1992

Applications and Meaning of Inheritance in Software Specifications

Berzins, Valdis; Luqi; Lee, Yuh-Jeng

https://hdl.handle.net/10945/43622

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.
Applications and Meaning of Inheritance in Software Specifications

Valdis Berzins* Luqit Yuh-Jeng Lee

Computer Science Department
Naval Postgraduate School, Monterey, CA 93943

Abstract

We present a novel inheritance mechanism for a specification language. This mechanism supports stepwise refinement by combining constraints that can be inherited from several sources. Inheritance in specifications differs from inheritance in programming languages. The proposed mechanism has been designed specifically to support computer-aided requirements analysis. The main design issues for the mechanism are explained, and the application of the mechanism to requirements analysis is illustrated via examples.

1 Introduction

The value of inheritance mechanisms in object-oriented programming languages has been widely recognized. Inheritance provides an easy way to extend and adapt reusable software components without modifying their source code, and helps make software systems more flexible. Inheritance introduces a subclass structure that is useful for organizing complex systems to make them easier to understand. The subclass structure also provides an important kind of generalization that can significantly simplify programs: a single polymorphic program that depends only on the attributes (instance variables) and operations (methods) of a general class can be applied to instances of all of its subclasses, regardless of possible differences in the data structures used to represent instances of different subclasses.

Inheritance is useful in the context of specification languages as well as programming languages, and enables new and useful software development paradigms in that context. This paper discusses the inheritance mechanism provided by the specification language Spec [5, 7], and explores its application to requirements analysis.

Spec is a language for representing conceptual domain models in the problem definition phase of requirements analysis and for representing black-box specifications in the functional specification and architectural design of software systems. Black-box specifications are essential for realizing the benefits of abstractions in the software development process [2]. The critical early stages of software development are dominated by the tasks of building conceptual models of the proposed software and defining its interfaces. A precise notation for such models is needed as a medium for organizing and communicating conceptual models and as a means for supporting computer-aided design. Such conceptual models are essential for the application of abstractions, because they enable a designer to use a complex subsystem without examining its implementation. Some of the tasks that should be supported by a CAD system based on Spec include inheritance expansion, rearrangement of inheritance structures, automatic propagation of design implications, error checking [17, 18], materializing implied aspects of a design, prototyping, formal proofs, and automated classification of test results [12].

Spec has evolved from earlier specification languages [1, 19] based on extensive classroom experience in using formal specifications in multi-person projects [2]. A general description of the language can be found in [5], while a complete treatment and extensive examples of its use can be found in [8]. In this paper we concentrate on the inheritance mechanism of Spec, extending the formulation in [8], making some of its formal properties explicit, and illustrating some novel uses of inheritance in requirements analysis via examples.

An inheritance mechanism induces a generalization hierarchy, which can be useful for organizing a library of reusable components. The generalization structure can be exploited when automating the bookkeeping associated with design by stepwise refinement, which in practice involves considering alternatives [16] and occasionally backtracking to explore parallel paths. Inheritance also has some uses which are more important for specifications than for programs, and may be less familiar.

View Integration Large systems are designed by groups of people, who must work on different aspects of the same problem simultaneously. When
Standardization

A common problem in developing very large software systems is to ensure that the same commands in different subsystems have consistent interfaces and interpretations. One way to achieve this is by means of a skeleton specification for the common interfaces, which is defined once by the system architect and inherited by all of the subsystems.

Design History

Inheritance supports an incremental style of documenting the evolution of a system specification. Each enhancement in the history corresponds to another layer in the inheritance structure. This structure is most useful during requirements formulation via exploratory analysis and design, which specifications are changing rapidly and alternatives are being explored.

Factoring

Complex conceptual models can be partitioned into small, logically related groups of concepts that can be inherited in all of the contexts in which they are used. This helps make the sharing of conceptual dependencies explicit, helps make complex descriptions humanly understandable by separating them into independent parts that are smaller than system boundaries, and provides a mechanism for controlled sharing of definitions. We expect CAD tools to support rearrangement of inheritance structures corresponding to the design history into cleaner logical groupings. The tools should preserve a copy of the design history, and maintain the mapping between the two forms to support continued evolution of the specification. The factoring process is appropriate after a specification module has been fully refined and stabilized. Since proper factoring can make a specification easier to read and analyze, this process should be performed before design reviews.

Reuse

Standard building blocks for specifications can be reused by inheriting them and specializing them. Generalized types and partial constraints are essential for making interface structures and domain models reusable: if the components are too specific, they will not correspond to a wide variety of applications, and will fail to be reusable in practice. Examples of such libraries of reusable components for conceptual modeling and for defining system interfaces can be found in [8].

There has been a great deal of previous work on providing programming language support for inheritance [9, 14, 21]. Parameterization is an important mechanism for reusing designs and code [13], which is important for realizing the full benefits of inheritance.

Section 2 describes the specification language Spec used in this paper. Section 3 gives an informal explanation of the inheritance mechanism and discusses some of the design issues that affected its design. A formal transformational semantics [11, 10] for the inheritance mechanism of the Spec language can be found in [4]. Section 4 illustrates the application of inheritance to requirements analysis via examples. Section 5 presents our conclusions.

2 The Spec Language

The Spec language is based on the event model of computation and uses (second order, temporal) predicate logic for the precise definition of desired behavior. The event model is an extension of the actor model that extends the familiar precondition/postcondition style of specification to concurrent, distributed, and real-time systems in a natural way [8]. The primitives of the simplified event model are modules, messages, events, and alarms.

A module is a black box that interacts with other modules only by sending and receiving messages. Modules can represent software systems, such as Ada tasks or packages, as well as people and hardware devices. A message is a data packet that is sent from one module to another. Messages represent interactions between modules, such as calls on Ada task entries or subprograms declared in Ada package specifications. Messages can also be realized by I/O, exceptions, and other mechanisms. An event occurs when a module receives a message at a particular instant of time. Events represent both stimuli and responses, and serve as reference points for timing constraints. An alarm is a point in time that triggers a temporal event.

Spec provides language features supporting development of complex systems, such as controlled name spaces. The most important ideas of this language are modules, messages, events, localized state models, atomic transactions, parameterization, inheritance, and defined concepts.

In Spec modules are classified as functions, machines, and types. System behavior is defined using Spec MESSAGE declarations. Each MESSAGE declaration defines the required responses for all events in which a message of the declared form arrives at the module. A response consists of a set of outgoing messages that represent required future events. Responses of modules with internal states can also include an optional state transition, which is defined via a local state model. An event can have several different responses that are guarded by preconditions. Requirements on the contents of outgoing messages and the next state of the module are defined by postconditions. Preconditions and postconditions are logical assertions marked by the keywords WHEN and WHERE, respectively. For modules with internal states, the part of the postcondition specifying the requirements on state transitions is separated and marked with the keyword TRANSITION to improve readability and to syntactically distinguish intended state transitions.
Spec modules can also define concepts. A CONCEPT declaration defines a logical symbol representing a predicate, function, type, or constant. Concepts are used to concisely represent assertions in preconditions and postconditions. They support hierarchical structuring of complex definitions, and help make formal logic humanly understandable. Messages directly represent observable aspects of system behavior. Concepts represent abstract ideas that are used to state the requirements but do not appear as part of the system behavior.

3 The Spec Inheritance Mechanism

This section informally explains the Spec multiple inheritance mechanism. A Spec module can inherit the messages, concepts, and data or state model from a set of other modules. A novel aspect of inheritance in the Spec language is that each inherited definition is combined with the corresponding local definition, if there is one. This choice differs from most object-oriented programming languages, in which local definitions supersede inherited definitions with the same name. The purpose of this interpretation for inheritance in Spec is to support design by stepwise refinement as well as the applications listed in Section 1.

In cases where several definitions for the same operation are inherited, these are also combined. In most programming languages that support multiple inheritance, situations in which several methods are inherited for the same operation are either resolved by precedence rules that pick one of the methods to supersede the others (e.g. in LISP flavors) or are reported as errors (e.g. in Eiffel). Inheritance in programming languages provides a refinement capability only for virtual messages, which declare an operation but do not provide a method. Virtual messages can be inherited by subclasses that supply methods for realizing them. One reason for these restrictions on inheritance in programming languages is that reliable automated techniques for combining different versions of a program are still the subject of active research [3, 15, 6]. This paper describes a sound method for consistently combining specifications, and explains its application to an inheritance mechanism for the Spec language.

Inheritance in the Spec language can be used in ways similar to an include directive for a compiler, but it is a semantic operation rather than a text manipulation process. Specifications can be combined by forming the union of the interfaces they introduce, and forming the intersection of the associated behavioral constraints. This is illustrated by a very simple example in Figures 1 - 3. For brevity the example shows only a fragment of a complete specification, focusing on the requirements for the equal operation of an abstract data type representing rational numbers. Figure 1 shows the specification written by the analyst, which inherits the standard properties of mathematical equality, common to the equality operations of all data types, from the library unit shown in Figure 2. Figure 3 shows an inheritance-free specification equivalent to the specification in Figure 1, which is derived via an expansion process. The effect of the INHERIT clause in Figure 1 is to add some constraints to the postcondition of the message equal in addition to adding the message not_equal and the concept Transitive. The effect on the postcondition of the equal operation illustrates the combination of inherited con-

```
CONCEPT Transitive(f: function{t, t, boolean})
MESSAGE not_equal(x y: t) REPLY(b: boolean)
WHERE b <=> equal(x, y)
CONCEPT Transitive(f: function{t, t, boolean})
VALUE(b: boolean)
WHERE b <=>
   ALL(x y z: t :: f(x, y) & f(y, z) <=) f(x, z))
```

Figure 3: Equivalent specification

66
straints with constraints from a local definition. In the Spec language, a list of constraints denotes the conjunction of the individual constraints in the list. The effect of the INHERIT on the postcondition of equal illustrates the intersection of constraints, while the addition of the message not.equal illustrates the union of the interfaces. Some of the finer points in the design of the Spec inheritance mechanism are discussed in the remainder of this section. The next section gives examples of the use of these mechanisms in the context of requirements analysis and architectural design.

3.1 Algebraic properties

Inheritance in Spec is transitive: if module A inherits module B and B inherits module C, the A inherits C. This property is common with most of the inheritance mechanisms in object-oriented programming languages.

Multiple inheritance in Spec is associative and commutative: the path by which a module is inherited and the order of inheritance are irrelevant. To be precise, the specifications shown in Figure 4 are all equivalent with respect to module A, although there are differing effects on modules B and C. In the above, MOD-

MODULE A INHERIT B INHERIT C END
MODULE B ... END
MODULE C ... END

MODULE A INHERIT C INHERIT B END
MODULE B ... END
MODULE C ... END

MODULE A INHERIT B C END
MODULE B ... END
MODULE C ... END

MODULE A INHERIT C B END
MODULE B ... END
MODULE C ... END

MODULE A INHERIT B END
MODULE B INHERIT C ... END
MODULE C ... END

MODULE A INHERIT C END
MODULE B ... END
MODULE C INHERIT B ... END

Figure 4: Equivalent inheritance paths

ULE is a schema parameter ranging over the Spec keywords FUNCTION, MACHINE, TYPE, INSTANCE, and DEFINITION.

Multiple inheritance in Spec is idempotent: inheriting a module more than once is the same as inheriting it once. Thus inheriting the same module more than once via different paths does no harm, and neither does circular inheritance.

3.2 Monotonicity: upwards compatibility

The basic inheritance mechanism of Spec is monotonic. This means that if a module A inherits a module B, then every interaction with A that appears in the interface of B satisfies the specification of B. In particular, if A and B are type modules, then A defines a subtype of B, and if A and B are function modules, then A defines a subfunction of B. This means that A can be used in any context where B can be used, although A may provide services that B does not. In the context of software configuration management, this is called an upwards compatibility relationship.

The Spec INHERIT statement optional clauses that can introduce non-monotonic (not upwards compatible) behavior. An inherit statement of the form "INHERIT m RENAME n1 AS n2" substitutes the name n2 for all free occurrences of the name n1 in the module m before inheriting it. The names that are affected by RENAME clauses are messages names, concept names, state variable names, and instance variable names. This mechanism is used to resolve unintended name collisions, and is especially useful when inheriting reusable modules from a library.

The mechanism can also be used to introduce more specific meaningful names for general-purpose operations that are inherited in a specialized context. For example, when the set module inherits the partial ordering module, it renames the less.or.equal operation as subset.

The RENAME mechanism is not intended to affect module behavior, in the sense that the behavior of the renamed module should be isomorphic to the behavior of the original, although strictly speaking this property holds only in the absence of name collisions. A name collision occurs when two different messages with the same signature are coalesced by being renamed to the same name. Name collisions do not occur for messages with differing signatures because of operator overloading. An example of a name collision is "INHERIT integer RENAME plus AS combine RE NAME times AS combine". A CAD tool for detecting unintended name collisions and producing warning messages is useful.

Intentional non-monotonic changes are made via statements of the form "INHERIT m HIDE n". This hides the name n and removes all structures bound to the name in the module m before inheriting the result. Hiding is legal for the same set of names as for the renaming operation. This mechanism is used most often to supersede an inherited definition with an incompatible reformulation. This situation is common in exploratory design and analysis, and in system evolution in response to requirements changes.

The Spec language has chosen to syntactically highlight all incompatible changes via the keywords RENAME and HIDE to make it easier for CAD tools to detect incompatible changes. This supports a refinement to tools that manage induced changes in a software configuration [20]. Dependencies between source modules can be managed based on principles similar to those used by tools to keep mechanically derived files up to date, such as the UNIX "make" tool. However, make recomputes all dependent files whenever there
is any kind of change to a source file, based just on a timestamp. If an interface specification for a module is changed in an upwards compatible way, then there is no need to rederive all places where the module is used or to rederive all modules that depend on the interface specification. Since checking dependencies and rederiving source modules dependent on other source modules can be a tedious manual operation in the maintenance of a large software system, tool support for keeping track of all the places potentially impacted by a software change is useful. Avoiding rederivation in cases where it is not necessary can have significant practical value.

3.3 State and instance variables

Spec defines the behavior of types and machines based on conceptual representations consisting of sets of variables representing the state of the machine (state variables) or the information content of the values of the type (instance variables). These are called conceptual representations because they are abstract data models that need not be the same as the data structures used in the implementation. For this reason instance variables in Spec differ from the instance variables of most object-oriented programming languages.

The Spec inheritance mechanism combines instance variables by matching up variable names (coalesced union). An alternative interpretation (disjoint union) makes instance variables that are inherited from different modules artificially distinct, by using (module name, variable name) pairs as variable identifiers. The disjoint interpretation is plausible for programming languages because we might not want implementations of different aspects of the behavior to interfere with each other. We did not adopt the disjoint interpretation for the Spec inheritance mechanism because we expect different partial views of machines to interact. The application of this aspect of inheritance in the context of requirements analysis is illustrated in Section 4.

The constraints associated with a state model, such as type declarations, invariants, and initial value constraints, are combined by intersecting constraints. For logical assertions such as invariants and initial value constraints the intersection is realized as a conjunction. For type declarations the intersection is realized as the most general common subtype in the subtype lattice. For example, the result of combining a vehicle type with an airplane type is the airplane type. The result of combining two incompatible types is an empty type, which is generally an indication of a design error.

3.4 Overloading and interface refinement

Concepts and messages in the Spec language can be overloaded if they have different signatures. The signature of a concept or message consists of its name and the sequence of the types of its input parameters. The Spec inheritance mechanism matches up concepts and messages by their signatures. Operations with different signatures are inherited separately, while operations with a common signature are combined. The result of combining a message and a concept with the same signature is an error.

Each message represents a kind of stimulus recognized by the module. The response to a stimulus can be different in different cases, where each case is defined by a logical assertion called a precondition or a guard. When several different definitions for the same operation are defined, we take the conjunction of the postconditions for each case. The set of possible cases is defined by the cross product of the sets of preconditions for each of the definitions to be combined. If the preconditions defining the cases for all the definitions are the same, then most of the terms in the cross product simplify to the false guard, and can be dropped. However, inheriting operations defined by cases can cause a significant increase in the number of cases in the equivalent expanded specification.

For each case, the response can consist of zero or more outgoing messages and a state transition. The postconditions for the state transitions are combined by conjunction, as are the postconditions associated with corresponding outgoing messages. Outgoing messages are matched up by destination module, message name, and message type (normal, exception, or generator). Output parameters are matched by position, and type declarations are combined by intersection, which is explained in Section 3.3. This mechanism is useful for factoring complex decisions into independent views, thus avoiding combinatorial explosions in the number of cases to be specified.

3.5 Module type restrictions

There are five module types in Spec: FUNCTION, MACHINE, TYPE, INSTANCE, and DEFINITION. Any module can inherit the same type of module. In addition, any module can inherit a DEFINITION module, since such a module contains only concept definitions, which are compatible with all of the other module types. A MACHINE module or a TYPE module can also inherit a FUNCTION module, since everything that can appear in a function can also appear in a type or a machine. All other combinations of module types are prohibited because of potential incompatibilities.

3.6 Visibility of names

Name scoping in Spec is not affected by inheritance. Message names are globally visible in Spec. Spec concepts are local by default, but can be made visible to other modules via EXPORT and IMPORT declarations. An EXPORT declaration makes the specified concepts visible from those modules with matching IMPORT declarations. The purpose of an IMPORT declaration is to specify the location where an imported concept is defined. To avoid the need to follow chains of indirect references to find the definition of an imported concept, the Spec language does not allow inherited concepts to be exported by the inheriting modules. For this reason, EXPORT declarations are not inherited.

For example, the EXPORT declaration in Figure 2 is not shown in Figure 3. The module rational therefore does not implicitly export the concept Transitive, and cannot explicitly export it. Other modules that want to use this concept must import it directly from the module equality.
4 Using Inheritance in Requirements Analysis

This section illustrates the use of inheritance in requirements analysis using a case study of an information system for a discount store. We illustrate two main themes: using inheritance for coordinating teams of analysts, and using inheritance to record design history. We expand on the coordination aspect first.

The initial analysis identifies the systems and interfaces shown in the context diagram in Figure 5. We decompose the requirements model into three views of the proposed system, as seen from the viewpoint of each external system. These views are defined by different analysts, each of whom creates and modifies separate sets of specification modules. These specification modules are linked via inheritance relations, as shown in Figure 6. The three views are refined in parallel as shown in Figures 7 - 9. The definitions of various aspects of the state model are inherited by the external interface views because the state models are shared by the different interfaces as illustrated by the inheritance diagram shown in Figure 10. The definitions of the state views are shown in Figure 11 and Figure 12.

The work of different analysts interacts only because of shared state models. The inheritance structure makes these dependencies explicitly visible, and supports systematic coordination of their efforts. The inheritance structure also records the dependencies between different aspects of the requirements model. In particular, we can see that the receiving department is not concerned with the cash flow aspect in the current version of the requirements model.

The required coordination can be accomplished via policies and manual procedures, but the inheritance structure also makes software tool support feasible. For example, the design entry tools can be aware of the special role of state models, and of which analysts are working on which projects. Whenever a new component of a state model or a change to an existing state model is proposed, the tools can alert the other designers in the team. The important part of this approach is that the analysts do not have to examine each other's work except in the situations where the system alerts them to do so.

The analysts can examine the already-defined state views when they discover that the interface they are elaborating requires a new kind of state information. If this kind of information has already been modeled by some other analyst for a different purpose, the existing structure can be shared and reused. If the existing structure is almost but not quite appropriate,
the information provided by the tool can identify the other analyst working on the same aspect of the problem, and the two can communicate directly to explore generalizations or reformulations that can cover both purposes. However, note that such direct communication is required only to resolve incompatibilities. If the existing formulation is adequate, the second analyst can just use the first analyst’s results by adding an inheritance link, without disturbing the first analyst’s thought processes. This supports non-intrusive communication, and helps improve the efficiency of the analysts.

To illustrate the use of inheritance to record design history, we consider the repair of a deficiency in the initial requirements model. If we examine the clerk view, we can see that a sales transaction is supposed to tell the clerk how much change to provide, but the requirements do not explain the constraints on the amount of change to be reported. The analysts considers this, and realizes that information about the prices of the items is needed in order to formulate the requirements. This in turn leads to the question of where the price information comes from, and whether it can change.

It is not reasonable for the price information to be supplied as an input by the clerk, and it is also not reasonable to require all prices to remain fixed. Therefore the price information must come from the state model. Examining the components of the state model defined by all the other analysts so far, we find that this aspect of the problem has not yet been considered, and propose the new state component shown in Figure 13. This information is used to create the refinement of the clerk view shown in Figure 14. The new version of the clerk view inherits the previous version.

This modification is a consistent refinement because the previous version is inherited without any HIDE or RENAME clauses, as discussed in Section 3.2. The net effect of the refinement is to add a new postcondition to the REPLY. In order to formulate this new postcondition, it is necessary to refer to the new component of the state model (the price map) and to define a new concept (the total price for a collection of items), as shown in Figure 13. The requirements model is still by no means complete. Some of the unresolved issues are how the prices can change, what happens if the price of an item is undefined, and how the amount of a refund is related to prices of items. Some reasonable reactions to these issues are to add another external interface, to the
MACHINE clerk.view.1
INHERIT clerk.view

MESSAGE process.sale(items_sold: items,
                      payment: money)
      REPLY(change: money)
      WHERE payment =
              total.price(items_sold) + change

CONCEPT total.price(i: items)
      VALUE(t: money)
      WHERE t =
              SUM(u: upc.code :: i[u] * price[u])
END

Figure 14: Specification of refined clerk view

manager of the store, and a mechanism for updating prices.

The second issue leads to a coupling between two components of the state model: the price should be defined for every item that is in stock. This restriction is a state invariant, which cannot be stated in the current state model structure because the inventory and the price are not both visible in the same module. A revised state model structure in which the new invariant can be expressed is shown in Figure 15. The diagram shows a somewhat surprising consequence of the analysis: the receiving department interacts with the price component of the model, even though it has nothing to do with the cash flow. The interaction stems from violations of the newly discovered invariant. One possible form for this interaction is to delay processing shipments of items that do not have prices defined for them, and to send a problem report message to the manager, requesting the price to be defined so that the shipment can be processed.

We do not complete the analysis of the example here, since our goal is to illustrate the use of inheritance in requirements analysis rather than to provide a complete explanation of our systems analysis procedure. Complete examples and explanations of a method for requirements analysis via formal specifications and abstractions can be found in [8].

5 Conclusions

Inheritance is especially useful in the context of specification languages. We have explained the meaning of the inheritance mechanism of the Spec language and have illustrated its use for view integration and recording development history in the context of requirements analysis. The most important aspect of the Spec inheritance mechanism is its support for gradual refinement of requirements, by allowing inherited requirements to be further constrained.

Some of the main issues in the design of the mechanism are related to the treatment of data types and state variables. When combining data models, we have coalesced state or instance variables in cases where variables of the same name are inherited from several different parents. This interpretation is necessary to properly model interactions between decisions made by several analysts who are concurrently refining different but partially overlapping views (projections) of a requirements specification for a complex software system.

For example, updates to state variables visible in a view can affect other views that read those state variables. When state variables from different views are merged, the constraints on them are combined. Thus the type of a merged state variable must be the greatest common subtype of all the inherited versions, and the merged data invariant must be the conjunction of all the inherited invariants. Logical dependencies between aspects of the system that were thought to be independent can be discovered via this mechanism, as illustrated in the previous section.

Similar considerations apply to output variables and postconditions, but not to preconditions. The type signatures of the input variables serve to identify overloaded variants of stimuli, and preconditions serve to identify disjoint cases in the required response behaviors.

Inheritance can be formulated on semantic grounds, so that the boundaries between specification modules are almost transparent to the process. This allows the specifications to be factored and reorganized freely. Such reorganization is necessary to keep a complex specification humanly understandable, because the analysts rarely have sufficient insight to organize the specification in exactly the right way when they first start the analysis. After the analysis is complete, the shape of the problem domain becomes apparent, and with the benefit of hindsight, it becomes possible to impose a rational structure on the specification.
The one exception to this transparency is the treatment of name scoping via the import/export mechanism. In order to simplify the necessary support environment for the language, and to enable analysts to read printed specifications without resorting to mechanical aid, we have required that non-local references to defined symbols refer to the physical location of the definition, and have not allowed modules to export inherited definitions. This rules out chains of indirect references and reduces the need for page flipping when reading a large specification. This amounts to simplifying the tools for browsing a specification at the expense of complicating the tools for reorganizing the specification. We believe this is appropriate because specifications will be read much more frequently than printed specifications without resorting to mechanisms for the language, and to enable analysts to see the refinement on algorithms that complicate the semantics.

We have defined the Spec inheritance mechanism so that it supports multiple inheritance and gradual refinement. We have been able to afford clean semantics that is convenient for the users because combining constraints on specifications is much easier than combining constraints on algorithms (code merging). A useful inheritance mechanism must be realizable by the language processor as well as providing convenience to the users of the language.

Combining logical postconditions is relatively easy: all it takes is forming the proper conjunctions and computing the greatest common subtypes of the signatures. Performing the corresponding transformations on algorithms that realize the specifications is considerably more difficult. Some results on merging different versions of code can be found in [3, 15, 6]. Since the process of merging programs is not yet well understood in the general case, multiple inheritance mechanisms in some programming languages have been subjected to implementation restrictions that complicate the semantics.

We believe that it is preferable to report conflicts in cases where it is not possible to produce merged code corresponding to the semantics of the merged specifications, rather than to produce code with behavior that is hard to predict. A conservative practical design can report a conflict whenever two different algorithms for the same message must be merged if more powerful code merging techniques are not available to the compiler writers.

References


