9-1993

Inheritance of Interface Specifications

Gary T. Leavens
Iowa State University

Inheritance of Interface Specifications
(Extended Abstract)

Gary T. Leavens

TR #93-23
September 1993

Keywords: specification, inheritance, subtype, subclass, modularity, object-oriented, abstract data type.
1992 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and verifying and reasoning about programs — pre- and post-conditions, specification techniques;

Submitted to the Workshop on Interface Definition Languages.

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Inheritance of Interface Specifications (Extended Abstract)

Gary T. Leavens*
Department of Computer Science, 229 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu

September 14, 1993

Abstract

Four alternatives for the semantics of inheritance of specifications are discussed. The information loss and frame axiom problems for inherited specifications are also considered.

1 Introduction

An *interface specification language* (ISL) defines both how to call a module and its (functional) behavior [Win83] [Win87] [Lam89] [GHG+93]. The details of how to call a module and some aspects its behavior are specific to the particular programming language; hence in the Larch approach to interface specification [GHG+93], each ISL is tailored to a particular programming language.

What does this tailoring involve?

- The syntax for specifying interfaces is a subset of the syntax for the programming language, so it can be directly compared to the interface of a candidate implementation. This means that the ISL must use the type system of the programming language.
- Most of the semantic concepts of the programming language should be reflected in the ISL's semantics.
- The ISL should allow the specifier to use the programming language's abstraction mechanisms. For example, an ISL tailored to an object-oriented programming language (OOPL) should allow inheritance of specifications.

*This work was supported in part by the National Science Foundation under Grant CCR-9108654.

- The ISL should ease ways of reasoning about the correctness of code that are common or otherwise important. For example, an ISL for an OOPL should ease reasoning that uses *supertype abstraction* (thinking only about the types written in the program, not about the dynamically possible subtype that expressions may denote [LW90]).

It follows that when one is designing an ISL for an OOPL, one must consider both subtyping and inheritance.

1.1 Modularity

The last point in the list of ways to tailor an ISL is especially important in an object-oriented context. In order to support supertype abstraction one must pay attention to modularity. An ISL is *modular* if:

- at any point where an argument is specified to have type T, an actual argument of some subtype of T is allowed,
- when adding new subtypes of existing types, or when using specification inheritance, one need not change any existing specifications, and the meaning of the existing specifications does not change (except for allowing the new subtypes to be used),
- to specify a type, one need not be aware of the specification of any other types, except those of the supertypes of the type being specified (and their supertypes, etc.), and the argument and result types of the operations.

1.2 Subtype versus Subclass

It is important to carefully define the terms "type", "class", "subtype" and "subclass". By a *type* we mean an abstract data type (ADT), as characterized by a (behavioral) specification. By a *class* we mean an implementation module (such as classes in Smalltalk or C++).

A *subclass* is formed from another class (its superclass) by inheritance of code; this has no particular relation to behavior, as an object of a subclass may behave quite differently from an object of one of its superclasses. For example the class `IntStack` may be a subclass of `IntDEQueue`, although a `IntStack` object cannot respond to all the messages that one would want to send to a `IntDEQueue` [Sny86].

If one were to make an analogy between the notions of subtype and subclass, one would say that, by contrast, a subtype is formed from another ADT by inheritance of specifications, not code. For example, one might specify the type `IntDEQueue` by inheriting from `IntStack`, even if one chooses to implement the class `IntStack` as a subclass of the class `IntDEQueue`. The inheritance analogy is only approximate, however, as subtyping has to do with specified behavior,

not with how that behavior is specified. That is, subtyping is independent of specification inheritance. More precisely, a type S is a *subtype* of T if one can use objects of type S in a program where objects of type T are expected without any surprising results. This certainly implies that each object of type S acts like some object of type T [Sny86] [SCB+86] [Lea89]. In many practical cases it also implies that there is a homomorphic coercion function, $f_{S,T}$, that maps the values of objects of type S to values of objects of type T in such a way that for all instance operations of the supertype and for all x of the subtype S,

$$\text{SuperPreCond}(f_{S,T}(x)) \Rightarrow \text{SubPreCond}(x) \quad (1)$$

$$\text{SuperPostCond}(f_{S,T}(x)) \Leftarrow \text{SubPostCond}(x) \quad (2)$$

where "SuperPreCond" is the precondition of the supertype's instance operation, etc. [Ame87] [Ame89] [Ame91] [LW93a] [LW93b].

1.3 Plan

In the following we discuss inheritance of specifications in ISLs. Our ideas come from our work on the ISLs Larch/Smalltalk (for Smalltalk) [Che91] and Larch/C++ (for C++) [LC93b] [CL93] [LC93a], and our work on the semantics of subtyping in OOPLs [Lea89] [LW90] [Lea90] [LP91] [LW92].

2 Inheritance of Specifications

For an example, consider the types `BankAccount` and `PlusAccount`. The supertype, `BankAccount`, has just a savings account. The subtype, `PlusAccount`, also has a ("free") checking account. We want to specify instance operations such as `balance` and `pay_interest`, for `BankAccount` and have these specifications be inherited by `PlusAccount`.

The Larch/C++ interface specification of `BankAccount` is given in Figure 1. The LSL [GHG+93] trait `BankAccountTrait` it uses is presented in Figure 2. The trait `Rational` which is included by `BankAccountTrait`, is found in the Larch Shared Language Handbook [GHG+93, Appendix A.16]. The member functions are specified as `virtual`, which means that the code executed in a call such as `ba->pay_interest()` will execute code determined by the dynamic class of the object pointed to by `ba`.

In order to state the inheritance problem as clearly as possible, we specify the subtype `PlusAccount` in Figure 3 without using inheritance. The LSL trait `PlusAccountTrait` is specified in Figure 4.

2.1 The Specification Inheritance Problem

The specification inheritance problem is to state the specification of types like `PlusAccount` as succinctly as possible, and to give the specification with inheri-

```
class BankAccount {
   uses BankAccountTrait(BankAccount for Acct);
   public:
    BankAccount(double amt) {
        requires (1/100) ≤ rational(amt);
        constructs self;
        ensures approximates(amt, balance(self'), 1/100);
    }
    virtual double balance() const {
        ensures approximates(result, balance(self^), (1/100));
    }

    virtual void pay_interest(double rate) {
        requires (0/1) ≤ rate ∧ rate ≤ (1/1);
        modifies self;
        ensures approximates(toDouble(balance(self')),
                             ((1/1) + rational(rate)) × balance(self^),
                             1/100);
    }

    virtual void update(double amt) {
        modifies self;
        ensures approximates(toDouble(balance(self')),
                             balance(self^) + rational(amt), 1/100);
    }
};
```

Figure 1: Larch/C++ interface specification of the (super)type `BankAccount`.

```
BankAccountTrait(Acct): trait
  includes Rational  % defines the sort Q
  introduces
    createAcct: Q → Acct
    balance: Acct → Q
  asserts ∀ q: Q
    balance(createAcct(q)) == q
```

Figure 2: The trait that specifies the abstract values of `BankAccount` objects.

tance a semantics that matches the intended specification as nearly as possible.

For `PlusAccount`, what one wants to write is something like the specification in Figure 5. In this specification, the type `PlusAccount` inherits the specifications of `balance`, `pay_interest`, and `update`. It is hard to imagine the interface specification of `PlusAccount` being more succinct. The question is, what does

```
class PlusAccount : public BankAccount {
   uses PlusAccountTrait(PlusAccount for PA);
   public:
    PlusAccount(double savings_balance, double checking_balance) {
        requires (1/100) ≤ rational(savings_balance)
                 ∧ (0/1) ≤ rational(checking_balance);
        constructs self;
        ensures approximates(savings_balance, savings(self'), 1/100)
                ∧ approximates(checking_balance, checking(self'), 1/100);
    }
    virtual double balance() const {
        ensures approximates(result, savings(self^)+checking(self^), (1/100));
    }

    virtual void pay_interest(double rate) {
        requires (0/1) ≤ rate ∧ rate ≤ (1/1);
        modifies self;
        ensures approximates(toDouble(savings(self')),
                 ((1/1) + rational(rate)) × savings(self^), 1/100)
               ∧ approximates(toDouble(checking(self')),
                 ((1/1) + rational(rate)) × checking(self^), 1/100);
    }

    virtual void update(double amt) {
        modifies self;
        ensures approximates(toDouble(savings(self')),
                             savings(self^) + rational(amt), 1/100)
                ∧ checking(self') = checking(self^);
    }
};
```

Figure 3: Larch/C++ interface specification of the (sub)type `PlusAccount`, done without using inheritance.

Figure 5 mean? And how close is that meaning to the meaning of Figure 3?

2.2 Possible Semantics of Specification Inheritance

A little reflection is enough to convince one that what should be done is to copy each inherited operation specification from the parent specification to the inheriting type's specification [CDD+89] [DD90] [Cus91] [LC93b]. The semantic question then becomes: given that the parent's specification was written in terms of the abstract values of the parent type, how can one interpret it for the abstract values of the inheriting type? We have identified four potential answers to this question in the context of a Larch-style ISL.

```
PlusAccountTrait: trait
  introduces
    savNchk: Q,Q → PA
    checking: PA → Q
    savings: PA → Q
  asserts
    ∀ q1,q2: Q, pa: PA
    savings(savNchk(q1,q2)) == q1;
    checking(savNchk(q1,q2)) == q2
```

Figure 4: The trait that specifies the abstract values of `PlusAccount` objects.

```
class PlusAccount : public BankAccount {
   uses PlusAccountTrait(PlusAccount for PA);
   public:
    PlusAccount(double savings_balance, double checking_balance) {
        requires (1/100) ≤ rational(savings_balance)
                 ∧ (0/1) ≤ rational(checking_balance);
        constructs self;
        ensures approximates(savings_balance, savings(self'), 1/100);
                ∧ approximates(checking_balance, checking(self'), 1/100);
    }
};
```

Figure 5: Ideal interface specification of `PlusAccount` as a subtype of `BankAccount`, using specification inheritance.

1. Use the same sort of abstract values (i.e., extending the same LSL trait) for the subtype as for the supertype [GM87] [MOM90]. Since the abstract values of the types are the same, there is no problem in interpreting the parent type's specification.

2. Define a homomorphic[1] coercion function that maps the abstract values of the inheriting type to the abstract values of the parent type [Ame87] [Ame89] [Ame91] [LW93a] [LW93b]. The parent type's specification is interpreted by using this function to coerce the inheriting type's abstract values to the types assumed in its specification.

[1] Homomorphic in the sense that it commutes with the trait functions of the parent type; for subtypes it should also commute with the instance operations of the supertype in the sense of Formulae (1) and (2) above.

3. Define a homomorphic relation that relates each inheriting type's abstract value to at least one parent type abstract value, which is used to coerce the abstract values of the inheriting type to the parent type [Lea89]. The parent type's specification is interpreted by using this relation to obtain a set of parent type abstract values, and these are all used to interpret the parent type's specification.

4. Overload each trait function that takes an argument of the parent type's abstract values so that it is defined on abstract values of the inheriting type [LW90] [Lea91] [LW92]. For the abstract values of an inheriting type, these overloaded trait functions are used to interpret the inherited specification.

These approaches are discussed and compared below.

2.2.1 Using the Same Sort of Abstract Values

The approach of using the same sort for the inheriting type's specification as for the parent type specification is slick when it works. It often works when the inheriting type is a simple restriction on the parent type; for example, when a subtype's abstract values are a subset of the supertype's. However, it does not work well when the subtype's objects contain more information than the supertype's, as is the case with `PlusAccount`. While it is always possible to specify both sets of abstract values by using a disjoint union as the abstract value set of the parent type, doing so is not modular.

2.2.2 Using a Coercion Function

In our example, one can define a trait function, `toAcct`, in `PlusAccountTrait` that maps values of sort `PA` to values of sort `Acct`. There are infinitely many such mappings. The mapping which makes the meaning of Figure 5 closest to the meaning of Figure 3 is the one defined by the following axiom.

```
toAcct(savNchk(q1,q2)) == createAcct(q1+q2)
```

Since there are so many possible mappings, the desired coercion function should be specified for the inheriting type. In Larch/C++ this can be done by specifying `toAcct` in the trait `PlusAccountTrait`, and adding to the interface specification of Figure 5 a line of the following form.

```
simulates BankAccount by toAcct
```

However the requirement that the coercion be a function, and that it be homomorphic, sometimes makes specification inconvenient. Consider the specification of an abstract class `Graph`, which has no way to create objects, but is intended as the common supertype of `DirectedGraph` and `UndirectedGraph`.

One way to describe the abstract values of `Graph` is as a pair of sets: a set of nodes and a set of edges. The edges cannot be undirected, as this would make them useless for `DirectedGraph`. But then one cannot specify the abstract values of `UndirectedGraph` by identifying the edges `[n,m]` and `[m,n]`, because doing that makes having a homomorphic function from the abstract values of `UndirectedGraph` to the abstract values of `Graph` impossible. So the specifier of `UndirectedGraph` is forced to specify the abstract values without making this identification, which certainly complicates the specification of `UndirectedGraph` (see [CL93] for how this is done). Note, however, that this is not a modularity problem.

2.2.3 Using a Coercion Relation

A homomorphic relation is a generalization of a homomorphic function. Because it can coerce an inheriting type's abstract value to a set of abstract values of the supertype (viewing the relation as a set-valued function), one can avoid the inconvenience described in the previous paragraph. However the disadvantage of homomorphic relations is that there is much to prove before one is convinced that assertion evaluation is well-defined, because of the possible ambiguity in dealing with sets of abstract values [Lea89] [Lea90].

2.2.4 Overloading the Trait Functions

This approach attacks the problem of how to interpret the parent type's specification directly. It is clearly more general than the approaches above, because the others also, in effect, overload the trait functions so that they are defined on abstract values of the inheriting type. Permitting the trait functions to be overloaded in any way at all allows more flexibility, which can be exploited to partially solve the information loss problem (described in Section 3 below).

However, we recently discovered that this approach is not completely modular. Consider the specification of a type `Node`, which is used in the specification of the type `Graph`. Let us suppose that there is a trait function in the trait defining the abstract values of type `Graph` called `includesNode`, with the signature: `includesNode: Graph, Node -> Node`. When one defines a subtype of `Node`, say `ColoredNode`, then one must overload the trait function `includesNode` so that it is defined when its second argument is a `ColoredNode`. But this violates the definition of modularity of specifications, because the specifier of `ColoredNode` should not have to know about the specification of `Graph`, which is not a supertype of `Node`.

Another problem with this approach is that it requires a nonstandard interpretation of equality (=).

2.3 Discussion

Recall that in an OOPL, inheritance of code does not have to be used to make subtypes. Should inheritance of specifications necessarily be used to make subtypes? We can see no good reason for this in general. All of the approaches mentioned above work for inheriting specifications, even if the inheriting type is not a subtype of the parent type. Indeed this is a plus, as one would want the specification to be well-defined so that one can prove whether a claimed subtype relationship is legal or not.

There is also the semantic issue of what to do when multiple specifications of the same operation are inherited from different parent types. For the sake of brevity, we only offer our opinion that the best thing for a specification language to do is to prohibit the use of inheritance for such specifications; after all, the understandability of the specification is not decreased by giving the specification explicitly.

3 The Information Loss Problem

The information loss problem occurs when an inheriting type's abstract values contain more information than its parent type's. Consider what the inherited specification of `pay_interest` in Figure 5 says compared to its specification in Figure 3. From Figure 5 one can conclude that the total balance is increased by the specified interest, but the distribution between checkings and savings is not specified. Thus, besides paying interest, an implementation of `pay_interest` for `PlusAccount` is allowed to transfer money between checking and savings!

For post-conditions of the form `self' = tf(self^)`, where `tf` is a trait function, the problem may be solved by specially overloading the trait function `tf` to avoid the information loss. However, not all post-conditions take this form—witness `pay_interest`. So overloading the trait functions is not a general solution.

Meyer's OOPL Eiffel has a way to avoid information loss without resorting to complete respecification. In subtypes, the Eiffel specifier can (only) conjoin an additional assertion to the post-condition using the keyword `then` [Mey92]. For example, one would specify `pay_interest` by specifying that the ratio of the checking to the savings to the checking parts of a `PlusAccount` is unchanged by `pay_interest`, as in the following.

```
virtual void pay_interest(double rate) {
  ensures then if checking(self^) = 0 then checking(self') = 0
               else (savings(self')/checking(self'))
                    = (savings(self^)/checking(self^));
}
```

However, there is no reason to limit such shorthands to the specification of subtypes.

The information loss problem may also be amenable to solutions similar to those proposed for the frame problem (see below).

4 The Frame Problem

The frame problem is how to say "and nothing else changes" in a specification [BMR93]. In Larch/C++, a function specification has a `modifies` clause that says what objects the function is allowed to change. However, when a `modifies clause` of the form `modifies self` is inherited, it means that the abstract value as a whole may change. This may be less restrictive than intended, as extra information that in the subtype's abstract values may or may not be intended to change. The approach advocated in [BMR93] looks promising as a way to solve this.

5 Summary Position

For each inheriting type, the specifier should state a coercion function (or relation). This avoids all modularity problems. For maximum flexibility, a specification language could also allow individual trait functions to be defined, not by the coercion, but by explicit overloading. That way only overloadings that are different than those produced by the coercion function need be defined. In effect this gives inheritance with overriding to trait functions.

Allow adding conjuncts to post-conditions to ease the information loss problem without requiring complete respecification.

Acknowledgements

These ideas were developed in conjunction with Yoonsik Cheon, and other members of the local Iowa State formal methods community, especially Tim Wahls, K. Kishore Dhara, and Albert Baker. Thanks to Yoonsik, Tim, and Kishore for comments on drafts.

References

[Ame87] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, N.Y., June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[Ame89] Pierre America. A Behavioural Approach to Subtyping in Object-Oriented Programming Languages. Technical Report 443, Philips

Research Laboratories, Nederlandse Philips Bedrijven B. V., April 1989. Revised from the January 1989 version.

[Ame91] Pierre America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[BMR93] Alex Borgida, John Mylopoulos, and Raymond Reiter. '... And Nothing Else Changes': The Frame Problem in Procedure Specification. In *Proceedings Fifteenth International Conference on Software Engineering, Baltimore*, May 1993. Preliminary version obtained from the authors.

[CDD+89] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In *Formal Description Techniques (FORTE '89), Vancouver*, pages 281–296. North-Holland Publishing Co., December 1989.

[Che91] Yoonsik Cheon. Larch/Smalltalk: A Specification Language for Smalltalk. Technical Report 91-15, Department of Computer Science, Iowa State University, Ames, IA, June 1991. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[CL93] Yoonsik Cheon and Gary T. Leavens. A Quick Overview of Larch/C++. Technical Report 93-18, Department of Computer Science, Iowa State University, June 1993. To appear in the *Journal of Object-Oriented Programming*. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[Cus91] Elspeth Cusack. Object Oriented Modelling in Z For Open Distributed Systems. In *International Workshop on Open Distributed Processing*, October 1991. Obtained from the author.

[DD90] D. Duke and R. Duke. Towards a Semantics for Object-Z. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90: VDM and Z — Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG*, volume 428 of *Lecture Notes in Computer Science*, pages 244–261. Springer-Verlag, April 1990.

[GHG+93] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[GM87] Joseph A. Goguen and Jose Meseguer. Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 18–29. IEEE, June 1987.

[Lam89] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LC93a] Gary T. Leavens and Yoonsik Cheon. Larch/C++ Reference Manual. Available by anonymous ftp from ftp.cs.iastate.edu., 1993.

[LC93b] Gary T. Leavens and Yoonsik Cheon. Preliminary Design of Larch/C++. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch, July, 1992*, Workshops in Computing. Springer-Verlag, New York, N.Y., 1993.

[Lea89] Gary Todd Leavens. Verifying Object-Oriented Programs that use Subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[Lea90] Gary T. Leavens. Modular Verification of Object-Oriented Programs with Subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[Lea91] Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software*, 8(4):72–80, July 1991.

[LP91] Gary T. Leavens and Don Pigozzi. Typed Homomorphic Relations Extended with Subtypes. Technical Report 91-14, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, June 1991. Appears in the proceedings of *Mathematical Foundations of Programming Semantics '91*, Springer-Verlag, Lecture Notes in Computer Science, volume 598, pages 144-167, 1992.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about Object-oriented Programs that use Subtypes (extended abstract). *ACM SIGPLAN Notices*, 25(10):212–223, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[LW92] Gary T. Leavens and William E. Weihl. Subtyping, Modular Specification, and Modular Verification for Applicative Object-Oriented Programs. Technical Report 92-28, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 1992. Submitted for publication. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[LW93a] Barbara Liskov and Jeannette M. Wing. A New Definition of the Subtype Relation. Programming Methodology Group Memo 76, Massachusetts Institute of Technology, Laboratory for Computer Science, May 1993. To appear in the proceedings of ECOOP '93.

[LW93b] Barbara Liskov and Jeannette M. Wing. Specifications and Their Use in Defining Subtypes. To appear in the proceedings of OOPSLA '93. Obtained from the Authors, May 1993.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[MOM90] Narciso Marti-Oliet and Jose Meseguer. Inclusions and Subtypes. Technical Report SRI-CSL-90-16, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, Calif., December 1990.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. *ACM SIGPLAN Notices*, 21(11):9–16, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Win83] Jeannette Marie Wing. A Two-Tiered Approach to Specifying Programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Win87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR93-23
Submission Date: October 13, 1993