A type system for prototyping languages*

Dinesh Katiyar David Luckham John Mitchell †

{katiyar,luckham,mitchell}@cs.stanford.edu

Stanford University
California, USA

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. We describe the main features of the type language, its intended use in a prototyping environment, and rationale for selected design decisions.

1 Introduction

RAPIDE is a programming language framework with an associated toolset. The framework consists of a set of languages, while the toolset provides related program development and diagnostic tools. The framework includes a type language, a default executable language, a default specification language, and an architecture language. Although these languages together provide a complete programming language, RAPIDE is intended to accommodate additional implementation and specification languages, besides the default ones provided by the framework. 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. In a sense, the type system provides a minimal *lingua franca* for assembling or experimenting with multi-language systems. The architecture language collects together those features of the type and executable languages that are typically used in defining the interfaces of system components and the dataflow connections between them. This document gives an overview of the RAPIDE type system and describes some of the motivation and rationale for the design. Full language reference manuals for the type, executable, specification and architecture languages are available separately.

The central construct of the RAPIDE type system is the *interface type.* An interface type lists the visible types and objects of a set of *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.*

In addition to interface types, the main concepts of the RAPIDE type system are several forms of parameterization, interface derivation mechanisms, and the use of subtyping. Parameterization may be used to define a family of related types, depending on either type parameters or value (such as integer) parameters. A type parameter in an interface or function declaration may be constrained by requiring it to be a subtype of some given type. The derivation mechanisms of the type language allow interfaces to be combined and reused; they are essentially a form of inheritance mechanism appropriate for defining related interfaces.

Besides its unified view of type, the RAPIDE type system has several novel and interesting features. *(i)* It separates types of components from their module implementations. This is a direct consequence of the framework having separate types and executable languages. *(ii)* The subtyping hierarchy is independent of the inheritance mechanisms and is purely structural – it can be inferred by examining the types. *(iii)* Subtyping constraints can be imposed on type parameters of functions and interfaces, giving rise to a very useful and powerful form of polymorphism. Similar constraints can be imposed on type constituents in interfaces, allowing "partially abstract" types. *(iv)* Type constituents in interfaces can also be specified completely, giving RAPIDE the expressiveness of languages such as Standard ML [MTH90] while still allowing modules to be first-class. *(v)* The derivation mechanisms of the language can be applied to parameterized interfaces as well.

Section 2 summarizes general concepts and design decisions in the RAPIDE framework and Section 3 gives an overview

*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.

of the type system. Section 4 describes how the predefined types of the RAPIDE framework are defined as interface types, and summarizes the semantics associated with action and reference types. Section 5 describes extensions to interfaces that enhance their expressiveness. Section 6 illustrates the type system through larger examples. Appendix A briefly summarizes the underlying calculus that is chosen as the medium for theoretical studies of the RAPIDE type system.

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 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 [TC93], the X-open distributed transaction processing architecture [Ken93], 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++.

2 Prototyping in RAPIDE

2.1 Rapid prototyping

The primary goal of RAPIDE is to support rapid construction of prototypes of large concurrent and distributed real-time systems.

Prototyping is the incremental development of systems by an iterative design and development process. In a typical project, partial system components, called *prototype components,* are built. These are analyzed separately and combined into a *system prototype,* which is analyzed to determine the adequacy of the design, to refine the specifications of the components, and to estimate system performance. The analysis of a prototype provides useful feedback for subsequent revision and refinement of the system design or of individual components. Prototyping is often an experimental activity. Alternate system designs can be evaluated by building prototypes for each and analyzing and comparing them. This allows crucial design decisions to be made as early in the design process as possible. An advantage for software development under contract is that the purchaser may obtain early information about the functionality of the system before committing to full-scale development of production-quality code.

A prototyping framework must (*i*) provide the capability to construct large components easily, often by reusing or adapting existing components, (*ii*) provide tools to perform quick and meaningful analyses of the execution of prototypes, and (*iii*) support easy revision and refinement of prototypes.

2.2 The languages of the RAPIDE framework

The RAPIDE language framework consists of a *type language* to define the interfaces of components, a *specification language* to specify and constrain the behavior of components, and an *executable language* to construct modules that implement system components. The *pattern language* is used by the executable and specification languages to characterize the execution of RAPIDE programs. The *architecture language* provides features for easily "wiring up" prototype components – *i.e.*, for defining explicitly the (possibly dynamic) communication links between modules.

Some languages of the RAPIDE framework are better understood in the context of the computational model associated with the default executable language. Computations are viewed as partially ordered sets of events (*posets*). The synchronization, concurrency, and timing aspects of a prototype are explicitly represented in such computations. Requirements on computations can be specified formally as constraints on these posets, and violations of these can be detected automatically. This model is described in greater detail in [LVB+93].

The standard languages of the framework are summarized below.

The type language: This language provides constructs for defining the interfaces of components and for deriving new interfaces from prior ones. It also provides *typing rules* which are checked automatically to safeguard against common mistakes that arise while using such a language. The notion of subtyping adds flexibility to these conformance rules.

The executable language: The executable language is a concurrent reactive programming language in which one defines the executable behavior of components. Its principle constructs are independent (or concurrent) reactive processes that activate when patterns of events occur during execution. The executable language also provides standard kinds of Algol control structures, subprograms, exception raising and handling constructs, and timing features. The result of executing a RAPIDE program is a set of events with various partial orderings (such as time and causality) relating the events. This is referred to as the *poset execution model.*

The specification language: This language provides constructs for abstract specification of the behavior of a distributed system, including timing requirements. It is based on the poset execution model. The constraint-based specification language supplies a rich set of constructs for defining patterns of events that should or should not occur in any computation of a distributed system. It is also designed so that straightforward algorithms can be used to check any actual execution of a prototype for violations of pattern constraints.

The pattern language: This sublanguage of the default specification and executable languages provides a convenient notation for characterizing posets. This capability is used in the specification language to impose constraints on event-based behaviors, and in the executable language to generate, detect and react to certain patterns of events. It is used by the architecture language to define communication between components.

The architecture language: This language provides basic mechanisms to wire together system components by defining the connections between them. Architectures are sets of component interfaces and connections. Connections allow components to synchronize and to communicate data. The need for this language is motivated by the prototyping of large distributed real-time systems, where the early stages of prototyping concentrate on system architecture. The separate architecture language allows the prototyping process to be started without the full knowledge of the rest of the framework.

The Rapide toolset provides several design and diagnostic tools to aid program development and refinement. The architecture language has a graphical interface to allow component interfaces and connections to be defined easily. Graphical browsers facilitate navigating through module hierarchies. A separate tool allows inspection of the output of simulations (*i.e.* partially-ordered set of events) in several useful ways. Constraint checkers detect violations of constraints during execution of Rapide programs.

2.3 Interaction between the languages

The languages of the framework must satisfy certain compatibility requirements and conventions. These constraints are of different kinds:

- Certain restrictions apply globally. The languages of the framework must share certain visibility, scoping and naming rules. (These are just the rules of block-structured, statically-scoped languages.)
- The type system imposes syntactic and semantic restrictions on any executable or specification language used within the framework. The syntactic constraints essentially guarantee that the grammars of the framework languages can be combined to form an unambiguous context-free grammar. The semantic conditions ensure that the operations provided by the executable language behave in a manner that is, loosely speaking, consistent with respect to the type system.
- The specification and executable languages may impose additional constraints on each other, independent of the types language. For instance, in the default set of languages, the executable language uses a computation model based on partially ordered sets of events. In this context, a specification language is most useful if it can meaningfully refer to partially-ordered computations. This is achieved in the default executable and specification languages by sharing a pattern language for defining and recognizing partially-ordered patterns of events.

A formal study of these interactions is a subject of current research.

3 Main concepts of the Rapide type system

Rapide has a strong static typing discipline. The key notions associated with the type system are interface types (Section 3.1), two reuse mechanisms – parameterization (Section 3.2) and derivations (Section 3.3), and the subtyping discipline (Section 3.4). Other details of the type system are discussed in Section 4.

3.1 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 name, 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 declares 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, and cannot be used as run-time values).

The objects of an interface type are modules containing 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 object constituents must have the type indicated in the interface. A type constituent may be *(i)* any type when the interface contains only a type name declaration, *(ii)* any subtype of the type given in the interface, if a subtype designation is used, or *(iii)* must 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 3.4.

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.
□

As mentioned briefly above, a type declaration in an interface may be *(i)* a declaration of a type name only, *(ii)* a type declared to be a subtype of some designated type, or *(iii)* a type specified exactly by declaring that a type name is equal to some type. This three-level flexibility allows relatively fine control (in the interface) over the information that will be exported from modules.

Type name declarations are used for complete information hiding, as with traditional abstract data types.

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 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' is 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 T_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. (Note that one couldn't define Ext_T_Package using the interface derivation mechanisms of Section 3.3, since that would result in a new independent abstract type.)
□

Subtype designations provide an intermediate level of information hiding that allows certain operations on the type to be revealed. When a type constituent of an interface is not declared fully in the interface, that type may only have subtypes if another type in the same interface is declared a subtype or equal to it. Such non-subtypeable types are discussed in more detail in Section 3.4.

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 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. The examples in Section 6 illustrate some uses of specifications. Executions can be checked for violations of specifications by an automated checker at runtime.

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 5.

Interface types have been used to write the interfaces for all the predefined types supported by the executable language of the framework (details in [RPr]). These include several types with interesting semantics such as action and reference types (Section 4.1). The exercise of specifying these predefined types as interfaces has illustrated the expressiveness and flexibility of the type system.

3.2 Type constructors

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 3.4.

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.
□

3.3 Reuse and Interface derivation mechanisms

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. In RAPIDE, interface derivation declarations define textual dependencies between interfaces. Any change to an interface imposes similar changes to all interfaces derived from it. This is an important feature for rapid prototyping since it simplifies program modification and maintenance by reducing the amount of redundant or repeated code, but it must be used carefully to avoid unexpected changes.

In contrast with Smalltalk [GR83], C++ and other object-oriented languages, RAPIDE has distinct 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].

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 3.1.

Example: *Deriving Colored Point from Point*

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

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 Col.
□

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, 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
            rename x to new_x,
                   y to new_y
```

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.
□

Another use for type constructors in derivation declarations arises when specializing recursively-defined interfaces. In languages without derivations, interface definitions can simply use the name of the interface being defined to make recursive interface definitions. However, when one permits these definitions to be reused in defining new interfaces, one would like the recursive occurrences of the interface name to now refer to the new interface being defined. Hence, such languages typically provide a separate keyword for these references – for example, Eiffel [Mey92] uses the term "*like current*". In RAPIDE, however, we can use type constructors to overcome the need for a new feature. More specifically, instead of writing

```
type T is interface ... like current...;
```

one would write

```
type TGen(type t) is interface ...t...;
type T is TGen(T);
```

Derivations are then done using the constructor TGen instead of T. Since RAPIDE has no special type symbol *like current*, there is no need for special typing rules of the form found in Eiffel, for example, or the concomitant typing difficulties (see [Coo89]). The constructor TGen is also useful for imposing subtype constraints, as described in Section 3.4.

There is an important restriction on the use of derivation declarations. In RAPIDE, types may be formal parameters to functions and other constructs. However, as mentioned earlier, derivation must be resolved at compile time. Therefore, formal parameters may not be used in derivation declarations. For instance,

```
type TGen(type t) is interface
    include Foo(t);
    ...
end interface;
```

is considered illegal.

If a type name T is used in a derivation declaration, then the interface named by T must be determined at compile time.

3.4 Subtyping and subtype constraints

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 primary motivation for subtyping in a prototyping language is to allow functionality to be added to a prototype with minimal modification to the system. This is widely appreciated in the object-oriented programming community. In short, if objects of some type B have some behavior which crudely approximates part of the system being prototyped, then it may be desirable to replace objects of type B with objects of another type A that have more realistic or accurate behavior. In many cases, the type A will be a subtype of B. By designing the language so that substitutivity is allowed, one may add functionality in this way without any modification to the original program.

An example illustrating the use of subtyping is in prototyping an airport scheduling system. In an early prototype, one would define an interface airplane with constituents such as position and orientation and accelerate that would allow a control tower module to affect the approach of an airplane. In a later prototype, it is likely that different types of airplanes would be used. If one adds interfaces for Boeing 757's and Beechcrafts, these would be subtypes of airplane, containing extra constituents reflecting features specific to these aircraft. By virtue of this subtyping relation, all Beechcrafts are now instances of airplanes and all of the general control algorithms that apply to all airplanes could be used for Beechcrafts without modification or recompilation.

Another advantage of subtyping is uniform operation over various types of data, regardless of whether these types are added incrementally. In particular, subtyping makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some type. For example, one could have a queue containing airplane modules, each belonging to different subtypes of airplane. This is generally not possible in strongly typed languages without subtyping.

RAPIDE provides rules for determining, by compile-time analysis of the structure of the types, if one type is a subtype of another. RAPIDE chooses this implicit structural approach over two other prevalent trends – *(i)* determining the subtype relationships directly from the derivation hierarchy, and *(ii)* requiring all subtype relationships to be explicitly stated. This is done for the following reasons:

- The subtyping-from-inheritance approach either compromises the substitutivity principle or places strong restrictions on the derivation mechanisms [CHC90]. It also causes the subtyping relationship to be greatly affected by the order in which the program components are written.

- The explicit subtyping approach is rather cumbersome, since many "obvious" relationships would have to be made explicit. Also, it is hard to fit a new type into an existing hierarchy. This approach does avoid the problem of "accidental" subtyping (caused when one type is determined to be a subtype of a completely unrelated type simply because they happen to have similar structures). However, accidents do not seem to be troublesome enough to warrant the added responsibility on the programmer. Some circumstantial evidence for this is provided in the work of [Rit91, RT91], where it is shown that types can be used to search for functions in libraries, generating very few accidental "hits".
- The RAPIDE approach makes it easier to reorganize code and to combine independently constructed subprograms. There is no need to change or introduce any subtyping relationships - they are all automatically determined from the program text.

The subtyping problem is fairly crucial to the process of type-checking and is the topic of recent research [Pie92, KS92, Kat92, GP94].

3.4.1 Subtyping and type parameters

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;
                      compare : function (X,Y : T)
                                return Boolean;
                      L : List(T))
                       return List(T);

type Ordered(type T) is interface
    compare : 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 compare function constituent. But the type of compare 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 compare. For any type T, the interface Ordered(T) will have a comparison function compare of the type **function** (X : T) **return** Boolean. Consequently, if Y, Z : T, then Y.compare(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.
□

3.4.2 Subtyping and type constituents of modules

When a type in an interface is declared as a type name only, the corresponding type constituent of any module is highly abstract, in the sense that outside the module, this type is not equal to nor related by subtyping to any other type in the program. Such non-subtypeable types have an important pragmatic consequence. An implementation of such a type can assume that all objects of that type will be represented in exactly the same way, and therefore can implement the operations of the object more efficiently.

Consider, for example, an interface definition of the predefined type for integers. If integers are defined as an explicit interface, the integer operations have to be implemented as calls to module constituent functions. While this allows a user to define his or her own "integers" and have them work interchangeably with the predefined integers, it does result in an inefficient implementation of integers. If the integer type were declared in the manner specified above, then its implementation could assume that all integers will be represented in the same way, and could use standard machine operations to implement the integer operations efficiently.

However, there is a slight inconvenience with such abstract types by virtue of their not having any supertypes either. Since nothing is known about their representation, objects of such types cannot be used in any context but of the operations on the abstract type. The use of subtype constraints on type constituents provides a useful compromise. In this case, the types are only "partially" hidden, since some information about the operations on the type is specified in the subtype designation. This makes the type non-subtypeable, while still allowing supertypes, and hence permitting its use in contexts that are satisfied with the partial information about the type. This is illustrated in the following example:

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 new_clock 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();
            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 the same unique internal representation for events. Therefore, the precedes function can access the internal representation of its argument – in this case, this is done through the event_rep constituent.
□

Non-subtypeable types are analogous to the "sealed classes" of Dylan [App92].

4 Interfaces for specific types

4.1 Predefined types

The *predefined types* and *predefined type constructors* of RAPIDE are described in [RPr]. These types include integers, booleans, strings, arrays, records, and other common types of Pascal-like languages. They are presented in a separate document, rather than as a basic part of the type language, because these are all particular examples of interface types. Moreover, different executable languages will require different sets of predefined types, and specific application domains may merit specialized types. Therefore, the predefined type definitions are presented as a type preamble, expressible in the core type language, rather than as an essential, fixed part of the language itself. The preamble also defines the special syntax for predefined types. It also makes use of specifications to define the normal semantics of objects of the predefined types.

Sections 4.2 and 4.3 describe two of the more interesting predefined types that are provided with the default executable language.

4.2 Action types

Actions provide a mechanism for asynchronous communication between components. Communication takes place through entities called *events*. An event can be thought of as a tuple of objects. An action name declares a type of events and two associated operations – *generate* and *read*. An action call generates an event – the contents of the tuple are the parameters to the call, plus other information about the caller, the current time, etc. Events are detected by *patterns*. Patterns can be used to match sets of events. The pattern matching process relies on calls to the *read* operation on events.

As mentioned above, an action declaration corresponds to two operations on the corresponding event type. However, there is a slight subtlety in this, depending on where the action declaration is located.

- If an action name is declared in the public interface, this is taken to indicate that events of that type may be generated either by the component itself or its containing module, but these may be read only by the component itself. This means that a public action is essentially declaring a public *generate* operation and a hidden *read* operation.
- An action declaration in the private interface behaves similarly, except that it is visible to modules of the same interface type.
- An action appearing as an external constituent declares a public *read* and a hidden *generate* operation. In other words, events of the corresponding type can be generated only by the component itself but can be read by the containing module also.

4.3 Variables and reference types

It becomes hard to provide type safety when objects may change state while executing concurrently. This is apparent in Ada, where task types are limited private types that do not have an assignment operation or an equality test – this leads to two kinds of types, "normal" types and "limited" types.

The RAPIDE types language permits the executable language of the framework to distinguish between variables and constants of the same type. A simple, uniform view of variables that works for all types of objects (including modules with changing states and multiple threads) is that a variable denotes a *reference* to an object. So when a variable is changed by assignment, the variable refers to a different object. Giving variables a different type from constants also meshes well with the subtyping since the operations possible on a variable are different from those on a constant.

There is a predefined type constructor called *reference*. For each type A, **ref**(A) is the type of references to objects of A. RAPIDE references are identical to ML references [MT91, MTH90, Pau91], and are almost the same as pointers of Pascal and related languages, except for the difference in testing for reference or pointer equality.

5 Private and external interfaces

In Section 3.1, 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*.

5.1 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.

5.2 External interfaces

An interface may also have an external part which consists 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.

Consider the interface Timer that 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 a clock be made available to every instance of the type Timer.

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

6 Illustrative examples

6.1 An example from the predefined type library

As a more comprehensive example, this section presents the interface definitions that lead to the complete definition of the interfaces for the numeric types in the predefined types library. These definitions, in their full glory, would contain several constraints that are imposed on the type definitions. Since the reader may not be familiar with the specification language, these examples will use simple English statements to describe the constraints wherever necessary.

First, since equality and other ordering relations are part of several predefined types, these are defined as separated interfaces, and included wherever they are needed. Note that the Boolean predefined type is being assumed in these examples.

The interface for the equality relations defines two functions - one for checking equality and the other for inequality.

```
type Equality(type T) is interface
    = : function(X : T) return Boolean;
    /= : function(X : T) return Boolean;
constraints
        -- reflexive, transitive and symmetric
        -- equality and inequality are complementary
end interface;
```

The interface for ordering relations includes the equality operations and then defines the usual four comparison operators that one might expect.

```
type Order(type T) is interface
    include Equality(T);
    <= : function(X : T) return Boolean;
    < : function(X : T) return Boolean;
    > : function(X : T) return Boolean;
    >= : function(X : T) return Boolean;
constraints
        -- reflexive, transitive and anti-symmetric
        -- axioms for the interdependencies
        --       between the operations
end interface;
```

Using these interfaces, one now defines the numeric types – integers, naturals and positive integers.

```
type Numeric_Pkg is interface
    type Integer <: interface
        include Order(Integer);
        Pred : function() return Integer;
        Succ : function() return Integer;
        + : function() return Integer;
        - : function() return Integer;
        + : function(N : Integer) return Integer;
        - : function(N : Integer) return Integer;
        * : function(N : Integer) return Integer;
        ** : function(N : Integer) return Integer;
        / : function(N : Integer) return Integer;
        % : function(N : Integer) return Integer
        Abs : function() return Integer;
        Iszero : function() return Boolean;
        IsPositive : function() return Boolean;
    constraints
        -- an integer is greater than its pred
        --     and less that its succ
        -- pred and succ are inverse operations
        -- axioms for the behavior of zero with
        --     the various operators
        -- axioms for the interaction of succ/pred
        --     with the different operators
    end interface;
    MaxInt : Integer;
    MinInt : Integer;

    type Natural <: Integer
    constraints
        -- all naturals are greater than or
        --     equal to zero

    type Positive <: Integer
    constraints
        -- all positives are greater than zero
end interface;
```

6.2 An architectures example

This next example attempts to give a feel for the kind of expressive interfaces that may be built using the RAPIDE type system. It uses action types, which are described briefly in Section 4.2, and other architecture-language dependent keywords that will be elaborated to the extent required to illustrate their use.

The example describes how the interfaces of an automobile and a driver might be specified, and how the two could be hooked to get an operational vehicle.

First, the interface for the controls of an automobile is described.

```
type AutoControls is interface
public
    Speedometer : function() return MPH;
    Gas_Gauge : function() return Gallons;
    action Steering_Wheel(A : Angle);
    action Accelerator(P : Position);
    action Brake(P : Pressure);
extern
  action Warning_Light(S : Status);
  ...
end interface;
```

The instruments Speedometer and Gas_Gauge are specified as *public* functions, which, in effect, *output* their readings when asked.

Controls such as Steering_Wheel, Accelerator and Brake are specified as *public* actions. An AutoControls module can receive (and react to) events defined by these actions. The component that uses such a module must generate external events containing position, pressure or angle data. These external actions of a user should be connected by the containing architecture to the public actions so that the Autocontrols component receives the data in its public events. One way of doing this is mentioned later.

AutoControls assumes the presence of Warning_Light as an action whose events it can generate with appropriate status data. This allows AutoControls to send out warning events; the assumption is that the containing architecture will do something useful with them.

An automobile interface can be defined using different subinterfaces, including the one for the automobile controls.

```
type Automobile is interface
public
    service Controls : AutoControls;
    service Doors_and_Windows : AutoPorts;
    ...
end interface;
```

Declaring Controls and Doors_and_Windows as public *services* is a way of indicating that the constituents of these interface are "waiting" to be hooked up externally through the connection mechanisms of the architecture.

For instance, the AutoControls service of an Automobile can now be connected, using a single declaration, to any other object whose interface contains AutoControls as an *extern* service. Such an object would be capable of using the speedometer and gas gauge readings, and of generating events corresponding to changes in steering, acceleration, and braking. An automobile driver, as one might anticipate, is one such object.

The interface of an automobile driver would look as follows:

```
type Driver is interface
public
    include Person;
    Permit : Driving_License;
    ...
extern
    service DrivingSkills : AutoControls;
    ...
end interface;
```

Given the interfaces for automobiles and drivers, one can define an architecture that takes an instance of each and connects them correctly as follows.

```
architecture Vehicle_Test is
    Dummy : Driver;
    TestCar : Automobile;
connect
    Dummy.DrivingSkills to TestCar.Controls;
    ...
end architecture;
```

The connection in Vehicle_Test is shorthand for a set of basic connections between each pair of constituents with the same name, one in the *public* service and the other in the *external* service. So, for example, whenever Dummy calls its Speedometer function, the call is connected to the *public* Speedometer function in TestCar. Similarly, whenever the Dummy generates an *external* SteeringWheel(angle) event, TestCar will receive a *public* event with an identical tuple. When TestCar generates Warning_Light events, Dummy will receive them as public warning light events.

7 Conclusions

This document describes some of the salient features of a type system designed for use in a prototyping environment. The design has been based on recent and ongoing research in the theory of type systems as well as on implementation experience.

As mentioned earlier, any effective prototyping environment must provide the capability to construct large components easily, and must also support easy reuse and refinement of prototypes. RAPIDE's interface types provide a very convenient language to define components. The different flavors of type constituents, the parameterization mechanisms coupled with the subtyping constraints on type parameters, and the private and external interfaces add to the expressiveness of the types language. The interface derivation mechanisms provide flexible means of reuse, while the inferred structural subtyping discipline permits code to be used for a prototype and its refinements.

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

References

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

[CHC90] William Cook, Walt Hill, and Peter Canning. Inheritance is not subtyping. In *Proc. 17-th ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.

[Coo89] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[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.

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

[GP94] Castagna Giuseppe and Benjamin Pierce. Decidable bounded quantification. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[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.

[Kat92] Dinesh Katiyar. Subtyping F-bounded types. In *ANSA Workshop on F-bounded quantification, Cambridge*, Sept 1992. Position paper.

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

[Ken93] John. J. Kenney. Banking on X/Open. To appear as a Stanford University Technical Report, 1993.

[KS92] D. Katiyar and S. Sankar. Completely bounded quantification is decidable. In *ACM SIGPLAN Workshop on ML and its Applications*, 1992.

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

[LVB+93] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 1993.

[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.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[MT91] R Milner and M Tofte. *Commentary on Standard ML*. MIT Press, 1991.

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

[Pau91] L.C. Paulson. *ML for the working programming*. Cambridge Univ. Press, 1991.

[Pie92] Benjamin Pierce. Bounded quantification is undecidable. In *Proc. 19-th ACM Symp. on Principles of Programming Languages*, pages 305–315, 1992.

[Rit91] M. Rittri. Using types as search keys in function libraries. *J. Functional Programming*, 1(1):71–90, 1991.

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

[RT91] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *J. Functional Programming*, 1(2):191–212, 1991.

[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.

[TC93] W. Tracz and L. Coglianese. An adaptable software architecture for integrated avionics. In *Proceedings of NAECON 93*, pages 1161–1168, Dayton, Ohio, May 1993. IEEE.

[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.

A Type-theoretic foundations for RAPIDE

It is rather tedious to study and prove properties of the RAPIDE type system if one works with the framework in its entirety. Instead, a relatively simple and concise calculus that embodies the key concepts of the type system is chosen, and is made the focus of such theoretical studies. This section describes the types of this calculus.

The calculus has function types and record types that correspond rather loosely to the function types and interfaces of the RAPIDE type system. However, functions in RAPIDE can be polymorphic and their type arguments can be constrained using subtyping. Similarly, interfaces can be polymorphic and can also contain type constituents that may or may not be abstract. These "extra" notions are captured rather nicely by the polymorphic and abstract types of the calculus. The details are summarized briefly below.

The grammar for the types of the language is as follows:

$$
\begin{array}{rcll}
T & ::= & t & \textit{type variables} \\
 & \mid & T_1 \rightarrow T_2 & \textit{function types} \\
 & \mid & \mu t.T & \textit{recursive types} \\
 & \mid & \{\ell_1 : T_1, \ldots, \ell_n : T_n\} & \textit{record types} \\
 & \mid & \forall t{<:}F(t).T & \textit{polymorphic types} \\
 & \mid & \exists t{<:}F(t).T & \textit{abstract types} \\
 & \mid & \exists t = T_1.T_2 & \textit{transparent types} \\
 & \mid & \mathtt{fst}(e) & \textit{type selection} \\
 & \mid & \mathtt{Top} & \textit{supertype of all types}
\end{array}
$$

- The function type $T_1 \rightarrow T_2$ is the type of all functions that take arguments of type T_1 and return values of type T_2.
- The type $\mu t.T$ denotes type definitions that are recursive in the type variable t.
- The record type $\{\ell_1 : T_1, \ldots, \ell_n : T_n\}$ is the type of all records with fields $\ell_1, \ldots, \ell_n$ having types $T_1, \ldots, T_n$ respectively.
- Types of the form $\forall t{<:}F(t).T$ denote bounded polymorphic types. These types denote expressions that are polymorphic over all the subtypes of a given type. The bounds imposed on the polymorphic type variable can contain the variable itself – these are the so-called F-bounds.
- Abstract types in this calculus have the form $\exists t{<:}F(t).T$. It is easy to view such terms as being pairs (the first component being the hidden type and the second the term over which the type is abstracted). Once again, bounds can be imposed on the type variable that is being abstracted on.
- A slightly different variant of abstract types are types where the type component is not really hidden, and is known and visible outside the type. These types have the form $\exists t = T.T'$.
- For abstract and transparent types, one needs a way to extract the type component, and this is done using the term $\mathtt{fst}(e)$. (Correspondingly, the expression language uses the term $\mathtt{snd}(e)$ to select the value component)
- `Top` is the supertype of all types. This type allows one to write unbounded polymorphic types and completely opaque abstract types as special cases of bounded polymorphic types and bounded abstract types respectively (by using `Top` as the bound in each case). `Top` also has ramifications on the decidability of subtype checking, and these have been studied in [KS92].