Polymorphism and subtyping in interfaces*

Dinesh Katiyar David Luckham John Mitchell†

Department of Computer Science
Stanford University, USA
{katiyar, luckham, mitchell}@cs.stanford.edu

Sigurd Meldal‡

Department of Informatics
University of Bergen, Norway
sigurd@ii.uib.no

Abstract

RAPIDE is a programming language framework designed for the development of large, concurrent, real-time systems by prototyping. The framework consists of a type language and default executable, specification and architecture languages, along with associated programming tools. *Interfaces* are the central construct of the type system. This document describes RAPIDE interfaces, their expressiveness and their use as a typing construct.

1 Introduction

RAPIDE is a programming language framework consisting of a types language, default executable and specification languages, and a set of tools. Although the languages, together, provide a complete programming language, RAPIDE is intended to accommodate additional implementation and specification languages. The constant of the framework is the type language, which is intended to be general enough to allow the types of relevant implementation languages to be expressed within it. The types language is a general interface definition language, expressive enough to describe the interaction and communication between fairly complex and heterogeneous program entities.

The central construct of the RAPIDE type system is the *interface type.* An interface type lists the types and objects associated with one or more *modules.* While the word "module" may suggest a static grouping of related declarations, the word is used to refer to any aggregation of objects, functions, types, or other definable program entities. In the default executable language, for example, modules serve as both compile-time groupings of related types and operations, resembling Ada packages [US 80], for example, and as run-time objects of the form found in C++ [ES90] and other object-oriented languages. A central, unifying idea is that *all types in* RAPIDE *are expressible using interface types and function types.*

The RAPIDE type system supports several forms of parameterization, which coupled with the notion of subtyping, result in very expressive forms of polymorphism. Subtyping constraints on type constituents

*This research was supported by DARPA under ONR contract N00014-92-J-1928 and by the Air Force Office of Scientific Research under Grant AFOSR-91-0354.

†Also supported by an NSF PYI Award, with matching funds from AT&T, Digital Equipment Corporation and the Powell Foundation, NSF Grant CCR-9303099 and the Wallace F. and Lucille M. Davis Faculty Scholarship.

‡Also supported by Norges forskningsråd.

of interfaces allows flexibility in the definition of non-subtypable types. RAPIDE also allows very flexible derivation mechanisms that can be used to combine and reuse existing interfaces to define new ones. RAPIDE interfaces can also express subtle visibility requirements through their *private* and *external* declarations. The rest of this document discusses these aspects at greater length.

The RAPIDE type system provides a basic set of mechanisms to describe system components and their interdependencies. It is envisioned that any language using this type system would provide connection mechanisms to make these dependencies explicit. This would allow components to be defined and wired up together easily to define larger architectures. For instance, the RAPIDE architectures language defines restricted interfaces called *services* which could be viewed, rather loosely, as defining the "pins" and "sockets" of an interface. Since RAPIDE services are just the interfaces of connectors, these are expressible as interface types. The language also provides simple constructs to connect "complementary" services, automatically ensuring that corresponding pins and sockets get wired up. This turns out to be a very useful application of RAPIDE interfaces.

This paper talks selectively about ideas from the RAPIDE type system that apply to interface languages, and attempts to motivate them in the setting of an architecture definition language. A more detailed description of the design and rationale of the type system can be found in [KLM94]. A preliminary design of the RAPIDE type system was described in [MMM91]. Some additional technical points are discussed in [KLM+92]. An early overview of the RAPIDE approach to prototyping appears in [BL90]. Details of the architecture language can be found in [RAr].

2 Interface types

An *interface* is defined by a list of declarations. These may be declarations of object names (where an object is either a module or function) or types. A declaration of an object name associates a type with the identifier, but does not give an implementation. A type declaration in an interface may give only a type name, a type name with a subtype designation, or a complete type declaration. A type name only produces a hidden "private" type, while a complete type declaration results in a "public" type, to use Ada terminology. When a subtype designation is used, the result is a form of "partially hidden" type. Overall, an interface is similar in spirit to an Ada package specification (except that packages are not types in Ada) [US 80], the signature part of a C++ class [ES90], or a Standard ML signature [MTH90] (except that Standard ML modules are only compile-time entities).

The objects of an interface type are modules with type and object constituents matching the declarations listed in the interface. (We use the term *constituent* for components of a module to avoid conflict with the term *component* for component of a system or architecture.) All objects constituents must have the type indicated in the interface. A type constituent may be any type, when the interface contains only a type name declaration, any subtype of the type given in the interface, if a subtype designation is used, or may be forced to be exactly the type given explicitly in the interface.

The interface of a module governs access to its constituents. The constituents that correspond to declarations in its interface are *visible,* which means that they may be referred to outside the module. If a module declares identifiers that do not occur in the module interface, then these are *hidden*, *i.e.*, not accessible from outside. In the case of visible type constituents, the subtype or type equality information given in the interface determines whether the type constituent will be considered a subtype of or equal to any other type in the program. This is discussed briefly below and addressed in more detail in Section 4.

Example: *A Point interface*

type Point **is interface**
    X_Val : Integer;
    Y_Val : Integer;

```
    Distance : function(P : Point)  return  Integer;
end  interface;
```

The interface type Point contains three name declarations. Modules of type Point must have two integer constituents, called X_Val and Y_Val, and a function constituent called Distance. A function computing the distance between two points requires only one argument, since the function belongs to a particular point and therefore may compute the distance between the point passed as an actual parameter and the particular point to which the function belongs. In other words, the intended use of the Distance function constituent of a point object x is to compute the distance between x and another point object y, by a call of the form x.distance(y). Of course, since the interface gives only signature information, the Distance constituent is not forced to compute the distance between two points. If the user wishes to specify that Distance must compute distance, this may be added to the interface using the specification sublanguage.

□

Example: *A Stack package*

```
type  Int_Stack_Package  is  interface
    type  Stack;
    New : function()  return  Stack;
    Push : function(X : Integer,  S : Stack)  return  Stack;
    Pop : function(S : Stack)  return  Stack;
    Top : function(S : Stack)  return  Integer;
end  interface;
```

This interface illustrates the traditional abstract data type style. An object of Int_Stack_Package must have a type constituent Stack and the function constituents listed above. It is intended that these constituents provide the standard operations on stacks. Since only the type name, Stack, is given in the interface, the type constituent of an object of type Int_Stack_Package will be hidden. More specifically, outside the body of a module Stack_Impl of type Int_Stack_Package, the type Stack_Impl.Stack will not be recognized as equal to or a subtype of any other type in the program. Using Ada terminology, the type Stack will be a private type of any Int_Stack_Package module.

□

Full type declarations permit the type-checking of incremental definitions of related types, as in the style developed for Standard ML [Mac86, Tof89]. (The relationship between Standard ML type constituents, which are considered “public” by default, and full type declarations in interfaces is discussed in [HL94, Ler94].)

Example: *Extending the functionality of abstract types*

```
type  T_Package  is  interface
    type  T;
    f  :  function(x:T)  return  T;
end  interface;

type  Ext_T_Package(t:T_Package)  is  interface
    type  T  =  t.T;
    g  :  function(x:T)  return  T;
end  interface;
```

The type T_Package defines an abstract type and a function on that type. Given an instance t of this package, one can define values of type t.T and apply the function t.f to them.

Now suppose we wish to extend the functionality of the package type with another function g such that the new function can be applied to instances of the old package type. This can be done as shown above using a full type declaration. For each implementation of T_Package, Ext_T_Package defines an interface with a type constituent that is equal to that of the particular T_Package. (Due to the standard typing rules for abstract types, the function g must be definable from constituents in the interface T_Package of the module parameter t.)

□

Subtype designations provide an intermediate level of information hiding that allows certain operations on the type to be revealed. One reason that subtype designations are particularly useful in RAPIDE is that subtype constraints may be attached to type parameters of polymorphic (or generic) functions. Therefore, giving a subtype designation may allow a type to be used more effectively in conjunction with polymorphic functions. This is illustrated in Section 4.

When a type is not declared fully in an interface, it will generally not have any subtypes. An important ramification of this is that this allows the representation of objects to be optimized, and simplifies the implementation of binary operations in many cases. Section 5 explains this in greater detail.

In all of the examples given in this section, all interface declarations are *public*. In general, a RAPIDE interface may also declare names as *private* or *external*. These are described in Section 7.

In addition to name declarations, an interface may also contain *specifications*. These are constraints on the behavior of the executable objects, and are written in the RAPIDE specification language. The purpose of specifications in interfaces is to document the intended semantics of modules formally and more precisely than can be achieved by types alone, *e.g.,* input/output constraints on parameter values of operations or restrictions on the order in which the operations can occur. Specifications are not considered during compile-time type checking, but may play an important role in the prototyping process. Besides serving as the basis for more-or-less formal reasoning about the correctness of particular implementations or refinements, one may apply other tools to the specification part of an interface declaration. Of particular interest in a prototyping context is the possibility that a compiler may *synthesize* a module's behavior, given only its type's specification. RAPIDE is intended for the development of distributed systems, with event-based programming as a central concept. The specification language of RAPIDE reflects this. A specification may consist of a set of non-constructive (but run-time checkable) event patterns the type's modules must adhere to, and a set of reactive processes sharing an abstract state. The latter may, in the absence of programmer-defined modules, serve as a generator of modules of the type which then behave as required by the specification.

When further refinement of a prototype introduces specific module implementations, the specifications are in turn interpreted as requirements of the modules, and another part of the RAPIDE toolset comes into play – to a great extent the specifications may be checked at run-time or post-mortem for consistency between the system execution as it occurred and as it was required by the specifications to occur. By selectively turning on or off the checking of particular specification elements one may combine decent execution speed with efficient, specification-based high-level error detection and error cause determination [LST91].

3 Parameterized interfaces

It is useful to define functions that work uniformly over many types of objects – for instance, a quicksort routine that works for all types whose objects can be compared with each other. Similarly, one often needs to define interfaces uniformly for various types – consider writing stack packages for integers, for reals, for strings, and so on. This may be done in Ada using *generics*, or in C++ with *templates*. RAPIDE has a similar capability that is more flexible in two ways. The first is that type parameters need not be known at compile time. The second is that type parameters may be constrained to be subtypes of given types. Subtyping constraints involving type constructors provide a very flexible and useful form of polymorphism. This technique is illustrated in Section 4.

A generic set interface uses a type parameter as the type of elements of the set.

Example: *A Set package*

```
type SetType(type Elem) is interface
    Insert : function(E: Elem);
    Delete : function(E: Elem);
    Union : function(S: SetType(Elem));
    Is_Member : function(E: Elem) return Boolean;
end interface;
```

The above declaration defines a type constructor. When applied to a particular type T, this gives an interface type SetType(T). Each object of type SetType(T) will have the four function constituents listed above. The type of each function constituent may be computed by substituting T for Elem in the declaration above. For example, if S has type SetType(T), then the type of S.Insert is **function**(E : T), meaning that S.Insert is a function that requires an actual parameter of type T and returns no result. The intended use of this type constructor is to give the general form of set type, where each module implements a single set in the usual object-oriented way.

□

4 Subtyping and interface parameterization

The fundamental relation between RAPIDE types is the subtype relation. The constraint that A is a subtype of B is written as "A <: B". The main principle associated with subtyping is *substitutivity*: if A is a subtype of B, then any expression of type A may be used without type error in any context that requires an expression of type B.

The following examples illustrate the flexible style of polymorphism supported by subtype constraints. The examples show a general pattern that may be applied in several situations. The main idea is to identify the required operations on objects of the type parameter and formulate its constraint accordingly – the interface containing exactly these operations is used as a subtype bound on the type parameter. In the simple case where the type of the required operations does not depend on the type parameter, one can use a simple interface type as in the first example. If the type of a required operation does depend on the type parameter, then a type constructor is used in the subtype bound instead, as illustrated in the second example.

Example: *Printable Objects*

```
type Printable_Object is interface
    Print : function() return String;
end interface;

Display : function(type T <: Printable_Object, X : T) return ... ;
```

This example shows how one might write a general function Display that displays objects on a screen. Given an object, such a function must somehow determine an appropriate representation of it that is printable. For this, it must invoke some operation on the objects passed to it. For simplicity, assume that each object to be displayed has a Print constituent that returns a string representation of the object. Then one would like to explicitly state this restriction that each actual parameter must have a Print constituent. This constraint on actual parameters is expressed by constraining the type of objects that can be passed to Display. One first defines a type Printable_Object as above. The parameters passed to Display are now restricted to be of a type that is a subtype of Printable_Object. The definition of subtyping guarantees that if T is a subtype of Printable_Object (written "T <: Printable_Object"), then every X : T will have a function constituent Print returning a string.

□

Example: *Generic Sorting*

```
Naive_Sort : function(type T;
                      < : function (X,Y : T) return Boolean;
                      L : List(T))
                      return List(T);

type Ordered(type T) is
interface
    < : function(X : T) return Boolean;
end interface;

Sort : function(type T <: Ordered(T); L : List(T)) return List(T)
```

This example illustrates the use of type constructors in subtype constraints. It is required to sort a list of elements of some type. The elements of any such type have some ordering relation on them.

A simple solution would be to define a function that takes both the list of elements as well as the ordering relation as arguments, and returns the sorted list. This is the Naive_Sort function type defined above.

However, in a language where objects may carry a comparison operation as one of their function constituents, it is possible to write the sort function such that the comparison operation does not have to be passed as an argument. To do this, one must make sure the actual type parameter is a type whose objects have a $<$ function constituent. But the type of $<$ is not the same for all types of ordered objects. For this reason, one needs to use the type constructor Ordered whose type parameter appears in the type of $<$. For any type T, the interface Ordered(T) will have a comparison function $<$ of the type **function** (X : T) **return** Boolean. Consequently, if Y, Z : T, then Y.<(Z) determines whether Y is less than Z. One then defines the Sort function using Ordered(T) as the bound for the type argument T.

□

5 Subtyping and interface type constituents

RAPIDE interfaces allow for multiple implementations to be provided, and these can coexist and interact with each other. However, the possibility of having more than one implementation can be a problem as well - occasionally one needs to have some information about the representation chosen for implementing an interface, and very often, many optimizations can be made based on partial or complete information about the representation. Section 7 describes how private interfaces can be used to allow some visibility to the internal details of modules. This section describes how one can enforce unique implementations and thereby be more efficient about the implementation.

When a type in an interface is declared as a type name only, the corresponding type constituent of any module will generally not have any subtypes. (The only exceptions are when another type in the same interface is designated to be a subtype or equal to that type.) This has an important pragmatic consequence, both for the predefined types of an executable sublanguage and user-defined types. We illustrate some of the consequences by discussing predefined types. The predefined types of an executable sublanguage must be definable using the RAPIDE types language. This means that common types such as integers and strings will be defined by interfaces. A useful aspect of RAPIDE is that this may be done in such a way that the predefined integers and strings have no subtypes. This allows the implementation of the executable sublanguage to assume that any object of type integer, for example, will be represented in the standard way chosen by the compiler. This leads to much more efficient implementations of integer operations than would be possible if RAPIDE did not have any mechanism for defining non-subtypable types. For example, integer operations may be implemented using the standard machine operations instead of requiring calls to arbitrary module constituent functions. Of course, if it is considered more useful for integers and strings to have user-defined subtypes, then these types may be given as explicit interfaces. The result in this case would be a Smalltalk-like situation in which a user could define a module with the appropriate function constituents, and use this new "integer" in place of any of the predefined integers.

Types that are given only with subtype designations in the interface are similarly "non-subtypable" – if the only information exported about a type is that it is a subtype of some other type, one cannot define a new type that would be a subtype. However, since the subtype designation gives some information about the operations on the type, these types are only "partially" hidden. These types are analogous to the "sealed classes" of Dylan [App92].

Example: *Partially hidden clock type*

```
type Clock_Spec is interface
      type Clock <:  interface
                          tick : function();
                          read : function() return integer;
                     end interface;
      new_clock : function() return Clock;
end interface;
```

This example shows how to specify some constituents, without giving an explicit type definition. The interface declares a type Clock together with a function that returns a "new" clock. If Clock_Impl is a module with interface Clock_Spec, then Clock_Impl.Clock is a type that is known to be a subtype of the interface given explicitly above. Therefore, if C is an object of type Clock_Impl.Clock, we can call function constituents C.tick() and C.read(), the first returning no value and the second returning an integer. Since the type Clock is not given explicitly in the interface, one cannot define a type that is a subtype of Clock_Impl.Clock. Therefore, the only elements of type Clock_Impl.Clock will be those produced by calls to Clock_Impl.new_clock.

□

Example: *Time-stamped events*

```
type Event_Spec is interface
      type Time_Stamped_Event <:
               interface
                    event_name : string;
                    precedes :  function(E:Time_Stamped_Event) return boolean;
               end interface;
      new_event : function(Name:string) return Time_Stamped_Event;
end interface;

module Event_Impl() return Event_Spec is
      type Time_Stamped_Event =
               interface
                    event_id : integer;
                    event_name : string;
                    precedes :  function(E:Time_Stamped_Event) return boolean;
               end interface;
      new_event : function(Name:string) return Time_Stamped_Event is
                          module
                               event_id : integer = System.Clock().read;
                               event_name : string = Name;
```

```
                              precedes : function(E:Time_Stamped_Event) return boolean =
                                         return(event_id < E.event_id);
                          end module;
end module;
```

This example illustrates how non-subtypable types can lead to more efficient implementations. Event_Impl is an implementation of Event_Spec. As explained earlier, one cannot define subtypes of Event_Impl.Time_Stamped_Event. This allows Event_Impl to assume that the argument E of the precedes function has not only access to the constituents given in the subtype requirement of the Event_Spec declaration, but also to those given in the full type declaration in the specific implementation of an Event_Spec instance in Event_Impl. In particular, the precedes function there can access the event_id constituent, and use that in the determination of whether one event precedes another.

□

6 Subtyping and interface derivation mechanisms

The RAPIDE type system has taken an extensional approach to the question of when two interface types are related – if all the objects of one interface type conform to the interface given for another type, then the former should in principle be a subtype of the latter. This results in a *structural* subtyping mechanism, where the declared name (if any) of a type has no influence on whether it is a subtype of another or not. This is in contrast to subtyping (or subclass) mechanisms relying on explicitly stating the type relations between named types (such as in Ada [US 80], Simula67 [DMN70] and C++ [Str86]). Structural subtyping gives the software designer and prototyper a flexible tool when progressing through the development of new software, while retaining the security of safe typing, and avoiding the growth of unmanageable or confusing type libraries (where possibly a large proportion of the types in the libraries are retained for no other reason than to maintain the proper type relations between other types).

The RAPIDE types language requires that the validity of all assertions about subtype relationships be determinable at compile time. Consequently the structural subtyping scheme extends only to the fully visible types – the declaration of a type as partially or fully hidden effectively limits the information available for reasoning about its structure, and as a result intentionally breaks subtyping relationships that might otherwise hold.

If a software designer would like to make sure that no accidental subtyping occurs (as it may, with structural subtyping), she may declare types as wholly or partially hidden, making use of this feature for limiting the type checker's insight into the structure of any particular type.

Example: *A Subtype of Printable_Objects*

```
type Graphics_Window is interface
    Init: function(upper_left, lower_right: coordinates);
    ...
    Print: function() return String;
end interface;
```

There is no need to explicitly state that objects of the type Graphics_Window are also of the type Printable_Object. The fact that all the objects required of a Printable_Object instance are also required of a Graphics_Window instance ensures that. The function Display, which requires a Printable_Object as parameter, may be used also on Graphics_Window objects.

In contrast, the type Clock of a particular Clock_Spec instance (see 5) cannot be established as the supertype of any other type (though we *do* know that it is a subtype of an interface type with a tick and a read functions).

□

Though the subtyping scheme of RAPIDE does not rely on explicit type derivation, it is useful to be able to base one type definition on another – reusing the type declarations. This is a mechanism quite distinct from that of subtyping, and the fact that one type is *derived* from another in does not necessarily indicate that there is a subtype relationship between them.

RAPIDE derivation mechanisms provide a flexible form of reuse. The purpose of a derivation (or inheritance) mechanism is to allow a declaration made in one place for one purpose to be used in another place for a similar, related, or perhaps even different purpose. One important aspect of derivation is that when one entity is derived from another, changes to the first declaration affect the other. This may seem to make programs complicated, but when used carefully, derivation can be a powerful means to maintain consistency in code that is evolving over time. It also simplifies program modification and maintenance by reducing the amount of redundant or repeated code in a program.

In contrast with Smalltalk [GR83], C++ and other object-oriented languages, RAPIDE has distinct but parallel inheritance mechanisms for interfaces and implementations. This has proven very useful, for the general reason that interface and implementation inheritance are often used independently and/or in opposite ways. For instance, one would want to define the interface for double-ended queues by extending that for simple queues, whereas the latter can be easily implemented from the former [Sny86]. Similarly, the Smalltalk library shows several cases of conflict between the implementation hierarchy and a reasonable interface hierarchy [Coo92].

In RAPIDE, interface derivation declarations are essentially used to copy declarations from one interface into another. The simplest form of derivation declaration is

include type_expression

Essentially, A declaration "**include** A" inside an interface type B has the effect of inserting all the declarations of A into B. Since this requires knowing the interface A completely, the type expression that is **include**d must denote an interface type that can be determined completely at compile time.

A simple example is the derivation of the interface for colored points from the Point interface given in Section 2.

Example: *Deriving Colored Point from Point*

```
type Colored_Point is
interface
    include Point;
    Col : function() return Color;
end Point;
```

Colored points are derived from points using an **include** declaration. The result is that the Colored_Point interface has four function constituents, X_Val, Y_Val, Distance and Color.

□

It is often useful to be able to include only a part of a previously declared interface. RAPIDE provides the following forms of derivations for this purpose

include type_expression **only** identifier_list
include type_expression **except** identifier_list

which include only the listed constituents, or include all constituents except those listed (respectively).

Unlike other languages that allow a single entity to be derived from more than one other (*i.e.*, multiple inheritance), RAPIDE does not have any standard or default rules for resolving conflicts. One reason is that such rules are generally complicated, and unpredictable in certain cases. Moreover, what seems a reasonable resolution strategy varies greatly from one person to the next, and it seems unreasonable to impose a particular one – there is a limit to how "cleverly" a software design tool should attempt to interpret the unstated wishes of a user. Finally, it is anticipated that the RAPIDE type system will be used in conjunction with multiple executable languages, and these may each have their own conflict resolution rules for modules.

However, RAPIDE makes it possible to avoid conflicts by renaming included constituents. For instance, one can write

include A **only** x,y **replace** x **with** new_x, y **with** new_y

Such a renaming has consequences for the subtyping relationship between the derived type and the type it is derived from – the set of declarations of the two interfaces become distinct. There are various ways in which the renaming mechanism could be extended so as not to necessarily break the subtyping relationship between the two types. However, this would introduce another set of mechanisms into an already rich type system, and is not felt to be worth it.

It is often useful to include constituents of an interface that is defined using a type constructor. By applying a type constructor to the interface that is being defined, one may specialize the types of included constituents. This, in fact, gives RAPIDE the power to program so-called *mixins* (in the terminology of CLOS [Kee89]) without requiring a separate concept. This is illustrate in the following example, which adds equality to points.

Example: *Points with Equality*

```
type Equality(type T) is
interface
    function Eq(X : T) return Boolean;
end interface;

type Eq_Point is
interface
    include Point;
    include Equality(Eq_Point);
end interface;

type Eq_Point_Elab is
interface
    function X_Val return Integer;
    function Y_Val return Integer;
    function Eq(X : Eq_Point_Elab) return Boolean;
end interface;
```

The type constructor Equality(type T) returns an interface with one function constituent. The type of this constituent contains the formal type parameter T, and therefore will depend on the actual parameter of the type constructor. Interface Eq_Point includes the constituents of Point and Equality(Eq_Point). Since Equality is applied to the actual type parameter Eq_Point, the resulting Eq function constituent has type **function** (X : Eq_Point) **return** Boolean, obtained by replacing the formal parameter T with the actual parameter Eq_Point. Therefore, the Eq function constituent of an Eq_Point object accepts another Eq_Point as an argument. The interface Eq_Point is equivalent to Eq_Point_Elab, which is written directly without any derivation declarations.

□

7 Private and external interfaces

In Section 2, it was said that interface types consist of a list of declarations. This was a slightly simplified view of interfaces. The constituents mentioned there are part of what is called the *public interface*. Constituents of the public interface are visible to the entire system. Interfaces can contain two other subparts - a *private interface* and an *external interface*.

Private interfaces

The private interface contains a list of declarations similar to that in a public interface. However, these constituents are visible only within modules of this interface type. The need for this arises from the difficulty in implementing binary operations in the object-oriented style of programming. Usually implementing a binary operation efficiently requires some information about the representation of the two arguments. However, in the object-oriented style, binary operations only have access to the internals of their first implicit argument; they can only use the publicly available functions to look at the second. Private interfaces provide a means of "leaking" as many details of the internal representation as may be required, but only to objects of the same type. This is made clear in the following example:

Consider an interface for sets of integers.

```
type IntegerSetType is interface
    Insert : function(N : Integer);
    Delete : function(N : Integer);
    Union : function(S : IntegerSetType);
    Is_Member : function(N : Integer) return Boolean;
end interface;
```

Now consider how one might implement the union function. The natural and efficient way of doing it requires the ability to iterate over the elements of the set passed as an argument. But the publicly available functions on sets don't provide an apparent way of doing this. The most realistic solution would be to add a new function in the interface which returns a list of all the members of the set. However, this means making more operations public than required. Also, adding a new constituent during program development may require changes in certain derived interfaces. Using the private interface, one can write:

```
type IntegerSetType is interface
public
    Insert : function(N : Integer);
    Delete : function(N : Integer);
    Union : function(S : IntegerSetType);
    Is_Member : function(N : Integer) return Boolean;
private
    List_of_Members : function() return Integer_List;
end interface;
```

Now, only other integer sets can invoke the List_of_Members functions.

Private interfaces are similar in spirit to the C++ friend concept.

External interfaces

An interface may also consist of a set of external declarations. This is a list of object names with specified types that modules belonging to this interface can assume to be visible. The executable language is expected to provide means of associating actual objects with these declarations.

The use of external declarations is strongly motivated by the use of the RAPIDE type system to specify architectures and their connections.

Example: *Relying on an externally given Clock*

```
type Timer is interface
public
    Start : function();
    Stop : function();
    Reset : function();
    Time_Elapsed : function() return Time;
extern
    Clk : Clock;
end interface;
```

The Timer interface describes timing devices. A timer can provide various timing utilities as long as it has access to some clock. This may be specified by using the external declarations to require that every instance of the type Timer may freely refer to Clk, expecting it to denote some instance of the Clock type. In the RAPIDE architecture language [RAr] the denotation of Clk for a particular Timer instance may change as the execution progresses, in contrast to the effect of making Clk a formal parameter of a function returning new Timers (where the denotation would become fixed at function invocation time).

The *external* mechanism together with the *connection* mechanism of the architecture language allows one to model dynamic architecture topologies in a perspicuous manner, making clear how a set of modules may connect, and how these connections may change over time.

□

8 Current Status

Over the past four years, the RAPIDE design has been tested, modified and evaluated in a variety of ways. An early "0.2" implementation, based on a translation to Ada, was completed in 1991. Since then, several programming tools have been implemented, including a graphical interface to allow component interfaces and connections to be defined easily, a graphical browser to facilitate navigating through module hierarchies and to check subtype relationships and a separate tool to inspect the partially-ordered set of events in several useful ways. This allowed experimentation with the concurrent aspects of the language and some evaluation of the program-structuring mechanisms. Several large systems, including the ADAGE avionics system, the X-open distributed transaction processing architecture, a "trusted" X-windows definition, as well as many small ones, have been expressed in the RAPIDE type system. The diagnostic information provided by the tools significantly contributed to the refinement of the framework design, including the type system. The second generation language and tool implementations, now well underway, are being done in C++. This change of language has given us an opportunity to reflect on interoperability considerations and evaluate aspects of our approach to multi-language prototyping.

The interface types of the RAPIDE type system have been used to write the interfaces for all the predefined types supported by the executable language of the framework. These include several types with interesting semantics such as action and reference types (for details, see [KLM94]). The exercise of specifying these predefined types as interfaces has illustrated the expressiveness and flexibility of the type system. The detailed interfaces for the predefined types and type constructors are outlined in [RPr].

Acknowledgements: *We are thankful to the other members of the* RAPIDE *project for continuing insightful comments and criticisms.*

References

[App92] Apple Computer Inc. *Dylan, an object-oriented dynamic language*, Nov 1992.

[BL90] F. Belz and D.C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proc. ACM Tri-Ada'90 Conference*, December 1990.

[Coo92] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

[DMN70] O-J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA Common Base Language*. Norwegian Computing Center S-22, Oslo, Norway, 1970.

[ES90] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual*. Addison-Wesley, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation*. Addison Wesley, 1983.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[Kee89] S.E. Keene. *Object-oriented programming in Common Lisp*. Addison-Wesley, 1989.

[KLM+92] D. Katiyar, D.C. Luckham, N. Madhav, S. Meldal, J.C. Mitchell, and S. Sankar. Subtyping, assignment, and cloning in a concurrent object-oriented language. In *Proceedings of the DARPA Software Technology Conference, Los Angeles, California.*, pages 458–470, April 1992.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[LST91] D. C. Luckham, S. Sankar, and S. Takahashi. Two dimensional pinpointing: An application of formal specification to debugging packages. *IEEE Software*, 8(1):74–84, January 1991. (Also Stanford University Technical Report No. CSL–TR–89–379.).

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.

[MMM91] J.C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 270–278, January 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[RAr] The PAVG group, Stanford University. *The Rapide Architecture Language Reference Manual.*

[RPr] The PAVG group, Stanford University. *The Rapide Predefined Types Reference Manual.*

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. 1-st ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Str86] B. Stroustrop. *The C^{++} Programming Language*. Addison-Wesley, 1986.

[Tof89] M. Tofte. Four lectures on Standard ML. Technical Report ECS-LFCS-89-73, Lab. for Foundations of C.S., University of Edinburgh, 1989.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.