NOTICE

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

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

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

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

AVIS

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

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

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

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

Canada

Formal Specification of C++ Class Interfaces for Software Reuse

Piero Colagrosso

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements for

the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

March 1993

ISBN 0-315-87256-X

ABSTRACT

Formal Specification of C++ Class Interfaces for Software Reuse

Piero Colagrosso

Software reuse is widely believed to have the potential of greatly improving software development productivity and quality. One of the much touted and elusive promises of the object-oriented development approach has been its great support for reuse and the potential productivity and quality increases which could result from this. Unfortunately, these promises do not seem to have materialized yet and widespread software reuse still remains an enduring dream.

In this thesis we argue that to provide better support for software reuse, object-oriented programming languages must be supplemented with a specification language which can provide precise semantic specifications of reusable components. We identify the properties which a component specification language should have in order that it can best promote reuse and capitalize on the advantages of the object-oriented paradigm.

As a first step towards the realization of such a specification language for the programming language C++, we show that the formal specification language Larch/C++ has many of the desired properties and can be used to specify the behavior of C++ class interfaces. We also present a formal definition of class behavior and we use this definition to develop a criterion for evaluating the completeness of class interface specifications. We then present a methodology for applying this criterion to Larch/C++ specifications. Finally, we present a proposal for extending Larch/C++ with a proof theoretic semantics for determining subtyping relations and a mechanism for specification inheritance.

To my Wife Sonia and Daughter Emilia

Acknowledgments

I express my deepest gratitude to my supervisor Professor V.S. Alagar. His sustaining guidance and encouragement made my thesis work a very pleasant and educational experience.

I thank Ramesh Achuthan for his friendship, encouragement and for the many long and stimulating discussions we had on formal methods, object-orientation and on the material contained in this thesis. His many pertinent questions and insightful comments on my work were instrumental in helping me clarify and refine my ideas into the material presented in this thesis. His help was invaluable.

I also wish to thank Professor Gary Leavens of Iowa State University for answering my numerous questions regarding Larch/C++ and for providing me much helpful information on Larch/C++.

I gratefully acknowledge the financial support provided by BNR (Bell-Northern Research) and the Natural Sciences and Engineering Research Council of Canada. I thank Michael Payette, Jack Dyment, Hugh Cameron and Serge Fournier for having made it possible for me to obtain a study leave from BNR to work on this thesis. I also thank Michael Payette for providing me with a good working environment at BNR where I could conduct part of my research.

Finally, and most of all, I would like to thank my wife Sonia and my daughter Emilia for their patience and for the encouragement which they brought me through their confidence in my abilities and through their cheerful presence. This work would not have been possible without them.

Contents

List of Figures

Chapter 1

Introduction

It has been widely believed for some time now that software reuse has the potential of greatly improving software development productivity and quality [HM84, Sta84, Jon84, LG87, BR87, Kru92]. This view is based on the observation that considerable savings can be realized by assembling software systems using reusable components rather than by building them entirely from scratch. Underlying this view is the assumption that reusable components can be located, understood and integrated within an application much more economically than by writing the components entirely from scratch.

Proponents of the object-oriented programming approach have argued very convincingly that the data abstraction, encapsulation, inheritance and polymorphism mechanisms provided by object-oriented (OO) languages are very conducive to software reuse [Cox86, Mey87, Mey88, KM90, G.B91]. Despite this, widespread large-scale software reuse still remains an elusive dream [Cox90].

One of the principal hurdles which must be overcome before this dream can become a reality is the specification of the abstractions provided by classes in a succinct, implementation independent, and complete manner. If programmers are to reuse classes effectively, they must be able to develop a precise understanding of the abstractions implemented by classes with a minimum of effort.

It is the premise of this thesis that the principal reason that OO languages have failed to promote widespread, successful software reuse until now is that they have failed to address the issue of specifying the abstractions implemented by classes. The mechanisms provided by the OO paradigm address only the syntactic support required for effective reuse but fail to address semantic support issues. In this thesis, we describe how this void can be filled by supplementing OOPLs (object-oriented programming languages) with a specification language which can provide a precise semantic description of components. We will refer to such a specification language

as a *ROOCSL* (Reusable Object-Oriented Component Specification Language).

Most OOPLs rely on natural language specifications to document the semantics of class interfaces.[1] Because of their informal nature, these specifications may be ambiguous, incomplete, and not sufficiently precise. This makes them inadequate for the purposes of describing the behavior of reusable classes to potential reusers.

The reuser of a class needs a precise mental model of a class's behavior in order to reuse it correctly. If the informal specification does not provide such a model, then a developer must often turn to the implementation of a class to *really* understand the class's behavior. This effort can offset part or all of the productivity increases which should have been obtained by reusing a class rather than rewriting it. Much of the advantage of reuse is lost if it is necessary to understand and test a class as carefully as if it had been written from scratch.

The effort required to understand the behavior of a class by reading its implementation code should not be underestimated. Some studies suggest that *program understanding time* may be the dominant time in the entire software life cycle and thus the dominant cost [Sta84]. In view of this, it might be construed that software reuse will be effective in increasing productivity only so far as it will be effective in reducing or eliminating the need for programmers to read and understand the source code of modules which they reuse.

Researchers in the area of software reuse have recognized the critical role which the ability to specify abstractions and to make these easily understandable to reusers plays in ensuring a successful software reuse methodology [HM84, BR87, Cox90]. In [Kru92] it is emphasized that any activity of software reuse involves four dimensions of *abstracting*, *selecting*, *specializing* and *integrating* software components. Among these, abstraction plays a key role in reusing a software component, since abstraction is necessary for easy selection, specialization and integration.

An abstraction for a software component is a succinct description that is free of irrelevant details but generalizes and emphasizes the information that is important. It is our view that the most suitable medium for describing module abstractions is a formal specification language accompanied by an informal natural language explanatory

[1] The one notable exception is Eiffel. However, for reasons discussed in section 3.1, we do not feel the specification style used in Eiffel is adequate for specifying reusable classes.

text. The formal specification provides the necessary precision and formal semantics while the explanatory text improves the understandability of the specification.

Formal specification languages such as Larch [GHW85b, Win87, GHM91, GH91], Z [Hay87, Spi88, Spi89], VDM [Jon90], and OBJ [FGJM84, GW88] are precise and unambiguous. They are based on mathematics and have a formal syntax and semantics. They remove any area of doubt in a specification. Because of their declarativeness and lack of concern for implementation details, formal specifications can describe a class abstraction far more succinctly than an informal specification or the implementation code itself.

In this thesis, we focus on the identification of a ROOCSL for the programming language C++ [Str91]. C++ is a superset of C providing support for data abstraction, encapsulation, polymorphism, and inheritance while retaining the efficiency and portability of C. Because of its characteristics, C++ has become the object-oriented language of choice in industry for the implementation of large production software system. Although there are no known statistics to support this claim, it is supported by anecdotal evidence such as the volume of traffic on the `comp.lang.c++` usenet news group as well as by the author's involvement with software development projects in industry. As a result of the widespread use of C++ in industry, any approach intended to promote software reuse amongst practitioners is most likely to succeed if it is targeted at C++. For these reasons, we have chosen to focus on C++ as a concrete example. However, most of the analysis and results discussed in this thesis are sufficiently general to be adapted and applied to other object-oriented programming languages.

In order to identify an appropriate ROOCSL for C++, we first identify the properties which a general ROOCSL should have in order that it can best promote reusability and capitalize on the advantages of the object-oriented paradigm. We then survey existing work aimed at providing semantic descriptions of object-oriented components and assess to what extent these proposals satisfy the properties we have identified. As a first step towards the realization of a ROOCSL for C++, we show that the formal specification language Larch/C++ has many of the desired properties and can be used to specify the behavior of C++ class interfaces for software reuse.

Having selected a formal specification language, it is also necessary to define precisely *what* needs to be specified as part of a class interface specification. Without such a definition, there can be no systematic methodology for evaluating the completeness of a specification. Such an evaluation is required to ensure that a specification provides a developer with all the information required to reuse a class in a truly black box fashion.

In the absence of a systematic method for evaluating the completeness of a class specification, the evaluation becomes a subjective matter, leaving it to the designer/implementor's discretion what the specification should and should not state. This, in turn, can very adversely affect the quality of a reusable class. Poor documentation may confuse, or worse, mislead a prospective reuser as to how a class should be used. From the reuser's point of view, having an incomplete specification is just as undesirable as attempting to reuse a class whose implementation is defective.

In order to develop a criterion for evaluating the completeness of a class interface specification, it is necessary to formalize the notion of *behavior of a class* which the specification is intended to capture. In this thesis, we present a formal definition of the behavior of a class and we use this definition to develop a criterion for evaluating the completeness of a class interface specification. We also use some examples to show how this criterion can be applied to evaluate the completeness of some C++ interface specifications written in Larch/C++.

The organization of the remainder of this thesis is as follows. In chapter 2 we survey the critical issues involved with the formal specification of classes and define the scope for the remainder of this thesis by identifying which of these issues will be treated. In chapter 3 we survey existing specification languages for specifying classes (modules) and we justify our choice of using Larch/C++ for the specification of reusable C++ class interfaces.

In chapter 4 we present our formal definition of class behavior and use this definition to develop a criterion for determining when a class interface specification is complete. In chapter 5 we present a methodology for applying this completeness criterion to Larch/C++ specifications. In chapter 6 we present a proposal for extending Larch/C++ with a proof theoretic semantics for determining subtyping relations and

a mechanism for specification inheritance.

Finally, in section 7 we conclude by giving our assessment of the potential usefulness and impact of the work described in this thesis, summarizing the contributions of this thesis and identifying future work.

Chapter 2

Survey of Critical Issues and Scope of This Thesis

In this chapter we survey the critical issues involved with the formal specification of class interfaces for reuse and we define the scope of this thesis by identifying which of these issues will be addressed herein.

In subsections (2.1 to 2.4) we consider various aspects relating to the object-oriented paradigm (data abstraction and encapsulation, inheritance, polymorphism, and subsystems and frameworks) and identify how OO programming languages fail to provide semantic support for reuse in the context of each of these aspects. We also describe how the use of a formal specification language could help overcome these shortcomings for each of these aspects.

In section 2.5 we discuss the need to evaluate the completeness of class interface specifications and in section 2.6 we discuss the need to verify the correctness of candidate class implementations with respect to these specifications.

Based on all these observations, in section 2.7 we summarize the properties which an ideal ROOCSL should have. Finally, in section 2.8 we define the scope of this thesis and identify which of the critical issues will be tackled herein.

2.1 Data Abstraction and Encapsulation

In object-oriented programming languages, the basic software module is the *class* and it is used to implement an abstract data type [Mey88, KM90, Bud91]. One of the cornerstones of object-oriented programming is *data abstraction* and refers to the fact that a class's interface is completely independent from its implementation. Classes implemented by one programmer (the *producer* or *implementor*) can be used by other programmers (the *clients*) without these clients having to concern themselves with understanding a class's implementation details.

The client of a class needs only to understand the behavior of the class as specified by the method interfaces and may view the method implementations as being contained in a *black box* hidden from view. This approach is conducive to reuse, as a module can be used by a client without concern for the implementation details of this module; much less effort is required than to write a module from scratch or to reuse an existing module by understanding its implementation details.

However to reuse code, one must understand its behavior precisely. The black box approach to reuse assumes that a class's behavior can be described succinctly and without having to refer to an implementation. Object-oriented programming languages offer no support for this ability. Instead, programmers must rely on the informal comments in class interface files to understand the semantics of a class.

OO languages provide support for encapsulation, i.e. the ability to make data accessible only to a specified set of operations. This provides the syntactic support required for creating reusable black boxes. However, the semantic support required to describe the behavior of these black boxes is not provided in most object-oriented languages.

Informal documents and program comments may be something of a relief. However, their imprecise, verbose, and potentially ambiguous and incomplete nature often prevents them from being much help [Mey85, Win90, IA90]. A natural language such as English is not sufficiently precise to describe the exact functioning of a software module as a black box, and leads to ambiguities and differences of interpretation from one programmer to the next. What appears obvious to the implementor of a class may not be so obvious to a client that is supposed to rely on nothing more than the class's method signatures and informal comments to learn how to use it.

The result of this inability to adequately specify the behavior of reusable components is that programmers are forced to turn to the implementation of a class to fully understand its behavior. However, the sheer volume of existing classes, their complex interactions, and their implementation details frustrate programmers who, by inspecting the code, must try to understand the behavior of potentially useful classes.

The net effect of all these problems is that a client may ultimately make improper

use of a class, resulting in a defective program. The client's only avenue to solve the program defect is then often to "debug" the code by tracing its execution to try to determine the cause of the interfacing (i.e. usage) problem. As a result, a substantial amount of time is spent trying to learn how to use a class correctly. All this wasted effort offsets part or all of the productivity increases which should have resulted by reusing the module rather than rewriting it.

The above observations indicate the need for a formal semantic definition of reusable OO component interfaces. Because of its precision and conciseness, a formal specification language has the potential of overcoming the problems associated with the use of natural languages for specifying the semantics of reusable OO components.

Formal specifications also have another advantage over natural language in that they can be automatically processed. This opens the door for such things as automatic searching in component libraries, specification syntax checkers, and partially automated semantic analysis of specifications [Som89, Win90, dCAC+91]. Because formal specifications are mathematical objects, it is also possible to develop formal criteria and algorithms to evaluate properties of these specifications such as completeness and consistency.

Using a formal specification language to specify reusable components also has the advantage of providing the precision required to be able to distinguish easily between *incidental* and *intended* features of a reusable component [Wil91].

Intended features of a reusable component are those which are intrinsic to the component and which are guaranteed not to change in subsequent versions of a component. In contrast, the incidental features of a module are those which are merely the byproduct of a specific implementation of the component and which may change in future versions of the component. For example, in the case of a list union method, the fact that the resulting list will contain all the items in the two original lists is an intended feature and is guaranteed not to change in future versions of the component. However, the fact that the items in the resulting list retain their original order can be either an incidental or an intended feature of the module. If the specification states that the order is preserved then the feature is intended. However, if the specification does not make any reference to the order of the items then the feature is incidental

and should not be assumed in client code. If updated versions of a component are to be distributed and incorporated into systems which use it, the systems' designers must be able to make the distinction between incidental and intended features to avoid becoming dependent on "features" which are nothing more than artifacts of a given version of a component.

Based on the above observations, it is apparent that a ROOCSL should be able to specify the behavior of an OO component independently of its implementation. In other words, such a specification should never refer to any of the implementation variables of a component. There are several other reasons why this is important, and we summarize these here:

(i) **Implementation Updates, Fast Prototyping:**

One of the benefits of the OO approach is that it makes it possible to modify the implementation of a class (e.g. for efficiency reasons, for porting to a new platform, for fast prototyping) without modifying its interface. This means that the clients of a class will not be affected by such an implementation change. This is very practical, as it means that the implementations of classes can be modified with very minimal impact to the system in which these classes are embedded. This approach accounts for the good support which OOP offers for the rapid prototyping of systems and the ability to evolve such prototypes rapidly. It also accounts for the ease of maintenance and modification attributed to the OO approach. During the maintenance and modification phases of a system, data structures and representations are likely to change, either to improve performance or to accommodate changing requirements. The data abstraction mechanism provided by the OO approach localizes the changes to the implementation of a class. Clients should not be affected by the change of implementation of the classes they use.

If the specification of a component refers to its implementation and if this implementation is updated, then the specification will likely also have to be updated. This is very impractical for both the reusers and maintainers of reusable classes:

- **reusers:** For the reusers of the class it means they have to understand a new description of the class (i.e. change their mental model of the class), even if the semantics of that class have not changed.

- **implementors:** For implementors, this means that it is now necessary to update the specification, in addition to the source code, when the implementation is modified. In addition to being error-prone, this task also imposes the additional burden of ensuring that the two specifications are equivalent. This is by no means a trivial problem [Cox90].

As a result, all the benefits of the OO approach relating to fast prototyping, ease of maintenance, ease of modification, and evolvability are lost unless implementation independent specifications are used.

(ii) **Abstract Classes:**

For abstract classes [WBJ90, KM90], there is usually no implementation to be described. This means that the specification of abstract classes cannot possibly refer to their implementation. However, as we will discuss in section 2.3, for the purposes of polymorphic code it is important to document what constraints the types associated with abstract classes impose on their subtypes. Without an implementation independent specification, this cannot be accomplished.

(iii) **Ease of Understanding:**

The principal reason that software reuse can lead to increased productivity is that it provides a higher level of abstraction for a software developer. It permits a developer to benefit from the services of a software module without having to concern himself with the implementation details of this module. To be consistent with this philosophy, the specification of such modules should not refer to implementation details.

Behavioral specifications (i.e. those that do not refer to an implementation) have the advantage of fully preserving the encapsulation of a class. Because such specifications can be made highly declarative, they have the potential of

being much more concise and of more readily conveying the precise semantics of a class interface than informal comments or implementation code.

(iv) **Separation of concerns:**

From a software design point of view, the description of the behavior of a class and the description of the implementation of a class represent two different concerns. The former corresponds to a *high-level design* while the latter corresponds to a *detailed design.*

For ease of reuse, it is important to keep these two concerns separate. While the high-level design of an OO system is being developed, it is necessary to identify the required classes and the functionality which they provide (i.e. their behavior). To maximize opportunities for reuse, a designer should attempt to identify potentially useful classes from a reusable library at this stage so that they can be included as part of the high-level design. Since the designer is only interested in the functionalities provided by reusable classes, it would be unreasonable to refer to the implementation in the specification.

At the later stages of detail design, a developer will be interested in "sketching" the implementation of the objects identified in the high-level. At this stage, there is also potential for identifying classes from a reusable library which can be useful in this context. However, once again, a developer is interested in the services provided by the reusable classes and *not* in their implementation.

(v) **Design:**

Implementation independent specifications are also a useful design tool. The data abstraction mechanism provided by the OO paradigm makes it possible to defer decisions about data structures until the uses of the data are fully understood. Instead of defining data structures in the high-level design, it is possible to focus on classes and their operations. Implementation independent specifications provide a powerful way to document the behavior of operations without committing to a particular implementation. Decisions about how to implement a class can be made later, when all its uses are well understood.

It is important to note that although behavioral specifications are required for specifying the semantics of class interfaces for reuse, specifications which refer to the implementation, called *implementational specifications*, are useful in other contexts:

- Such specifications are useful for clarifying the intentions of the implementor of the class, so that maintainers of the class can understand and modify the implementation more easily.
- They can be useful in the development of debugging aids. For example, assertions (pre/post conditions) in Eiffel [Mey88] refer to the implementation of classes and can be automatically tested at run-time. A++, a specification language for C++, also provides debugging tools based on implementational specifications [CL90a, CL90b].
- Finally, such specifications are useful in constructing formal proofs of correctness of the implementation with respect to its specification. In Fresco [Wil91], both behavioral and implementational specifications are used. The latter are used to prove that the implementation of a class correctly implements the former.

2.2 Inheritance and Subtyping

It has now become clear in the object oriented research community that the original object-oriented concept informally known as *inheritance* actually encompasses two distinct, orthogonal concepts [Sny86, Ame87, Lis88, DT88, WZ88, Ame89, CHC90, Cus91]:

(i) **Implementation Inheritance**: This is a mechanism for the incremental definition of new classes based on existing ones. It can be defined as the carrying of features (methods and instance variables) from a parent class definition to its child class and the possible overriding of methods in the child class.

By sharing implementation code which describes the internal representation of classes, the total amount of code in a system can sometimes be reduced drastically. In general, this form of inheritance does not provide any guarantees that a newly-derived class will be a specialization of its parent class. This is because a method may be overridden in the derived class in a way which is not consistent with the parent class.

(ii) **Subtype Inheritance (conformance)**: This is a relationship between the specification of two classes and characterizes the informal *is-A* relation between classes. That is, subtyping captures the notion of behavioral compatibility between two classes by requiring that members of a subtype are also members of the supertype.

The subtyping property ensures that a subtype can be reliably substituted for a supertype in a specification or program. This notion, which we will refer to as the *substitutivity principle*, is especially crucial in the context of polymorphism (as discussed further in section 2.3).

We emphasize that the subtyping which we are referring to corresponds to semantic subtyping based on specifications. This is in sharp contrast to the notions of subtyping or conformance which are determined based on purely syntactic considerations, (i.e. the signatures of a class's methods), in many typed object-oriented languages (e.g. Eiffel, C++). In the remainder of this

thesis, we will always take the term subtyping to refer to semantic subtyping.

Implementation and subtype inheritance each define a different class hierarchy. Implementation inheritance defines an implementation hierarchy which is based on the incremental sharing of code. It describes the construction of the internal structure of objects and is useful to the implementors of classes [Cox90, DT88, Ame89]. The implementation hierarchy says nothing, or worse misleads, about the class's behavioral specification — the properties that the class offers its consumers [Cox90]. Subtype inheritance, in contrast, defines a conceptual hierarchy amongst the behavioral specifications of classes. This hierarchy is useful to reusers of classes since it is based on the externally observable behavior of objects (i.e. the behavior which can be observed by sending messages to objects) [Cox90, DT88, Ame89].

For a long time, the prevailing view in the object-oriented community was that the implementation and subtype hierarchies were in fact the same [Ame89]. This belief was based on the intuitive idea that if a class `B` inherits from a class `A`, each instance of class `B` will have at least all the variables and methods that instances of class `A` have. This seems to suggest that whenever an object of class `A` is required, an instance of class `B` would do equally well so that instances of class `B` can be regarded as *specialized versions* of the instances of class `A`. However, within the past few years many researchers have pointed out that the two hierarchies are distinct and that confusing them can lead to several problems [Sny86, Ame87, DT88]. In fact, researchers have also advocated that OOPLs should explicitly separate the two hierarchies [Sny86, Ame87, DT88]. POOL [Ame90], a recently designed OOPL has adopted this approach.

The above observations suggest that the classes in a reusable library should be organized according to the specification hierarchy, to make explicit the behavioral relationships between classes to reusers. However, just the opposite is currently true since reusable class libraries are arranged in such a way as to give a high degree of code sharing among the classes in the library, not according to their conceptual relationships [LTP86, DT88, Lal89, Cox90].

Brad Cox was co-founder and chief technical officer of Stepstone, a company which spent the past nine years developing and marketing reusable object-oriented

components and promoting the mass scale reuse of such components. While relating his company's experiences in [Cox90], Cox concluded, after seven years of operation, that one of the major stumbling blocks in making the behavior of reusable classes understandable to clients was the fact that class libraries were arranged according to the implementation hierarchy rather than the specification hierarchy. Cox now advocates the organization of reusable class libraries according to the specification hierarchy [Cox90].

Cox also reported that the inability to provide precise specifications of the behavior of classes, or *making classes tangible* as Cox describes it, was the most serious problem faced by the company. The marketing department experienced the problem when trying to explain the value of a component to potential customers. The development team, on the other hand, experienced the problem when changing a released component in any fashion such as porting it to a new machine, repairing a defect, or extending it with new functionality. Without a mechanism for precisely describing the behavior of classes, it became difficult to make the commercial sale of reusable classes viable:

> "Without tools to express the old specification independently from the new and then determine if the old specification is intact while independently testing the new one, development quickly slows to a crawl. All available resources become consumed in quality assurance." [Cox90]

Organizing reusable classes according to the subtype hierarchy provides a view of the classes which is more logical and much more useful to clients. However, behavioral subtyping relations between classes cannot be deduced without a precise specification of the behavior of these classes [Sny86]. Only a formal semantic specification of behavior affords the necessary precision.

Several formal definitions of subtyping based on the formal specification of classes have been proposed [BW87, Ame89, Lea90, Wil91, DL92]. A ROOCSL should provide a suitable semantics for subtyping and it should distinguish between the notions of subtyping and inheritance. Ideally, the semantics should be simple enough to permit programmers to formally or rigorously verify whether a given specification defines a subtype of another.

OOP provides the constructs for the reuse of implementation code and for the specification of behavioral relationships (i.e. subtyping) but does not provide semantic or syntactic support to distinguish between these two notions. Subtype relationships in many statically typed OOPLs such as Eiffel are based on inheritance relationships. From a reuse perspective this is undesirable because the implementation hierarchy is not, in general, useful in helping to infer behavioral relationships which exist between classes and may actually mislead programmers about the actual behavior of a class.

In the context of inheritance, the lack of semantic support for effective black box reuse in OOPLs can be remedied, once again, by using a formal specification language to specify the behavior of classes. The formal specifications can be used as a basis for organizing the classes according to a specification (i.e. subtype) hierarchy. The specification language itself can also incorporate the concept of hierarchy in such a way that the specifications of a supertype are inherited by a subtype and need not be repeated in the subtype. This allows reuse at the level of specifications, in addition to reuse at the level of code, so that new specifications can be developed incrementally based on existing specifications.

2.3 Polymorphism and Message Passing

One of the key features of OOP which facilitates software reuse is the support for polymorphism, as provided by it *message-passing* (or late binding) paradigm. The type of polymorphism provided in OOP has been extensively studied and characterized as *inclusion polymorphism* [CW85] or *subtype polymorphism* [LW90, Lea91]. It has been shown that this type of polymorphism, popularized by its ability to support code reuse in Smalltalk-80 [GR83], is fundamentally different from the *universal parametric* polymorphism found in functional languages such as ML [CW85, DT88, Lea90].

Subtype polymorphism is distinguished from other kinds of polymorphism by two features: (i) the dynamic binding of operation names to operations based on the run-time types of their arguments, and (ii) the possibility that a given expression may denote objects with different types (i.e. different subtypes) at run-time [LW90]. These properties allow programs which make use of subtype polymorphism to abstract over a set of heterogeneous objects that have similar behavior. This makes it possible to extend, or customize, an existing program by introducing new types of objects on which the program can operate. In turn, this ability to easily extend a program is very conducive to software reuse; when such an extension is made the existing code of a program is entirely reused, without modification, and only the code implementing new types needs to be added.

Polymorphic code in OOP can manipulate objects of several different types, provided that the actual (dynamic) type of an object *conforms* to the static (or nominal) type of the expression which denotes it. This capability has important consequences on the ability to support reuse. It means that new subtypes can be easily added to a system without modifying existing generic (polymorphic) modules. This is to be contrasted with non-polymorphic typed languages (e.g. C, Pascal, Modula) where the addition of new types typically involves the addition of a "type tag" on the new data type, and the verification of this type tag in a "case" statement. This latter approach is very dangerous and error-prone, as it is easy to forget updating one of the potentially many case statements. This leads to software that is difficult to reuse and maintain [Cox84].

In fact, it has been argued very convincingly that subtype polymorphism is essen-

tial for the development of extensible, reusable software components [Cox84, Cox86, Mey88]. The non-polymorphic approach to code reuse makes black-box reuse of code impossible, since to make an extension to a system (i.e. to add a new type) it is necessary to modify existing code as well as to add new code [Cox84]. In the subtype polymorphic OO approach, existing polymorphic code can be left intact since it is sufficiently generic to accommodate extensions.

However, polymorphic programs that use message passing can be difficult to reason about, because the effect of a *message send* depends upon the type of the receiving object. There may be many different operations that could be executed by a *message send* and the same piece of code may result in the execution of different method implementations during different executions. One approach to reasoning about polymorphic programs would be to perform an exhaustive case analysis by considering all possible object types that a message selector can involve. However, this approach is impractical in large systems. This approach also has the severe disadvantage that adding a new type of object to a system can require additional case analysis of message invocations in existing polymorphic code. To obtain the advantage of extensibility promised by object-oriented methods, unchanged program modules should not have to be respecified or reverified when new types of objects are added to a program. Since one does not have to update the code (because of late binding), it would be tiresome if one had to reverify the implementation of existing polymorphic code.

In the object-oriented culture, programmers have traditionally dealt with this problem by reasoning about polymorphic programs informally. Subclass relationships are used to classify the behavior of objects of different classes and a superclass acts as a representative for all its subclasses. Intuitively, programmers reason that the correctness of a polymorphic construct is dependent on the requirement that "an instance of a subclass `B` can be substituted for an instance of the parent class `A` provided that `B` *behaves like* `A` ".

In other words, if the static type of some polymorphic expression is `A`, then the intended behavior of `A` imposes some restrictions on the behavior of the objects which can be handled correctly by the polymorphic construct. These latter objects must conform to the static type's behavior. However, since OOPLs provide no means

to specify precisely what the behavior of a class is, programmers must rely on informal comments or on the implementation code to determine the behavior which must be conformed to. Moreover, the static type of a polymorphic expression often corresponds to an *abstract class* [WBJ90, KM90, Bud91] for which there is no implementation. In such a case, the programmer must rely exclusively on informal comments since there is no implementation code to fall back on.

In a small, closed system which is written entirely by one designer, it may be acceptable to document these restrictions informally or not at all. However for the purposes of black box software reuse, this approach is unacceptable. If polymorphic code is to be distributed and reused with subclasses its designers have never conceived of, it is imperative that the precise constraints on client-subclasses be documented, and that the code be guaranteed to work with any client subclass which conforms to those constraints. Otherwise, the polymorphic code cannot be reused in a black box fashion and much of the incentive for reusing this code, rather than writing it from scratch, is lost.

As discussed in section 2.2, there is no guarantee that a subclass `B` will be behaviorally compatible to its superclass `A`, unless `B` is also a semantic subtype of `A`. If a message is sent to an object whose actual type is not a semantic subtype of its nominal [Lea91] type, then there is no guarantee that the message will have the intended behavior. If an instance of subclass is used where instances of a superclass are expected, and the subclass is not a subtype, then the resulting program may behave in unexpected ways. This is further evidence that the distinction between subtypes and subclasses is not just an academic curiosity; this distinction is crucial in understanding whether a piece of polymorphic code will behave as intended for a given subtype or when a new subtype is added.

Formal specifications of the semantics of class behavior hold much potential for coping with all these problems. They make it possible to specify precisely the properties of the objects which a piece of polymorphic code is capable of working with, allowing this code to be reused in a black box fashion. As we have noted in section 2.2, subtype relationships cannot be deduced without a formal semantic description of the classes involved. However, the key to being able to reuse polymorphic code in

a black-box fashion is based on the observation that the informal notion *B behaves like A* (or *B is substitutable for A*) can be formally characterized by requiring that B be a semantic subtype of A.

Using this approach, it is possible to realize the advantages of extensibility promised by OOP. Leavens [Lea90, LW90, Lea91] has done work in this direction and has arrived at a *modular* specification and verification technique for OO programs which makes it unnecessary to respecify and reverify unchanged polymorphic code when new types are added. His verification technique is sound and complete and can be used to formally verify the correctness of an implementation involving polymorphic code with respect to its specification. However, Leaven's formal results are limited to applicative OO languages involving immutable types.

Ideally, a ROOCSL should permit modular verification, even for imperative OO languages with mutable objects such as C++. The existence of such a technique would be evidence that the underlying specification framework is sound and precise. In practice, such techniques would also enable developers to reason rigorously about the behavior of polymorphic programs, without having to resort to a fully formal verification approach.

2.4 Subsystems and Frameworks

> "The big lie of object-oriented programming is that objects provide encapsulation. In order to accomplish anything, objects must interact with each other in complex ways, and understanding these interactions can be difficult." [Hog91]

Subsystems. Recent literature on object-oriented software development has begun to recognize the importance of inter-object behavior in OO systems and this has been expressed in terms of *subsystems and collaborations*, *responsibilities* and *mechanisms* [BC89, WBJ90, WBWW90, G.B91]. It is clear that an understanding of inter-object behavior is necessary to understand the design of an object-oriented system. In all but the simplest cases (e.g. ADTs such as stacks, lists, and tables) objects do not act in isolation from one another but are part of a *subsystem* [WBJ90] of classes which cooperate to fulfill a larger purpose. The behavior of classes involved in such cooperations can be fully understood only in the context of their relationship to the other classes in the subsystem to which they belong.

From the point of view of reuse, subsystems are a very attractive concept since they represent a higher level unit of reuse in comparison to stand-alone classes. Reusing a subsystem implies reusing the design and implementation of a group of classes and their interactions, rather than just reusing the implementation of a single class. In fact, because of the central importance of groups of cooperating objects in OO systems it can be argued that a class, in general, represents too fine grained a construct to adequately serve the purposes of a reusable object-oriented component [WBJ90].

Interobject interactions between the classes in a subsystem must be well understood before a subsystem can be reused. Current OOPLs provide no support for the specification and abstraction of the interactions between objects; they only provide the syntactic mechanism for implementing subsystems. The existence of inter-object behavior in a system, and in particular the *behavioral dependencies* [HHG90] which they imply, cannot be easily inferred. Instead, they are spread across many class definitions in method implementations.

For the same reasons discussed in section 2.1, it is desirable to use formal specifications, rather than informal specifications or the implementation code itself, to describe the inter-object behavior in a subsystem. It is also important that the interactions be described in a black-box manner, for the same reasons as it is necessary to describe class behavior in a black-box manner.

[HHG90] presents pioneering work on the specification of inter-object behavior. The approach presented in [HHG90] appears promising but no formal syntax and semantics has been given for the specification language used to specify *contracts*. In addition, the work does not consider the specification of the behavior of individual classes, but only class interactions.

To obtain an integrated specification language which can specify both the behavior of individual classes as well as inter-object behavior, it would be necessary to adapt the approach of [HHG90], or another approach capable of achieving the same result, and appropriately combine it with a specification language capable of specifying the behavior of individual classes. An ideal ROOCSL should provide such an integrated capability.

Frameworks. Frameworks are similar to subsystems and go a step further in the direction of promoting reusability in OOP [Deu87, Deu89, WBJ90, HHG90]. They are an attempt at providing a means to reuse entire designs and implementations for a given application domain (e.g. user interfaces [JM89], VLSI routing algorithms [SA89], and operating systems [RC89]). Frameworks not only provide all the basic functionality required for a given application domain, but also the flexibility of being able to customize and refine most of this functionality to suit the needs of a particular application.

Frameworks are characterized by the following properties [WBJ90]:

- They consist of a collection of abstract and concrete classes. Part of the definition of each abstract class is its responsibilities.
- They contain a description of the collaborations between the objects in its abstract classes.

- They are designed to be refined. This is the principle difference between frameworks and subsystems.
- They are more than a collection of classes, but include instructions for making new subclasses and for configuring applications.
- They are used by writing a program which configures together a set of objects belonging to the classes in the framework. This is accomplished by creating a set of objects and performing operations to interconnect them.
- A framework can be refined by changing the configuration of its components or by creating new kinds of components (i.e. new subclasses of existing classes).
- Even when new subclasses are needed, these are very easy to produce because the abstract superclasses provide their design and much of their code.

Because of their ability to be refined and customized, frameworks offer further opportunities for reuse. However, this comes at a price. Reusing frameworks introduces new complications not associated with the reuse of subsystems. To obtain a given behavior it is necessary to determine:

- From which existing classes should new classes be derived ?
- Which methods should be overridden in those new classes ?
- Which new objects should be created ?
- How should objects be initially interconnected ?

For the effective reuse of frameworks, it is necessary to be able to answer such questions with minimal effort and without having to inspect source code. The work presented in [HHG90] shows how this can be accomplished. *Contracts* are introduced to specify behavioral compositions and the obligations on participating objects. *Conformance declarations* are used to specify how specific classes, and thus their instances, support the role and obligations of participants in a contract. These two mechanisms combined allow for the explicit representation of application frameworks. An ideal

ROOCSL should provide such capabilities for specifying frameworks as part of its integrated specification approach.

2.5 Evaluating Completeness of Specifications

We have already stated repeatedly that a ROOCSL must be able to specify the behavior of classes. However, up to now we have side-stepped the issue of defining what we mean by the behavior of a class. Without a precise definition of *class behavior*, there can be no systematic methodology for evaluating the completeness of a class interface specification. Such an evaluation is required to ensure that a specification provides a developer with all the information required to reuse a class in a truly black box fashion.

In the absence of a systematic method for evaluating the completeness of a class specification, the evaluation becomes a subjective matter, leaving it to the designer/implementor's discretion what the specification should and should not state. This, in turn, can very adversely affect the ability with which a class can be reused. Incomplete documentation may confuse, or worse, mislead a prospective reuser as to how a class should be used. From the reuser's point of view, having to refer to an incomplete specification is just as undesirable as attempting to reuse a class whose implementation is defective.

The availability of a methodology for evaluating the completeness of a specification also makes it possible to formally assess the expressiveness of a candidate ROOCSL. Ideally, a ROOCSL should be able to provide a complete specification of any arbitrary class.

2.6 Verifying Correctness

With reusable components acquired from many different places, a designer must be especially careful not only to have a precise, unambiguous understanding of what each component is supposed to do, but also some assurance that it will do that. Much of the advantage of reuse is lost if it is necessary to test each component as carefully as if it had been developed from scratch. In this context also, formal specifications provide an advantage over natural language specifications since they provide a very precise description of the intended behavior of a given module against which the implementation can be verified.

The correctness of a class's implementation with respect to its specification can be formally demonstrated using the approach described in [Hoa72, Ame89]. The ease with which such a formal proof of correctness can be developed depends on the complexity of the semantics of the programming language, the semantics of the specification language, and the complexity of the class's implementation.

In an industrial context, it may not always be feasible or practical to develop formal proofs of correctness for class implementations. The availability of complete formal behavioral specifications can still lead to significant improvements in the correctness, and hence the quality, of such classes. Black box testing and code walkthroughs can be used to verify the correctness of the implementations of classes against these formal specifications. This represents a pragmatic engineering compromise for industry and could be used to "certify" that the implementation of a class is correct within a certain tolerance using statistical quality control techniques [CM90]. A useful compromise between these two approaches would be to formally prove the correctness of certain critical components of a system and then use inspection, black box testing and statistical quality control for the remaining components.

Testing or inspecting the implementation of a class against a formal specification is much more meaningful than it is against an informal specification. The formal specification provides a precise, unambiguous description of the behavior which a candidate implementation should satisfy. In contrast, an informal specification does not provide an adequate gauge against which an implementation can be evaluated. The specification is potentially vague, incomplete and ambiguous, leaving it up to

the tester to interpret just what the behavior of a class should be. This later impacts the effectiveness with which developers can correctly reuse such a class. Without a precise specification of a component's intended behavior, determining whether or not a certain class behavior is a "bug" can be quite a subjective matter. It is difficult to tell what is and isn't an error when it isn't clear what is being said in the specification. As a result, the correctness or reliability of a component can only be as good as its specification.

2.7 Desirable Properties of a ROOCSL

In summary, an ideal ROOCSL should have the following properties:

(i) It should have a formal syntax.

(ii) It should have a formal semantics.

(iii) It should provide the capability to write specifications which are implementation independent (2.1).

(iv) It should define an appropriate semantics for determining subtype relations and make a distinction between subtyping and implementation inheritance (2.2).

(v) It should permit inheritance of specifications (2.2).

(vi) It should permit modular specification (2.3).

(vii) It should provide a methodology or proof procedure to verify the correctness of a candidate implementation in a modular fashion (2.6,2.3).

(viii) It should be able to specify interobject behavior so as to permit the specification of reusable subsystems and frameworks (2.4).

(ix) It should provide a methodology for evaluating the completeness of specifications (2.5).

(x) It should be sufficiently expressive to write complete specifications of any arbitrary class (2.5).

2.8 Scope of This Thesis

For the remainder of this thesis, we will deal with the identification and adaptation of a suitable formal ROOCSL for C++ as well as with the evaluation of the completeness of specifications written in the selected ROOCSL. However, we will restrict our attention to class designs which do not involve interobject behavior.

We will consider the distinction between subtyping and implementation inheritance and define an appropriate semantics for determining subtyping relationships for the ROOCSL of our choice. This permits modular specification but not modular verification since we do not consider formal verification in this thesis.

For code reuse through inheritance, we assume a disciplined approach in which heirs cannot directly access the instance variables of their parent classes and access them only through the use of methods. This paradigm is supported in languages such as CommonObjects, Self, and Trellis/Owl and can also be enforced in C++ by using *private* data members exclusively. As it has been explained in [WBW89], this disciplined approach makes code reuse more effective because directly referring to the variables of a class severely limits the ability to refine subclasses. This behavioral approach to reuse via inheritance also preserves a black box approach to reuse where it is unnecessary to read the source code of a class to reuse it.

The issues we have chosen to consider in this thesis constitute a first logical step towards the larger goal of developing a ROOCSL having all the properties identified in the previous section. These issues must be tackled before the problem of providing the remaining properties can be addressed.

Chapter 3

Choice of A Specification Language

3.1 Survey of Module Specification Languages

In this section we survey different formal specification languages which have been proposed for the specification of object-oriented modules and assess to what extent each of the approaches satisfies the requirements we have outlined in section 2.7. The results of this analysis are summarized in the table of figure 3.1. Each row in the table corresponds to a property identified in section 2.7. Each column in the table refers to one of the the specification approaches described below.

3.1.1 Fresco

Fresco [Wil91, Wil92b, Wil92c] is a Smalltalk-based interactive environment supporting the specification and proven development of reusable Smalltalk software components. Fresco specifications are model-oriented and based on VDM. The verification of components is accomplished using a mixture of formal proofs and informal arguments.

In Fresco, no strong distinction is made between type specifications (behavioral descriptions of classes) and class specifications (specifications which describe the implementations of classes). One syntactic framework, the *type/class description* (TCD) serves both purposes. A TCD can describe a specification or an implementation or, most commonly, a mixture of the two. As a result, model variables may be hypothetical (i.e. abstract) or they may correspond to actual program variables. Specification assertions may also be interweaved within the implementation code. The code is written in a variant of Smalltalk which has been modified for a more convenient integration with specification constructs.

According to the designer of Fresco, the lack of a clean separation between specifi-

cations and implementations increases the flexibility and ease of program specification and verification. However, this lack of separation also has the very undesirable effect that specifications are implementation dependent, making them ill-suited for reuse.

3.1.2 Eiffel

The object-oriented language Eiffel [Mey88] provides constructs for writing assertions which attempt to specify the behavior of Eiffel classes. The assertions in each class consist of boolean expressions written using the same syntax as program statements, and are used to write pre- and post-conditions as well as a class invariant.

Since there are no abstract values in the Eiffel specification sublanguage, specifications refer to the instance variables and operations of a class. Implementational specifications are used because the emphasis is more on providing runtime checks for debugging and on providing a precise description of a class's implementation. Unfortunately, this makes the resulting specifications implementation dependent and unsuitable for describing reusable classes in a black box fashion.

Strictly speaking, it is possible to write specifications which are implementation independent in Eiffel. This involves referring only to the methods of a class in a specification. The use of this technique is advocated for the specification of *deferred* (i.e. abstract) classes but, unfortunately, as Meyer himself acknowledges [Mey88, p. 239] this style of specification is not even sufficiently expressive to specify a stack class. This is due to the absence of universal quantification and model variables in the Eiffel assertion language.

3.1.3 Anna

Anna [Luc91, LST91], is an extension to the Ada language which adds facilities that are useful for specifying the intended behavior of Ada programs. The extensions to Ada take the form of *annotations* which appear as formal comments within the Ada source text. The textual position of formal comments is used to relate the specifications written in the formal comments to program fragments.

Anna annotations can be used to specify both the interfaces and implementations of Ada *packages* (modules). There are two principal styles for specifying the behavior

of Ada packages using Anna. The first involves the use of functions, called *basic concepts*, which are used to define the behavior of the visible operations in a package. These operators are assumed to be so basic that their semantics can be informally understood with clarity, and hence no specification is provided for them. This assumption has the severe disadvantage that the specification is not completely formal, although deemed acceptable to the designers of Anna because the implementation of these functions can be executed to "debug" the specifications. For the purposes of specifying the interface of reusable module interfaces this approach is not adequate. Such specifications do not provide all the information required to reuse the module in a black box fashion.

The second approach for specifying the behavior of Ada packages involves the use of *virtual theory packages*. This is almost identical to the Larch [GHW85a] approach to module specification. Theory packages consist of equational axioms, as in an algebraic specification, which define the properties of types. The specification of an *actual package* imports theory packages which provide a precise semantics for the basic concepts required in the specification. This is analogous to inclusion of LSL traits in Larch interface specifications.

Because Ada does not support inheritance, Anna does not provide any features relating to semantic subtyping and inheritance of specifications.

3.1.4 A++

A++ (annotated C++) [CL90a, CL90b] is both an annotation formalism and a proposed C++ CASE tool supporting object-oriented annotations for C++. The annotation formalism contains features inspired by Anna and Eiffel, but with constructs specifically designed around C++. Some of the goals of A++ are: (i) To increase code safety, by formally verifying code fragments against their annotations (ii) To increase code clarity by replacing explicit error checks in the code with higher-level annotations which automatically generate the corresponding checks at run-time when necessary. (iii) To increase code performance by using the annotations as an aid to optimization.

As in Anna, annotations are used to specify both class implementations and inter

faces. Class interfaces are specified by giving pre- and post-conditions for each of the methods. Also as in Anna and Eiffel abstract model variables are not used and the assertions can refer only to the operators and variables of the class. As in Anna, the semantics of "basic concepts" need not be explicitly provided by the specification and can be left undefined resulting in a specification which is not semantically complete.

A++ also provides some support to assist one in attempting to fully characterize the semantics of "basic concepts". However, A++ differs from Anna in the manner in which this support is provided. Rather than following the Larch approach, A++ provides constructs to emulate an algebraic specification style in addition to the available axiomatic specification style.

Both specification styles can be used within the same specification. There is no clean two-tiered separation as in Larch or Anna. There is no distinction between algebraic operators and methods, these are taken to be one and the same. Moreover, the emulation of the algebraic style is very limited. Relationships amongst methods are specified by giving axioms which apply to a sequence of operators. This approach lacks the ability to functionally compose operators, as methods of a class are typically not functions since they mutate the receiving object. As in Eiffel, this specification approach appears unable to fully characterize the behavior of certain simple classes like Stack.

3.1.5 Larch/C++

Larch/C++ [LC92] is part of the Larch [GHW85b, Win83, Win87] family of specification languages. Larch languages are formal specification languages geared towards the specification of the observable effects of program modules, particularly modules which implement abstract data types. Larch provides a two-tiered approach to specification:

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

is designed for a specific programming language. The LIL for C++ is called Larch/C++.

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

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

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

Thus, a Larch specification of a program module consists of two distinct specification components: an LIL interface specification and one or more LSL traits which describe the underlying traits. The philosophy behind this two-tiered approach is summarized in [Win87] as:

> We believe that for specifications of program modules, the environment in which a module is embedded, and hence the nature of its observable behavior, is likely to depend in fundamental ways on the semantic primitives of the programming languages. ... Thus we intentionally make an interface language dependent on a target programming language, and keep the shared language independent of any programming language. To capitalize on our separation of a specification into two tiers, we isolated programming language dependent issues — such as side effects, error handling, and resource allocation — into the interface language component of a specification.

As a result of this philosophy, Larch's two-tiered approach makes it possible to express programming language dependent module properties using a syntax and semantics which reflects the underlying programming language. This is achieved by providing constructs for expressing programming language dependent module properties such as parameter passing, side effects, exceptions, and concurrency using the

syntax and semantics of the underlying programming language.

The syntax and semantics of each LIL also takes into account the syntax and semantics of the type system and the memory model of the underlying programming language. For example, the syntax and semantics of C pointers, arrays, structs and other basic data types are built into Larch/C++ and C operators such as '*' and '→' are also built-in. As a result, the formal parameters in the specification of an operation can be referenced using the same syntax and semantics as in the underlying language. This built-in support facilitates the task of writing formal specifications for class interfaces.

Like other Larch languages, Larch/C++ has the advantage that interface specifications are implementation independent. This is because they do not need to refer to the instance variables of a class to specify the semantics of its interface. Instead, the interface specifications refer to abstract sorts whose properties are defined axiomatically in traits.

3.1.6 Larch/Smalltalk

Larch/Smalltalk [Che91] is a Larch interface specification language for Smalltalk. It follows the same two-tiered approach as other Larch languages, but the interface language has been tailored to Smalltalk.

The design for Larch/Smalltalk presented in [Che91] provides a formal syntax for the language but the semantics is presented informally. Subtype relations are included as part of the design of Larch/Smalltalk, but these are based only on syntactic conditions. No semantic definition of subtyping is provided. As well, no mechanism for specification inheritance is clearly defined.

Like other Larch languages, Larch/Smalltalk has the important benefit of providing abstract, implementation independent specifications.

3.1.7 LM3

LM3 [Jon91] is a Larch interface specification language for the Modula-3 programming language [CDG+89]. The design of LM3 described in [Jon91] provides a formal syntax but no formal semantics. Inheritance of specifications and a semantic definition of

subtyping are not treated either. Like other Larch languages, LM3 has the important benefit of providing abstract, implementation independent specifications.

3.2 Justification for Choosing Larch/C++

Based on the considerations discussed in the previous section, the data in table 3.1, and the design principles presented in section 2.7, Larch/C++ can be seen to be the preferred choice for specifying the behavior of C++ class interfaces. In this section, we provide a detailed analysis justifying this choice.

In section 2.1 it was emphasized that the ability to provide implementation independent specifications is of utmost importance for a ROOCSL. The ability to write implementation independent specifications is one of the most important considerations in selecting a ROOCSL for C++. In contrast to the Fresco and Eiffel approaches, Larch/C++ can be used to write implementation independent specifications. Although A++ and Eiffel can be used to write implementation independent specifications, they are not sufficiently expressive to express even simple examples like a Stack class. The Larch two-tiered approach as used in Larch/C++, Larch/Smalltalk, LM3 and Anna which satisfies the implementation independent requirement while providing a sufficiently expressive language.

Larch/C++ and A++ are the languages which provides built-in syntactic and semantic support for specifying C++ class interfaces. Using one of the other approaches (e.g. Eiffel, Fresco) to specify C++ class interfaces would require extending and adapting the syntax and semantics of these approaches for C++. Such an adaptation is by no means trivial and would require a considerable amount of effort, almost as much as designing a specification language for C++ from scratch.

An alternative to using one of the languages discussed in the previous section on which to base a ROOCSL would have been to use a general purpose specification language such as Z or VDM. In contrast to Larch/C++, such *universal* specification languages do not offer the built-in support for C++ syntax and semantics. The result is that Larch module interface specifications have the potential of being shorter than specifications written in a universal specification language. They can also be generally clearer and more natural to developers who are accustomed to the syntax

feature	Fresco	Eiffel	Anna	A++	Larch/ C++	Larch/ Smalltalk	LM3
Formal Syntax	Y	Y	Y	Y	Y	Y	Y
Formal Semantics	Y	N	N	N	P^a	N	N
Impl. Indep.	N	N	P^b	P^c	Y^d	Y	Y
Subtyping	Y	Y	–	N	Y^e	N	N
Inher. of Specs	P^f	Y	–	Y	Y^g	N	N
Modular Spec.	N	N	–	N	Y	N	N
Modular Verif.	N	N	–	N	N	N	N
Interobj. Beh.	N	N	N	N	N	N	N
Completeness	N	N	N	N	Y^h	N	N
Expressiveness	Y^i	N	P^{ij}	N	Y^i	Y^i	Y^i

Y : Feature is supported
P : Feature is partially supported
N : Feature is not supported
– : Feature is not applicable

a: The formal semantics is currently being defined.
b: Only the package interface component of nna is implementation independent.
c: Only the behavioral specification portion of A++ is implementation independent.
d: Excluding the portion of Larch/C++ used to generate implementation header files (i.e. private member and function declarations).
e: No specific formal semantics for subtyping was provided in the preliminary design of Larch/C++ [LC'92] The extensions proposed in this thesis (chapter 6) provide one.
f Because specifications may refer to instance variables in Fresco, the inheritance of specifications will not be well defined in cases where the instances variables are not used by overriden methods of a subclass
g: The work presented in this thesis (chapter 6) extends the specification inheritance mechanism presented in the preliminary design of Larch/C++ [LC92] and remedies several problems identified with that preliminary proposal.
h: The work presented in this thesis (chapter 3 and 4) provides a methodology for evaluating the completeness of Larch/C++ specifications.
i: This is only a conjecture It would be necessary to prove it formally to make this a definite claim. However, to state the negative can be done with certainty since some examples exist which disprove this.
j: Applies only to the Anna approach which uses theory packages to define basic concepts.

Figure 3.1: Support for Properties in Different Specification Approaches

and semantics of C++. This avoids having to learn an entirely new syntax.

One of the advantages which has been attributed to model-oriented languages such as Z and VDM is their ease of use, intuitiveness, and expressiveness. Experience in the formal methods community suggests that model-oriented specifications are easier to read and write than other types of specifications such as algebraic specifications [Wil91]. Strictly speaking, Larch is a definitional language with equational axioms in the LSL-tier and Hoare-style axiomatic specifications in the interface tier. However, its two-tiered approach makes it as intuitive and expressive as other model-oriented specification languages such as Z and VDM. The abstractions defined by traits play the same role as the abstract models (e.g. sets, maps) in these languages. From this point of view, the interface tier of Larch is very similar to a model-oriented specification language such as VDM.

The Larch two-tiered approach also has the important advantage of providing a useful separation of concerns between the two tiers. Mathematical abstractions are defined in the LSL tier while programming language specific issues are specified in the LIL tier. This makes it possible to reuse LSL specifications in different LIL module specifications. Many common traits are available in the LSL Handbook [GHW85b]. Larch also has a powerful theorem prover, LP (Larch Prover), which can be used to perform semantic analysis of LSL traits [GG89, GG90, GGH90, GGH90].

In addition to the many advantages of Larch/C++ discussed above, the work presented in this thesis also proposes the following extensions so as to further increase the usefulness of Larch/C++ as a ROOCSL for C++:

(i) A definition of completeness for class interface specifications and a method for evaluating the completeness of Larch/C++ specifications.

(ii) A proof-theoretic definition for verifying subtyping relations as well as a methodology for applying the definition.

(iii) A mechanism which permits the inheritance of specifications.

3.3 A Larch/C++ Tutorial

In this section we provide a brief introduction to LSL and to Larch/C++. Instead of presenting a formal description of Larch, we focus on providing an intuitive understanding of Larch/C++ specifications through some simple examples without going through an exhaustive description of LSL and Larch/C++. We present only the features which are required to understand the examples discussed in this thesis. For a more thorough and rigourous treatment of the semantics of Larch interface languages and of LSL the interested reader is referred to [Win83, Win87, GHM91, GH91] and for more information on Larch/C++ the reader is referred to [LC92]. The summary below is based primarily on [Win87] and [LC92].

As we mentioned in the previous section, Larch is a property-oriented specification language that combines both axiomatic and algebraic specifications into a two-tiered specification. The axiomatic component (LIL) specifies the state-dependent behavior (for example, side effects and exceptional termination) of programs. The algebraic component specifies state-independent properties of the data accessed by programs.

The unit of encapsulation in LSL is the trait. Figure 3.2 shows an LSL trait which specifies the properties of a set. This example is similar to a conventional algebraic specification in the style of [EM85, GH78, Bid88]. A trait contains a set of operator declarations, or signatures, which follows the **introduces** keyword, and a set of equational axioms, which follows the **asserts** keyword. A signature consists of the sorts and the domain and range of an operator. The equational axioms specifies a set of constraints on the defined operators.

There are a few notable differences between Larch traits and conventional algebraic specifications:

(i) The name of a trait (e.g. SetTrait) is distinct from the name of all sort and operator identifiers defined in the trait (e.g. Set).

(ii) The names of sorts are not explicitly declared. They are implicitly declared by appearing in a signature.

(iii) Larch makes use of the clauses **partitioned by** and **generated by** to increase

the expressive power of traits.

(iv) The semantics of = and == are exactly the same in LSL; only their syntactic precedence differs to ensure that expressions parse in an expected manner without having to use parentheses. The operator = binds more tightly than the operator ==.

(v) Equations of the form *term == true* can be abbreviated to *term*; thus the third equation in figure 3.2 is an abbreviation for *subset(emptyset, s) == true* and the first equation is an abbreviation for *not(member(x,emptyset)) == true*.

(vi) The semantics of Larch traits is based on multisorted first order logic with equality rather than on an initial, final or loose algebra semantics used by other algebraic specification languages [EM85, GTW78, ST87, Bid88]. Each trait denotes a theory[1] in multisorted first-order logic with equality. The theory contains each of the trait's equations, the conventional axioms of first order logic with equality, everything which follows from them, and nothing else. This means that the formulas in the theory follow only from the presence of assertions in the trait — never from their absence. The theory of a trait can also be strengthened by adding a **generated by** or a **partitioned by** clause.

(vii) A trait definition need not correspond to an abstract data type (ADT) definition since an LSL trait can define any arbitrary theory of multisorted first-order equational logic. For example, a trait can be used to define the first order theory of mathematical abstractions such as equivalence relations which do not correspond to abstract data types.

(viii) LSL traits can be augmented with checkable redundancies in order to verify whether intended consequences actually follow from the axioms of a trait. These checkable redundancies are specified in the form of assertions which are included in the **implies** clause of a trait and can be verified using LP.

In the trait of figure 3.2, the **generated by** clause states that all values of the sort Set can be represented by terms composed solely of the two operator symbols

[1] A *theory* is a set of logic formulas having no free variables

emptyset and *insert*. In other words, saying that sort *S* is **generated by** a set of operators, Ops, asserts that each term of sort S is equal to a term whose outermost operators is in Ops. The operators in the set Ops are referred to as the *generators* of the sort S. A **generated by** clause strengthens the theory of a trait by adding an inductive rule of inference which can be used to prove properties which hold for all Set values.

For LSL traits which define an ADT, there is a sort referred to as the *distinguished sort*, sometimes also called the *principal sort* or *data sort*. For example, for the trait of figure 3.2 the distinguished sort is **Set**, which is the sort corresponding to the set ADT.

The **partitioned by** clause provides additional equivalences between terms. Intuitively, it states that two terms are equal if they cannot be distinguished by any of the functions listed in the clause. For the Set example, this property could be used to show that order of insertion in the set is commutative. That is, it could be shown that the terms *insert(i, insert(j, s))* and *insert(j, insert(i, s))* are equal for all values of i, j: Int and s: Int.

The **exempting** clause documents the absence of right-hand sides of equations for delete(x, emptyset); this incompleteness is dealt with in the interface specification. The **converts** and **exempting** clauses together provide a way to state that this algebraic specification is sufficiently complete [GH78]. Intuitively, what the **converts** and **exempting** clauses are saying is the following: "the specification of the operators *delete*, *unionn*[2], *delete*, *inter*, *member*, *subset*, *size*, *isEmpty* is complete in the sense that any terms involving these operators can be reduced to terms not involving these operators. The only exception to this rule is for terms which involve a subterm of the form *delete(x, emptyset)*. For example, any term *t* whose outermost operator is *unionn* or *inter* can be reduced to a term *s* involving only *emptyset* and *insert*, provided that *t* has no subterms of the form *delete(x, emptyset)*.

LSL also provides a ways of putting traits together, one of which is through an **includes** clause. A trait that includes another trait is textually expanded to contain all operator declarations, **constrains** clauses, **generated by** clauses, and axioms of

[2]We use the identifier *unionn* rather than *union* because *union* is a reserved LSL word.

the included trait. The meaning of the including trait is the meaning of the textually expanded trait. In the Set example, the signature and meaning of the '+' operator comes from the `Integer` trait. Boolean operators (true, false, not, $\vee$, $\wedge$, $\rightarrow$, and $\leftrightarrow$) as well as some heavily overloaded operators (if-then-else, =) are built into the language; that is, traits defining these operators are implicitly included in every trait.

Figures 3.3 and 3.4 show a Larch/C++ interface specification for IntSet, a class which implements set of integers. For each IntSet operation, the specification consists of a *header* and a *body*. The header specifies the name of the operation, the names and types of the parameters, as well as the return type, and uses exactly the same notation as in C++. The body of the specification consists of an **ensures** clause as well optional **requires** and **modifies** clauses.

The **requires** and **ensures** clauses specify the pre- and post-condition respectively. For each IntSet operation, the **requires** and **ensures** clauses specify the pre- and post-condition respectively. The identifier *self* in the pre- and post-condition assertions denotes the object which receives the message corresponding to the specified method. The **modifies** clause lists those objects whose value may change as the result of executing the operation. Hence, for example, *add* and *remove* are allowed to change the state of an IntSet object but *size* and *isIn* are not. An omitted **requires** clause is interpreted to mean "**requires** *true*" and an omitted **modifies** clause is interpreted to mean that no objects are modified by the corresponding method (neither *self*, nor any parameter objects).

The link between the IntSet interface specification and the `SetTrait` LSL specification is indicated by the clause **uses** *StackTrait(IntSet for Set, int for E)*. The *used trait* IntSet provides the names and meaning of the operators *emptyset*, *insert*, *delete*, *unionn*, *intersect*, *member*, *subset*, *size*, and *isEmpty* as well as the meaning of the equality symbol, =, which are referred to in the pre- and post-conditions of IntSet's method specifications. The uses clause also specifies the *type to sort* mapping which indicates which abstract values the objects involved in the specification (e.g. *self* and parameter objects) can range over. For example, the abstract values of `IntStack` objects are represented by terms of the sort `Set`.

Associated with each member function specification is the predicate over two states,

$$\text{Pre} \rightarrow (\text{Modifies} \wedge \text{Post})$$

where Pre and Post are the assertions in the **requires** and **ensures** clauses, respectively, and Modifies is the implicit assertion associated with the **modifies** clause. The clause **modifies** x_1, ..., x_n implicitly asserts that the method changes the value of no object in the environment of the caller except possibly some subset of $[x_1, \ldots, x_n]$.

It is important to note the following points about a Larch/C++ class interface specification:

(i) *self* is an abbreviation for *(*this)*. In C++, *this* represents a pointer to the receiving object so that self = (*this) is a name representing the receiving object itself.

(ii) A distinction is made between an object and its value by using a plain object identifier (e.g. `s`) to denote an object, and a superscripted object identifier (e.g. s' or s^) to denote its value in a state.

(iii) The operators ^ and ' are used to extract values from objects. An object identifier superscripted by ^ denotes an object's initial value and an object superscripted by ' denotes its final value. This is similar to the use of superscripts and decorations in VDM and Z. Thus, the assertion self' = self^ says that the value of the object `self` is left unchanged.

(iv) The headers of a Larch/C++ member function specification are deliberately chosen to be exactly the same as C++ member function prototypes.

(v) The **modifies** clause is an assertion whose meaning is given by considering it to be conjoined to the postcondition. It is syntactically separated from the postcondition to highlight a procedure's potential side effect on the values of objects. It is an example of a *special assertion*. Each Larch interface language comes equipped with its own set of special assertions. For example, in Larch/C and Larch/C++, there is a keyword `trashed` which is used to indicate dealloca-

tion of component objects in the destructor of a class[3]. These special assertions can be regarded as syntactic sugar for first-order assertions about state.

3.3.1 Inheritance in Larch/C++

For the purposes of type checking in C++, each class name is the name of a type. The only way to make S a subtype of T is to declare the class S a public subclass of T. It is possible to have a purely implementation relationship between S and T by declaring S to be a private or protected subclass of T. However, it is not possible to have a subtype relationship without a subclass relationship. As a result, the designers of Larch/C++ decided to equate the notions of public subclass and subtype. Syntactically, this decision makes sense because the C++ type system allows a public subclass instance to be substituted anywhere a base class instance is expected. However, semantically, there is no guarantee that a public subclass will actually be behaviorally compatible to its superclass. The use of precise specifications and of a semantic subtyping relation based on these specifications can be used to ensure such behavioral compatibility.

[3]Unlike in Larch/CLU where all special assertions have been given a fully formal semantics [Win83], the formal semantics for the **trashed** keyword has not been given a formal semantics yet in Larch/C and Larch/C++.

```
IntSetTrait(S, E): trait
  includes Integer
  introduces
    emptyset: → Set
    insert: E, Set → Set
    delete: E, Set → Set
    unionn: Set, Set → Set
    inter: Set, Set → Set
    member: E, Set → Bool
    subset: Set, Set → Bool
    size: Set → Int
    isEmpty: Set → Bool
  asserts
    Set generated by emptyset, insert
    Set partitioned by member
    ∀ x, y : E, s, t : Set
      not(member(x, emptyset))
      member(x, insert(y,s)) == if (x = y) then true else member(x,s)
      subset(emptyset, s)
      subset(s, emptyset) == (size(s) = 0)
      subset(insert(x,s), t) == if member(x,t) then subset(s,t) else false
      size(emptyset) == 0
      size(insert(x,s)) == if member(x,s) then size(s) else 1 + size(s)
      size(unionn(s,t)) == (size(s) + size(t)) - size(inter(s,t))
      delete(x, insert(y,s)) == if (x = y) then s else insert(y, delete(x,s))
      unionn(s, emptyset) == s
      unionn(emptyset, s) == s
      unionn(insert(x,s), insert(y,t)) == if (x = y) then insert(x, unionn(s,t))
                                          else insert(x, insert(y, unionn(s,t)))
      inter(s, emptyset) == emptyset
      inter(s, insert(y,t)) == if member(y,s) then insert(y, inter(s,t)) else inter(s,t)
      isEmpty(s) == size(s) = 0
  implies
    converts delete, unionn, intersect, member, subset, size, isEmpty
    exempting ∀ i : E
      delete(i, emptyset)
```

Figure 3.2: LSL Trait for Set

```
class IntSet
{
    uses IntSetTrait(IntSet for S);
public:
    IntSet()
    {
        modifies self;
        ensures self'= emptyset;
    }
    ~IntSet()
    {
        modifies self;
        ensures trashed(self);
    }
    int size()
    {
        ensures result = size(self^);
    }
    void add(int i)
    {
        modifies self;
        ensures self'= insert(i, self^);
    }
    void remove(int i)
    {
        requires ¬isEmpty(self^);
        modifies self;
        ensures self'= delete(i, self^);
    }
```

Figure 3.3: Larch/C++ Specification for Set (Part I of II)

```
    IntSet* intersect(IntSet* pS)
    {
        ensures (*result) = inter(self^, (*pS)^);
    }
    IntSet* unionn(IntSet* pS)
    {
        ensures (*result) = unionn(self^, (*pS)^);
    }
    bool isIn(int x)
    {
        ensures result = member(x, self^);
    }
    bool isEmpty()
    {
        ensures result = isEmpty(self^);
    }
};
```

Figure 3.4: Larch/C++ Specification for Set (part II of II)

Chapter 4

Formal Definition of Class Behavior and Complete Specification

As we have mentioned previously, the availability of a *complete* class interface specification is of utmost importance in ensuring the effective reuse of a class. However, there are many possible sources and types of *incompleteness* in a specification [AK92]. We have stated informally that a complete specification should provide a developer with all the information required to reuse a class in a black-box fashion. In this section, we will make precise and formalize this notion of completeness in the context of software reuse. This will then make it possible to develop a criterion for evaluating the completeness of class interface specifications.

For the purposes of software reuse, the principal type of incompleteness which is important to avoid is *partial specification* [AK92]. Such behavior results when a specification does not fully characterize the behavior of the entity which is being specified. For a class interface specification, the specificand of interest is the behavior of a class. As a result, we need to formalize the notion of *behavior of a class* in order to formalize the notion of completeness in the context of software reuse.

The definition of class behavior presented here is inspired by the trace assertion specification technique of [BP86]. Our approach makes it possible to define the behavior of a class in a manner which is both independent of a class's implementation and independent of the choice of a particular specification language.

For clarity of exposition, we motivate the definition by appealing to the reader's intuition. Starting from an informal definition, we successively refine the definition until a concise and precise formalization of class behavior is obtained.

4.1 Informal Definition of Class Behavior

In accordance with the philosophy of the object-oriented paradigm, we define the behavior of a class independently of its implementation details. We view a class as a black box whose behavior can be perceived only by sending messages corresponding to methods defined in the public interface. Operations and data structures which are internal to the class implementation are not the client's concern and should not be referred to when describing the class behavior.

Informally, the externally observable behavior of a class is perceived by an object's response to messages and by the constraints which apply to messages which can be sent to this object. In formalizing this notion, the following questions arise:

- How does an object respond to messages ?
- How are these responses perceived ?
- What constraints apply to the invocation of methods ?

Informally, these questions can be answered as follows:

- An object responds to a message by performing one or more of the following actions:
  - Modifying its internal state variables.
  - Modifying the state of the external environment (e.g. graphical display, database, communication network).
  - Returning a value.

  It should be noted that there is another possible action which an object can perform in response to a message. In response to a message m_1, an object **a** of class `A` may send a message m_2 to some other object **b** of a class `B`.

  In many cases, the invocation of m_2 will be an implementation detail of class `A` which should be transparent to the user of m_1 to preserve the encapsulation of `A`. In such cases, the invocation of m_2 is not externally observable and so is of no concern when discussing the externally observable behavior of `A`.

In some cases, it will be necessary to consider the invocation of m_2 to describe the externally observable behavior of A. In such instances the user of class A must know about class B to fully understand the semantics of method m_1. This corresponds to the case of *interobject behavior* [HHG90].

As we do not treat interobject behavior in this thesis, we assume that all behavior which is observable by the client is limited to the receiving object.

- Each type of response is perceived as follows:
  - Modifications to the internal state of an object cannot be directly perceived.
  - Modifications to the external environment can be perceived through direct observation (e.g. a graphical screen) or by *probing* (e.g. a database) to inspect it.
  - A returned value can be perceived by manipulating it in a program.

  Thus, an object's observable behavior can be perceived in two distinct ways by a client: by noting changes in the state of the external environment or by invoking inspector methods defined in the public interface.

  We will refer to the former behavior as *environmental behavior* and the latter as *interface behavior*. Thus, an object's total behavior consists of the combination of its environmental behavior and its interface behavior. Some classes define objects which exhibit only environmental behavior, only interface behavior, or both.

- The constraints which apply to methods are such that some methods can be invoked only when the object has a particular internal state. An alternate behavioral characterization of this is that certain methods can be invoked only when the sequence of methods which precedes it satisfies a given property.

Based on these considerations, and assuming a sequential, deterministic model of computation, we can informally define a *complete specification of class behavior* to be a specification which satisfies the following properties:

- For any sequence of messages sent to a class, it is possible to deduce from the specification whether the sequence of messages is legal (i.e. respects constraints).

- For any legal sequence of messages sent to a class, it is possible to deduce the effect (i.e. change in state) caused in the object's external environment (e.g. graphical display, database, communication network).

- For any legal sequence of messages sent to a class which terminates in a message returning a value, it is possible to determine the *value* returned based on the specification. We will make precise our notion of *value of an object* in the following section.

4.2 Towards a Formal Definition

In this section we make more precise our definition of complete specification of class behavior. First, we require the following definitions.

Abstract Value of an Object. In the previous section, we remarked that it is necessary to refer to the values of objects returned by methods. This poses the problem of deciding how we wish to represent object values. It is undesirable to represent the values of objects by their concrete values (i.e. the values of their instance variables). This would compromise object encapsulation and would provide too low a level of abstraction. Instead, we use a technique which is very common in dealing with abstract data types [Hoa72, Ame89]. We model the abstract conceptual state of an object by an element of some mathematical domain.

The use of such abstract values to represent object values is consistent with the approach used in most formal specification languages. For example, our abstract object values correspond to the values of the abstract models of specification languages such as Z and VDM or to the values of *sorts* of specification languages such as Larch or OBJ. As we will show later, this correspondence is very useful in evaluating the completeness of a class interface specification written in such a formal specification language.

For convenience, we can also use literal values to represent the abstract values of

"basic" objects such as integers and real numbers. For example, '1' can be used to denote the abstract value of the integer whose value is 1.

Trace of a Class. A trace of a class C represents a sequence of *message sends* on an object of class C, starting in the initial state. In the context of trace elements, we will refer to the method corresponding to a message m simply as "the method m" where this is convenient.

Informally, a trace of a class C is a sequence where each element is the name of a method of C along with its parameters. In addition, we make the requirement that the first element of the sequence be the name of a constructor of C and that the other elements be any method name except the constructor.

Formally, we define a trace of a class C to be any sequence whose items are elements of $IPMN(C)$, whose first item is an element of $ICN(C)$, and whose other items are elements of $IPMN(C) \setminus ICN(C)$. $IPMN(C)$ and $ICN(C)$ are defined as follows:

IPMN(C): An *IPMN* (instantiated public method name) of a class C is the name of a public method defined in C or inherited from an ancestor of C, where the abstract values of actual parameters have been substituted for the formal parameters. $IPMN(C)$ is the set of all *IPMN*s of C.

ICN(C): Similarly, an *ICN* (instantiated constructor name) of a class C is defined as the name of a constructor of C where the abstract value of actual parameter values have been substituted for all the formal parameters of the constructor, and we denote the set of all *ICN* of a class C by $ICN(C)$.

Out of all the possible traces of a class, only a certain subset of these correspond to a correct usage of the class's methods. We call this subset the *legal* traces of a class. The legal traces of a class cannot be determined from the implementation of a class alone and must necessarily be based on a faithful specification of the class's behavior. The failure of a client to respect the correct sequence and parameters values will typically result in an error condition or in unpredictable behavior.

The set of all traces of a class C is denoted by $trace(C)$, the set of all legal traces of C is denoted by $LegalTr(C)$, and the set of all illegal traces is denoted by $IllegalTr(C)$.

Any given trace $t \in trace(C)$ is either legal or illegal: either $t \in LegalTr(C)$ or $t \in IllegalTr(C)$.

V-action, S-action, O-action. Inspired by the work presented in [BP86], we use the following categories to characterize the possible actions of a method.

- **V-action:** A method which returns a value belongs to this category.
- **S-action:** A method which produces an observable side effect in the object's external environment (e.g. display a window, emit a beep, write to a file, transmit a packet on a network) belongs to this category.
- **O-action:** A method which changes the abstract state of an object belongs to this category.

Our V-action and O-action categories correspond to the V-function and O-function categories of [BP86]. However, our characterization differs from that of [BP86] in many important ways: (i) We use this characterization as a means to define what the behavior of a module is rather than to write a specification of the behavior of a particular module. (ii) We use abstract values to denote the values of method parameters and return values. This makes it possible to consider methods that take as parameters or return values of complex objects rather than just basic values like integers and floating point numbers. (iii) In addition to considering methods which modify the internal state of an object, we also consider methods which modify the state of the external environment. This makes it possible to provide a more precise definition of class behavior.

We use the above categories to classify the methods of a class. For example, a method which belongs to the O-action and V-action categories is referred to as an O-V method.

We assume that a method which belongs to the S category must always also belong to the O category so that any change in the environmental state of an object will be reflected by a change in the abstract state of the object. Therefore, there can be no S methods, only O-S methods or O-S-V methods.

This assumption is motivated by pragmatic concerns. In existing specification

languages (e.g. Larch, Z, VDM) there is only one abstract state which is identified with the object or module being specified. There is no distinction between those abstract states which correspond to the object state and those abstract states which correspond to environmental states. Moreover, in practice, a modification to the external environment is usually mirrored by a modification to the computer memory. For example, modifying the contents of the graphics display involves modifying the contents of the display buffer. The abstract state of an object is an abstraction for the computer memory used, directly or indirectly, to implement an object. It is only natural that this abstract state reflect such changes which result in a modification of the environmental state.

The remaining possible combinations for a method are O, O-V, O-S, V, and O S-V. Intuitively, it is clear that these can be reduced to the *canonical combinations* O, O-S, and V. This is because it can be shown that any method which belongs to any other combination (i.e. O-V, O-S-V) can be reduced to two methods each belonging to a canonical combination. That is, any method which belongs to the V category and to some other category can be split into two methods one which is a pure function (the V portion) and another which is a procedure (the remaining O or O-S portion). Details of a formal proof of this would require defining the semantics of the object-oriented programming language being considered. To circumvent this and to avoid having to commit to a specific programming language, we choose to accept the validity of our assumption as an axiom.

For simplicity, we therefore assume that all the methods which we must deal with are of the canonical combinations O, O-S and V. In view of the above comments, this results in no essential loss of generality.

Complete Specification of Class Behavior. A specification of a class's externally observable behavior is *complete* iff the following conditions are satisfied:

(i) For any given trace of a class, it is possible to determine whether or not the trace is legal.

(ii) For every legal trace ending with a call to a V-method, the abstract value returned can be derived from the specification.

(iii) For every legal trace ending with a call to an S-method, the side effect which will occur as a result of this last method can be derived from the specification.

This definition assumes that the specification is a faithful description of the implementation, i.e. that the implementation is correct with respect to the specification.

4.3 Formalizing the Definition

To make the definition fully formal, it is necessary to postulate the existence of two sets. Given a class C, we assume that the following sets are available:

RVAL_C: the set of all abstract object values which can be returned by the methods of C. For each return type of the V-methods of C, it is assumed that the corresponding classes have been assigned abstract value representations for each of their possible objects.

EXT_C: the set of all possible abstract states of the external environment. We use the elements of a mathematical domain to model the abstract state of the external environment, in the same manner as we have done for modeling the abstract conceptual values of objects.

The elements of EXT_C are assumed to be determined *a priori* and will depend on the level of abstraction which is relevant to a particular application. For example, in the case of a database update, it will be necessary to decide if updates should be described in terms of modifications to relations, to physical files which store the relations, or to the bytes on the disk which stores the physical files.

We can now formally define the *behavior* of a class as a triple ($LegalTr(C)$, A_s, A_v), called the *behavior-triple*, where A_s and A_v are partial functions:

$$A_s : LegalTr(C) \nrightarrow (EXT_C \rightarrow EXT_C)$$

defined by:

$$A_s(t) : \begin{cases} \text{envState} & \text{if isOSMethod(last(t))} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where last, envState, and isOSMethod have the following interpretations:

last: The function *last* takes as argument a trace of length ≥ 1 and returns the last element of the trace.

envState: The function envState : $EXT_C \rightarrow EXT_C$ is defined by envState(S_0) = S' where S_0 is the initial environmental state (before any methods of the trace were executed) and S' represents the current environmental state (after all methods in the trace, including the final S-method, have been executed).

isOSMethod: The predicate *isOSMethod* takes a method name as input and returns true if this method is an S-method and false otherwise.

A_v is a partial function

$$A_v\text{: } LegalTr(C) \nrightarrow \text{RVAL}_C$$

defined by:

$$A_v(t) : \begin{cases} \text{val} & \text{if isVMethod(last(t))} \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the above definitions val and isVMethod have the following interpretations:

val: The abstract value *val* represents the value returned by the final V-method of the trace.

isVMethod: The predicate *isVMethod* takes a method name as input and returns true if this method is an V-method and false otherwise.

We provide examples involving the above operators in section 5 where we introduce some example class interfaces from which traces can be derived.

We now define a complete specification of the behavior of a class C to be a specification S_C such that for all traces $t \in trace(C)$ the following hold:

(i) S_C can be used to determine whether $t \in LegalTr(C)$ or $t \in IllegalTr(C)$.

(ii) If $t \in LegalTr(C)$, then S_C can be used to compute the value of $A_s(t)$ and $A_v(t)$.

4.4 Applying the Definition in Practice

In the absence of a specification for a given class C, the behavior triple (*LegalTr*(C), A_s, A_v) which formally characterizes the class's behavior is not explicitly available but, rather, can be determined by exercising the class's implementation. This triple represents the behavior of a class's implementation, or of any equivalent implementation. A class interface specification attempts to describe this behavior.

To evaluate the completeness of the formal specification of a class's behavior, we need to show that S_c can be used to derive the behavior-triple. We will refer to this as the *completeness criterion*. This criterion is sufficient to determine completeness because we have assumed that the implementation of C is correct with respect to S_c.

Chapter 5

Evaluating the Completeness of Larch/C++ Specifications

In this chapter, we present a methodology for evaluating the completeness of a C++ class interface specification written in Larch/C++. This methodology is based on the completeness criterion described in section 4.4 and is intended to verify that a class interface specification fully captures class behavior as we have defined it.

Although the methodology we present in this section is tailored to Larch/C++, the reader should have no difficulty in seeing how it could be adapted and extended to other programming languages and specification languages.

5.1 The Methodology

To evaluate the completeness of a Larch/C++ interface specification S_c for a class C, it is necessary to show that S_c can be used to construct the behavior triple (*LegalTr*(C), A_v, A_s). We can decompose this process into three separate steps, one for each component of the behavior triple. Each step constitutes a proof obligation which demonstrates that the corresponding component of the behavior triple can be derived from S_c.

STEP I. The first step consists in showing that S_c is used to construct *LegalTr*(C). This is shown by proving that for any arbitrary trace t, we can determine whether t $\in$ *LegalTr*(C) or t $\in$ *IllegalTr*(C) based on S_c. If this cannot be shown for all possible traces, then S_c is incomplete.

To formally demonstrate the required result for all traces, we use a proof by induction on the length of traces. In order to do this, it is necessary to "link" the traces of a class to the specification S_c.

This is achieved by defining a recursive function *Absval* whose domain is *LegalTr*(C) and whose range is VAL_C, the set of all abstract values for the objects of class C.

Given a particular trace t, *Absval* returns the abstract value of the state which results after executing all the methods of the trace on an object. These methods are applied in the order and with the parameters specified by the trace.

In Larch, the abstract values of the objects of a class C are represented by the terms which denote the values of the sort SORT_C corresponding to class C. Therefore, the range of *Absval* will be SORT_C. SORT_C corresponds to the *distinguished sort* [GHM91] of the trait used in the specification of C.

In a Larch/C++ interface specification, the postconditions of methods describe, among other things, how the abstract state of `self`[1] is modified by the execution of methods. The definition of *Absval* is based on these postconditions so as to determine the state changes of `self` resulting from the various messages.

Absval is written in such a way as to simply emulate the effect of the postconditions in the interface specification of C. For the O-methods and OS-methods of C, this means that *Absval* must transform the *current state* in the manner specified by the postcondition of the last method of the trace. The current state is the abstract state resulting from the execution of all the previous operations. For V-methods, *Absval* does not need to transform the abstract state; it remains the same as before.

Based on the definition of the function *Absval*, a given trace t can be inductively tested for legality. This is accomplished by determining whether the precondition in the Larch/C++ interface specification corresponding to each message in the trace is satisfied. This process is described by the steps in figure 5.1.

It should be noted that the decision to use the preconditions of the interface specification to determine the legality of traces in this methodology is specific to the needs of Larch/C++. If some other formal specification language were used, then the legality of traces could be defined according to some other criterion appropriate to that formalism.

In the inductive proof we do not actually make use of the steps described in Figure 5.1, but the proof is motivated by an understanding of those steps. Given a trace of length n we can suppose, by induction, that the legality of the subtrace of length $n-1$ is determined. In the inductive step, it is then necessary to show that

[1] In Larch/C++ `self` is syntactic sugar for (*this), which represents the current object in C++

```
FOR I := 1 to length(t) −1 DO
    BEGIN
        let st be the subtrace of t
            consisting of the first I elements;
        let AV be the abstract value Absval(st);
        let m be method corresponding to t(I+1);
        Evaluate the precondition of m using AV and the
            values of the parameters specified for m in t;
    END
IF each precondition above could be evaluated
    THEN
    IF each precondition evaluated above was satisfied
        THEN trace t is legal
        ELSE trace t is illegal
    ELSE it is not possible to determine whether t is legal
        and the specification is incomplete
```

Figure 5.1: Steps for Verifying Legality of a Trace

the precondition corresponding to the n^{th} message in the trace is evaluated.

This is accomplished by considering all possible cases for the n^{th} message and proving that, in each case, the precondition is evaluated. This, in turn, is accomplished by making use of the fact that the abstract value of `self`, the object being considered, is evaluated using *Absval*.

STEP II. The second step consists in showing that S_c can be used to construct A_v. This is shown by proving that for any trace $t \in LegalTr(C)$ which ends in a V-method, S_c is used to determine the value of $A_v(t)$.

The proof involves a case analysis on the V-methods of C. For each case, it is necessary to show that for all traces which end in the particular V-method, the abstract value representing the returned value is determined using S_c.

This requires using the postcondition of the method being considered to determine the returned value. Typically, the postcondition will refer to `self`, in which case the function *Absval* is used to determine the abstract value of `self`. The abstract value of `self` can then be used to evaluate the abstract value of `result`, in the postcondition.

It is also necessary to ensure that the value of `result` so obtained does not correspond to a term which is identified as exempted (i.e. not convertible) in the corresponding trait. This corresponds to the notion of *protective specification* described in [Win83].

STEP III. The third step consists in showing that S_C can be used to construct A_s. This is shown by proving that for any trace $t \in LegalTr(C)$ which ends in an OS-method, S_C is used to determine the value of $A_s(t)$.

The method here is similar to Step II, except that the case analysis is done on OS-methods rather than on V-methods.

5.2 A Stack Example

The first example we consider is a class which implements a stack of integers. Fig. 5.2 shows the LSL trait for the sort Stack and Fig. 5.3 shows the Larch/C++ interface specification for class IntStack. We will refer to the entire stack specification, consisting of both the trait and the interface specification, as $S_{IntStack}$. We have prefixed the C++ member function names in figure 5.3 with 'I' to distinguish them from the corresponding LSL trait operators in figure 5.2. This is not strictly necessary, as the context in which an identifier appears can be used to unambiguously determine whether an LSL operator or member function name is being referred to. However, this convention will make the ensuing discussion more clear.

The methods of IntStack are categorized as follows: the V-methods are {Itop, IisEmpty}, the O-methods are {IntStack, Ipush, Ipop} and there are no OS-methods.

The abstract values of the class IntStack, $VAL_{IntStack}$, are represented by values of the sort $SORT_{IntStack}$ = Stack. These values are equivalence classes of terms denoted by canonical terms of the form:

`new, push(new, i), push(push(new, i), j), ...`

where `i` and `j` are arbitrary integers.

The set of abstract values for the return types of IntStack, $RVAL_{IntStack}$, consists of the abstract values representing integers and booleans. Abstract values for these are defined using LSL traits. However, for simplicity we will denote integer values using the standard literal symbols '1', '2', and booleans using the values 'true' and

StackTrait(E, Stack) : **trait**
introduces
new: → Stack
push: Stack, E → Stack
top: Stack → E
pop: Stack → Stack
isEmpty: Stack → Bool
asserts
Stack **generated by** new, push
Stack **partitioned by** top, pop, isEmpty
∀ s: Stack, e: E
top(push(s,e)) == e
pop(push(s,e)) == s
isEmpty(new)
¬ isEmpty(push(s,e))
implies
converts top, pop, isEmpty
exempting top(new), pop(new)

Figure 5.2: LSL Trait for Stack

'false'.

The set of abstract values for the environmental state, $EXT_{IntStack}$, is the empty set since IntStack does not interact with the environment.

Intuitively, the legal traces of IntStack are those for which each occurrence of **pop** and each occurrence of **top** is preceded by a subtrace where the number of occurrences of **push** exceeds the number of occurrences of **pop**.

We can use also an example trace of IntStack to exemplify the operator *val* introduced in section 4.3. For a trace t = ⟨ IntStack, push(1), push(2), pop, top ⟩ we have val(t) = 1.

We now proceed to show that the specification $S_{IntStack}$ is complete by applying the above methodology.

STEP I. Given any $t \in trace$(IntStack), $S_{IntStack}$ can be used to determine whether $t \in LegalTr$(IntStack) or $t \in IllegalTr$(IntStack).

Proof: The proof of this property is by induction on the length of the trace

Basis Step: Consider traces of length 1. The only legal trace of length one is ⟨ IntStack ⟩. Since the precondition of any constructor is *true*, this trace is legal. All other traces of length 1 are illegal.

Inductive Hypothesis: Suppose, that the property is true for all traces of length $k-1$.

Inductive Step: We must now show that the property is true for all traces t of length k. By the inductive hypothesis, for the subtrace composed of the first $k-1$ elements of t it is possible to determine whether or not this trace is legal. If this subtrace is illegal, then the entire trace t is illegal.

If on the other hand the subtrace is legal, then we must consider the different possible cases for the last (i.e. k^{th}) element of the trace t. The possible choices in our example are: `Ipush(int)`, `Itop`, `Ipop`, and `IisEmpty`.

Figure 5.4 shows the definition of the function

$$\text{Absval: LegalTr(IntStack)} \rightarrow \text{Stack}$$

which is needed to prove this inductive step. Notice that this function is based on the postconditions of the IntStack operations and updates the abstract state of the IntStack (i.e. `self`) based on these postconditions.

We now consider the following cases for the final element of t:

Ipush, IisEmpty: For these cases, the precondition is always satisfied hence making t legal.

Itop: In this case the precondition is `not(isEmpty (self^))`. This precondition is evaluated by substituting *Absval(front(t))* for `self^` in the precondition predicate[2]. This yields the expression *not(isEmpty(Absval(front(t))))*. If this condition evaluates to true, then the precondition is satisfied and so t is legal. Otherwise, the precondition is not satisfied and so t is not legal. Here, `front` is a function which takes as argument a trace of length > 1 and returns a trace with the last element removed.

Ipop: This case is exactly same as for Itop.

STEP II. Given any $t \in$ *LegalTr*(IntStack) which ends in a V-method, $S_{IntStack}$

[2]More precisely, we are actually substituting the value denoted by the term which the expression *Absval(front(t))* evaluates to.

can be used to determine the value of A_v(t).

Proof: We need to consider only legal IntStack traces which end in a V-method. There are only two such methods, `Itop` and `IisEmpty`.

Itop: In this case, the value of A_v(t) is defined to be *top(Absval(front(t)))*.

IisEmpty: Here, the value of A_v(t) is defined to be *isEmpty(Absval(front(t)))*.

It is also straightforward to prove that the values returned by A_v are all well defined in the sense that they do not correspond to terms involving exempted values. For example, A_v will never return a term of the form `top(new)`. To prove this it is only necessary to recall that the input trace t is assumed to be legal. By construction, this means that it satisfies the preconditions of the corresponding methods of the trace. A case analysis of the possible exempted terms easily demonstrates that a contradiction results if it is assumed that any one of these exemptable values could be returned by A_v.

STEP III. Since there is no OS-method in the above example this step is trivially satisfied.

5.3 A Screen Example

We now present an example involving a Screen class. One notable difference between this example and the previous is that it involves environmental behavior. The Screen class provides methods to move the cursor on a textual screen. The LSL and interface specification for the Screen class[3] are defined in Fig. 5.5 and 5.6.

The methods of Screen are categorized as follows: the OS-methods are {moveLeft, moveRight, moveUp, moveDown} and there are no O-methods or V-methods.

The abstract values of the class Screen, VAL_{Screen}, are represented by values of the sort SORT_{Screen} = Scrn.

The set of abstract values for the return types of Screen, RVAL_{Screen}, is the empty set since there are no return values.

The state of the physical screen can be modelled by an integer which records

[3]Adapted from a specification obtained through personal communication with Gary Leavens November 1992.

the current position of the cursor on the screen. The addressable locations on the screen are numbered from 0 to $(length \times width) - 1$. Hence, the set of abstract values for the environmental state, EXT_{Screen}, is the set of integers in the range $0, \ldots, length \times width - 1$

Intuitively, the legal traces of Screen are those which keep the cursor within the screen.

We can also use an example trace of Screen to exemplify the operator *envState* introduced in section 4.3. For a trace t = ⟨ Screen(5,5), moveRight, moveDown, moveRight, moveDown, ⟩ in the screen example, envState(t) = 12.

We now proceed to show that the specification S_{Screen} is complete by applying the methodology.

STEP I. Given any $t \in trace$(Screen) the specification S_{Screen} can be used to determine whether $t \in LegalTr$(Screen) or $t \in IllegalTr$(Screen).

Proof The proof of this property is by induction on the length of the trace.

Basis Step: Consider traces of length 1. The only legal trace of length one is ⟨ Screen(x,y) ⟩. Since the precondition of any constructor is *true*, this trace is legal.

Inductive Hypothesis: Suppose that the property is true for all traces of length $k - 1$.

Inductive Step: We must now show that the property is true for all traces t of length k. By the inductive hypothesis, for the subtrace composed of the first $k - 1$ elements of t it is possible to determine whether or not this trace is legal. If this trace is illegal, then the entire trace t is illegal.

If on the other hand the subtrace is legal, then we must consider the different possible cases for the last (i.e. k^{th}) element of the trace t. The possible choices are: `moveLeft`, `moveRight`, `moveUp`, and `moveDown`.

The definition of the function

$$\text{Absval: LegalTr(Screen)} \rightarrow \text{Scrn}$$

is in figure 5.7. As in the previous example, the function *Absval* is based on the postcondition of the interface specification operations. It returns the abstract value denoting the state of the resulting object.

We now consider the possible cases for the last element of t.

moveLeft: In this case the precondition is `getCursor(self^) > 0`. This precondition is evaluated by substituting *Absval(front(t))* for `self^` in the precondition predicate. This yields the expression `getCursor((Absval(front(t)))) > 0` which is evaluated using the trait axioms for Screen, as the resulting term *Absval(front(t))* has sort `Scrn`. If this condition evaluates to true, then the precondition is satisfied and so t is legal. Otherwise, the precondition is not satisfied and so t is illegal.

The proofs for the remaining cases (`moveRight`, `moveUp`, and `moveDown`) are similar to the proof for `moveLeft`.

STEP II. Since there is no V-method in this example this step is trivially satisfied.

STEP III. Given any $t \in$ *LegalTr*(Screen) which ends in an OS-method, we must show that S_{Screen} can be used to determine the value of A_s.

Proof: We need to consider only legal Screen traces which end in a OS-method. For all these cases, we can define the function
A_s(t):

$$\text{LegalTr(Screen)} \not\rightarrow (EXT_{Screen} \rightarrow EXT_{Screen})$$

to be given by $A_s(t) = \text{getCursor}(Absval(t))$

Here, the definition of Absval provides an explicit link between the environmental state and the abstract state of Screen objects.

Hence S_{Screen} is complete with respect to the definition of class behavior.

```
class IntStack
{
    uses StackTrait(IntStack for Stack, int for E);
    public:
        IntStack()
        {
            modifies self;
            ensures self'= new;
        }
        ~IntStack()
        {
            modifies self;
            ensures trashed(self);
        }
        void Ipush(int i)
        {
            modifies self;
            ensures self'= push(self^, i);
        }
        bool IisEmpty()
        {
            ensures result = isEmpty(self^);
        }
        int Itop()
        {
            requires not(isEmpty(self^));
            ensures result = top(self^);
        }
        void Ipop()
        {
            requires not(isEmpty(self^));
            modifies self;
            ensures self'= pop(self^);
        }
};
```

Figure 5.3: Larch/C++ Interface Specification for IntStack

$$
Absval(t): \begin{cases} \text{new} & \text{if } \mathcal{L}(t) = 1 \\ \text{push(Absval(front(t)),j)} & \text{if } \mathcal{L}(t) > 1 \text{ and } \text{last}(t) = \text{Ipush(j)} \\ \text{pop(Absval(front(t)))} & \text{if } \mathcal{L}(t) > 1 \text{ and } \text{last}(t) = \text{Ipop} \\ \text{Absval(front(t))} & \text{otherwise (if isVMethod(last(t)))} \end{cases}
$$

where $\mathcal{L}(t)$ is the length of the trace t

Figure 5.4: The function Absval for Stack

```
ScreenTrait: trait
  introduces
  includes Integer
    newScreen: Int, Int → Scrn
    height, width: Scrn → Int
    setCursor: Scrn, Int → Scrn
    getCursor: Scrn → Int
  asserts
    Scrn generated by newScreen, setCursor
    Scrn partitioned by getCursor, height, width
    ∀ s: Scrn, h,w,i:Int
        height(newScreen(h,w)) == h
        width(newScreen(h,w)) == w
        height(setCursor(s,i)) == height(s)
        width(setCursor(s,i)) == width(s)
        getCursor(newScreen(h,w)) == 0
        getCursor(setCursor(s,i)) == i
  implies
    converts height, width, getCursor
```

Figure 5.5: LSL Trait for Screen

```
class Screen
{
  uses ScreenTrait(Screen for Scrn);
  public:

      Screen(int h, int w)
      {
          modifies self;
          ensures self'= newScreen(h,w);
      }
      ~Screen()
      {
          modifies self;
          ensures trashed(self);
      }
      void moveLeft()
      {
          requires getCursor(self^) > 0;
          modifies self;
          ensures self' = setCursor(self^, getCursor(self^) - 1)
      }
      void moveRight()
      {
          requires getCursor(self^) < height(self^) * width(self^);
          modifies self;
          ensures self' = setCursor(self^, getCursor(self^) + 1)
      }
      void moveUp()
      {
          requires getCursor(self^) ≥ width(self^) ;
          modifies self;
          ensures self' = setCursor(self^, getCursor(self^) - width(self^))
      }
      void moveDown()
      {
          requires getCursor(self^) < (height(self^) - 1) * width(self^);
          modifies self;
          ensures self' = setCursor(self^, getCursor(self^) + width(self^))
      }
};
```

Figure 5.6: Larch/C++ Interface Specification for Screen

$$Absval(t): \begin{cases} \text{newScreen(h,w)} & \text{if } \mathcal{L}(t) = 1 \\ \text{setCursor(A, getCursor(A) - 1)} & \text{if } \mathcal{L}(t) > 1 \text{ and last(t)} = \text{moveLeft} \\ \text{setCursor(A, getCursor(A) + 1)} & \text{if } \mathcal{L}(t) > 1 \text{ and last(t)} = \text{moveRight} \\ \text{setCursor(A, getCursor(A) - width(A))} & \text{if } \mathcal{L}(t) > 1 \text{ and last(t)} = \text{moveUp} \\ \text{setCursor(A, getCursor(A)+ width(A))} & \text{if } \mathcal{L}(t) > 1 \text{ and last(t)} = \text{moveDown} \end{cases}$$

where A is an abbreviation for Absval(front(t))
$\mathcal{L}$(t) is the length of the trace t
h,w are the parameters of the first element of the trace

Figure 5.7: The function Absval for Screen

Chapter 6

Dealing With Inheritance in Larch/C++

In section 2.2 we discussed the importance of distinguishing between subtype inheritance and implementation inheritance. We mentioned that it was important for a ROOCSL to distinguish between the notions of subclassing (implementation inheritance) and subtyping (subtype inheritance). A subtype relationship is a behavioral relationship which could be proven by examining the specifications of the types involved, while a subclass relationship is purely an implementation relationship.

In section 2.2 we also discussed the importance of permitting inheritance of specifications. Specification inheritance avoids having to re-specify the methods inherited from superclasses by making it possible to construct larger specifications from smaller specifications.

It is important to note that although determining subtyping relations and inheritance of specifications are somewhat related concepts, they can exist independently of one another. It is possible to define a semantics for determining whether subtype relations hold between classes even if the specification language does not support inheritance of specifications. Conversely, it is possible to provide inheritance of specifications without defining a semantics for determining subtype relationships between classes.

There is some interaction between the semantics which we propose for determining subtype relations and the mechanism we propose for dealing with inheritance of specifications. However, to maintain a clear distinction between the two proposals we first present the two concepts separately and then discuss how they interact.

In section 6.1 we present a proof theoretic semantics for determining subtyping relations between Larch/C++ class specifications. In section 6.2 we describe a mechanism which permits inheritance of specifications. Although some preliminary work

on this was described in [LC92], many questions were left unanswered. Our proposal attempts to answer these questions as well as to solve some new problems which have been identified with the preliminary proposal described in [LC92].

6.1 Proof Theoretic Semantics for Subtyping Relations

Although several semantics for subtyping have already been proposed [BW87, Ame89, Lc .90. Lea91, Dha92], none has been selected yet in the preliminary design for l .ch/C++ [LC92]. The semantics we propose is based on that of America [Ame89], which we have adapted for Larch/C++.

America's approach is much simpler than other approaches which have been proposed, being based on a very simple proof-theoretic definition rather than on the complex model-theoretic definitions used in the other approaches (e.g. [Lea90, DL92]). The manner in which we have adapted the approach to the LSL tier of Larch/C++ also considerably simplifies the process of demonstrating the existence of subtyping relationships. These factors make the use of our approach viable and practical by developers in industry. Also, because the semantics is proof-theoretic, the verification process can be partially automated and we describe how this can be achieved using the theorem prover LP [GG89, GG90, GG91].

Given the Larch/C++ class interface specifications I_1 and I_2 for some classes C_1 and C_2, where C_2 is a public subclass of C_1, we wish to determine when C_2 is actually a semantic subtype of C_1. Let us denote the interface specification of a method m in I_1 by

$$\{\mathrm{Pre}_{super}\}\ m\ \{\ \mathrm{Post}_{super}\}$$

where Pre_{super} and Post_{super} denote the assertion of the corresponding **requires** and **ensures** clauses respectively. Similarly, let us denote the interface specification of a method m in I_2 by

$$\{\mathrm{Pre}_{sub}\}\ m\ \{\ \mathrm{Post}_{sub}\}$$

where Pre_{sub} and Post_{sub} denote the assertion of the corresponding **requires** and **ensures** clauses respectively. The interface specification I_1 makes use of a trait Tr_1 to define the LSL terms referred to in the assertions of Pre_{super} and Post_{super} while I_2 makes use of a trait Tr_2 to define the LSL terms referred to in the assertions of Pre_{sub} and Post_{sub}. This situation is depic d in figure 6.1. We now wish to develop a definition for specifying when class C_2 is a semantic subtype of class C_1.

Following the approach of America [Ame89], for C_2 to be a semantic subtype of C_1we require that for every public method specification

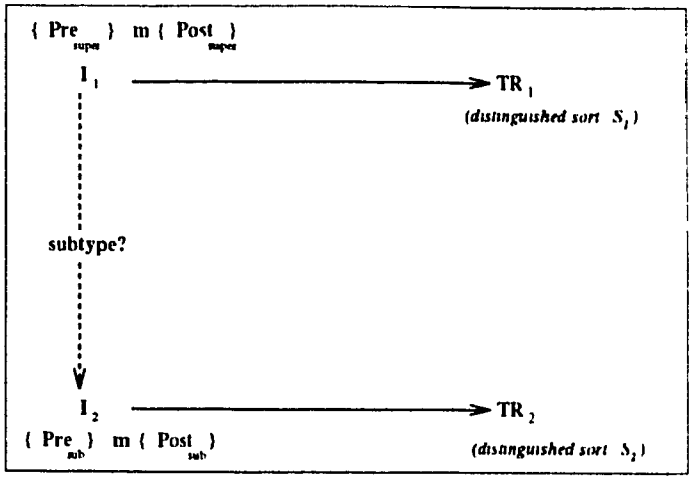

Figure 6.1: Defining A Subtyping Relation

$$\{\text{Pre}_{super}\}\ \text{m}\ \{\text{Post}_{super}\}$$

in I_1 there should be a method specification

$$\{\text{Pre}_{sub}\}\ \text{m}\ \{\text{Post}_{sub}\}$$

in I_2 such that the former implies the latter, which can be expressed as:

$$\text{Pre}_{super} \rightarrow \text{Pre}_{sub} \quad \text{and} \quad \text{Post}_{sub} \rightarrow \text{Post}_{super} \quad (1)$$

It should be noted that this requirement is the same as that used in Eiffel's *assertion redefinition rule* for preserving conformance relations between a subclass and a superclass [Mey88]. It appears that the requirement was developed independently by America and Meyer. The requirement ensures that we can indeed use an object of class C_2 wherever an object of class C_1 is expected: when we send an object of class C_2 a message m, using it as an object of class C_1 guarantees that the precondition Pre_{super} will hold (since we must respect the specification I_1 to use a C_1 object correctly). By the implication $\text{Pre}_{super} \rightarrow \text{Pre}_{sub}$ we can conclude that the precondition Pre_{sub} in C_2's specification I_2 will also hold. After the execution of method m the postcondition Post_{sub} will hold since the precondition Pre_{sub} was respected and the implementation of C_1 is assumed to be correct with respect to I_1. By the implication $\text{Post}_{sub} \rightarrow \text{Post}_{super}$, we can conclude that Post_{super} also holds as required by the specification I_1 for an object of class C_1.

Although the above requirement provides an adequate definition of when a class C_2 is a subtype of a class C_1, it fails to take into account the fact that the specifications

I_1 and I_2 might be expressed using different mathematical domains (i.e. different sorts). In general, we must assume that I_1 is expressed using a sort S_1 while I_2 is expressed using a different sort S_2.

Following [Ame89], we solve this problem by requiring the existence of a coercion function ϕ which maps the mathematical domain associated with I_2 to the domain associated with I_1. In [Ame89], America uses abstract mathematical models like sets and sequences to represent the abstract values in the class interface specifications. He associates a mathematical domain Γ with the supertype specification and a mathematical domain Σ with the subtype specification and then requires the existence of a function ϕ: $\Sigma \rightarrow \Gamma$ to coerce Σ values to Γ values .

In the case of Larch/C++, the abstract values are represented by the values of sorts defined in LSL traits. Thus, it is necessary to require the existence of a function

$$\phi: S_2 \rightarrow S_1.$$

Before proceeding further, it is necessary to discuss how such a function can be defined in the context of the LSL traits which are used to define the abstract models of Larch/C++ interface specifications. To address this problem, we note the following observations regarding LSL sorts and traits:

- Values of the sorts S_1 and S_2 are represented by algebraic terms defined in the LSL traits used by the interface specifications I_1 and I_2. Therefore, a mapping between the sorts S_1 and S_2 can be defined by establishing a mapping between the LSL terms denoting the sort values.

- Each value of the sort S_2 and S_1 corresponds to an equivalence class of LSL terms defined in the corresponding trait. All these equivalence classes, and hence sort values, can be denoted by terms involving only the generators of a sort. Hence, to establish a mapping from values of S_2 to values of S_1 it is sufficient to establish a mapping between the terms involving generators of the traits.

- In LSL, the generators of a trait are explicitly identified by the **generated by** clause. Let us denote the trait which defines sort S_1 by Tr_1 and the trait which defines sort S_2 by Tr_2. Then the function ϕ can be defined by considering only

t ms which involve only the operators identified in the **generated by** clause of Tr_2 and mapping these to terms involving only the operators identified in the **generated by** clause of Tr_1.

We can now illustrate the process of defining the coercion function ϕ using an example. Figure 6.2 is an LSL trait which defines the properties of `Table` an abstract data type in which (key, value) pairs can be inserted, removed, and retrieved. The distinguished sort of trait `Table` is `Tab`. Figure 6.3 is an LSL trait which defines the properties of `OrdTable`, an ordered table abstract data type. `OrdTable` is the same as `Table` except that it incorporates a notion of ordering based on the order of insertion. It provides additional operations to retrieve the n^{th} key of value from the table.

Note that we have repeated all the definitions of the operations defined in the `Table` trait of figure 6.2 in the `OrdTable` Trait of figure 6.3. For clarity of exposition, we have renamed all the repeated operators by prefixing them with an 'O' in the `OrdTable` trait. This renaming is not strictly necessary as LSL operators can be overloaded, permitting several operators having the same name but different signatures. Usually, no ambiguity results from such overloading as the context in which an operator is being used is sufficient to unambiguously determine which of the overloaded operators is being referred to. In the cases where ambiguity can still result regardless of the context, it is possible to disambiguate an operator name by appending the suffix ':<sig>', where <sig> is the full signature of the operator. For example, `delete:Tab,Key->Key` provides a more precise name for the operator `delete` defined in the `Table` trait and can be used to disambiguate it from another operator having the same name.

In view of all the above observations concerning the definition of the coercion function ϕ in the context of LSL traits, we can define ϕ for this example to be given by the trait operator `toTable` defined in figure 6.4.

Having shown how a coercion function ϕ can be defined between the sorts used by two different class interface specifications, we can now show how this function is useful in determining subtyping relations. We modify the requirement (1) to handle interface specifications in which two different distinguished sorts are used by requiring

Table : **trait**
  includes Integer
introduces
  empty: → Tab
  insert: Tab, Key, Val → Tab
  delete: Tab, Key → Tab
  hasKey: Key, Tab → Bool
  lookUp: Tab, Key → Val
asserts
  Tab **generated by** empty, insert
  Tab **partitioned by** hasKey, lookUp
  ∀ k, k_1 : Key , v, v1 : Val, t : Tab, n : Int
    lookUp(insert(t,k,v), k_1) == **if** k = k_1 **then** v **else** lookUp(t, k_1)
    not(hasKey(k, empty))
    hasKey(k, insert(t, k_1, v)) == **if** k = k_1 **then** true **else** hasKey(k, t)
    delete(insert(t,k,v), k_1) == **if** k = k_1 **then** t **else** insert(delete(t, k_1), k, v)
  implies
    converts delete, hasKey, lookUp
    exempting ∀ i: Key lookUp(empty, i)

Figure 6.2: LSL Trait for Table

OrdTable : **trait**
  includes Table
introduces
  Oempty: → OTab
  Oinsert: OTab, Key, Val → OTab
  Odelete: OTab, Key → OTab
  OhasKey: Key, OTab → Bool
  OlookUp: OTab, Key → Val
  size: OTab → Int
  getNthVal: Int, OTab → Val
  getNthKey: Int, OTab → Key
asserts
  OTab **generated by** Oempty, Oinsert
  ∀ k : Key , v: Val, t : OTab, n : Int
    OlookUp(insert(t,k,v), k_1) == **if** k = k_1 **then** v **else** OlookUp(t, k_1)
    not(OhasKey(k, Oempty))
    OhasKey(k, Oinsert(t, k_1, v)) == **if** k = k_1 **then** true **else** OhasKey(k, t)
    Odelete(insert(t,k,v), k_1) == **if** k = k_1 **then** t **else** Oinsert(Odelete(t, k_1), k, v)
    size(Oempty) = 0
    size(Oinsert(t,k,v)) = 1 + size(t)
    getNthVal(n, Oinsert(t,k,v)) == **if** n = size(t) + 1 **then** v **else** getNthVal(n, t)
    getNthKey(n, Oinsert(t,k,v)) == **if** n = size(t) + 1 **then** k **else** getNthKey(n, t)

Figure 6.3: LSL Trait for OrdTable

CoerceTrait : **trait**
  includes Table
  includes OrdTable
introduces
  toTable: OTab → Tab
asserts
  ∀ k : Key , v: Val, t : OTab, n : Int
    toTable(Oempty) = empty
    toTable(Oinsert(t,k,v)) = insert(toTable(t), k, v)

Figure 6.4: LSL Trait for coercion function `toTable`

that for every method specification

$$\{\mathrm{Pre}_{super}\}\ \mathrm{m}\ \{\mathrm{Post}_{super}\}$$

occurring in I_1 there should be a method specification

$$\{\ \mathrm{Pre}_{sub}\}\ \mathrm{m}\ \{\ \mathrm{Post}_{sub}\}$$

in I_2 such that:

$$\begin{aligned} \mathrm{Pre}_{super} \circ \phi &\Rightarrow \mathrm{Pre}_{sub} && (2)\\ \mathrm{Post}_{sub} &\Rightarrow \mathrm{Post}_{super} \circ \phi && (3) \end{aligned}$$

Here, $\mathrm{Pre}_{super} \circ \phi$ stands for the formula in which each occurrence of a variable x of sort S_1 is replaced by the expression $\phi(x')$, where x' is a fresh variable of sort S_2. We emphasize that the sorts S_1 and S_2 are the distinguished sorts of the traits and are those which `self` gets mapped to in the interface specifications I_1 and I_2 respectively.

The intuition behind this definition is that we wish to evaluate the truth value of the expressions $\mathrm{Pre}_{super} \rightarrow \mathrm{Pre}_{sub}$ and $\mathrm{Post}_{sub} \rightarrow \mathrm{Post}_{super}$ for some arbitrary value of sort S_2. The motivation for doing this is that given an arbitrary value of sort S_2, we want to determine if it is also a value of sort S_1. Since the expressions Pre_{super} and Post_{super} refer to values of sort S_1 we must first coerce the S_2 value to an S_1 value before we can evaluate the truth value of the expression. Since Pre_{sub} and Post_{sub} are already referring to S_2 values, no coercion is necessary and the predicates can be evaluated "as is" for the given value of sort S_2.

We now proceed to apply this definition to an example which involves the traits `Table` and `OrdTable` discussed previously. Figure 6.5, 6.6 and 6.7 show Larch/C++ interface specifications for the classes `Dict` and `OrdDict` which implement a table and ordered table abstract data type respectively. We can determine whether class `OrdDict` is a subtype of class `OrdDict` according to our definition by applying (2) and (3) to the common methods of these classes (i.e. `add`, `retrieve`, `remove`, and `includesKey`). Doing so, we obtain the following proof obligations which must be discharged to prove that a subtype relationship between `Dict` and `OrdDict` class holds:

add:	1.	true ⇒ true
	2.	s = Oinsert(t, k, v) ⇒ toTable(s) = insert(toTable(t), k, v)
retrieve:	3.	hasKey(k, toTable(s)) ⇒ OhasKey(k, s)

```
class Dict
{
    uses Table(dict for Tab, char* for Key, void* for Val);

    Dict()
    {
        modifies self;
        ensures self'= empty;
    }
    ~Dict()
    {
        modifies self;
        ensures trashed(self);
    }
    void add(char* key, void* pVal)
    {
        modifies self;
        ensures self' = insert(self^, key, pVal);
    }
    void* retrieve(char* key)
    {
        requires hasKey(key, self^);
        ensures result = lookUp(self', key);
    }
    void remove(char* key )
    {
        requires hasKey(key, self^);
        modifies self;
        ensures self' = delete(self^, key);
    }
    bool includesKey(char* key)
    {
        ensures result = hasKey(key, self^);
    }
    void clear(char* key)
    {
        modifies self;
        ensures self' = empty;
    }
};
```

Figure 6.5: Larch/C++ specification for Dict

```
class OrdDict : public Dict
{
    uses Table(OrdDict for OTab, char* for Key, void* for Val);
    OrdDict()
    {
        modifies self;
        ensures self' = Oempty;
    }
    ~OrdDict()
    {
        modifies self;
        ensures trashed(self);
    }
    void add(char* key, void* pVal)
    {
        modifies self;
        ensures self' = Oinsert(self^, key, pVal);
    }
    void* retrieve(char* key)
    {
        requires OhasKey(key, self^);
        ensures result = OlookUp(self', key);
    }
```

Figure 6.6: Larch/C++ specification for OrdDict (Part I of II)

```
    void remove(char* key )
    {
        requires OhasKey(key, self^);
        modifies self;
        ensures self' = Odelete(self^, key);
    }
    bool includesKey(char* key)
    {
        requires true;
        ensures result = OhasKey(key, self^);
    }
    void clear()
    {
        modifies self;
        ensures self' = empty;
    }
    char* retrieveNthKey(int n)
    {
        requires size(self^) ≥ n;
        ensures result = getNthKey(n, self^);
    }
    void* retrieveNthVal(int n)
    {
        requires size(self^) ≥ n;
        ensures result = getNthVal(n, self^);
    }
};
```

Figure 6.7: Larch/C++ specification for OrdDict (Part II of II)

	4.	r = Olookup(s, k) ⇒ r = lookup(toTable(s), k)
remove:	5.	hasKey(k, toTable(s)) ⇒ OhasKey(k, s)
	6.	s = Odelete(t, k) ⇒ toTable(s) = delete(toTable(t), k)
includesKey:	7.	true ⇒ true
	8.	r = OhasKey(k, s) ⇒ r = hasKey(k, toTable(s))

All the variables in the above expressions are implicitly universally quantified. These proof obligations can be discharged using the usual axioms and rules of inference of first-order predicate calculus with equality. This includes the rules for equational reasoning:

Reflexivity:

For all terms t, $t = t$.

Symmetry:

For all terms t_1 and t_2, $t_1 = t_2 \Rightarrow t_2 = t_1$.

Transitivity:

For all terms t_1, t_2, and t_3, $t_1 = t_2 \wedge t_2 = t_3 \Rightarrow t_1 = t_3$.

Substitutivity:

For all functions f of n arguments, if $t = t'$ then,

$f(t_1, \ldots, t, \ldots, t_n) = f(t_1, \ldots, t', \ldots, t_n)$

Instantiation:

If $t_1 = t_2$ and x is a variable, then t_1[t **for** x] = t_2[t **for** x],

where $t'[t$ **for** $x]$ stands for the term t' with all free occurrences of the variable x replaced by the term t.

For example, the second proof obligation for the **add** method (proof obligation (2)) can be proved as follows:

1. s = Oinsert(t, k, v)	(assume antecedent)
2. toTable(s) = toTable(Oinsert(t,k,v))	(by substitutivity)
3. toTable(s) = insert(toTable(t), k, v))	(by transitivity with axiom 2 of CoerceTrait)
4. s = OInsert(t,k,v) ⇒ toTable(s) = insert(toTable(t), k, v))	(discharge antecedent assumption by implication inference rule)
5. qed	

The proof for the other obligations, although somewhat more involved in some cases, are similar. To simplify the process of discharging the proof obligations, we can use a slightly more stylized format for the trait `OrdTable`. Figure 6.8 shows how this can be accomplished. In this trait, we define the trait operator `toTable` and use it to provide the definitions of the operators `Odelete`, `OhasKey`, and `OlookUp`. The resulting axiomatizations for these operations are equivalent to those specified in figure 6.3. However, the new style of definition considerably simplifies the process of discharging some of the proof obligations. As well, as it will be discussed in the following section, this new style of trait definition will facilitate the inheritance of interface specifications.

The fact that the process of discharging the proof obligations is simplified can be verified by considering proof obligations 3, 4, 5, and 8 above. In these cases, the required proof is much simpler. As an illustration, consider proof obligation 3, i.e. hasKey(k, toTable(s)) ⇒ OhasKey(k, s). This can be discharged as follows:

1. hasKey(k, toTable(s)) (assume antecedent)
2. OhasKey(k,s) (by 4^{th} axiom of figure 6.8)
3. hasKey(k, toTable(s)) ⇒ OhasKey(k.s) (discharge antecedent assumption by implication inference rule)
4. qed

As the reader can verify, this proof is considerably simpler than the one that would be required using the trait axioms of figure 6.3. For obligations 1 and 6, the same proof applies for either set of axioms (i.e. figure 6.3 and 6.8). As for obligations 2 and 7, they are so trivial that no proof is required in either case.

It should be noted that although our dictionary example does not involve any method overriding, the semantics for subtyping presented in this section is not limited to *strict inheritance*, where subtypes can only add methods. The approach is sufficiently general to handle cases where method overriding occurs, and ensures that methods are overridden in a subclass in a manner which is consistent with the definition of the method in the superclass. In other words, the approach ensures a monotonic redefinition of methods.

The proofs of the various obligations above can be automated using LP (Larch Prover) [GG89, GG90, GGH90, GG91]. LP is a theorem prover based on equational term-rewriting, for a fragment of first order logic. It has been used to analyze formal specifications written in Larch, to reason about algorithms involving concurrency, and to establish the correctness of hardware designs [GG89].

Appendix A shows a transcript of an LP session in which proof obligation (2) above is automatically discharged. Appendix B shows a transcript of an LP session in which all the above (nontrivial) proof obligations (i.e. 2-6 and 8) are automatically discharged. The ability to automate these proofs makes the use of the definition for verifying subtype relations viable by practitioners.

The ability to automate the discharging of the proof obligations results from the adaptation of America's approach to the LSL tier of Larch/C++. The proofs required to discharge the obligations can be carried out in a strictly formal manner since they involve only the manipulation of formulas of first order predicate calculus with equality. This is in contrast to the proofs used by America which make use of rigourous arguments, but which are not formal. The lack of a formal axiomatization of the abstract values used by America precludes the proofs from being fully formal, and hence from being automated. The axiomatization of the abstract values used in Larch/C++ (by LSL traits) overcomes this problem and thus offers a considerable advantage.

Another notable advantage of applying America's definition of subtyping in the context of LSL, as opposed to the mathematical models used by America, is that the process of defining the coercion function ϕ becomes much more straightforward. In the context of LSL, the coercion function can be defined by considering all the generators of a sort and writing the appropriate axiom. In the context of the abstract values used by America, there is no systematic manner for defining the coercion function ϕ. This depends on the nature of the mathematical models used as abstract values of the subtype and supertype and may sometimes require considerable ingenuity to define.

```
OrdTable : trait
    includes Table
introduces
    toTable: OTab → Tab
    Oempty: → OTab
    Oinsert: OTab, Key, Val → OTab
    Odelete: OTab, Key → OTab
    OhasKey: Key, OTab → Bool
    OlookUp: OTab, Key → Val
    size: OTab → Int
    getNthVal: Int, OTab → Val
    getNthKey: Int, OTab → Key
asserts
    OTab generated by Oempty, Oinsert
    ∀ k : Key , v: Val, t : OTab, n : Int
        toTable(Oempty) = empty
        toTable(Oinsert(t, k, v)) = insert(toTable(t), k, v)
        toTable(Odelete(t, k)) = delete(toTable(t), k)
        OhasKey(k,t) = hasKey(k,toTable(t))
        lookUp(t,k) = lookUp(toTable(t), k)
        size(Oempty) = 0
        size(Oinsert(t,k,v)) = 1 + size(t)
        getNthVal(n, Oinsert(t,k,v)) == if n = size(t) + 1 then v
                                                    else getNthVal(n, t)
        getNthKey(n, Oinsert(t,k,v)) == if n = size(t) + 1 then k )
                                                     else getNthKey(n, t)
```

Figure 6.8: Stylized LSL Trait for OrdTable

6.2 Mechanism for Inheritance of Specifications

In the dictionary example which we discussed in the previous section, the specifications for the public member functions `add`, `retrieve`, `remove`, and `includesKey` defined in class `Dict` were repeated in the specification for class `OrdDict`. This is undesirable, as the implementation of these operations are inherited from class `Dict` by class `OrdDict` and the specifications are almost exactly the same. It should not be necessary to have to repeat the specifications. Ideally, it should be possible for the specification of class `OrdDict` to inherit the specification of the inherited operations from `Dict`. In this section, we describe a mechanism for accomplishing this in general.

For clarity of exposition, we will develop the mechanism incrementally. At each step, we identify some limitations and propose modifications to overcome them. This approach will more clearly highlight all the subtle issues involved with inheritance of specifications.

If we consider the `Dict` / `OrdDict` example, the first problem preventing inheritance of the specifications for the inherited operations (i.e. `add`, `retrieve`, `remove`, and `includesKey`) is that the interface specifications in `Dict` and `OrdDict` are not even syntactically compatible. For example, the specification for the `add` member function in the `Dict` interface specification refers to the trait operator `insert` but in the `OrdDict` interface specifications it refers to `Oinsert`. This problem can be overcome simply by using the same names in traits `Table` and `OrdTable` for "common" trait operators. By "common" trait operators, we mean operators like `insert`, `lookUp`, `delete` and `hasKey` which are defined in trait `Table` and then simply redefined in trait `OrdTable` using a different name but in a semantically equivalent manner.

Figure 6.9 shows the modified trait `OrdTable` in which common trait operators are given the same names as in trait `Table`. As explained in the previous section, this is possible as LSL trait operators can be overloaded. This results in no ambiguity as far as the formal syntax and semantics of LSL are concerned. The particular operator being referred to can be determined from the context it is used. For example, in the fourth axiom of the trait, the occurrence of the operator `hasKey` on the left hand side of the equation refers to `hasKey:OTab,Key->Bool` while the occurrence on the right

OrdTable : **trait**
includes Table
introduces
toTable: OTab → Tab
Oempty: → OTab
Oinsert: OTab, Key, Val → OTab
delete: OTab, Key → Tab
hasKey: Key, OTab → Bool
lookUp: OTab, Key → Val
size: OTab → Int
getNthVal: Int, OTab → Val
getNthKey: Int, OTab → Key
asserts
OTab **generated by** Oempty, Oinsert
∀ k : Key , v: Val, t : OTab, n : Int
toTable(Oempty) = empty
toTable(Oinsert(t, k, v)) = insert(toTable(t), k, v)
toTable(delete(t, k)) = delete(toTable(t), k)
hasKey(k,t) = hasKey(k,toTable(t))
lookUp(t,k) = lookUp(toTable(t), k)
size(Oempty) = 0
size(Oinsert(t,k,v)) = 1 + size(t)
getNthVal(n, insert(t,k,v)) == **if** n = size(t) + 1 **then** v **else** getNthVal(n, t)
getNthKey(n, insert(t,k,v)) == **if** n = size(t) + 1 **then** k **else** getNthKey(n, t)

Figure 6.9: LSL Trait for OrdTable with Overloaded Operators

refers to `hasKey:Tab,Key->Bool`.

In general, given Larch/C++ interface specifications I_1 and I_2 for classes C_1 and C_2, where C_2 is a public subclass of C_1, we wish to determine under what conditions the operation specifications in I_1 can be inherited by I_2. The assertions in the pre- and post-conditions of those operations are expressed in the language defined by the used trait of Tr_1. They may not be syntactically valid if `self'` and `self^` correspond to (i.e. are mapped to) some sort other than the distinguished sort of Tr_1. To allow for inheritance of specification in general, it is necessary to determine conditions under which operation specifications defined in I_1 will be syntactically valid when inherited by I_2.

To illustrate this point, let us consider the dictionary example once again. In the interface specification of `Dict`, `self^` and `self'` are mapped to the sort `Tab` while in the interface specification of `OrdDict`, `self^` and `self'` are mapped to the sort `OTab`. In order for the specification of a `Dict` operation like `retrieve` to be "inheritable" by the interface specification of `OrdDict`, it is necessary for all the LSL terms used in the assertions of the pre- and post-condition of `retrieve`'s specification (i.e. `hasKey` and `lookUp`) to be defined for sort `OTab`. In [LC'92], it is suggested that this problem can be solved by requiring that trait operators be overloaded so as to accept arguments of the sort corresponding to `self^` and `self'` (i.e. the distinguished sort of the used trait). As we will illustrate below, this condition does not seem sufficient.

The first problem we identify with the condition stated above can be illustrated through an example. Consider the operation `remove` of class `Dict`. The **ensures** clause of the operation's specification refers to the LSL operator `delete`. When the specification of `remove` is to be inherited by the interface specification for `OrdDict` it is necessary to overload the LSL operator `delete` appropriately in `OrdTable` (the used trait of the interface specification of `OrdDict`). The signature for the operator `delete` is

delete: Tab, Key → Tab.

In [LC'92], it is mentioned that trait operators must be overloaded to allow arguments of the subtype's sort. However, no mention is made of whether the result sort must be overloaded and none of the examples involve overloading of the return sort. In the examples presented in [Lea91], the result sort is not overloaded. However, some reflection indicates that it is indeed necessary to overload the result sort. To realize this, consider the operator `delete`. If it is overloaded only for the arguments in the trait `OrdTable` then we obtain:

delete: OTab, Key → Tab.

However, this overloading is not adequate to guarantee syntactic compatiblity of the inherited assertions. For example, consider the assertion

self'= delete(self^, key).

When this assertion is inherited by `OrdDict`, the sort corresponding to self^ and self' is `OTab`. This requires the return sort of `delete` to be `OTab`. As a result, it is necessary

to overload trait operators in such a way that a return sort which corresponds to the distinguished sort of a trait be modified to correspond to the distinguished sort of the new trait.

In [LC92], it is mentioned that one has to be very careful with the treatment of equality in the context of subtyping and inheritance of specifications. It is suggested that the use of = should be prohibited and that assertions should use a user-defined trait function, called \eq, rather than =, to prevent potential problems that can arise with the use of =. Intutively, the problem with the operator = is that it is not always clear whether it should actually refer to equality of the abstract values of the supertype or of the subtype. Using a user-defined trait function \eq has the advantage that it can always be redefined in the traits used by a subclass, and thus can provide better constraints on subtypes.

However, recent experience with Larch/C++ suggests that prohibiting the use of = is unnatural and difficult to enforce[1]. As a result, it is necessary to find some satisfactory semantics for =. In [LC92] it is mentioned that an alternative to using \eq is to interpret "=" as equality of the coerced abstract values, where the values are coerced to the nominal type used in an assertion. This approach forces "=" to be interpreted as equality at the level of the supertype's abstract values. In many cases, this is not what is required and does not provide the same flexibility for redefining the meaning of equality as when \eq was used. We find it simpler and more natural to take the default interpretation of = to be the standard meaning defined by the LSL trait which defines the sorts involved in the expression where "=" is used. This useage is consistent with the standard semantics of = in LSL, and assumes that all uses of = are "well sorted", in the sense that both arguments of = will be of the same sort [GHM91, p. 11]. Below, we will introduce some requirements which will guarantee that this requirement on "=" is always satisfied in the context of inheritance of specifications. It should also be noted that our suggested approach still provides the freedom of introducing a \eq operator whenever it is necessary to override the "default" semantics of equality which we suggest.

To summarize and state the new requirements more precisely, it is necessary

[1]Private communications with Gary Leavens, March 1993.

to introduce a classification for the operators of LSL traits. Following [LG86], the operators in a trait which defines an ADT can be categorized as *basic constructors*, *extra constructors*, *basic observers*, and *extra observers*.

- *Basic Contructors:* A constructor is an operator whose range is the distinguished sort. The basic constructors are a minimal set of constructors which produce all values of the distinguished sort. In LSL, they are identified by the **generated by** clause. The basic constructors are also referred to as the *generators* of the distinguished sort.

- *Extra Constructors:* Remaining operators whose range is the distinguished sort. These are also referred to as the *extensions* of the trait.

- *Basic Observers, Extra Observers:* Operators whose domain is the distinguished sort and whose range is some other sort are called observers. The basic observers form a minimal set of such observers; all other observers (the extra observers) can be defined in terms of the basic observers.

The basic constructors and extra constructors are collectively referred to as the *constructors* of the trait while the basic observers and extra observers are collectively referred to as the *observers* of the trait.

Having introduced the above classification for trait operators, we can now give a more precise set of requirements for permitting inheritance of specifications. To do so, let us introduce the following notation:

- Let I_1 and I_2 be the Larch/C++ interface specifications for classes C_1 and C_2, where C_2 is a public subclass of C_1.

- Let Tr_1 be the used trait of I_1 and let S_1 be the distinguished sort of Tr_1.

- Let Tr_2 be the used trait of I_2 and let S_2 be the distinguished sort of Tr_2.

Then, the requirements for permitting inheritance of the operation specifications of I_1 by I_2 are as follows:

(i) Each observer operator `obsOp` defined in trait Tr_1 is overloaded in trait Tr_2 to accept input arguments of sort S_2 rather than sort S_1. Usually, the definition

of `obsOp` in Tr_2 is of the form:

$$\text{obsOp}(x_1, \ldots, x_i, \ldots, x_n) = \text{obsOp}(x_1, \ldots, \text{toUpper}(x_i) \ldots, x_n)$$

where x_i is a variable of sort S_2 and `toUpper` is the trait operator which implements the coercion function ϕ: $S_2 \rightarrow S_1$[2]. This ensures that `obsOp` is redefined in a consistent manner, so as to satisfy the semantic subtyping requirements of the previous section.

It should be noted that in this equation the two instances of the operator `obsOp` denote operators having different signatures. The instance on the left hand side of the equation denotes the operator which accepts an S_2 input argument, while the instance of the right hand side of the equation denotes the operator which accepts an S_1 input arguments.

(ii) Similarly, each extra constructor operator `consOp` defined in trait Tr_1 is overloaded in trait Tr_2 to accept input arguments of sort S_2 rather than sort S_1(if any). In addition, it must also be overloaded to return values of sort S_2 rather than sort S_1.

Usually, the definition of `consOp` in Tr_2 is of the form:

$$\text{toUpper}(\text{consOp}(x_1, \ldots, x_i, \ldots, x_n)) = \text{consOp}(x_1, \ldots, \text{toUpper}(x_i) \ldots, x_n)$$

where x_i is a variable of sort S_2 and `toUpper` is the trait operator which implements the coercion function ϕ: $S_2 \rightarrow S_1$. This ensures that `consOp` is redefined in a consistent manner, so as to satisfy the semantic subtyping requirements of the previous section.

It should be noted that in this equation the two instances of the operator `consOp` denote operators having different signatures. The instance on the left hand side of the equation denotes the operator which returns an S_2 value, while the instance of the right hand side of the equation denotes the operator which returns an S_1 value.

[2]There may be several argument variable x_i of sort S_2. The operator `toUpper` is applied to all such variables

(iii) The names of basic constructor operators defined in Tr_1 do not appear in Tr_2. This is because the basic constructors of a sort are those operators which define the basic abstract values of a sort. Because of their distinguishing characteristic, these must be different for each sort.

These requirements correspond to the process used to obtain the trait `OrdTable` of figure 6.9 from the trait `Table` of figure 6.2. However, the requirements still fall a little short of the final requirements. This can be realized by considering the specification for the `OrdTable` operator `clear`. The specification for this operator references the operator `empty` defined in trait `Table`. This is a problem because `empty` is a basic constructor for `Table`. According to the requirements above, this operator should not be redefined in trait `OrdTable`. This is intuitively clear, as `OrdTable` has its own basic constructors: `Oempty` and `Oinsert`. However, this example shows us that it is sometimes necessary to refer to LSL operators which are basic constructors in interface specifications. There are two ways to solve this problem:

(i) Override the basic constructors defined in Tr_1 in Tr_2. This approach is undesirable as it will cause an unnecessary proliferation of generators for a given sort, making the resulting trait specifications difficult to understand. For example, using this approach for `OrdTable` would result in the sort `OTab` having as constructors `Oempty`, `Oinsert`, `Odelete`, `empty`, and `insert`. The "new" extra constructors `empty` and `insert` do not serve a useful purpose in this trait. Such overloading should be avoided if possible, as it makes the resulting trait more cluttered and more difficult to understand.

(ii) Do not permit the use of basic constructors in interface specifications.

This is not as stringent a requirement as it may seem. Suppose it is desired to use some basic constructor `bOp` in an interface specification. Then, it is always possible to define in Tr_1 a new extra constructor `bOp2`, equivalent to `bOp`, which can be used in the interface specification instead. It is possible to define `bOp2` to be an operator equivalent to `bOp` by using an axiom of the form:

$$\text{bOp2}(x_1, \ldots, x_i, \ldots, x_n) = \text{bOp}(x_1, \ldots, x_i, \ldots, x_n)$$

which states that the definition of `bOp2` is exactly the same as the definition of *bOp*.

The advantage of this approach is that additional generators are added only when they are really necessary (i.e. when the interface specification needs to refer to it). In other cases, the proliferation of constructor operators can be avoided.

The second approach is clearly more desirable and we choose to add it to the other requirements. To illustrate the use of this, we can consider the dictionary example once again. We add a new constructor `new`, equivalent to `empty` to the trait `Table`, as shown in figure 6.10. We then override `new` just as we would any other extra constructor in `OrdTable`, as shown in figure 6.11. The specification of `clear` can now refer to `new` instead of `empty`, and thus provide syntactic consistency even when the specification is inherited by `OrdTable`.

With this final modification, we now have a complete set of requirements to ensure syntactic compatibility of trait operators when inheritance of interface specifications takes place. We emphasize that this mechanism of inheritance for specifications which we have described can be sensibly used only in the case of public inheritance, where the subclass implements a subtype of its superclass. Our interpretation of the inheritance of operation specifications is based on the overloading of trait operators, and this overloading is really meaningful only in the case of subtypes.

6.3 Extensions for Multiple Levels of Inheritance

For simplicity, the semantics for determining subtyping relations and the mechanism for inheritance of specifications described in the previous sections only considered the case where one class `A` is the subclass of another class `B`.

However, both notions can easily be extended to cases where the inheritance hierarchy is arbitrarily deep. For the case of subtyping relations, we can define a general subtyping relation $\leq^t$ which is the transitive closure of the subtyping relation $\leq$ which is determined using the proof theoretic semantics we have described. So, for example, to prove that a given class `z` is a subtype of another class `x`, where `z` is

Table : **trait**
  includes Integer
introduces
  empty: → Tab
  new: → Tab
  insert: Tab, Key, Val → Tab
  delete: Tab, Key → Tab
  hasKey: Key, Tab → Bool
  lookUp: Tab, Key → Val
asserts
  Tab **generated by** empty, insert
  Tab **partitioned by** hasKey, lookUp
  ∀ k, k1 : Key , v, v1 : Val, t, t1 : Tab, n : Int
    new == empty
    lookUp(insert(t,k,v), k1) == **if** k = k1 **then** v **else** lookUp(t, k1)
    not(hasKey(k, empty))
    hasKey(k, insert(t, k1, v)) == **if** k = k1 **then** true **else** hasKey(k, t)
    delete(insert(t,k,v), k1) == **if** k = k1 **then** t **else** insert(delete(t, k1), k, v)
  implies
    converts delete, hasKey, lookUp
    exempting ∀ i: Key lookUp(empty, i)

Figure 6.10: LSL Trait for Table with operator `new`

OrdTable : **trait**
includes Table
introduces
toTable: OTab → Tab
Oempty: → OTab
Oinsert: OTab, Key, Val → OTab
delete: OTab, Key → OTab
hasKey: Key, OTab → Bool
lookUp: OTab, Key → Val
new: → OTab
size: OTab → Int
getNthVal: Int, OTab → Val
getNthKey: Int, OTab → Key
asserts
OTab **generated by** Oempty, Oinsert
∀ k : Key , v: Val, t : OTab, n : Int
toTable(Oempty) = empty
toTable(Oinsert(t, k, v)) = insert(toTable(t), k, v)
toTable(delete(t, k)) = delete(toTable(t), k)
toTable(new) = new
hasKey(k,t) = hasKey(k,toTable(t))
lookUp(t,k) = lookUp(toTable(t), k)
size(Oempty) = 0
size(Oinsert(t,k,v)) = 1 + size(t)
getNthVal(n, insert(t,k,v)) == **if** n = size(t) + 1 **then** v **else** getNthVal(n, t)
getNthKey(n, insert(t,k,v)) == **if** n = size(t) + 1 **then** k **else** getNthKey(n, t)

Figure 6.11: LSL Trait for OrdTable with operator **new**

a public subclass of **y** and **y** is a public subclass of **x**, it is necessary to prove that $y \leq x$ and $z \leq y$.

Similarly, for the inheritance of specifications, the mechanism can be extended to the more general case by incrementally constructing the specification of a class one level at a time, starting from the topmost ancestor in the hierarchy. At each level, the trait defining the abstract values of a class overloads the trait operators of its immediate superclass as described in the previous section.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have argued that to provide better support for software reuse, object-oriented languages must be supplemented with a ROOCSL. We have provided the design for an ideal ROOCSL by identifying all the properties which it should have. In light of our design goals, our comparative survey of existing approaches for specifying interfaces of reusable classes indicates that Larch languages are the preferred choice. Specifically, for C++ we have selected Larch/C++.

As well, we have presented a proposal for extending Larch/C++ with a proof-theoretic semantics for verifying subtype relations and a mechanism for inheritance of specifications. We have also shown how discharging the proof obligations resulting from our semantics for verifying subtype relations can be automated using the theorem prover LP.

Although several approaches have been proposed for formally specifying the behavior of classes [CL90a, Mey88, LST91, Wil91, Ame89, LC92], none of these proposals has provided a formal definition of what class behavior is. The work presented in this thesis attempts to bridge this gap and, in so doing, demonstrates how a formal definition of class behavior can be used to evaluate the completeness of class interface specifications.

From a theoretical standpoint, an advantage of the formal definition of class behavior and completeness of behavioral specifications presented in this thesis is that they are not dependent on any specific formalism. As a result, the completeness evaluation criterion can be successfully applied to most of the existing formal specification languages.

In summary, the contributions of this thesis are:

- Identification of the critical issues involved with the reuse of classes and design

of an ideal ROOCSL.

- Survey and comparative evaluation of existing class interface specification languages and selection of Larch/C++ as an appropriate ROOCSL for C++.
- Formal definition of class behavior.
- Formal definition of complete class interface specification.
- Illustration of how the completeness of Larch/C++ class specifications can be evaluated using the formal definition of complete class interface specification.
- Proposal for extending Larch/C++ with a proof-theoretic semantics for subtype inheritance and a mechanism for inheritance of specifications.

7.2 Comparisons to Related Work

The work most closely related to the one presented in this thesis is Larch/C++. The model-theoretic subtyping semantics being developed by the developers of Larch/C++ [Lea90, Lea91, DL92] are very complex and do not offer the possibility of being automatable and usable by practitioners. Our adaptation of America's proof theoretic subtyping semantics remedies this problem.

Fresco [Wil91, Wil92b, Wil92c, Wil92a] is another approach aimed at promoting the reuse of object-oriented software components through the use of formal methods. However, the focus of Fresco is more on proving the correctness of class implementations. Specifications in Fresco can be both implementational or behavioral, depending on the context, and no essential distinction is made between the classes and types of Fresco's TCDs. In contrast, our emphasis is on the development of behavioral specifications and on the use of such specifications to promote software reuse.

The work presented in [AP92] uses VDM specifications to derive an object oriented design. In contrast, we use Larch/C++ specifications to document the design of the interfaces of reusable classes. Whereas implications relating the pre- and post-conditions of operations are used to infer relationships between the functional requirements expressed in VDM, we use implications relating the pre- and post-conditions of operations to define subtyping relations between classes.

7.3 Assessment and Relevance to Industry

The work presented in this thesis has the potential of improving the effectiveness with which C++ classes can be reused by practitioners.

First of all, this thesis has argued that Larch/C++ has many of the features required to formally document the behavior of C++ class interfaces and has demonstrated this using several examples. Such formal documentation is necessary to provide users with a precise understanding of the class's behavior. Moroever, this formal documentation provides a solid basis for testing or verifying the implementation of reusable classes, to provide potential reusers some assurance that classes will indeed behave as specified.

Although Larch/C++ does not provide support for the specification of inter-object behavior, the work presented in this thesis has extended Larch/C++ to provide support for subtype inheritance. This provides Larch/C++ with the expressive power required to specify commercially available C++ ADT classes such as the Rogue Wave^{TM} library, thereby increasing the effectiveness with which such libraries can be reused.

The characterization of class behavior introduced in this thesis closely matches a user's intuitive understanding, making it appealing to practitioners. Moreover, the level of mathematics required to apply the completeness evaluation methodology should be quite accessible to software developers. This makes the use of our approach appealing and viable in an industrial context.

Even if completely informal specifications are used to document reusable class interfaces, the work presented in this paper can still be used to improve the ability to identify instances of incompleteness in the specifications. Understanding the formal details presented in this paper can help guide one to reason informally about the completeness of these specifications. Identifying and removing instances of incompleteness in informal specifications improves the quality of these specifications, thereby improving the effectiveness with which classes can be reused.

An important feature of our definition of class behavior is that it enforces an explicit distinction between the abstract states of an object and the abstract states of the object's external environment. Existing specification languages do not make a

distinction between these two states, making the effect of a class on its environment unclear. Our methodology for evaluating the completeness of a class interface specification provides a way to make explicit and document this distinction using existing specification languages.

7.4 Future Work

As it was stated in section 2.8, the work in this thesis does not address all the desirable properties of an ideal ROOCSL. The property which is currently most lacking is the ability to specify inter-object behavior using the ROOCSL. The approach of [HHG90], briefly descibed in section 2.4, presents some promising ideas for the specification of inter-object behavior but these ideas are not fully formalized. Moreover, these ideas do not make use of a model-oriented approach and violate the encapsulation of classes. Extending Larch/C++ with an approach based on a full formalization and model oriented adaptation of the ideas presented in [HHG90], or some equivalent approach, will make it possible to specify more sizeable and complex reusable components.

Another area which this thesis has not addressed is the verification of the correctness of a class with respect to its specification. Towards that end, an interesting research problem would be to investigate the use of the characterization of class behavior described in this thesis as a means for testing class correctness. This could possibly be accomplished by the automatic generation of test suites (i.e. test traces) based on class interface specifications. Ideally, it could be shown that it is sufficient to concentrate on certain canonical traces to obtain full coverage. Failing that, statistical quality control techniques could be used to determine the test suites required for guaranteeing a predetermined quality target.

To handle a broader range of classes, the formal definition of class behavior also needs to be extended to deal with inter-object behavior [HHG90] and exceptions. The ability to deal with inter-object behavior will make it possible to investigate the feasibility of applying the completeness evaluation approach presented in this thesis to subsystems and frameworks [JF88, WBJ90] of classes.

References

[AK92] V.S. Alagar and D. Kourkopoulos. (In)completeness in specifications. Technical report, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1992.

[Ame87] P. America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bézevin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, Paris, France, 1987. Springer-Verlag. Lecture Notes in Computer Science 276.

[Ame89] P. America. A behavioral approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratory, Nederlandse Philips Bedrijven, January 1989.

[Ame90] P. America. A parallel object oriented language with inheritance and subtyping. In *Proceedings of OOPSLA/ECOOP 90'*, 1990.

[AP92] V.S. Alagar and K. Periyasamy. A methodology for deriving an object-oriented design from functional specifications. *Software Engineering Journal*, pages 247-263, July 1992.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA 1989*, 1989.

[Bid88] M. Bidoit. A generalization of initial and loose semantics. Technical Report 402, Orsay, France, 1988.

[BP86] W. Bartussek and D.L. Parnas. Using assertions about traces to write abstract specifications for software modules. In Gehani and McGettrick, editors, *Software Specification Techniques*. Addison Wesley, 1986.

[BR87] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Transactions on Software Engineering*, March 1987.

[Bud91] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 1991.

[BW87] K.B. Bruce and P. Wegner. An algebraic model of subtype and inheritance. In F. Bancilhon and P. Buneman, editors, *Database Programming Languages*. Addison-Wesley, Reading, Mass, August 1987.

[CAA] P. Colagrosso, R. Achuthan, and V.S. Alagar. Evaluating the completeness of class interface specifications for software reuse. Submitted to OOPSLA 93.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kaslow, and G. Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation Systems Research Center, 1989.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages*. ACM Press, January 1990.

[Che91] Y. Cheon. Larch/Smalltalk: A specification language for Smalltalk. Master's thesis, Iowa State University, Ames, Iowa, 1991.

[CL90a] M. Cline and D. Lea. The behavior of C++ classes. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*. Marist College, 1990.

[CL90b] M. Cline and D. Lea. Using annotated C++. In *Proceedings of the 1990 C++ At Work Conference*, 1990.

[CM90] R.H. Cobb and H.D. Mills. Engineering software under statistical quality control. *IEEE Software*, November 1990.

[Cox84] B. Cox. Message-object programming: An evolutionary technology. *IEEE Software*, January 1984.

[Cox86] B. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, Ma., first edition, 1986.

[Cox90] B. Cox. Planning the software industrial revolution. *IEEE Software*, November 1990.

[Cus91] E. Cusack. Inheritance in object oriented Z. In *Proceedings of ECOOP '91, LNCS 512*, 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[dCAC+91] D. de Champeaux, P. America, D. Coleman, R. Duke, D. Lea, and G. Leavens. Formal techniques for OO software development (panel). In *Proceedings of OOPSLA 1991*, 1991.

[Deu87] L.P. Deutsch. Levels of reuse in the Smalltalk-80 programming system. In P. Freeman, editor, *Tutorial: Software Reusability*. IEEE Computer Society Press, Washington, D.C., 1987.

[Deu89] L.P. Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In T.J. Biggerstaff and A.J. Perlis, editors, *Software Reusability, Volume II*. ACM Press, 1989.

[Dha92] K.K. Dhara. Subtyping among mutable types in object-oriented programming languages. Master's thesis, Ames, Iowa, May 1992.

[DL92] K.K. Dhara and G.T. Leavens. Subtyping for mutable types in object oriented programming languages. Technical Report TR #92-36, Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040, USA, November 1992.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1), March 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Alebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, 1985. EATCS Monographs on Theoretical Computer Science, vol. 6.

[FGJM84] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of OBJ2. *Communications of the ACM*, 1984.

[G.B91] G.Booch. *Object-Oriented Design With Applications.* 1991.

[GG89] S.J. Garland and J.V. Guttag. An overview of LP, the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science vol. 355*, Chapel Hill We., New York, 1989. Springer-Verlag.

[GG90] S.J. Garland and J.V. Guttag. Using LP to debug specifications. In *Proceedings IFIP TC2/W62.3/WG2.3 Working Conference on Programming Concepts and Methods*, New York, 1990. Elsevier.

[GG91] S. Gardland and J. Guttag. *LP Reference Manual.* Massachussets Institute Of Technology, Laboratory for Computer Science, 1991. Available as part of the LP distribution package.

[GGH90] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering.* 16(9), September 1990.

[GH78] J.V. Guttag and J.J. Horning. The algbraic specifiction of abstract data types. *Acta Informatica*, 6:27–52, 1978.

[GH91] J.V. Guttag and J.J. Horning. Introduction to LCL, a Larch/C interface language. Technical Report 74, Digital Equipment Corporation Systems Research Center, July 1991.

[GHM91] J.V. Guttag, J.J. Horning, and A. Modet. Revised report on the Larch Shared Language (version 2.3). Technical Report 58, Digital Equipment Corporation Systems Research Center, July 1991.

[GHW85a] J.V. Guttag, J. J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5), September 1985.

[GHW85b] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, 1985.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co, Reading, Mass., 1983.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, pages 80–144. Prentice-Hall, 1978.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-889, SRI International, 1988.

[Hay87] I. Hayes. In *Specification Case Studies*. Prentice-Hall International, 1987.

[HHG90] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90*, Ottawa, Canada, 1990.

[HM84] E. Horowitz and J.B. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, 10, September 1984.

[Hoa72] C.A.R Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), 1972.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of OOPSLA 91*, 1991.

[IA90] D. Ince and D. Andrews. *The software Life Cycle*. Open University Press, 1990.

[JF88] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, August/September 1988.

[JM89] Vlissides J.M. and Linton M.A. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies '89 Conference*, November 1989.

[Jon84] T.C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, 10(5), September 1984.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, second edition, 1990.

[Jon91] K.D. Jones. Lm3: A Larch interface language for modula-3, a definition and introduction. version 1.0. Technical Report 72, Digital Equipment Corporation Systems Research Center, 1991.

[KM90] T. Korson and J.D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9), September 1990.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24, 1992.

[Lal89] W.R. Lalonde. Designing families of data types using examplars. *ACM Transactions on Programming Languages and Systems*, 11(2), April 1989.

[LC92] G.T. Leavens and Y. Cheon. Preliminary design of Larch/C++. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch*. Springer-Verlag, 1992. Workshops in Computer Science Series.

[Lea90] G.T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, July 1990.

[Lea91] G. T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4), July 1991.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specifications in Software Development*. The MIT Press, Mcgraw-Hill Book Company, 1986.

[LG87] R.G. Lanergan and C.A. Grasso. Softwre engineering with reusable designs and code. In P. Freemen, editor, *Tutorial: Software Reuse*. The IEEE Computer Society Press, 1987.

[Lis88] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5), May 1988. Revised version of the keynote address given at OOPSLA'87.

[LST91] D. Luckham, S. Sankar, and S. Takahashi. Two dimensional pinpointing: Debugging with formal specifications. *IEEE Software*, January 1991.

[LTP86] W.R. Lalonde, D.A. Thomas, and J.R. Pugh. An exemplar based Smalltalk. In Norman Meyrowitz, editor, *Proceedings of OOPSLA '86 Conference*, Portland, Oregon, 1986.

[Luc91] D. Luckham. *Programming With Specifications: An Introduction to Anna, A Language for Specifying ADA Programs*. Springer-Verlag, 1991.

[LW90] G.T. Leavens and W.E. Weihl. Reasoning about object-oriented programs that use subtypes. *SIGPLAN Notices*, 25(10), October 1990. Proceedings of ECOOP/OOPSLA '90.

[Mey85] B. Meyer. On formalism in specification. *IEEE Software*, January 1985.

[Mey87] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, March 1987.

[Mey88] B. Meyer. *Object-Oriented Software Constructions*. Prentice-Hall, 1988.

[RC89] V. Russo and R.H. Campbell. Virtual memory and backing storage management in multiprocessor operating systems using class hierarchical design. In *Proceedings of OOPSLA '89*, 1989.

[SA89] Gossain S. and D.B. Anderson. Designing a class hierarchy for domain representation and reusability. In *Proceedings of TOOLS '89*, Paris, France, November 1989.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, volume 21, Portland, Oregon, September 1986.

[Som89] I. Sommerville. *Software Engineering*. Addison Wesley, third edition, 1989.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Prentice-Hall International, 1988.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 1989.

[ST87] D.T. Sanella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150 178, 1987.

[Sta84] T.A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5), September 1984.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading MA, second edition, 1991.

[WBJ90] R. Wirfs-Brock and R.E. Johnson. Surveying current research in object oriented design. *In Communications of the ACM*, 33(9), September 1990.

[WBW89] A. Wirfs-Brock and B. Wilkerson. Variables limit reusability. *JOOP*, pages 34–40, May/June 1989.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[Wil91] A. Wills. Capsules and types in Fresco: Program verification in Smalltalk. In *Proceedings of ECOOP '91, LNCS 512*, 1991.

[Wil92a] A. Wills. *Formal Specification of Object-Oriented Programs.* PhD thesis, October 1992. Preliminary Draft.

[Wil92b] A. Wills. Refinement in Fresco. In K.D. Lano, editor, *Case Studies in object-oriented refinement.* Prentice-Hall, 1992. To appear.

[Wil92c] A. Wills. Specification in Fresco. In S. Stepney, editor, *Comparative Study of Object-Oriented Specification Methods.* Prentice-Hall, 1992. To appear.

[Win83] J. Wing. A two-tiered approach for specfying programs. Technical Report TR-299, Massachussets Institute of Technology, Laboratory for Computer Science, 1983.

[Win87] J. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1), January 1987.

[Win90] J. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9), September 1990.

[WZ88] P. Wegner and S. Zdonik. Inheritance as an incremental modification technique, or what like is and isn't like. In *European Conference on Object Oriented Programming*, Norway, August 1988.

Appendix A

Using LP To Discharge One Sample Subtyping Proof Obligation

In this appendix we demonstrate how the Larch theorem prover LP can be used to discharge the subtyping obligations for the proof obligation s = Oinsert(t, k, v) ⇒ toTable(s) = insert(toTable(t), k, v) discussed in section 6.1.

The LP command file which can be used to successfully accomplish is shown below. This is followed by the LP log file which results from the execution of the commands. The log file indicates that the required proof obligations were successfully discharged by LP.

LP command file:

```
set script OTable
set log OTable


declare sorts
  Tab, Key, Val, OTab
  ..

declare operators
  \neq: Bool, Bool -> Bool
  \neq: Tab, Tab -> Bool
  \neq: Key, Key -> Bool
  \neq: Val, Val -> Bool
  \neq: OTab, OTab -> Bool
  Oempty: -> OTab
  Oinsert: OTab, Key, Val -> OTab
  Odelete: OTab, Key -> OTab
  OhasKey: Key, OTab -> Bool
  OlookUp: OTab, Key -> Val
  toTable: OTab -> Tab
  empty: -> Tab
  insert: Tab, Key, Val -> Tab
  delete: Tab, Key -> Tab
  hasKey: Key, Tab -> Bool
  lookUp: Tab, Key -> Val
  ..

set automatic-ordering off


declare variables
  k: Key
  k1: Key
  v: Val
```

```
  v1: Val
  t: OTab
  t: Tab
  t1: OTab
  t1: Tab
  ..


% main trait: OTable

set name OTable

assert
  OTab generated by Oempty, Oinsert
  ..
assert
  OTab partitioned by OhasKey, OlookUp
  ..
assert
  toTable(Oempty) = empty
  toTable(Oinsert(t:OTab, k, v)) = insert(toTable(t:OTab), k, v)
  toTable(Odelete(t:OTab, k)) = delete(toTable(t:OTab), k)
  OhasKey(k, t:OTab) = hasKey(k, toTable(t:OTab))
  OlookUp(t:OTab, k) = lookUp(toTable(t:OTab), k)
  ..

% subtrait 1: Table

set name Table

assert
  Tab generated by empty, insert
  ..
assert
  Tab partitioned by hasKey, lookUp
  ..
assert
  lookUp(insert(t:Tab, k, v), k1) == if(k = k1, v, lookUp(t:Tab, k1))
  hasKey(k, empty) == false
  hasKey(k, insert(t:Tab, k1, v)) == if(k = k1, true, hasKey(k, t.Tab))
  delete(insert(t:Tab, k, v), k1) == if(k = k1, t:Tab, insert(delete(t:Tab,
    k1), k, v))
  ..

set automatic-ordering on

%% End of input from file '/users/piero/thesis/myThesis/app/proofs/OTable_Axioms lp'.


declare variables
  k: Key
  v: Val
  s: OTab
  x: OTab
  b: Bool
  ..


% main trait: OTable

set name OTableTheorem

prove
  s = Oinsert(x, k, v) => toTable(s) = insert(toTable(x), k, v)
  ..
```

```
  resume by =>
  <> 1 subgoal for proof of =>
    complete
    [] => subgoal
  [] conjecture
qed
```

Log File:

```
Larch Prover (27 November 1991) logging on 11 March 1993 19:47:42 to
'/users/piero/thesis/myThesis/app/proofs3/OTable.lplog'.

LP1.3:

LP1.4:

LP1.5: declare sorts
  Tab, Key, Val, OTab
  ..


LP1.6:

LP1.7: declare operators
  \neq: Bool, Bool -> Bool
  \neq: Tab, Tab -> Bool
  \neq: Key, Key -> Bool
  \neq: Val, Val -> Bool
  \neq: OTab, OTab -> Bool
  Oempty: -> OTab
  Oinsert: OTab, Key, Val -> OTab
  Odelete: OTab, Key -> OTab
  OhasKey: Key, OTab -> Bool
  OlookUp: OTab, Key -> Val
  toTable: OTab -> Tab
  empty: -> Tab
  insert: Tab, Key, Val -> Tab
  delete: Tab, Key -> Tab
  hasKey: Key, Tab -> Bool
  lookUp: Tab, Key -> Val
  ..


LP1.8:

LP1.9: set automatic-ordering off

Automatic-ordering is now 'off'.

LP1.10:

LP1.11:

LP1.12: declare variables
  k: Key
  k1: Key
  v: Val
  v1: Val
  t: OTab
  t: Tab
  t1: OTab
  t1: Tab
  ..
```

```
LP1.13:

LP1.14:

LP1.15: % main trait: OTable

LP1.16:

LP1.17: set name OTable

The name-prefix is now 'OTable'.

LP1.18:

LP1.19: assert
  OTab generated by Oempty, Oinsert
  ..


Added 1 induction rule named OTable.1 to the system.

LP1.20: assert
  OTab partitioned by OhasKey, OlookUp
  ..


Added 1 deduction rule named OTable.2 to the system.

The system now contains 1 deduction rule.

LP1.21: assert
  toTable(Oempty) = empty
  toTable(Oinsert(t:OTab, k, v)) = insert(toTable(t:OTab), k, v)
  toTable(Odelete(t:OTab, k)) = delete(toTable(t:OTab), k)
  OhasKey(k, t:OTab) = hasKey(k, toTable(t:OTab))
  OlookUp(t:OTab, k) = lookUp(toTable(t:OTab), k)
  ..


Added 5 equations named OTable.3, ..., OTable.7 to the system.

The system now contains 5 equations and 1 deduction rule.

LP1.22:

LP1.23: % subtrait 1: Table

LP1.24:

LP1.25: set name Table

The name-prefix is now 'Table'.

LP1.26:

LP1.27: assert
  Tab generated by empty, insert
  ..


Added 1 induction rule named Table.1 to the system

LP1.28: assert
```

```
  Tab partitioned by hasKey, lookUp
  ..


Added 1 deduction rule named Table.2 to the system.

The system now contains 5 equations and 2 deduction rules.

LP1.29: assert
  lookUp(insert(t.Tab, k, v), k1) == if(k = k1, v, lookUp(t:Tab, k1))
  hasKey(k, empty) == false
  hasKey(k, insert(t:Tab, k1, v)) == if(k = k1, true, hasKey(k, t:Tab))
  delete(insert(t:Tab, k, v), k1) == if(k = k1, t:Tab, insert(delete(t:Tab,
    k1), k, v))
  ..


Added 4 equations named Table.3, ..., Table.6 to the system.

The system now contains 9 equations and 2 deduction rules.

LP1.30.

LP1.31: set automatic-ordering on

Automatic-ordering is now 'on'.

The system now contains 9 rewrite rules and 2 deduction rules.

All equations have been oriented into rewrite rules.  The rewriting system is
NOT guaranteed to terminate.

LP1.32:

LP1.33:

LP1.34:

LP1.35: declare variables
  k: Key
  v: Val
  s. OTab
  x: OTab
  b. Bool
  ..


LP1.36:

LP1.37:

LP1.38: % main trait: OTable

LP1 39:

LP1.40: set name OTableTheorem

The name-prefix is now 'OTableTheorem'.

LP1.41:

LP1.42: prove
  s = Oinsert(x, k, v) => toTable(s) = insert(toTable(x), k, v)
  ..
```

```
The current conjecture is OTableTheorem.1.

Conjecture OTableTheorem.1:
  (Oinsert(x, k, v) = s) => (insert(toTable(x), k, v) = toTable(s)) == true
Proof suspended.

LP1.43:   resume by =>

Conjecture OTableTheorem.1: Subgoal for proof of =>
New constants: xc, kc, vc, sc
Hypothesis:
  OTableTheoremImpliesHyp.1: Oinsert(xc, kc, vc) = sc == true
Subgoal:
  OTableTheorem.1.1: insert(toTable(xc), kc, vc) = toTable(sc) == true


The current conjecture is subgoal OTableTheorem.1.1.

Added hypothesis OTableTheoremImpliesHyp.1 to the system.

Deduction rule lp_equals_is_true has been applied to equation
OTableTheoremImpliesHyp.1 to yield equation OTableTheoremImpliesHyp 1.1,
  Oinsert(xc, kc, vc) == sc,
which implies OTableTheoremImpliesHyp.1.

The system now contains 10 rewrite rules and 2 deduction rules.

Subgoal OTableTheorem.1.1: insert(toTable(xc), kc, vc) = toTable(sc) == true
Proof suspended.

LP1.44:   <> 1 subgoal for proof of =>

LP1.45:     complete

The following equations are critical pairs between rewrite rules
OTableTheoremImpliesHyp.1.1 and OTable.4.
  OTableTheorem.2: toTable(sc) == insert(toTable(xc), kc, vc)

The system now contains 1 equation, 10 rewrite rules, and 2 deduction rules

Subgoal OTableTheorem.1.1: insert(toTable(xc), kc, vc) = toTable(sc) == true
[] Proved by normalization.

The current conjecture is OTableTheorem.1.

Conjecture OTableTheorem.1:
  (Oinsert(x, k, v) = s) => (insert(toTable(x), k, v) = toTable(s)) == true
[] Proved =>.

The system now contains 10 rewrite rules and 2 deduction rules.

LP1.46:     [] => subgoal

LP1.47:   [] conjecture

LP1.48: qed

All conjectures have been proved.

End of input from file
'/users/piero/thesis/myThesis/app/proofs3/OTable.lp'.

LP2: quit
```

Appendix B

Using LP To Discharge All Sample Subtyping Proof Obligations

In this appendix we demonstrate how the Larch theorem prover LP can be used to discharge the subtyping obligations for the proof obligations 2-6 and 8 discussed in section 6.1. These proof obligations are:

2. s = Oinsert(t, k, v) ⇒ toTable(s) = insert(toTable(t), k, v)
3. hasKey(k, toTable(s)) ⇒ OhasKey(k, s)
4. r = Olookup(s, k) ⇒ r = lookup(toTable(s), k)
5. hasKey(k, toTable(s)) ⇒ OhasKey(k, s)
6. s = Odelete(t, k) ⇒ toTable(s) = delete(toTable(t), k)
8. r = OhasKey(k, s) ⇒ r = hasKey(k, toTable(s))

The LP command file which can be used to successfully accomplish is shown below. This is followed by the LP log file which results from the execution of the commands. The log file indicates that the required proof obligations were successfully discharged by LP.

LP command file:

```
set script OTable
set log OTable


declare sorts
  Tab, Key, Val, OTab
  ..

declare operators
  \neq: Bool, Bool -> Bool
  \neq: Tab, Tab -> Bool
  \neq: Key, Key -> Bool
  \neq: Val, Val -> Bool
  \neq: OTab, OTab -> Bool
  Oempty: -> OTab
  Oinsert: OTab, Key, Val -> OTab
  Odelete: OTab, Key -> OTab
  OhasKey: Key, OTab -> Bool
  OlookUp: OTab, Key -> Val
  toTable: OTab -> Tab
  empty: -> Tab
  insert: Tab, Key, Val -> Tab
  delete: Tab, Key -> Tab
  hasKey: Key, Tab -> Bool
```

```
  lookUp: Tab, Key -> Val
  ..

set automatic-ordering off


declare variables
  k: Key
  k1: Key
  v: Val
  v1: Val
  t: OTab
  t: Tab
  t1: OTab
  t1: Tab
  ..


% main trait: OTable

set name OTable

assert
  OTab generated by Oempty, Oinsert
  ..
assert
  OTab partitioned by OhasKey, OlookUp
  .
assert
  toTable(Oempty) = empty
  toTable(Oinsert(t:OTab, k, v)) = insert(toTable(t:OTab), k, v)
  toTable(Odelete(t:OTab, k)) = delete(toTable(t:OTab), k)
  OhasKey(k, t:OTab) = hasKey(k, toTable(t:OTab))
  OlookUp(t:OTab, k) = lookUp(toTable(t:OTab), k)
  ..

% subtrait 1: Table

set name Table

assert
  Tab generated by empty, insert
  ..
assert
  Tab partitioned by hasKey, lookUp
  ..
assert
  lookUp(insert(t:Tab, k, v), k1) == if(k = k1, v, lookUp(t:Tab, k1))
  hasKey(k, empty) == false
  hasKey(k, insert(t:Tab, k1, v)) == if(k = k1, true, hasKey(k, t:Tab))
  delete(insert(t:Tab, k, v), k1) == if(k = k1, t:Tab, insert(delete(t:Tab,
    k1), k, v))
  ..

set automatic-ordering on

%% End of input from file '/users/piero/thesis/myThesis/app/proofs2/OTable_Axioms.lp'.


declare variables
  k: Key
  v: Val
  s: OTab
  x: OTab
  b: Bool
```

```
  ..

% main trait: OTable

set name OTableTheorem

prove
  s = Oinsert(x, k, v) => toTable(s) = insert(toTable(x), k, v)
  ..
  resume by =>
  <> 1 subgoal for proof of =>
    complete
    [] => subgoal
  [] conjecture
qed

prove
  hasKey(k, toTable(s)) => OhasKey(k, s)
  ..
  [] conjecture
qed

prove
  v = OlookUp(s, k) => v = lookUp(toTable(s), k)
  ..
  [] conjecture
qed

% duplicate of equation in main trait: OTable
prove
  s = Odelete(x, k) => toTable(s) = delete(toTable(x), k)
  ..
  resume
  resume by =>
  <> 1 subgoal for proof of =>
    complete
    [] => subgoal
  [] conjecture
%% quit

prove
  b = OhasKey(k, s) => b = hasKey(k, toTable(s))
  ..
  qed
```

Log File:

```
Larch Prover (27 November 1991) logging on 28 February 1993 12:12:00 to
'/users/piero/thesis/myThesis/app/proofs2/OTable.lplog'.

LP1.3:

LP1.4:

LP1.5: declare sorts
  Tab, Key, Val, OTab
  ..


LP1.6:

LP1.7: declare operators
```

```
  \neq: Bool, Bool -> Bool
  \neq: Tab, Tab -> Bool
  \neq: Key, Key -> Bool
  \neq: Val, Val -> Bool
  \neq. OTab, OTab -> Bool
  Oempty: -> OTab
  Oinsert: OTab, Key, Val -> OTab
  Odelete· OTab, Key -> OTab
  OhasKey: Key, OTab -> Bool
  OlookUp: OTab, Key -> Val
  toTable: OTab -> Tab
  empty: -> Tab
  insert: Tab, Key, Val -> Tab
  delete: Tab, Key -> Tab
  hasKey: Key, Tab -> Bool
  lookUp: Tab, Key -> Val
  ..


LP1.8:

LP1.9: set automatic-ordering off

Automatic-ordering is now 'off'

LP1.10:

LP1.11·

LP1.12: declare variables
  k: Key
  k1· Key
  v: Val
  v1: Val
  t: OTab
  t  Tab
  t1: OTab
  t1· Tab
  ..


LP1.13:

LP1 14.

LP1.15: % main trait: OTable

LP1.16:

LP1.17: set name OTable

The name-prefix is now 'OTable'.

LP1.18:

LP1.19: assert
  OTab generated by Oempty, Oinsert
  ..


Added 1 induction rule named OTable.1 to the system.

LP1.20: assert
  OTab partitioned by OhasKey, OlookUp
  ..
```

```
Added 1 deduction rule named OTable.2 to the system.

The system now contains 1 deduction rule.

LP1.21: assert
  toTable(Oempty) = empty
  toTable(Oinsert(t:OTab, k, v)) = insert(toTable(t:OTab), k, v)
  toTable(Odelete(t:OTab, k)) = delete(toTable(t:OTab), k)
  OhasKey(k, t:OTab) = hasKey(k, toTable(t:OTab))
  OlookUp(t:OTab, k) = lookUp(toTable(t:OTab), k)
  ..


Added 5 equations named OTable.3, ..., OTable.7 to the system.

The system now contains 5 equations and 1 deduction rule.

LP1.22:

LP1.23: % subtrait 1: Table

LP1.24:

LP1.25: set name Table

The name-prefix is now 'Table'.

LP1.26:

LP1.27: assert
  Tab generated by empty, insert
  ..


Added 1 induction rule named Table.1 to the system.

LP1.28: assert
  Tab partitioned by hasKey, lookUp
  ..


Added 1 deduction rule named Table.2 to the system.

The system now contains 5 equations and 2 deduction rules.

LP1.29: assert
  lookUp(insert(t:Tab, k, v), k1) == if(k = k1, v, lookUp(t:Tab, k1))
  hasKey(k, empty) == false
  hasKey(k, insert(t:Tab, k1, v)) == if(k = k1, true, hasKey(k, t:Tab))
  delete(insert(t:Tab, k, v), k1) == if(k = k1, t:Tab, insert(delete(t.Tab,
    k1), k, v))
  ..


Added 4 equations named Table.3, ..., Table.6 to the system.

The system now contains 9 equations and 2 deduction rules.

LP1.30:

LP1.31: set automatic-ordering on

Automatic-ordering is now 'on'.
```

```
The system now contains 9 rewrite rules and 2 deduction rules.

All equations have been oriented into rewrite rules.  The rewriting system is
NOT guaranteed to terminate.

LP1.32:

LP1.33:

LP1.34:

LP1.35: declare variables
  k: Key
  v: Val
  s: OTab
  x. OTab
  b. Bool
  ..


LP1.36:

LP1.37:

LP1.38: % main trait: OTable

LP1.39:

LP1.40: set name OTableTheorem

The name-prefix is now 'OTableTheorem'.

LP1.41:

LP1.42: prove
  s = Oinsert(x, k, v) => toTable(s) = insert(toTable(x), k, v)
  ..


The current conjecture is OTableTheorem.1.

Conjecture OTableTheorem.1:
  (Oinsert(x, k, v) = s) => (insert(toTable(x), k, v) = toTable(s)) == true
Proof suspended.

LP1.43:   resume by =>

Conjecture OTableTheorem.1: Subgoal for proof of =>
New constants: xc, kc, vc, sc
Hypothesis:
  OTableTheoremImpliesHyp.1: Oinsert(xc, kc, vc) = sc == true
Subgoal:
  OTableTheorem.1.1: insert(toTable(xc), kc, vc) = toTable(sc) == true


The current conjecture is subgoal OTableTheorem.1.1.

Added hypothesis OTableTheoremImpliesHyp.1 to the system.

Deduction rule lp_equals_is_true has been applied to equation
OTableTheoremImpliesHyp.1 to yield equation OTableTheoremImpliesHyp.1.1,
  Oinsert(xc, kc, vc) == sc,
which implies OTableTheoremImpliesHyp.1.
```

```
The system now contains 10 rewrite rules and 2 deduction rules.

Subgoal OTableTheorem.1.1: insert(toTable(xc), kc, vc) = toTable(sc) == true
Proof suspended.

LP1.44:   <> 1 subgoal for proof of =>

LP1.45:     complete

The following equations are critical pairs between rewrite rules
OTableTheoremImpliesHyp.1.1 and OTable.4.
  OTableTheorem.2: toTable(sc) == insert(toTable(xc), kc, vc)

The system now contains 1 equation, 10 rewrite rules, and 2 deduction rules

Subgoal OTableTheorem.1.1: insert(toTable(xc), kc, vc) = toTable(sc) == true
[] Proved by normalization.

The current conjecture is OTableTheorem.1.

Conjecture OTableTheorem.1:
  (Oinsert(x, k, v) = s) => (insert(toTable(x), k, v) = toTable(s)) == true
[] Proved =>.

The system now contains 10 rewrite rules and 2 deduction rules.

LP1.46:     [] => subgoal

LP1.47:   [] conjecture

LP1.48: qed

All conjectures have been proved.

LP1.49:

LP1.50: prove
  hasKey(k, toTable(s)) => OhasKey(k, s)
  ..


The current conjecture is OTableTheorem.3.

Conjecture OTableTheorem.3: hasKey(k, toTable(s)) => OhasKey(k, s) == true
[] Proved by normalization.

Deleted equation OTableTheorem.3, which reduced to an identity

LP1.51:   [] conjecture

LP1.52: qed

All conjectures have been proved.

LP1.53:

LP1.54: prove
  v = OlookUp(s, k) => v = lookUp(toTable(s), k)
  ..


The current conjecture is OTableTheorem.4.

Conjecture OTableTheorem.4:
  (OlookUp(s, k) = v) => (lookUp(toTable(s), k) = v) == true
```

```
[] Proved by normalization.

Deleted equation OTableTheorem.4, which reduced to an identity.

LP1.55:   [] conjecture

LP1.56: qed

All conjectures have been proved.

LP1.57:

LP1.58: % duplicate of equation in main trait: OTable

LP1.59: prove
  s = Odelete(x, k) => toTable(s) = delete(toTable(x), k)
  ..


The current conjecture is OTableTheorem.5.

Conjecture OTableTheorem.5:
  (Odelete(x, k) = s) => (delete(toTable(x), k) = toTable(s)) == true
Proof suspended.

LP1.60:   resume

Conjecture OTableTheorem.5:
  (Odelete(x, k) = s) => (delete(toTable(x), k) = toTable(s)) == true
Proof suspended.

LP1.61:   resume by =>

Conjecture OTableTheorem.5: Subgoal for proof of =>
New constants: xc, kc, sc
Hypothesis:
  OTableTheoremImpliesHyp.2: Odelete(xc, kc) = sc == true
Subgoal:
  OTableTheorem.5.1: delete(toTable(xc), kc) = toTable(sc) == true


The current conjecture is subgoal OTableTheorem.5.1.

Added hypothesis OTableTheoremImpliesHyp.2 to the system.

Deduction rule lp_equals_is_true has been applied to equation
OTableTheoremImpliesHyp.2 to yield equation OTableTheoremImpliesHyp.2.1,
  Odelete(xc, kc) == sc,
which implies OTableTheoremImpliesHyp.2.

The system now contains 11 rewrite rules and 2 deduction rules.

Subgoal OTableTheorem.5.1: delete(toTable(xc), kc) = toTable(sc) == true
Proof suspended.

LP1.62:   <> 1 subgoal for proof of =>

LP1.63:     complete

The following equations are critical pairs between rewrite rules
OTableTheoremImpliesHyp.2.1 and OTable.5.
  OTableTheorem.6: toTable(sc) == delete(toTable(xc), kc)

The system now contains 1 equation, 11 rewrite rules, and 2 deduction rules.
```

```
Subgoal OTableTheorem.5.1: delete(toTable(xc), kc) = toTable(sc) == true
[] Proved by normalization.

The current conjecture is OTableTheorem.5.

Conjecture OTableTheorem.5:
  (Odelete(x, k) = s) => (delete(toTable(x), k) = toTable(s)) == true
[] Proved =>.

The system now contains 11 rewrite rules and 2 deduction rules.

LP1.64:     [] => subgoal

LP1.65:   [] conjecture

LP1.66:

LP1.67: prove
  b = OhasKey(k, s) => b = hasKey(k, toTable(s))
  ..


The current conjecture is OTableTheorem.7.

Conjecture OTableTheorem.7:
  (OhasKey(k, s) = b) => (hasKey(k, toTable(s)) = b) == true
[] Proved by normalization.

Deleted equation OTableTheorem.7, which reduced to an identity.

LP1.68:   qed

All conjectures have been proved.

End of input from file
'/users/piero/thesis/myThesis/app/proofs2/OTable.lp'.
```