes*", ACM SIGPLAN Notices, 28(10):16-28, October 1993. OOPSLA '93 Proceedings.

[45] J.D. McGregor. "*Constructing Functional Test Cases Using Incrementally Derived State Machines*", Object Technology Group, Department of Computer Science, Clemson University.

[46] J.D. McGregor. "*Functional Testing of Classes*", Object Technology Group, Department of Computer Science, Clemson University.

[47] J.D. McGregor, D.M. Dyer. "*Selecting Functional Test Cases for a Class*", Proceedings of the Pacific Northwest Software Quality Conference, 1993.

[48] J.D. McGregor. "*Testing Object-Oriented Software Systems*", OOPSLA '95, October 1995.

[49] G. Myers. "*The art of Software Testing*", New York: Wiley, 1979.

[50] Kurt M. Olender, James M.Bieman. "*Algebraic Specifications and Sequencing: A defect Detection Method*", Software Testing, Verification and Reliability, Vol5, 1995, pp 49-70.

[51] T.J.Ostrand, M.J.Bacler. "*The Category-Partition Method for Specifying and Generating Functional Tests*" Communications of ACM, Vol 31, No 6, June 1988.

[52] D.E. Perry, G.E.Kaiser "*Adequate Testing and Object-Oriented Programming*", Journal of Object-oriented programming, January/February 1990.

[53] Rogue Wave. "*Tools.h++ Class Library v6.0*", Rogue Wave Software, 1993

[54] K.K.Sabmani, A.T.Dahura. "*A Protocol Testing Procedure*", Comput. Networks and ISDN Syst., vol 15, no 4, pp. 285-297, 1988.

[55] M.D.Smith, D.J.Robson, "*Object-Oriented Programming - the Problems of Validation*", Proceedings of the IEEE Conference on Software Maintenance, Sep 1990, pp. 370-281.

[56] J.M.Spivey, "*The Z Notation: A Reference Manual*", Prentice Hall, New York, 1989.

[57] B.Stroustrup. "*The C++ Programming Language*" second edition. Addison Wesley. 1992.

[58] B.Stroustrup. "*The Design and Evolution of C++*". Addison-Wesley. 1994.

[59] C.D.Turner and D.J.Robson. "*The State-based Testing of Object-Oriented Programs*". Proceedings of the IEEE Conference on Software Maintenance. Sep 1993. pp 302-311.

[60] I.Umansky. "*Completeness of Larch/C++ Specifications for Black-Box Reuse*". Master of Computer Science. Department of Computer Science. Concordia University. Montreal Canada. 1995.

[61] E.J. Weuyker. "*Axiomatizing Software Test Data Adequacy*". IEEE Transactions on Software Engineering. Vol. SE-12. December 1986.

[62] J.M. Wing. "*A Specifier's Introduction to Formal Methods*". IEEE Computer. 1990.

[63] J.M. Wing. "*Hints to Specifiers*". Proceedings of the Fourth International Conference AMAST '95. July 1995.

Appendix A

Design of the Test-Case Generation Tool

This Appendix describes briefly the design of the test-case generation tool developed as part of this thesis. This tool was designed to be integrated with a larger tool which will implement the entire methodology. Section 1 presents the assumptions that were made in order to develop the system. Section 2 presents a brief description of the system.

A.1 Assumptions

For the design and implementation of the test-case generation tool a few assumptions were made. These are:

- The existence of input files that contain the description of the finite state machines in a certain format.
- Rules and predicates may involve only integer arithmetic.

Input Files: There are two input files associated with each test suite generation. Both have the same name but different extension. For instance, in the Stack example, the two files will have the names: **Stack.mac** and **Stack.sem**. The file with extension **.mac** is a data file that contains the structure of the finite state machine. The number of states in the finite state machine is at the top of the file. Successively, the names of the states follow. After, the names of the states, the number of transitions in

the finite state machine is given. Following this, for each transition the number of the source state and the number of the destination state is given followed by the name of the transition. An example of such a file is given in Table 9 Stack mac. The file with extension .sem contains the semantics of each transition and state in

Stack.mac	Stack.sem
3	1 size int
I	I true.
E	E size==0.
NE	NE size>0.
5	size=0. Init_1 true.
1 2	size=size+1. push_1 true.
Init_1	size=size+1. push_2 true.
2 3	size=size-1. pop_1 size==1.
push_1	size=size-1. pop_2 size>1.
3 3	
push_2	
3 2	
pop_1	
3 3	
pop_2	

Table 9: .mac and .sem files for the Stack class

the finite state machine that was described in the corresponding .mac file. At the top of the file, the number of variables that may be contained in the predicates and rules is given, followed by the name and the type of the variable. Consecutively, for every state that was described in the .mac file, the associated name and predicate is given. Following, for every transition in the mac file, the associated rule, name, and predicate (condition) are given. An example of such a file is given in Table 9 (stack.sem).

A.2 System Specification

In this section, the object model and the interface specification for most of the classes will be shown. Some of the classes are best specified using a context free grammar (CFG) because they implement parsers.

A.2.1 Object Model

Following the *Object Modeling Technique* (OMT), this section presents the object model of the system which captures the static structure of the system. This static structure shows the relationships between the objects of the system. Figure 20, shows the object model for the test-case generation tool that implements the technique presented in Chapter 6.

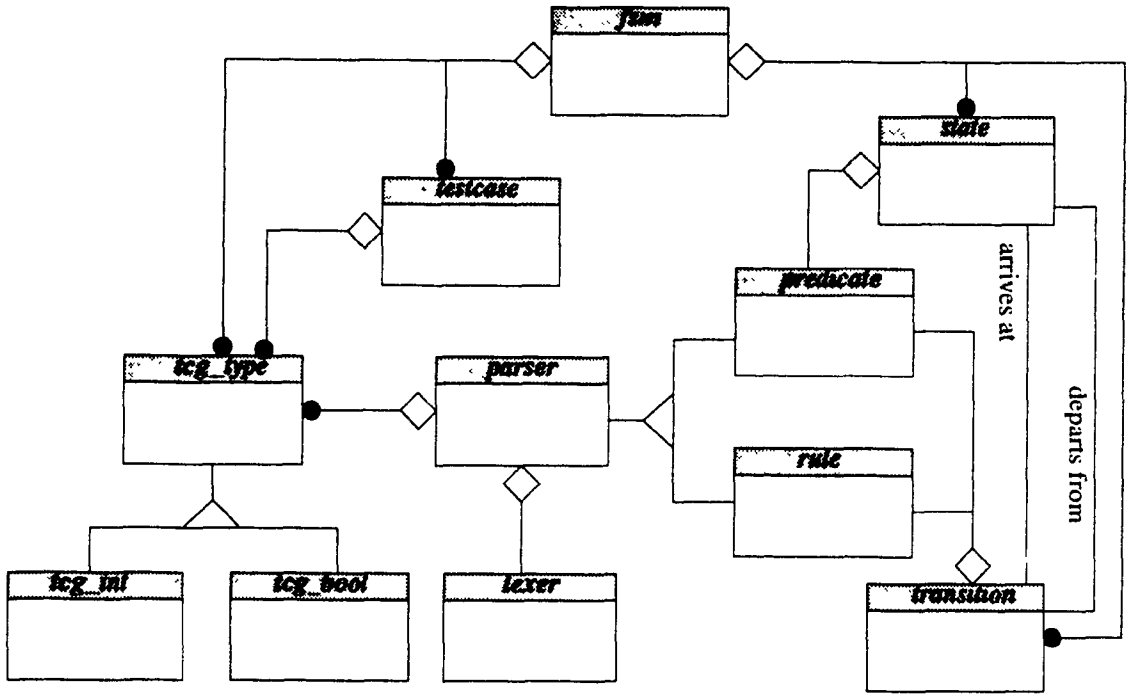


Figure 20: Object Model of the tool

A.2.2 Classes: fsm, state, transition

The class `fsm` represents the finite state machine that is inputed from the two data files .mac, .sem. It contains a list of `state` objects and a list of `transition` objects. These two lists of objects represent the states and transitions inputed from the two data files. An `fsm` object contains also a list of `test_case` objects which represents the test suite that is derived for the current `fsm` object. Also, an `fsm` object contains a list of `tcg_type` object which serves as a symbol table for the variables that are part of a predicate.

A `state` object contains a `predicate` object which represents the predicate that is associated with the `state` object.

A `transition` object contains a `predicate` object which represents the predicate that is associated with the `transition` object. It also contains a `rule` object which specifies how the symbol table variables may be modified when the associated transition is traversed.

A.2.3 Classes: parser, rule, predicate, lexer

As can be seen from the object model, the `rule` and `predicate` classes are both subclasses of the class `parser`. Therefore both `rule` and `predicate` are parsers but they may parse different expressions. Table 10 shows the LL(1) grammars for these two classes, along with the lookahead symbols. Clearly, many rules in these two tables are identical. These common rules are implemented by the superclass `parser`. A `predicate` object represents a predicate that may be associated with a state or a transition. A `rule` object specifies how the symbol table variables may be modified when a particular transition is traversed. The internals of both objects are inputed from the .sem input file.

A `lexer` object is a lexical analyzer that is associated with a `rule` or `predicate` object. It implements all the lexical conventions that follow:

- An identifier is a sequence of letters and digits. It may only start with a letter. It is specified as $l(l+d)*$, where l is any letter and d is any digit.
- A numerical constant (integer) is a sequence of digits specified as $dd*$.
- String constants are not defined.
- The language is case sensitive.
- White spaces may not appear in any expressions

predicate grammar	LL(1) symbols	rule grammar	LL(1) symbols
$Pred \rightarrow U.$	i ni ([true false	$Start \rightarrow S_1.$	i
$U \rightarrow C_1 W$	i ni ([true false	$S_1 \rightarrow SS_2$	i
$C_1 \rightarrow C$	i ni (	$S_2 \rightarrow : SS_2$	:
$C_1 \rightarrow [U]$	[	$S_2 \rightarrow \lambda$	.
$W \rightarrow ObUW$	AND OR	$S \rightarrow iK$	i
$W \rightarrow \lambda$	.]	$K \rightarrow = E$	=
$C \rightarrow EOrE$	i ni (	$E \rightarrow FJ$	i ni (
$E \rightarrow FJ$	i ni (	$F \rightarrow RJ_1$	i ni (
$F \rightarrow RJ_1$	i ni (	$J_1 \rightarrow OmRJ_1$	* /
$J_1 \rightarrow OmRJ_1$	* /	$J_1 \rightarrow \lambda$	+ - : .)
$J_1 \rightarrow \lambda$	) < > >= <= == ! = AND OR .]	$Om \rightarrow *$	*
$Om \rightarrow *$	*	$Om \rightarrow /$	/
$Om \rightarrow /$	/	$R \rightarrow i$	i
$R \rightarrow i$	i	$R \rightarrow ni$	ni
$R \rightarrow ni$	ni	$R \rightarrow (E)$	(
$R \rightarrow (E)$	(	$J \rightarrow OaFJ$	+ -
$J \rightarrow OaFJ$	+ -	$J \rightarrow \lambda$	: .)
$J \rightarrow \lambda$	) < > <= >= == ! = AND OR.]	$Oa \rightarrow +$	+
$Oa \rightarrow +$	+	$Oa \rightarrow -$	-
$Oa \rightarrow -$	-		
$Ob \rightarrow AND$	AND		
$Ob \rightarrow OR$	OR		
$Or \rightarrow <$	<		
$Or \rightarrow >$	>		

Table 10: CFG for predicate and rule classes

A.2.4 Classes: tcg_type, tcg_int, tcg_bool

In Figure 20, it is shown that the `tcg_int` and `tcg_bool` objects are subclasses of `tcg_type`. These three classes define *integer* and *boolean* data types. These data types are used for the definition of the symbol table and manipulation of the values stored in it. In this version of the tool only integer and values are allowed. The system is designed in a way that more data types may be added in the future without having to make many changes in the entire system.

A.2.5 Class: test_case

A **test_case** object represents a test sequence generated from the methodology described in Chapter 6. It contains a snapshot copy of the symbol table. This facilitates in identifying a certain test-case as *legal/illegal* or a certain *TCG node* as *leaf/non-leaf*.