1987

The Semantics of Inheritance in Spec

Luqi; Berzins, Valdis

Naval Postgraduate School

V. Berzins and Luqi, ''The Semantics of Inheritance in Spec", Technical Report NPS 52-87-032, Computer Science Department, Naval Postgraduate School, 1987.
https://hdl.handle.net/10945/65241

NPS52-87-032

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THE SEMANTICS OF INHERITANCE IN SPEC

Valdis Berzins

Luqi

July 1987

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

[redacted]
VALDIS BERZINS
Associate Professor
of Computer Science

Reviewed by:

[redacted]
VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[redacted]
KNEALE T. MARSHALL
Dean of Information and
Policy Science

UNCLASSIFIED
ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release Distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-87-032	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (*If applicable*)
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943	7a. NAME OF MONITORING ORGANIZATION 7b. ADDRESS (*City, State, and ZIP Code*)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Chief of Naval Research	8b. OFFICE SYMBOL (*If applicable*)
8c. ADDRESS (*City, State, and ZIP Code*) 800 N. Quincy St. Arlington, VA 22217-5000	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N0001487WR4E011

10. SOURCE OF FUNDING NUMBERS			
PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61153N	RR014-01	10P-011	

11. TITLE (*Include Security Classification*)

The Semantics of Inheritance in Spec

12. PERSONAL AUTHOR(S)
Valdis Berzins, Luqi

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	July 87	12

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Spec is a language for writing black-box specifications in the early stages of software development which supports multiple inheritance. A formal transformational semantics for a subset of the language containing the inheritance mechanism is presented, along with examples and a discussion of the issues which make it difficult to design clean multiple inheritance mechanisms in programming languages.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*)	22c. OFFICE SYMBOL
Valdis Berzins	408/646-2461	

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-24-

The Semantics of Inheritance in Spec

Valdis Berzins
Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Spec is a language for writing black-box specifications in the early stages of software development which supports multiple inheritance. A formal transformational semantics for a subset of the language containing the inheritance mechanism is presented, along with examples and a discussion of the issues which make it difficult to design clean multiple inheritance mechanisms in programming languages.

1. Introduction

Spec is a language for writing 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 [3]. 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 both as a medium for organizing and communicating conceptual models and as a means for supporting computer-aided design. Some of the of the tasks that should be supported by a CAD system based on Spec include error checking, materializing implied aspects of a design, test case generation, and prototyping.

Spec has evolved from earlier specification languages [2,13] based on extensive classroom experience in using formal specifications in multi-person projects [3]. A general description of the language can be found in [5]. In this paper we concentrate on the inheritance mechanism of Spec.

Specifications share many of the uses of inheritance with programming languages. An inheritance mechanism induces a generalization hierarchy, which can be useful for organizing a library of reusable components. The inheritance mechanism can also be used to adapt the behavior of an existing specification by adding new features or by further constraining existing features, without modifying the original. A generalized solution to a design problem can thus be defined only once, to be shared and adapted by many different applications, rather than being re-developed from scratch each time. The generalization structure can be exploited when automating the bookkeeping associated with design by stepwise refinement, which in practice involves considering alternatives [12] 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.

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. Such an approach should be backed up by software tools for expanding and checking the consistency of the inherited constraints.

View Integration

Large systems are designed by groups of people, who must work on different aspects of the same problem simultaneously. When a specification language supporting multiple inheritance is available, each designer can specify the view of the system through one of the external interfaces as a separate module. The whole system is materialized in a trivial module

that inherits the features of the modules defining the views. View integration would also benefit from automated tools for expanding and checking the consistency of inherited constraints.

There has been a great deal of previous work on providing programming language support for inheritance [6,11,14]. Parameterization is an important mechanism for reusing designs and code [10], which is important for realizing the full benefits of inheritance.

Section 2 describes the simple language addressed in this paper. Section 3 presents a transformational semantics [7,8] for the inheritance mechanism of the language. Section 4 contains some examples, and section 5 presents our conclusions.

2. The Microspec Language

We will consider a simplified version of Spec called Microspec which includes parametrized modules and the inheritance mechanism, but does not include state changes, exceptions, or defined concepts. A grammar for Microspec is shown below.

```
spec = { module }
module = "function" id formals parents events "end"
       | "type" id formals parents model events "end"
formals = [ "{" id { "," id } "}" ]
parents = [ "inherit" id actuals { "," id actuals } ]
actuals = [ "{" typespec { "," typespec } "}" ]
model = "model" declaration
events = { "message" id declaration response }
declaration = "(" id ":" typespec { "," id ":" typespec } ")"
response = { "when" exp "reply" declaration "where" exp }
typespec = "?" | "any" | id actuals
exp = "true" | "false" | "~" exp | exp "&" exp | "for all" declaration "::" "(" exp ")"
```

Lexical details, such as the structure of identifiers (id), are omitted because they are not needed for the purposes of the paper. An example of a Microspec specification for an abstract data type is shown in section 4.

Microspec is based on the event model of computation, and uses predicate logic for defining the desired behavior of modules. In the event model, computations are described in terms of modules, messages, and events. A module is an active black box that interacts with other modules only by sending and receiving messages. A message is data that is sent from one module to another. An event occurs at the instant of time when a module receives a message. The response to an event consists of the messages sent to other modules that are directly triggered by the event. A more detailed description of the event model can be found in [5].

An event specification of the form

message f(x: t1) when pre(x) reply (y: t2) where post(x, y)

means that whenever the module receives a message with data components x of types t1 such that pre(x) is true, it must send back a reply message with data components y of types t2 such that post(x, y) is true, where x, y are vectors of data values and t1, t2 are vectors of data types. In normal usage the preconditions of the *when* clauses associated with a message are disjoint. If an incoming message satisfies the precondition of more than one *when* clause, then the reply must satisfy the postconditions of all such clauses simultaneously.

It is most convenient to describe transformations based on an abstract syntax rather than the concrete syntax given above. The abstract syntax of Microspec is defined as a heterogeneous algebra with one sort for each of the substantive syntactic classes of Microspec. The operations of the algebra include the constructor operations for each sort, as well as some extra constants representing ill-formed syntactic objects. The sort boolean is assumed to be primitive, with the

standard two-valued model, constants *true* and *false*, and operations *and*, *or*, and *not* with the usual interpretations. The sort *id* is also assumed to be primitive, and isomorphic to the natural numbers with the standard interpretation of equality.

```
sort spec
  operations
    empty: spec
    define(id, formals, module, spec): spec
```

The *empty* and *define* operations are the primitive constructors for specifications.

```
sort module
  operations
    undefined: module
    function(parents, events): module
    type(parents, declaration, events): module
  axioms
    if conflicting(d) then type(p, d, e) = undefined
```

Undefined is an extra value introduced to represent ill-formed modules. The axiom states that types with model declarations containing type conflicts are ill-formed. In Microspec a module is either a function or an abstract data type. The full Spec language contains several other kinds of modules, which are omitted here due to lack of space.

```
sort events
  operations
    empty: events
    message(id, declaration, response, events): events

sort response
  operations
    empty: response
    when(exp, declaration, exp, response): response
      % precondition, reply declaration, postcondition, other cases
  axioms
    when(false, d, e, r) = r
```

The axiom shows that *when* clauses with uniformly false guards can be deleted (dead code elimination).

```
sort parents
  operations
    empty: parents
    add(id, actuals, parents): parents
  axioms
    add(i1, a1, add(i2, a2, p)) = add(i2, a2, add(i1, a1, p))
```

The axiom states that the order of the ancestors in the inheritance list does not matter.

```
sort formals
  operations
    empty: formals
    add(id, formals): formals
```

This sort represents the list of formal parameters of a module.

```
sort declaration
  operations
```

conflict: declaration
empty: declaration
declare(id, typespec, declaration): declaration

The constant *conflict* represents an ill-formed declaration.

sort actuals
operations
conflict: actuals
empty: actuals
add(typespec, actuals): actuals

The constant *conflict* represents an ill-formed list of actual type parameters for modules.

sort typespec
operations
inconsistent: typespec
undefined: typespec
any: typespec
variable(id): typespec
type-name(id, actuals): typespec

The type constant *inconsistent* is an extra value to denote the result of merging two incompatible type specifications. The type constant *undefined* represents a data type that has not yet been specified and is used in representing unfinished specifications. The type constant *any* represents the union of all other data types, and is used in specifying polymorphic operators.

sort exp
operations
true: exp
false: exp
not(exp): exp
and(exp, exp): exp
all(declaration, exp): exp

Expressions represent logical assertions, and include quantifiers. Only a subset of the expression syntax is shown here due to lack of space.

3. Transformational Semantics of Inheritance

The semantics of inheritance are defined by showing how to transform specifications with inheritance into equivalent specifications that do not use the inheritance mechanism. This is done by enriching the *spec* sort of the algebra defined in the previous section with an *expand* operation that performs this transformation. This requires enriching the other sorts of the algebra with additional helping functions. To reduce the number of axioms, we will incorporate the following equations for all sorts that have an *equal* operation

equal(x, x) = true
equal(x, y) = equal(y, x)

The definitions of these operations are given below.

enrich spec
operations
expand(spec): spec
fetch(id, actuals, spec): module
axioms
expand(empty) = empty
expand(define(i, f, m, s)) = define(i, f, inherit(m, s), expand(s))

fetch(i, a, empty) = undefined
fetch(i1, a, define(i2, f, m, s)) = if equal(i1, i2) then subst(a, f, m) else fetch(i1, a, s)

The *expand* operation merely expands the inherited features of each module in the specification, by means of the *inherit* operation. The problem of circular dependencies cannot arise in Microspec because only previously defined modules can be inherited. The *fetch* operation retrieves the module associated with an identifier, and substitutes actual values for the generic type parameters of the module. For simplicity we have chosen to take the first definition in case there is more than one. A more practical approach, illustrated in our treatment of declarations below, is to treat multiply declared identifiers as errors.

enrich module
operations
inherit(module, spec): module
merge(module, module): module
subst(actuals, formals, module): module
axioms
inherit(undefined, s) = undefined
inherit(function(empty, e), s) = function(empty, e)
inherit(function(add(i, a, p), e), s) = merge(inherit(function(p, e), s), inherit(fetch(i, a, s), s))
inherit(type(empty, d, e), s) = type(empty, d, e)
inherit(type(add(i, a, p), d, e), s) = merge(inherit(type(p, d, e), s), inherit(fetch(i, a, s), s))
merge(undefined, m) = undefined
merge(m, undefined) = undefined
merge(function(empty, e1), function(empty, e2)) = function(empty, merge(e1, e2))
merge(type(empty, d1, e1), type(empty, d2, e2)) = type(empty, merge(d1, d2), merge(e1, e2))
merge(type(empty, d, e1), function(empty, e2)) = type(empty, d, merge(e1, e2))
merge(function(empty, e1), type(empty, d, e2)) = undefined
subst(a, f, undefined) = undefined
subst(a, f, function(p, e)) = function(subst(a, f, p), subst(a, f, e))
subst(a, f, type(p, d, e)) = type(subst(a, f, p), subst(a, f, d), subst(a, f, e))

The *inherit* operation merges the module with all of its ancestors, both direct and indirect. The *merge* operation combines the features of several modules, declarations, or event specifications. *Merge* is not symmetric on modules because a type can inherit a function, but a function cannot inherit a type.

enrich events
operations
merge(events, events): events
fetch(id, declaration, events): response
remove(id, declaration, events): events
subst(actuals, formals, events): events
axioms
merge(empty, e) = e
merge(message(i1, d1, r1, e1), e2) =
message(i1, d1, merge(r1, fetch(i1, d1, e2)), merge(e1, remove(i1, d1, e2)))
fetch(i, d, empty) = empty
fetch(i1, d1, message(i2, d2, r, e)) =
if equal(i1, i2) and equal(d1, d2) then r else fetch(i1, d1, e)
remove(i, d, empty) = empty
remove(i1, d1, message(i2, d2, r, e)) =
if equal(i1, i2) and equal(d1, d2) then remove(i1, d1, e)
else message(i2, d2, remove(i1, d1, e))
subst(a, f, empty) = empty

subst(a, f, message(i, d, r, e)) = message(i, subst(a, f, d), subst(a, f, r), subst(a, f, e))

Event specifications to be merged are matched up by message name and argument declarations, reflecting the fact that message names can be overloaded (cf. [9]). The *fetch* and *remove* operations are used because the order of the messages in an event specification is not significant. The declarations are used in addition to message names when matching up corresponding messages because messages can be overloaded.

```
enrich response
 operations
  merge(response, response): response
  append(response, response): response
  subst(actuals, formals, response): response
 axioms
  merge(empty, r) = r
  merge(r, empty) = r
  merge(when(e1, d1, e2, r1), when(e3, d2, e4, r2)) =
   append(when(and(e1, e3), coerce(d1, d2), and(e2, e4), empty),
          append(merge(when(e1, d1, e2, empty), r2)), merge(r1, when(e3, d2, e4, r2)))
  append(empty, r) = r
  append(when(e1, d, e2, r1), r2) = when(e1, d, e2, append(r1, r2))
  subst(a, f, empty) = empty
  subst(a, f, when(e1, d, e2, r)) = when(e1, subst(a, f, d), e2, subst(a, f, r))
```

The *when* clauses of a message specification are combined by forming the cross product and then taking conjunctions of corresponding predicates. The *coerce* operation is used to combine declarations because a message has a single reply, whose type is part of the interface. For the merged module to have a consistent interface with the originals, the types of the reply messages must agree. Such interface consistency is needed if the descendents of a type with respect to inheritance are to be subtypes with respect to the usual type consistency rules (an expression E of type T1 can be bound to a variable X of type T2 if T1 is a subtype of T2). Microspec has this property, but languages that allow hiding or renaming some of the operations as a part of the inheritance mechanism do not. In contrast, the model of a type is an internal feature that does not appear in the interface, so that the model declarations of two type modules can be combined using *merge* instead of *coerce* without impacting interface consistency. We have chosen to combine similarly named components of the model rather than treating them as local names that should be artificially made disjoint because we expect inheritance to be used in view integration, where the same name is should refer to the same attribute in all views. When using multiple inheritance in a programming language to combine the features of several abstract data types, the opposite convention is more appropriate, to prevent interference between the protected and unknown representations of the multiple parents.

```
enrich parents
 operations
  subst(actuals, formals, parents): parents
 axioms
  subst(a, f, empty) = empty
  subst(a1, f, add(i, a2, p)) = add(i, subst(a1, f, a2), subst(a1, f, p))

enrich declaration
 operations
  merge(declaration, declaration): declaration
  coerce(declaration, declaration): declaration
  fetch(id, declaration): typespec
  remove(id, typespec, declaration): declaration
```

equal(declaration, declaration): boolean
conflicting(declaration): boolean
in(id, declaration): boolean
subst(actuals, formals, declaration): declaration
axioms
merge(empty, d) = d
merge(declare(i, t, d1), d2) = declare(i, merge(t, fetch(i, d2)), merge(d1, remove(i, d2)))
coerce(d1, d2) = if equal(d1, d2) then d1 else conflict
fetch(i, conflict) = undefined
fetch(i, empty) = undefined
fetch(i1, declare(i2, t2, d)) = if equal(i1, i2) = false then t2 else fetch(i1, d)
remove(i, t, conflict) = conflict
remove(i, t, empty) = empty
remove(i1, t1, declare(i2, t2, d)) =
if equal(i1, i2) and equal(t1, t2) then remove(i1, t1, d) else declare(i2, t2, remove(i1, t1, d))
equal(conflict, empty) = false
equal(conflict, declare(i, t, d)) = false
equal(empty, declare(i, t, d)) = false
equal(declare(i1, t1, d1), declare(i2, t2, d2)) = equal(i1, i2) and equal(t1, t2) and equal(d1, d2)
conflicting(conflict) = true
conflicting(empty) = false
conflicting(declare(i, t, d)) = in(i, d) or equal(t, inconsistent) or conflicting(d)
in(i, conflict) = false
in(i, empty) = false
in(i1, declare(i2, t, d)) = equal(i1, i2) or in(i1, d)
subst(a, f, conflict) = conflict
subst(a, f, empty) = empty
subst(a, f, declare(i, t, d)) = declare(i, subst(a, f, t), subst(a, f, d))

The *merge* of two declarations declares all of the identifiers in each with the least restrictive type specification consistent with both declarations.

enrich actuals
operations
merge(actuals, actuals): actuals
equal(actuals, actuals): boolean
subst(actuals, formals, actuals): actuals
axioms
merge(a1, a2) = merge(a2, a1)
merge(conflict, a) = conflict
merge(empty, empty) = empty
merge(empty, add(t, a)) = conflict
merge(add(t1, a1), add(t2, a2)) = add(merge(t1, t2), merge(a1, a2))
equal(conflict, empty) = false
equal(conflict, add(t, a)) = false
equal(empty, add(t, a)) = false
equal(add(t1, a1), add(t2, a2)) = equal(t1, t2) and equal(a1, a2)
subst(a, f, conflict) = conflict
subst(a, f, empty) = empty
subst(a1, f, add(t, a2)) = add(subst(a1, f, t), subst(a1, f, a2))

enrich typespec
operations
merge(typespec, typespec): typespec

```
  equal(typespec, typespec): boolean
  subst(actuals, formals, typespec): typespec
  replace(actuals, formals, id): typespec
 axioms
  merge(t1, t2) = merge(t2, t1)
  merge(inconsistent, t) = inconsistent
  merge(undefined, t) = t
  if equal(t, undefined) = false then merge(any, t) = t
  merge(variable(i1), variable(i2)) = if equal(i1, i2) then variable(i1) else inconsistent
  merge(variable(i1), type-name(i2, a)) = inconsistent
  merge(type-name(i1, a1), type-name(i2, a2)) =
    if equal(i1, i2) then type-name(i1, merge(a1, a2)) else inconsistent
  equal(inconsistent, undefined) = false
  equal(inconsistent, any) = false
  equal(inconsistent, variable(i)) = false
  equal(inconsistent, type-name(i, a)) = false
  equal(undefined, any) = false
  equal(undefined, variable(i)) = false
  equal(undefined, type-name(i, a)) = false
  equal(any, variable(i)) = false
  equal(any, type-name(i, a)) = false
  equal(variable(i1), type-name(i2, a)) = false
  equal(variable(i1), variable(i2)) = equal(i1, i2)
  equal(type-name(i1, a1), type-name(i2, a2)) = equal(i1, i2) and equal(a1, a2)
  subst(a, f, inconsistent) = inconsistent
  subst(a, f, undefined) = undefined
  subst(a, f, any) = any
  subst(a, f, variable(i)) = replace(a, f, i)
  subst(a1, f, type-name(i, a2)) = type-name(i, subst(a1, f, a2))
  replace(empty, empty, i) = variable(i)
  replace(add(t, a), add(i1, f), i2) = if equal(i1, i2) then t else replace(a, f, i2)
  replace(empty, add(i1, f), i2) = inconsistent
  replace(add(t, a), empty, i) = inconsistent
```

The typespecs form a generalization lattice, and the *merge* operation is a least upper bound in this lattice. The same kind of approach extends easily to lattices with a richer subtype structure, such as the one induced by the inheritance relation. A more detailed discussion of type lattices can be found in [1].

```
enrich exp
 operations
  subst(actuals, formals, exp): exp
 axioms
  subst(a, f, true) = true
  subst(a, f, false) = false
  subst(a, f, not(e)) = not(subst(a, f, e))
  subst(a, f, and(e1, e2)) = and(subst(a, f, e1), subst(a, f, e2))
  subst(a, f, all(d, e)) = all(subst(a, f, d), subst(a, f, e))
```

4. Examples

Consider the following set of parameterized functions.

```
function equality {t}
 message equal(x y: t)
```

```
  when true reply (b: boolean) where for all (a: t :: a = a ) &
    for all (a b: t :: a = b => b = a) &
    for all (a b c: t :: (a = b & b = c) => a = c)
 message not_equal(x y: t)
  when true reply (b: boolean) where b <=> ~equal(x, y)
end
```

The example agrees with the grammar for Microspec given above, except for a richer expression syntax. These functions can be inherited by a data type by means of a PARENT clause as follows.

```
type complex
inherit equality{complex}
model (re: real, im: real)
 message one()
  when true reply(r: complex) where r.re = 1.0 & r.im = 0.0
 message i()
  when true reply(r: complex) where r.re = 0.0 & r.im = 1.0
 message plus(x: complex, y: complex)
  when true reply(z: complex) where z.re = x.re + y.re & z.im = x.im + y.im
 message minus(x: complex, y: complex)
  when true reply(z: complex) where z.re = x.re - y.re & z.im = x.im - y.im
 message equal(x: complex, y: complex)
  when true reply(b: boolean) where b <=> x.re = y.re & x.im = y.im
end
```

Expanding this module with respect to the previous specification results in the following.

```
type complex
model (re: real, im: real)
 message one()
  when true reply(r: complex) where r.re = 1.0 & r.im = 0.0
 message i()
  when true reply(r: complex) where r.re = 0.0 & r.im = 1.0
 message plus(x: complex, y: complex)
  when true reply(z: complex) where z.re = x.re + y.re & z.im = x.im + y.im
 message minus(x: complex, y: complex)
  when true reply(z: complex) where z.re = x.re - y.re & z.im = x.im - y.im
 message equal(x: complex, y: complex)
  when true reply(b: boolean) where (b <=> x.re = y.re & x.im = y.im) &
    for all (a: complex :: a = a ) &
    for all (a b: complex :: a = b => b = a) &
    for all (a b c: complex :: (a = b & b = c) => a = c)
 message not_equal(x y: complex)
  when true reply (b: boolean) where b <=> ~equal(x, y)
end
```

The general properties of equality are inherited by the *equal* operation of the type *complex*, and are combined with the more specific properties relating to the model of the type. The operation *not_equal* is also inherited, along with the standard relationship to *equal*. Note the use of generic parameters to adapt the interfaces of the generic equality operations to the particular data type which inherits them.

5. Conclusions

Inheritance is especially useful in the context of specification languages. We have precisely defined the semantics of the inheritance mechanism of the Microspec language, which closely

mirrors the inheritance mechanism of the full Spec language. While the Microspec language does not include the state changes, exceptions, or defined concepts of the full Spec language, the approach presented here can be extended to cover those aspects.

The surface level conflicts that can readily detected by the expansion process are type conflicts, as illustrated by the equations. Simplification of the assertions by means of term rewrite systems can sometimes eliminate unreachable cases (when a precondition can be simplified to the constant *false*) or to detect semantic inconsistencies (when a postcondition can be simplified to the constant *false*). Some deeper conflicts can sometimes be detected by trying to prove the theorems

for all (x: t1) :: (pre(x) => for some (y: t2) :: (post(x, y)))

that show the satisfiability of each *when* clause. While complete automatic procedures for these checks are impossible, reliable partial procedures can be very useful in practice. It is still a matter of research whether it is possible to have a language that is strong enough to be useful and has a decidable class of inheritance conflicts, although a positive answer seems unlikely. Tools for explicitly expanding inheritance structures and displaying the results in ways meaningful to human designers are desirable for near term applications of inheritance.

Defining the semantics of multiple inheritance for a specification language is much easier than defining the semantics of multiple inheritance in a programming language, because merging assertions is much easier than merging code. Some results on merging different versions of code in an applicative language can be found in [4]. Since the process of merging programs is not yet well understood in the general case, multiple inheritance mechanisms in programming languages are often 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 design would report a conflict whenever two different algorithms for the same message must be merged.

1. H. Ait-Kaci, "A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures", Ph.D. Thesis, Department of Computer and Information Science, University of Pennsylvania, 1984.
2. V. Berzins and M. Gray, "Analysis and Design in MSG.84: Formalizing Functional Specifications", *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985).
3. V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development ", *Comm. of the ACM 29*, 5 (May 1986), 402-415.
4. V. Berzins, "On Merging Software Extensions", *Acta Informatica 23* (1986), 607-619.
5. V. Berzins and Luqi, "Specifying Large Software Systems in Spec", *submitted to IEEE Software*, 1987.
6. G. Birtwistle, O. Dahl, B. Myrhaug and K. Nygaard, *Simula Begin*, Auerbach Publishers, 1973.
7. M. Broy, "Transformational Semantics for Concurrent Programs", *Inf. Proc. Letters 11*, 2 (Oct. 1980), 87-91.
8. M. Broy, M. Wirsing and P. Pepper, "On the Algebraic Definition of Programming Languagess", *Trans. Prog. Lang and Systems 9*, 1 (Jan. 1987), 54-99.
9. "Ada Programming Language", American National Standards Institute/MIL-STD-1815A, DoD, 1983.
10. J. A. Goguen, "Parameterized Programming", *IEEE Trans. on Software Eng. SE-10*, 5 (Sep. 1984), 528-543.
11. A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, 1983.

12. M. Ketabchi and V. Berzins, The Theory and Practice of Representing and Managing the Refinements, Alternatives, and Versions of Composite Objects.

13. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *to appear in IEEE TSE*, 1987.

14. B. Meyer, "Eiffel: A Language for Software Engineering", TRCS85-19, Computer Science Department, University of California, Santa Barbara, Nov. 1985.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 2000 N. Beauregard Street Alexandria, VA 22311	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
Valdis Berzins Code 52Be Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	100
Chief of Naval Research 800 N. Quincy St. Arlington, VA 22217-5000	1