An Implementation of Hybrid
A Concurrent, Object-Oriented Language*

D. Konstantas
O. Nierstrasz
M. Papathomas

Abstract

This paper is a report on a prototype implementation of *Hybrid*, a strongly-typed, concurrent, object-oriented language. The implementation we describe features a compile-time system for translating Hybrid object type definitions into C, a run-time system for supporting communication, concurrency and object persistence, and a type manager that mediates between the two.

1 Introduction

This paper reports on a prototype implementation of *Hybrid*, a language for programming with *active objects*. Hybrid is a general-purpose programming language intended to address application domains (such as office systems) where concurrency, distribution and code reusability are important issues. As such, the design of Hybrid attempts to integrate three different object models:

1. objects as instances of reusable *object classes*
2. objects as *typed entities*
3. objects as independent *active entities*

The object class model adopted by Hybrid is basically a variant of the Smalltalk model, supporting multiple inheritance and parameterized object classes. The type model insists on compile-type-time correctness of all expressions, while accommodating polymorphism and dynamic binding. The main innovation in Hybrid is the concurrency model, which provides for objects to use a uniform paradigm for communication regardless of their degree of independence. The mechanisms for creating, interleaving and scheduling threads of control are, respectively, *reflexes*, *delegation* and *delay queues* [Nierstrasz 1987].

The prototype that we will describe implements a significant subset of the Hybrid language, and of the functionality of an environment for active objects. In particular, pseudo-concurrency and object persistence are supported, whereas distribution is not. The implementation consists of three

*In *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988, pp. 61-105.

main components: a *type manager* that manages all information related to object types, a *compiler* that generates C language code from Hybrid type definitions, and a *run-time system* that handles communication, pseudo-concurrency, and persistence.

The following table outlines the structure of the report:

Readers who only wish to get an idea of the system and its implementation should look at sections 2, 3, and 7-10. Sections 4-6 contain more detailed descriptions of the three components of the prototype.

2 The Hybrid Language

This overview is intended to provide the background necessary to understand the discussion that follows on implementation issues. It does not provide a complete description of the language. A more detailed exposé of the language constructs and of the semantics of the concurrency mechanisms can be found in [Nierstrasz 1987a,b,c].

An application written in Hybrid is a collection of cooperating active objects, possibly distributed amongst a number of separate object environments. The granularity of concurrency is defined by *domains*, which can be thought of as independent "top-level" objects. All activity within a domain is sequential, but there may be many domains concurrently executing within an environment.

Objects are *active* when they are responding to a message. Since all objects are instances of abstract data types, this means that objects are active when they are responding to an operation invocation, or when they themselves receive a response to request they have issued. The basic model of communication is that of remote procedure calls. We can therefore trace threads of control, called *activities* as sequences of *call* and *return* messages between objects, whether they communicate within a domain or between domains.

"Programs" in Hybrid consist purely of object type definitions. The language requires that all expressions be statically type-correct, but provides for dynamic binding and type polymorphism.

We shall provide brief overviews of how types are defined, and of the mechanisms for manipulating threads of control.

2.1 Object Types

Every object in Hybrid is an instance of an object type. Type definitions have the following general form[1]:

```
type typeName parameters :
    typeSpec ;
private
    implementation
```

The *typeSpec* describes the interface to instances of the type. The interface is typically a set of operations that may be invoked (including the specification of argument and return types), but may also include "visible instance variables". In addition to named operations, one may define and overload a set of infix and prefix operators recognized by the language.

Interfaces are described using a number of type constructors, the most general of which is `abstract`, which requires the programmer to explicitly provide the list of visible operations associated with the type. The other constructors are `inherits`, for defining subtypes that inherit operations from multiple parents, `enum` for defining enumerated types, `oid` for defining object identifiers, `array` for defining homogeneous arrays, and `range` and `variant` for defining records and variant types. A type may also be defined as a range of integer values (ranges of values from enumerated types are not yet handled).

[1] See the appendix for the precise grammar of the implemented prototype.

Type parameters may be supplied to define generic types. The parameters may be used either in the interface definition or in the body of the implementation, but must be bound to actual types before instances can be created.

The implementation specifies the persistent instance variables for instances of the type, as well as the implementation of all visible and hidden operations (i.e., the *methods*). For all of the type constructors except `abstract`, the implementation is typically omitted, since it can be inferred from the *typeSpec*. Subtypes may, however, override inherited methods. One may also define abstract types with no implementation, but these types cannot be instantiated.

The interface to an object is its *effective* type. The *actual* type of an object is determined by its implementation. A type T_1 *conforms* to another type T_2 if supports at least the same interface, i.e., if it supports at least the same set of operations with the same specifications. (We also say T_1 is a *subtype* of T_2.) Expressions are type correct if operation invocations are consistent with the effective types of the target and argument subexpressions. Variables may be dynamically bound to the value of any expression that conforms to the declared type of the variable.

Type casting is required to inform the compiler that the effective type of an expression conforms to another, expected type. In the implementation, this step guarantees that the appropriate method lookup table will exist so that the type-cast object can efficiently respond to messages intended for objects of the type it conforms to. For example, instances of *Spline* must be cast to the more general type *GraphicalObject* before they can be passed to a tool that only recognizes objects of the latter type. Once type-casting has been performed, there is only a small, fixed overhead in looking up the method for, say, a *display* operation.

Hybrid also supports a `check` statement for disambiguating variant types at run-time, and for determining whether an object actually belongs to a subtype of its effective type (e.g., to check whether a *GraphicalObject* is really a *Spline*).

Hybrid has an `if` statement and a `switch` statement for selectively executing code. Repetition is provided by a `loop` statement, which may be repeated with a `continue` statement, or exited with a `break` statement. A `block` is similar to a loop, except that it can only be exited, not repeated. In case of nested loops or blocks, a label may be supplied to either `break` or `continue`.

2.2 Communication

Messages between objects are generally either *call* messages, requesting an object to execute one of its methods, or *return* messages sent after the successful completion of a method. A *start* message is used to initiate a new activity. (Exceptions were envisaged as a necessary alternative to *return* messages, but were not included in the initial language design.) An activity traces a thread of *call* and *return* messages amongst a set of objects. Messages may be delivered either synchronously when communication is between objects within the same domain, or asynchronously when communicating objects are independent. When an object sends a *call* message to a remote target, the object's domain ordinarily *blocks* until a response is received. (Recursive calls, related to the blocking activity, are permitted.) An activity can always be viewed as being at a unique location, either within an object executing a method, or buffered in a message queue. Similarly, domains can always be viewed as being in one of three states: *idle*, *running*, or *blocked*.

New activities are created by invoking a special kind of operation called a *reflex*. When a reflex is invoked, a *start* message is sent to the object, and accepted as soon as the object's domain is idle.

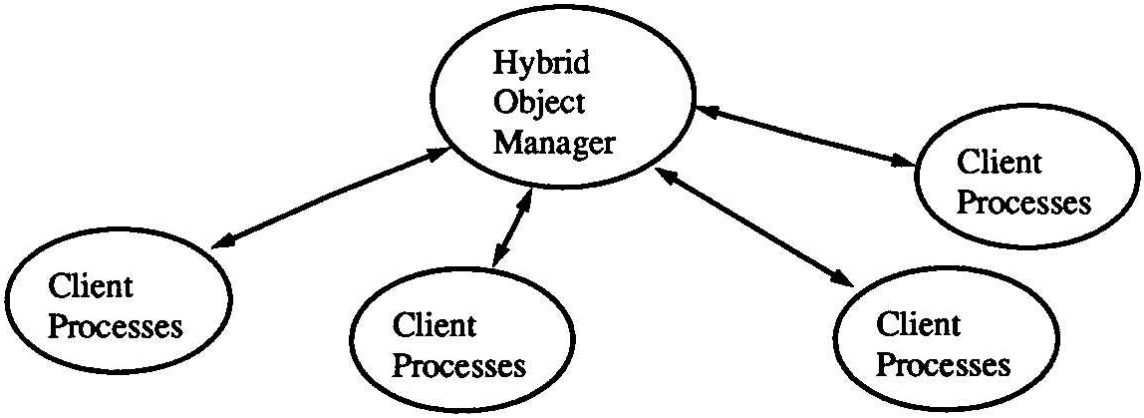

Figure 1: Multi-user support

Since reflexes do not return anything, the effect is to initiate a new thread of control.

Two additional mechanisms are required in order to be able to program interesting active objects. *Delegation* is a mechanism for interleaving threads of control. An expression of the form:

`delegate` (*target op args*)

will always be evaluated by asynchronous message-passing, and will leave the calling domain *idle*, that is, free to accept messages related to other activities. The context of the delegated expression is saved, and later resumed when the *return* message is eventually received. Aside from interleaving of activities, delegated expressions behave just like non-delegated expressions.

Delay queues are used to schedule activities when there are operations that cannot always be performed. These operations are declared as *using* a named delay queue, and the object manages the queue of buffered messages by *opening* and *closing* the queue during the execution of other methods. The delay queue is typically used to represent either the availability of a resource, or status of an awaited condition, much in the same way that condition variables are used in monitors. The main difference is that opening or closing a delay queue does not entail an immediate transfer of control, as is the case with *waits* and *signals* [Hoare 1974]

For examples of object types that use these mechanisms, the reader is referred to section 7 of this paper and to Appendix B.

3 System Overview

The Hybrid execution model is that of a distributed collection of object environments, each of which provides support for persistent active objects and for communication between objects in different environments. The prototype implementation is currently restricted to a single object environment, but with support for multiple users, as is shown in figure 1.

The Hybrid object manager effectively functions an "object server" for users' client processes. In the sample applications implemented using the prototype, the user processes are responsible for connecting to object manager, and for managing the user interaction objects (e.g., windows).

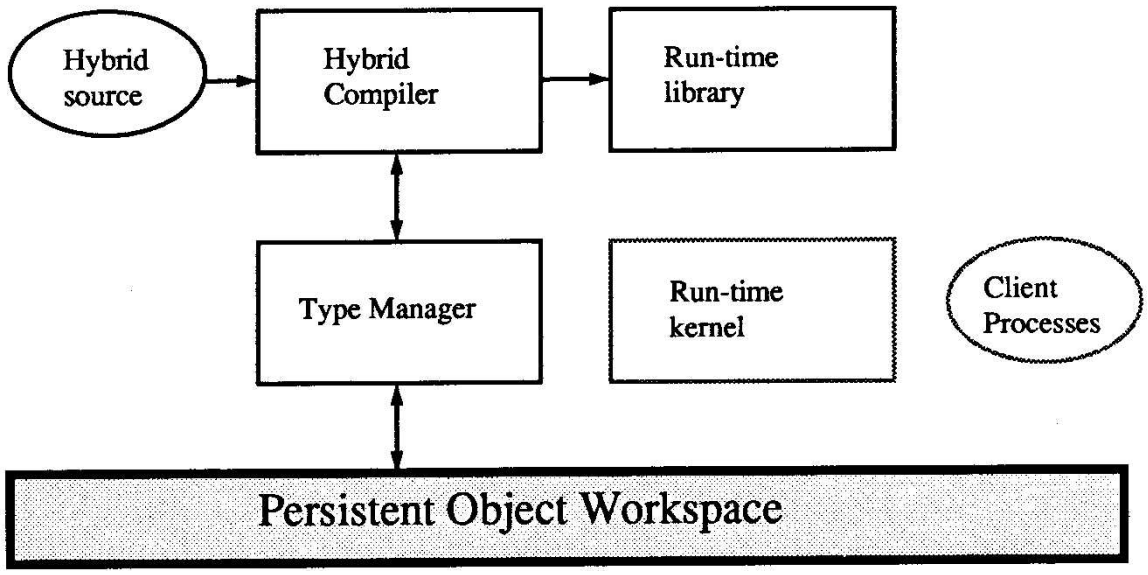

Figure 2: Compile-time view

Objects in the client's environment have corresponding "shadow" objects in the Hybrid object environment, which forward messages to the client.

The object manager is implemented as a single UNIXprocess that manages the workspace of active objects. Persistence is provided by storing the workspace in a file. The workspace is therefore limited by the size of virtual memory. Pseudo-concurrency is provided by light-weight processes implemented using a coroutine extension to the C language.

The system consists of three main components, the Hybrid *compiler*, the *type manager*, and the *run-time system*. After considering the alternatives, it appeared that the fastest and most flexible way to implement the compiler was to use the C programming language as a high-level "assembler". Dynamic linking was not considered a high-priority item for the prototype, so the present implementation does not integrate the Hybrid compiler into the object manager. We therefore distinguish between the compile-time and run-time views of the system.

In figure 2, we see Hybrid type definitions translated to C, compiled into run-time libraries, and linked in with the object manager. The type manager keeps track of a database of all information concerning object types, other than the actual executable code for the methods. The type database is stored directly in the persistent workspace. The type manager provides the mechanisms for the realization of multiple inheritance, code resubality, type parameterization, overloading and version management. The compiler communicates with the type manager in order to verify type-correctness of new type definitions, and generates information concerning new types to be stored in the type database for later use.

In figure 3 we see the run-time view of the system. The run-time system mediates between active objects and the client processes. The system implements Hybrid activities as light-weight processes. Communication with clients is supported by providing special object types that know how to communicate with the outside world. These types, as well as all basic Hybrid types, exist in the run-time type library. The type manager is responsible for the method lookup tables needed to support dynamic binding, and for the information needed to create and delete objects.

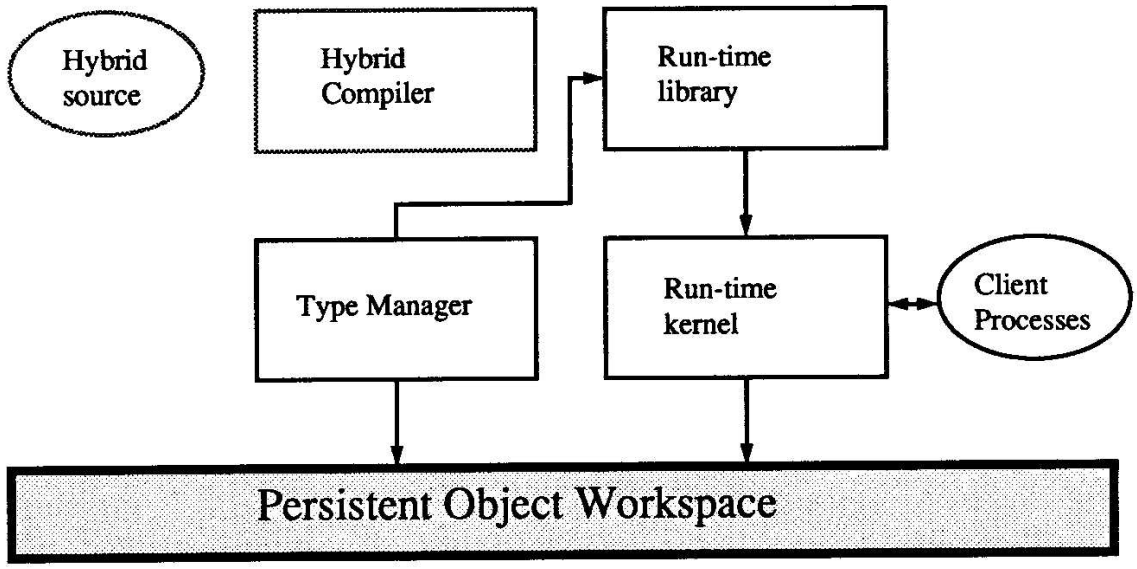

Figure 3: Run-time view

4 The Hybrid Compiler

The Hybrid compiler has been implemented as a three pass compiler: source code parsing and syntax checking occur during the first pass, semantic control during the second, and code generation during the third. Lexical analysis and parsing of the Hybrid source code, are performed with the help of the UNIX *lex* and *yacc* utility programs [Lesk and Schmidt 1975; Johnson 1975].

During lexical analysis, a small number of preprocessing statements is recognized and processed. These statements contain an underscore as the first character of the line. The statements recognized are `_include <file Name>`, for including the body of another file, and `_<number> [<file Name>]`, for changing the compiler's notion of the current line and file name.

Comments are stripped out during lexical analysis. Hybrid supports two styles of comments: single-line comments, preceded by `#`, and nested comments, surrounded by `#{` and `#}`. Nested and single-line comments can be combined to comment out one another.

During the *first pass* of the compiler, a parse tree and a symbol table are created for the Hybrid source program. Syntax errors are detected and reported. The parse tree also contains information regarding the line number and file name corresponding to each node. This information is used during the second pass for error reporting.

At the end of the first pass the symbol table is scanned and information concerning the referenced types is requested from the type manager. If unknown type names are referenced, error messages are given and the compilations ends. During the same scan, parameter types, operation arguments, methods and private instance variables are identified and indexing information is inserted in the symbol table, to be used during the third pass.

The *second pass* is responsible for *semantic* checking of the source code. For the prototype we have implemented only a very basic semantic control. That is, we only identify visible operations of types, obtain their return value type, and check basic expression type-correctness. Arguments passed to operations, consistency of private and public parts, and further semantic consistency are

not checked. If errors are encountered during this pass (i.e., unknown operations, expression-type mismatches etc.) error messages are given and the compilation stops at the end of the pass.

Code generation takes place during the *third pass*. When this pass is initiated, it is assumed that the program is correct, and that all information needed is available in the symbol table and the parse tree. Intermediate C code is generated for the methods and local procedures defined for the new type. The generated code is placed in a file whose name is that of the compiled type followed by a `.c` suffix. The created C file is then been compiled, and added to the type library. In addition, type information from the public and private parts is passed to the type manager and inserted into the type database.

Hybrid type definitions are stored, as mentioned, as UNIX text files. Although it is possible to have a single file containing several type definitions, the present implementation of the compiler does not recognize references between these new types. The reason is that information regarding the new types is inserted in the type database during the third pass *for all types in the file*. Consequently they are not known to the type manager during the first and second passes, when the references must be resolved.

4.1 Parse Trees and Type-checking

A question we had to answer when implementing the Hybrid compiler, was which aspects of the language can be checked on the syntax level and which on the semantic level. Our intention was to check as much as possible on the syntax level, that is at the *first pass*. We soon realized that the skeleton parser we had was not adequate for our needs. Many of its rules had ambiguous semantic interpretation, depending on the context in which they appeared. Furthermore, the skeleton parser was accepting features that either we did not intend to implement in the first prototype or were invalid. For example, one could write

```
type A : inherits { abstract { ..... }, variant { ... } } ;
```

which is not a valid type definition. For these reasons we modified the grammar and the parser, adding more rules and simplifying others so that each rule would have only one semantic interpretation.

The nodes of the parse tree are effectively variant records, with some common fields for all nodes, and variant fields depending on the kind of node. Leaf nodes corresponding to primitive expressions are either *type-independent*, meaning that their type is fixed, or *type-dependent*, meaning that the type of the expression depends on a neighbouring node. For example, in `x.y`, the type of `y` depends on the type of `x`, but `x` itself is type-independent. The nodes of the parse tree fall into the following four categories:

- *type-generating nodes:*
  These are leaf nodes, whether type-dependent or type-independent.
- *type-modifying nodes:*
  These nodes produce a new type, possibly making use of the types associated with their subtrees. An example is operation invocation.
- *neutral nodes:*
  Nodes that neither modify nor generate a type, but simply forward type information to higher

nodes. The node for *expr* is an example.

- *untyped nodes:*
  These nodes have no associated type. Typical examples are *loop* and *if*, which deal with statements rather than expressions.

Type-checking, which takes place in the second pass of the compiler, is performed by traversing the parse tree, starting with the type-independent leaf nodes, and passing type information up the tree. When reaching parent nodes with multiple children, type information may be passed down to unchecked subtrees to determine the parent's type.

4.2 Code Generation

After semantic checking has been completed, the parse tree and the symbol table are made available to the code generation pass of the compiler. During code generation two parallel activities take place: C code is generated, and the type manager is given information regarding the new type.

First the type definition part of the parse tree is traversed. From that all information regarding the interface of the type is passed to the type manager. Then, if the type is an *abstract* type, the private part is traversed. Information regarding instance variables is given to the type manager. At this point the type manager has *all* the information it needs concerning the new type.

Next, the branches of the parse tree describing methods are traversed, and code generation starts. First the C header files are printed (the same for all types). Methods (whether private or public) are translated according to a standard model. All information regarding the kind of the method (i.e., *reflex, infix* etc.) is ignored at this stage, since this is semantic information not relevant at run-time.

As an example, consider the method:

```
reflex method ( argument1 : A ; argument2 : B ) -> ...
```

Upon translation we will have:

```
static bl_Start method ( argList, curObject, ivOffset, paramOffset)
       bl_Start argList, curObject ;
       int ivOffset, paramOffset ;
```

The C type `bl_Start` is used for offsets within the workspace. So, the arguments of the Hybrid method are packed as a list, the pointer to the executing object is provided, and then we have the offsets to the instance variables and parameters (discussed later). The methods are translated to `static` functions so that we avoid name clashing when creating the methods library.

Next we deal with automatic variables and executable code. Automatic variables (i.e., local to a method) are instantiated at run-time. They are freed just before the return of the method. If the operation uses a delay queue, a call to check the state of the queue is inserted at the beginning of the generated code.

Statements consisting of an executable expressions are evaluated left to right (or as required by operator priority and parentheses). Since the result of an expression may be used as either a *target*

or an *argument* of an enclosing expression, a reference to the object is always generated rather than just a value. If the value of the expression needs to be sent in a message to another object, a temporary copy is made. When the statement has been evaluated, the temporary objects are freed. The code generated for the statement consists of a C block containing:

- temporary object declarations,
- temporary object initializations,
- expression executables, and
- freeing of temporary objects.

For integer literals and strings, we generate code that will create temporary instances of `integer` and `string`, and initialize them to the appropriate values.

Hybrid *loop* statements are translated to C `for(;;)` statements. If the statements have a label then the following statements are added

```
_CONTINUE_labelName : continue ;
_BREAK_labelName :    break ;
```

just before the end of the `for` loop. Hybrid *break* and *continue* statements are translated to `break` and `continue`, if they do not have a label, and to `goto _BREAK_labelName` and `goto _CONTINUE_labelName` if they do have a label.

Block statements are treated as *loop* statements, except that the statements added before the end of the `for` loop are:

```
_CONTINUE_labelName : ;
_BREAK_labelName :    break ;
```

This way a *continue labelName* statement will behave as a *break*[2]. The Hybrid *if - else , check* and *switch* statements are translated using C `if - else` statements. *Return* and *end* statements are translated to `return`. *Delegate, oidCalls, reflexes* etc. are translated to run-time library calls.

After translation of all methods is completed, a table containing the C functions that implement the methods is created, and a function that returns this table is defined. The function is public and has a unique name, constructed from the type name. It is called by the type manager whenever the system comes up, to initialize the function pointers in the method table.

5 The Type Manager

The type manager handles all information generated by the Hybrid compiler regarding object types. This information is stored in a type database within the persistent workspace, in order to be accessible to active objects at run-time.

The type manager consists of three components:

[2]Note that a *continue* statement is illegal inside a block. This is something that the semantic pass should find, but is not done in the current implementation.

- the *type interface manager*
- the *version manager*, and
- the *type manager Coordinator*.

The type interface manager handles all information concerning the visible interface of object types; the version manager keeps track of versions of type realizations; and the type manager Coordinator coordinates the other two, providing an interface to the external world. In addition to these modules, there is a *run-time interface* to the type manager that is mainly responsible for handling run-time errors. (The run-time interface is not actually part of the type manager, but cooperates closely with it.)

All items in the type database have a unique identifier (id), each containing three fields:

- the *item* field, indicating the kind of item the id refers to (i.e., a type, a version, a table, etc.)
- the *environment* or *index* field, identifying either the environment of the id (for types and versions) or the specific entry in a table (e.g., the fifth operation in the method table).
- the *key* field, which uniquely identifies the item in its environment.

5.1 Basic Types

Basic types are those provided directly within the system. Basic types are either *primitive* or *generic*. The primitive types are those at the lowest level in the type hierarchy, used to construct more complex types. Examples are integers, booleans, and delay queues.

For practical reasons, there are some restrictions on the way primitive types can be used:

- Primitive types cannot be inherited. For example, one cannot define a new type inheriting from integers and booleans.
- Instances of some of the primitives types cannot be copied, in particular, delay queues and displays.

Generic types are blue-prints for only defining other types. The generic types in Hybrid are `array`, `enum,` and `oid`. For example, all arrays have the same interface, but differ in the index type and in the type of elements they hold. Thus, arrays cannot be considered to belong to the same type, but rather each defines a new type, with a similar interface and methods. The generic array type holds all information needed for the creation of an array type. An array without its parameters bound is not a true type, and cannot be instantiated. Furthermore, `array` is a reserved word, so an attempt to define a variable as:

```
var x : array ;
```

will yield a syntax error. Types constructed using generic types share the pre-defined methods.

All basic types have been implemented in C and are integrated into the system. They are loaded by the type manager whenever the workspace is initialized.

5.2 Type Interface Manager

All types in the type database are referred to by their type id (described above). The *item* field directly indicates what type constructor was used to define the type, i.e., `abstract`, `record`, `oid`, etc. The *environment* field is intended for types defined in a distributed setting, and is therefore not used in this prototype. The *key* field is used to search a table containing the name of the type and the location of the type information structure. If a type is defined as an *alias* for an existing type, as in:

```
type newType : oldType ;
```

then `newType` will be given a new type id, but it will point to the the same entry as `oldType`.

The type information structures vary depending on the type constructor used in the definition. The most common is the *abstract type* structure. The three others are for: *enumeration* and *range* types, *variant* types and *array* types.

5.2.1 Abstract Types

Abstract types are the most common types in Hybrid. Their type information structure is the most general and the most complicated one! The following tables are needed for an abstract type:

- The *public operation table*.
  Each entry contains the name of the operation (or operator), the lists of type ids for the argument and return values[3], other information concerning the nature of the operation (i.e., infix, prefix etc. and whether it uses a delay queue), and the type id of the type where the operation was originally defined. This last piece of information is needed to support multiple inheritance, to be explained later.

- The *public instance variables table*.
  Each entry holds the name of the instance variable, its type id, and the type id of the type in which it was introduced (for supporting multiple inheritance).

- The *type parameter list*.
  Abstract types can be parameterized. The parameters are constrained to be subtypes of some given type. (NB: an unconstrained parameter is "constrained" to be a subtype of the most general type, *object*.) For each parameterized type we store the ids of the constraint types and the origin type of the parameter (for multiple inheritance).

- The *parameter offset table*.
  This table is used in support of multiple inheritance, and will be explained later.

- The *casting table*.
  Any object may be cast to an effective type to which its actual type conforms. The number and order of the visible operations and instance variables may be different after casting. Each entry of the casting table holds the type id of a type to which instances can be cast, and points to two lists containing the mapping between operations and instance variables of the two types (more details will be given in the section about type casting).

[3]Note that the language does not currently support compound return values.

- The *parent list*.
  For supporting multiple inheritance, we keep track of the list of type ids of ancestor types from which a child ultimately inherits.

- The *version list*.
  For each type we keep track of the list of its versions. The "top" entry is the most current version of the type. In order to allow sharing of the version list of generic types, the number of versions that a type can have is limited to a pre-defined maximum. Inheriting types have new versions automatically created when an ancestor is updated.

- The *type reference table*.
  Parameterized types used in type specifications must have all their parameters bound, either to existing types or to the parameters of the type being defined, as in:

  ```
  type newType : oldType of (integer, boolean) ;
  ```

  or:

  ```
  type newType of (parameterType :< boolean) :
          oldType of (integer, parameterType) ;
  ```

  The type reference table keeps track of the parameter bindings.

- The *string table*.
  For each type in the type database, all of the strings (names) used in the type definition are stored in the string table, which is a single, contiguous block. Occurrences of these strings are translated into *name ids*, which are offsets into the string table.

Finally, for each type there is a *tinfo table* that holding the pointers to all of the above tables. The type id is redundantly stored as a means to check consistency of the type database.

5.2.2 Enumeration and Range Types

Because the semantics of enumeration and range types similar, we use the same type information structure for both. Given an enumeration type, such as:

```
type colors : enum {white, yellow, red, green, blue, black} ;
```

we assign a number to each element of the enumeration type and use this number in compilation. This way the above is finally seen as a range from 1 to 6. Whenever the item `green` is referenced we instead use 4. This is done during the second pass of the compiler after ensuring the type correctness of an expression. Thus, the tables held for enumeration types are:

- The *enumeration element list*.
  This holds the names of the enumeration elements. Range types have a null list.

- The *range list*.
  This is a list of ranges in the form *from .. to*, specifying the values of a range type. (Multiple ranges could thus be handled, though the language does not support them.)

In addition to the above tables, the *tinfo* table includes pointers to the following generic enumeration type tables:

- The *operation table*.
- The *version list*.
- The *string table*.

These tables are as described for the abstract types. Note that for enumerated types, inheritance is not supported.

5.2.3 Variant Types

Variant types are *virtual types*, that is, no versions are attached or can be attached to them. Variant type objects are null objects (no operations, no instance variables) until they are bound (dynamically) to an instance of the list of variant types. Here we must note that message passing in variant type objects can be done *only* after performing a run-time type-check to disambiguate the actual type of the instance (i.e., using the `check` statement provided by the language).

The tables for the variant types are

- The *variant list*.
  This holds the names of the variant types and their type ids.
- The *string table*.
- The *tinfo table*.

Note that inheritance of variant types is not allowed in this implementation.

5.2.4 Array Types

Arrays are specially handled by the type manager. First, there is a generic type *array* that provides the array operations. Second, array types serve as blueprints for their instances. The number of instance variables is determined from the index parameter. The bounds are stored with the instance variables and are returned by the `lower` and `upper` operations. Array types have the following tables:

- The *parameter list*.
  Containing the parameter constraints, as for abstract types.
- The *array elements table*.
  This holds the upper and lower index bounds, and the type id of the array elements.
- The *operation table*.
- The *version list*.

- The *string table*.

The last three tables are those defined for the generic array, and are shared by all array types. Note that also for arrays, inheritance is not supported.

5.3 Version Management

Each version in the type manager is associated with a *version id*. The fields of the id are the same as for the type ids. In contrast with the type interface manager, the version manager uses a single structure for all versions. For each entry the following tables are held:

- The *method table*.
  This table holds the C function pointers identifying the executable code of the methods for a type.
- The *instance variable template*.
  This describes how to instantiate objects. It includes the names of the instance variables, their type ids, the original ancestor's type id and, if the type of the instance variable is parameterized, the list of parameter bindings.
- The *public to private instance variable mapping table*.
  This table identifies which actual instance variables in the realization of a type correspond to those defined publicly in the interface. This is necessary since there will typically be other, private instance variables as well, and different realizations may require different mappings.
- The *instance variable offset table*.
  The table is used for the support of multiple inheritance and will be described shortly.
- The *string table*.
  As before.
- The *vinfo table*.
  This table holds the pointer to the other tables and also the version id and the type id of the corresponding type.

5.4 Object Instantiation

An object, that is, an instance of an object type, is represented by a structure containing pointers to various tables. All objects, independently of their type, are represented using the same structure. In the discussion that follows, when we say "the pointer to the object" we mean the pointer to the structure containing the object's representation.

There are two kinds of tables in the object instance structure: private tables specific to that instance, and shared tables from the type database. The object instance structure contains the following fields:

- The *actual type id*.
  This identifies the type of which the object is an instance, i.e., the type providing its realization.

- The *effective type id*.
  This identifies the type that the instance currently appears to belong to. This must be a type to which its actual type conforms. The effective type is changed by type-casting.
- The *version id*.
  Identifies the realization version of the instance (which may be older than the most current version of that type).
- The *instance variable array*. (private)
  This is an array of pointers to the instance structures of the values bound to the instance variables.
- The *parameter binding list*. (private)
  Instances of parameterized types have a copy of the list of types bound to the parameters.
- The *method table*. (shared)
  From the version entry.
- The *operation casting table*. (shared)
  When the actual and effective types differ, the casting table must be used to correctly index the method table. This is a shared table from the type interface entry.
- The *public to private instance variable mapping table*.
  From the version entry.
- The *instance variable casting table*. (shared)
  From the type interface entry.
- The *instance variable offset table*. (shared)
  From the version entry.
- The *parameter offset table*. (shared)
  From the type interface entry.
- The *link counter*. (private)
  A counter indicating the number of objects that are referencing it. When the counter reaches zero, the object can be deleted.
- The *deletion flag*. (private)
  When an object issues a delegated call, the result of the call must be returned to the object. But if in the meantime the object attempts to delete itself, we defer deletion until the delegated call returns, raising the flag to indicate that.

We must note, however, that instances of basic types (i.e., integers, booleans, etc.) are handled differently. The instance variable array, instead of holding pointers to further instance structures, directly contains the C value corresponding to the value of the instance.

5.5 Implementation of Mechanisms

5.5.1 Multiple Inheritance

Multiple inheritance can be implemented in several ways, depending on when properties are to be inherited, and to what degree sharing is an issue. For example, one can create realizations

of children at compile-time, and automatically create new versions when parents are modified, or create the type interface at compile-time and defer realization creation until an instance is actually needed. There is a trade-off in the two approaches between compile-time and run-time overhead. Another implementation decision is whether children and parent should share not only source code but also executable. In our implementation we chose to defer realization creation until and instance was needed, and to generate code that can be shared between both parents and children.

The main problem with generating sharable methods is that instance variables may have a different relative location in parents and children. Suppose `parent1` is an abstract type with operations `op11, op12` and instance variables `iv11, iv12`. Similarly, `parent2` is an abstract type with operations `op21, op22` and instance variables `iv21, iv22`.

Instance variables are accessed by indexing an array holding the values they are currently bound to. Since all objects have a fixed size representation, this indexing is equivalent to the use of an offset into the instance variable array.

In an instance of type `parent1` the instance variable array will have entries `(iv11, iv12)`, and the method table will be `(op11, op12)`. If `op11` accesses `iv11`, then the index used to access the instance variable array will be 1, and it will be hardcoded in the executable method.

If we now define `type child : inherits (parent1 and parent2) ;`, then `child` will have four operations and four instance variables. The version's method table will point to the executable methods of the parents. The array of instance variables for an instance of `child` will be `(iv11, iv12, iv21, iv22)`, and the operations will be `(op11, op12, op21, op22)`. As we said the child will use the same code for the operations as the parent. So when `op21` is called (which uses `iv21`) it should access not the first instance variable, but the third. But the index is hardcoded to be 1. In order to solve this problem the *instance variable offset table* from the version is used (which is included in the instance). In this case the table will be `(0,0,2,2)` and upon compilation of the methods instead of directly using the instance variable index, we use `ivOffsetTab[<operation Index>] + index`, where `<operation Index>` is the index of the executing operation.

Back to our example now, `op21` has index number 3, and thus it will access the instance variable `2 + 1` which is `iv21`.

Obviously abstract types will have only zeros in their *instance variable offset table*.

Another problem that arises in multiple inheritance is *name clashing*. There are several criteria for uniquely identifying an operation name. If more than one operations have the same name, then the arguments of the operation are checked. If the argument types are also the same, then the originating type must be specified. [4]

It is also possible that a child may multiply inherit from the same parent. For example, in:

```
type child1 : inherits (parent1 and parent2) ;
type child2 : inherits (parent2 and parent3) ;

type child3 : inherits (child1 and child2) ;
```

we optimize `child3` by including `parent2` only once, thus avoiding the name clashing problem.

[4] This is not supported in the current implementation of the language.

5.5.2 Type Parameters

Abstract types may be parameterized. The type parameters may be used in the type specification and in the realization of the type as though they were actual types, but they must be bound, before instances can be created (just as with the basic type `array`). Type parameters were not difficult to implement, but combining them with multiple inheritance posed some problems that were not easy to solve.

We shall first describe how simple parameterization is implemented, and then we will extend the description to deal with multiple inheritance.

Parameters can be used in Hybrid as if they were actual types. Upon instantiation, a list of bound type ids is associated with the instance.

When we compile a parameterized type, we recognize the parameters, and instead of placing a type id in the code, we reference the corresponding entry in the parameter binding list of the instance. At run-time the correct type id will be found.

In the case of constrained parameters, the type that a parameter is bound to must be a subtype of the constraining type. Type casting is used to ensure that methods will be correctly accessed. Instances of the type bound to the parameter are cast to the constraining type, and accessed via the restricted interface.

Combining parameterization with multiple inheritance poses a problem similar to that of resolving instance variables and operations, namely, to have the parameterized parent access the correct parameters. The solution we provide is similar to the one used for instance variables and operations. We use an offset to correct accessing of the parameter binding list. The *parameter offset list* has one entry per operation and is indexed via the operation's index.

5.5.3 Type Casting

Recall that objects in Hybrid may have both an *actual* type, which is the type providing the realization of its instance variables and methods, and an *effective* type, which may be any supertype of the actual type. For example, the declared type of an instance variable is generally an effective type, since variables may be dynamically bound to values whose actual type is not known till run-time. Objects must be *type-cast* in order to change their effective type. This procedure is important not only for statically verifying that expressions are type-correct, but actually causes code to be generated that ensures that the object's instance variables and methods will be consistently accessed.

The following example illustrates the realization of type casting in the prototype.

Suppose that `type1` is a type with operations `op1, op2, op3, op4` and instance variables `iv1, iv2, iv3 iv4`, and that `type2` has operations `op2, op4` and instance variables `iv1, iv4, iv2`. If the common operations of the types take the same arguments and return the same values, and the common instance variables are of the same type, then `type1` is viewed as a subtype of `type2`, and so instances of `type1` may be cast to `type2`.

Suppose that we have a variable of declared type `type2` which is bound at run-time to an instance of `type1`. The code generated by the compiler, however, will be based on the variable's effective type. Since operation invocations are translated into indices in a method lookup table, a call to operation `op2` will be translated to the index 1. For an instance of `type1`, however, `op2` is the *second*

operation in the table. Access to methods and instance variables is therefore corrected through the use of a *casting table* that provides the mapping from the effective type of an expression to the actual type of its value. Casting tables are stored in the type database together with the type interface information concerning the actual type involved in the mapping.

The casting tables required for the example to cast from `type1` to `type2` are:

- operation casting table: (2, 4)
- instance variable casting table: (1, 4, 2).

Casting tables play a very important role in the prototype. They are not only used for accessing the method table, but also in every case where an operation index is needed. That is, for accessing parameters, instance variables, etc. In the next section we present a global view of the combined usage of the various mechanisms we have presented.

5.5.4 A Global View

A large number of transformation and mapping tables are used for reaching instance variables and methods. We have tables for offset correction due to inheritance, tables for the mapping parameters and tables for type casting. All these tables *must* be used together, since the mechanisms can be combined. That means that on code generation we must reference an operation through a number of tables, in order to end up with the correct method.

As an example, consider the following parameterized type:

```
type newType of (parameter :< Y ...) :
        abstract { ...  } ;
private {
        # instance variables
        var iv1 : A ;
        var iv2 : B ;
        ...
        operation : (arg1 : A ; arg2 : B) -> ;
        {
            var y : parameter ;
            var x : X ;
            ...
            x.op1() ;
            x.iv2 ;
            arg1 := y ;
            iv1 := z ;
            ...
        }
 }
```

The C code generated for the operation will be:

```
static bl_Start operation(argList, curObject, ivOffset, paramOffset)
```

```
bl_Start argList, curObject ;
int ivOffset, paramOffset ;
```

Inside the operation code we have two automatic variables. One is of a known type `X`, and the other is of the parameter type. Before starting execution the two variables will be created. Instantiating variable `x` is trivial since its actual type is known at compile-time. For `y`, however, we must refer to the object's parameter list. Since the object's type may have been defined through inheritance, the parameter list may contain inherited parameters from other parents. The part that refers to this type starts at offset `paramOffset`. Since the parameter is the first in the list, we will find the type id of the type it has been bound to at:

```
curObject->paramTab[paramOffset+1]
```

Instance variables are similarly accessed via `ivOffset`.

Accessing methods and instance variables of a remote object is more complicated. Casting tables need to be used at every step. The index of the method must be found via the operation casting table. Instance variable and parameter offsets are obtained from the corresponding tables via the operation index. But the index must be corrected via the casting table. Thus the translation of the `x.op1()` call will be (assuming that op1 has index 1):

```
*(x→opTab[x→castOpTab[1]])
    (<packed Arguments>, x,
    x→ivOffsetTab[x→castOpTab[1]],
    x→paramOffsetTab[x→castOpTab[1]])
```

For the instance variables another level of indirection is needed, since the instance variable table contains all the *private* instance variables, and we only know at compile-time the *public* ones. That is, in order to access `x.iv2`, we need:

```
x->ivTab[x->pub2real[x->ivCastTab[2]]]
```

Of course, one thing that we must not forget is that all tables (specifically, all persistent data) are referenced using offsets into the workspace. Thus, when we must access, say, the `argList`, we must translate the workspace offset to an actual address. The address of argList, for example, would be `(char *)(StartOfWorkspace + argList)`.

A final point to consider is that it is quite common for objects to have the same actual and effective type. The casting tables in these cases are trivial, i.e., (1, 2, 3, ...). Similarly, since many types are not defined using inheritance, the offset tables will also be trivial, i.e., (0, 0, 0 ...). Such tables are *null* tables, and can be omitted from the instance. When we access methods and instance variables for objects with null tables, no correction needs to be performed. Code is generated that checks at run-time for the presence of these null tables. If the table does exist, then it is used to correct the offset, otherwise no correction is necessary. For example, if we want to access the 4th entry in a casting table, the following code will be generated:

```
((castTab == 0) ? (4) : castTab[4])
```

6 Run-Time Support

The run-time system provides an execution environment for active objects. It is composed of:

- a *persistent workspace*,
- a *thread manager* that implements light-weight processes (threads),
- an *activity controller* that provides support for Hybrid activities in terms of threads,
- a *communication manager* that handles communication with the outside world, and
- support for *system objects*.

The run-time environment is implemented as a single UNIX process that acts as a server communicating with client processes. The persistent workspace is effectively the virtual memory of this process, which is saved in a file whenever a session ends (or as frequently as required by the running applications). The workspace contains all persistent data, other than the executable code of the run-time system. Since objects' methods are linked into the run-time system when new types are compiled, the workspace may be relocated in virtual memory from session to session. For this reason pointers in the workspace are represented as offsets rather than virtual memory addresses.

Concurrent threads execute within the shared workspace, and are themselves persistent. Separate, protected address spaces for threads are, in principle, not required, since Hybrid provides no mechanisms for accessing arbitrary memory locations (as is the case with C pointers). Furthermore, implementation of threads as, for example, UNIX processes, would make context switching and interprocess communication far more expensive. A similar approach to memory protection has been taken in the Cedar system [Swinehart et al. 1986] for much the same reasons.

Rather than implementing active objects as threads that communicate using a message-passing facility, it was far more natural to model objects as passive data structures that become active when a thread enters them. The reason is that objects may schedule and switch activities using the mechanisms of delegation and delay queues. Activities are therefore more readily modeled as threads that synchronize with respect to shared, passive data. (Multiple threads will be needed to represent activities in a distributed implementation.) The thread manager therefore provides monitor-like *passive objects* for implementing the shared, persistent data of Hybrid objects.

The activity controller provides an extra layer of support to threads that implement Hybrid activities. These threads must synchronize access to shared passive objects so that the semantics of active objects is preserved. The interface provided by this component is used in the code generated by the Hybrid compiler.

The communication manager acts as an intermediary between Hybrid objects and the outside world. It uses BSD sockets and the UNIX facilities for asynchronous I/O. The communication manager uses the synchronization primitives of the thread manager to suspend and resume threads requesting I/O.

Objects interact with the run-time system either *directly* through calls generated by the compiler when certain language constructs are encountered, or *indirectly* by communicating with pre-defined *system objects* provided by the run-time system. Other system objects supported by the run-time system include delay queues and object ids.

6.1 The Thread Manager

The thread manager implements lightweight processes (i.e., *threads*) that run within a (heavyweight) UNIX process. Monitor-like passive objects are provided for thread synchronization. Once the thread manager has been initialized at the beginning of a session, everything executes as a thread, that is, not just Hybrid activities, but also parts of the system, such as the communication manager.

Threads execute in a non pre-emptive fashion, yielding control to the thread manager when they terminate, when they execute a *hold* instruction (generated automatically by the Hybrid compiler), or when they must wait to access a passive object. They are scheduled in a simple round-robin fashion.

The use of *hold* instructions is similar to the approach taken to simulate concurrency in Concurrent Euclid [Holt 1983]. An alternative approach we experimented with was to interrupt executing threads using the UNIX alarm signals, and to schedule threads in a time-sliced fashion. Threads could be assigned different time slices according to need. Critical sections were provided by blocking alarm signals for short instruction sequences. A stand-alone time-slicing thread manager was implemented, but was not used for the Hybrid prototype because other parts of the system had already been developed without pre-emptive scheduling in mind.

Threads execute on the UNIX process stack. When a thread is suspended, its stack is copied into the persistent workspace, and the stack of the next thread to be scheduled is copied back to the process stack. Although this approach seems highly inefficient, the alternative would be to allocate stacks for threads directly in the workspace. The difficulty would be in ensuring that stacks do not overflow and corrupt the workspace. The overhead of copying stacks was preferred to that of repeatedly testing for stack overflows (at least for the purposes of a prototype implementation).

The thread manager provides a C type, called `monitor`, for synchronizing threads. It can be used to implement monitor-like behaviour for shared, passive objects. The following set of operations is used to simulate monitor behaviour:

- An `enter` operation used for gaining access to a `monitor` (i.e., in order to perform an operation on the shared data it protects).
- A `leave` operation to give up control of the `monitor`.
- A `wait` operation that implements waiting on a monitor's condition variable. This operation takes as an argument a queue that is used for suspending the thread.
- A `signal` operation, taking as argument a queue of threads, implements a signal on a monitor's condition variable. If the queue is not empty, the first thread in the queue is awakened, and the invoking thread is suspended.

Monitor-like behaviour for a C type is accomplished by including a `monitor` variable, and restricting access to functions that perform `enter` and `leave` on the *monitor* before and after accessing the type's variables. The functionality of condition variables for monitors is provided by applying `signal` and `wait` to variables of type `queue` in between the `enter` and `leave` calls. The C types `domain` and `port` described later have been implemented using this facility. In the rest of the section we shall refer to such objects simply as monitors.

6.2 The Activity Controller

The activity controller uses the threads and passive objects provided by the thread manager to implement the behaviour of Hybrid activities and domains. As we have outlined earlier, the prototype implements active objects by modeling them as passive objects shared by active threads (rather than as lightweight processes passing messages). The semantics of message-passing in Hybrid, which is based on remote procedure calls, led naturally to this approach. The activity controller provides the extra layer of control needed for threads to behave like activities. This layer is invoked through instructions generated by the Hybrid compiler implementing concurrent language constructs, and through pre-defined basic object types, such as delay queues and object ids, which are concerned with communication between concurrent objects.

6.2.1 Support for Activities and Domains

Hybrid domains are implemented as shared passive objects. Recall that a domain becomes *active* when it accepts a message. In the implementation, domains become active when they are accessed by activities. Synchronization is achieved by requiring activities to acquire a *domain lock* before accessing the domain.

All activities are assigned a unique *activity id* when they are created. This identifier is used for synchronization purposes. It also points to a data structure that contains information concerning the current status of the activity.

An activity may acquire a domain lock by calling an operation of a domain monitor (see 6.1) associated with the domain.

The domain monitor maintains the following information for activity synchronization:

- a *status*, indicating whether the domain lock has been acquired by some activity,
- an *activity id*, that identifies the activity currently holding the domain lock,
- a *lock counter* used for nested locking.,
- a *condition variable* used for suspending activities unable to acquire the domain lock when it is already held by another activity, and
- a *circular linked list* of condition variables, including those that represent open delay queues, and the one for activities waiting to acquire the domain lock.

When an activity wishes to enter a `domain`, it performs an operation to acquire the domain lock, providing its activity id. If the lock is available, it is granted. If it is held by another activity, the activity is suspended on a condition variable, i.e., it is added to a queue. If the lock is already held by the activity, the lock counter is incremented and the activity may proceed.

When an activity leaves a domain, it performs another operation that decrements the lock counter, and releases the domain lock when the counter falls to zero. When the domain lock can be released, the next non-empty condition variable in the circular list is signaled. If all conditions are empty the domain status becomes *idle*, and the lock becomes available to new requests.

6.2.2 Delay Queues

Delay queues are predefined system objects. Recall that they are used by objects that need to control when certain operations may be invoked – messages are accepted when the delay queue is *open*, and delayed when it is *closed*.

The information maintained by delay queue objects includes their domain, their status, and a unique identifier associating the delay queue with a condition variable in the corresponding domain monitor. Each domain monitor of a domain with delay queues maintains a circular list of condition variables representing the open queues. The domain monitor cycles through its list to avoid simple forms of starvation that could occur if certain delay queues were preferred.

Open and *close* operations on delay queues are implemented by inserting or removing the associated condition variable from the domain monitor's circular list. Removing the condition from this list means that the activities waiting on the condition will not be awakened when the domain becomes idle.

Since association of operations with delay queues is statically determined, the compiler generates special code for invoking these operations. This code first checks whether the delay queue is open or closed. If it is closed, the activity must decrement its lock counter and wait on the associated condition variable. (Deadlock is possible if the object has been poorly programmed.) If the queue is open, it checks whether there are other activities waiting on the associated condition. If there are it signals the condition and then waits on it.

6.2.3 Object Ids

Communication between objects in separate domains is performed using object identifiers, provided by the system type `oid`. As a consequence, all Hybrid operation invocations can be directly translated into C function calls, with remote communication between objects being handled directly in the implementation of object ids. When an operation is invoked using an object id, it checks whether the domain of the referenced object actually differs from the current one, and, if so, attempts to acquire the necessary domain lock. The oid contains the information needed to determine the referenced object's location and its domain.

Oids will play an important role in a distributed implementation of the Hybrid system. They will provide the mechanism for objects in different environments to communicate with one another. In this case oids will identify not only objects and their domains, but also an indication of the object's environment. Communication with objects in remote environments will require support from the communication manager for reliably delivering messages to the remote object manager.

6.2.4 Reflexes and Delegation

The Hybrid compiler translates ordinary operation invocations directly to C function calls. This is not the case with reflexes and delegated calls.

Recall that reflexes are special operations for starting a new activity. Whenever a reflex is invoked the compiler generates a call that allocates and initializes a structure representing the new activity and creates a thread for that activity. When the new thread starts it is bound to a function implementing the reflex on the target object.

If the target is a local object, then the new thread must first lock the domain in which it is to execute. If the target is remote, the operation is executed using an oid, which will take care of locking the domain. At the end of the operation, the domain is unlocked, the activity structure is freed and the thread terminates, returning control to the thread manager.

For delegated calls the compiler generates code that calls a domain operation that releases the domain lock and saves the value of the lock counter in the stack of the running thread. Then the thread calls the operation on the target object of the delegated call. When the operation returns, it first must re-lock the old domain (recall that delegation is intended to permit interleaving of activities within a domain), and resets the lock counter to the saved value.

See section 7 for an example using delegated calls.

6.3 Communication Manager

The communication manager handles all communication with client processes. Communication is asynchronous, so as not to interfere with the thread manager (the object manager must not block on I/O while there are runnable threads).

Threads generally interact with the communication manager through the medium of system-defined *port* objects that are implemented as monitors (see section 6.1). Ports hide the details of manipulating UNIX file descriptors, and provide the means for synchronizing threads that communicate with clients. The following information is associated with a port:

- the port's status, which may be *unused*, *active* or *suspended*,
- a file descriptor used for communication,
- a condition variable for suspending threads requesting unavailable input,
- a condition variable for suspending threads requesting service when the port itself is suspended, and
- a queue for incoming messages.

An *active* port is associated with a file descriptor actually being used to communicate with a client. Requests by threads to send a message through the port are handled immediately by writing the message using the file descriptor.

A *suspended* port is available to running threads, but no longer has an associated file descriptor. A thread requesting communication through that port will be suspended. The port may become *active* again when a client requests that connection to that port be resumed. Threads suspended on output may then be signaled.

When a message arrives on the file descriptor associated with the port, the message is inserted in the port's message queue and the condition of threads suspended for input is signaled.

Communication from the clients is handled by a special *input thread* created when the session starts. Whenever input is available, the object manager receives asynchronous notification via a UNIX SIGIO signal. The signal handler passes control to the input thread either directly, or by informing the thread manager that it needs to be scheduled. The input thread then determines

which file descriptors need to be read (using the UNIX *select* system call), and passes the input to the appropriate port objects.

Requests from clients wishing to connect to a port are handled using the BSD *socket* mechanism, which enables a process to receive messages asynchronously at a known address. The request may be either for a new port, in which case an *unused* port is allocated, or for a *suspended* port to be resumed.

7 A Hybrid Example

The capabilities and restrictions of the prototype Hybrid implementation can be best illustrated by an example. A simple example utilizing oids and delegated calls, is an *administator*. An administrator is an object that manages a set of *workers*. *Clients* (not shown) send requests to the administrator to have jobs performed, and the administrator distributes them to the workers. Since the administrator and the workers are implemented as independent Hybrid domains, they may execute concurrently. The administrator distributes jobs using delegation in order to be able to switch its attention to new requests without waiting for the workers to finish their jobs.

In our example the *worker* is very simple. It has just two methods: one for initialization and one for processing.

```
#
# A worker
#
type worker : abstract {
    init : (integer) -> ;
    doit : (integer, string) -> string ;
} ;
private {
    var workerId  : integer ;

    init : (id : integer)  -> ;
    {
        workerId := id ;
        ...
        return ;
    }

    doit : (fromClient : integer ; message : string) -> string ;
    {
        var returnMessage : string ;
        # Process the data
        return (returnMessage) ;
    }
}
```

The administrator also has two methods: one for initilization and one through which the clients make requests. A private method is used for selecting a *worker* to pass the request.

```
#
# The administrator
#
type administrator : abstract {
    init : -> ;
    work : (integer, string) -> string ;
} ;
private {
    var worker : workersArray ;
    var jobs : workersJobs ;

    init : -> ; # initialize the workers
    {
        var i : integer ;

        i := worker.lower ;
        loop {
            # create a worker as a new domain:
            worker[i] <- export(worker) ;
    # call via an oid:
            @(worker[i]).init(i) ;
            ++i ;
            if (i >? worker.upper) {
                break ;
            }
        }
        return ;
    }

    selectWorker : -> integer ; # select a worker to pass the work to.
    {
        #
        # A trivial selection algorithm: the one with the least jobs!
        #
        var i : integer ;
        var workerId : integer ;

        # select the least loaded one
        i := jobs.lower ;
        workerId := i ;
        loop {
            if (jobs[i] <? jobs[workerId]) {
                workerId := i ;
            }
            ++i ;
            if (i >? jobs.upper) {
                break ;
            }
```

```
        }
        ++jobs[workerId] ; # One more job for the worker
        return (workerId) ;
    }

    work : (fromClient : integer ; message : string) -> string ;
    {
        var workerId : integer ;
        var returnMesage : string ;

        workerId := selectWorker () ;
        returnMessage := delegate(
            @(worker[workerId]).doit(fromClient,message)
        ) ;
        --jobs[workerId] ;
        return (returnMessage) ;
    }

}
```

The administrator has two (private) instance variables. One that holds the oids of the workers, and one that holds the queue of jobs for each worker (used by the selection method). When the `init` method is called, the workers are created and initialized. The `export` call will create each worker as a separate domain, returning the oid of the new worker. The worker is initialized via an oid call (i.e., using the `@` symbol).

When a client makes a request to the *administrator*, it calls the `work` operation via the oid of the *administrator*. The *administrator* will select a worker (in the example the one with the smallest queue) and delegate the work to it. By delegating the call, the administrator may switch its attention to other clients' requests while the worker handles the job. That is, the administrator's domain does not block during call, as would be the case if the call were not delegated. When the job is finished and the administrator is ready to accept the worker's response, the activity is resumed within the administrator, and it delivers the finished job back to the client.

8 A Note on Design and Implementation

The ideas behind Hybrid developed over several years, beginning with the development of another experimental object-oriented language called *Oz* [Nierstrasz et al. 1983; Nierstrasz 1985]. Oz was also based on the idea of active objects, but the interface to an object was a set of *triggers* rather than operations. In Hybrid, triggers can be implemented using the more primitive mechanisms of delegation and delay queues.

An initial language design was completed by the beginning of 1987 [Nierstrasz 1987a]. A skeleton parser (i.e., recognizer and pretty-printer) for Hybrid was implemented by Oscar Nierstrasz, using the yacc and lex compiler-writing tools, as well the routines for managing the persistent workspace. Around this time, a co-routine package for C was implemented by Michalis Papathomas. The co-routines package was later used to provide concurrent threads for the object manager.

Implementation began in earnest in the spring of 1987, beginning with the implementation of the run-time support for active objects, and a parser and type manager for use by the compiler. At this point the plan was to implement object methods by translating them to an intermediate form that would be interpreted at run-time. It appeared at the time that this was the most flexible approach for supporting run-time addition of new object types. An abstract machine language was partially implemented.

A major turning point for the project took place in the fall of 1987, when we decided to abandon the abstract machine approach and translate Hybrid methods to C code. An abstract machine may be useful for enhancing the portability of a stable language, but it significantly complicates the prototype implementation of an experimemtal language. By choosing to translate to C, we avoided the problem of designing, implementing and debugging an abstract machine before getting to the real problem of building the Hybrid prototype. The C compiler together with the run-time system would provide us with a reliable abstract machine.

The most intensive implementation effort occurred during the final six months. The Hybrid compiler and the type manager were implemented by Dimitri Konstantas. The run-time system was implemented by Michalis Papathomas. The total implementation effort comprised roughly two man-years over the period from March 1987 to May 1988.

The source code lines of the major components of the Hybrid prototype are of the following sizes:

Compiler	18,102 lines
Type Manager	10,016 lines
Thread Manager	5,497 lines
Basic User Interface	5,426 lines

In addition, there were the following smaller components:

Run-time Type Manager Interface	1,882 lines
Persistent Workspace Module	1,969 lines
User Interface	1,806 lines
Test Programs	229 lines

The total size of the source code is 44,927 lines of C code.

Although the performance of the Hybrid examples implemented using the prototype appeared acceptable, we have not yet used Hybrid to implement realistic applications, nor do we have any measure yet of the performance as compared with other object-oriented or concurrent systems.

On the other hand, the performance of the compiler can be easily measured. The Hybrid compiler translates Hybrid source code to C at a rate of about 100 lines per second on a Sun 3/50 workstation. The C compiler is quite slow at compiling the translated code: around 200 seconds for 800 C code lines (or 100 hybrid source lines). The reason for this is that accessing of methods and instance variables in the generated code requires a large number of indirections through various lookup tables. For example, a method call expands to a 20 line C statement (more that 1500 characters) containing 23(!) *conditional expressions*.

9 Future Work

Since Hybrid was conceived as an experimental language and system rather than as a product, there are quite a few directions in which this work may continue.

First of all, the prototype needs to be extended to handle distributed object environments. In principle this should not be too difficult, since remote object managers can be viewed as special kinds of client processes. One difficulty, however, with a correct implementation would be the need for guaranteeing global consistency. Cooperating object managers need to provide the illusion of a global, persistent workspace. If an object manager's machine crashes, simply backing up to the last stable version of the local workspace may cause messages to be lost, and therefore leave remote objects blocked and waiting for a lost activity.

The current implementation of the compiler implements only a subset of Hybrid, and ignores a number of semantic issues. In-line type definitions, for example, are not supported. The type manager should be extented to allow *local* type definitions. This way we will be able to have in-line type definitions and also local type databases for each user. Each "user" (person or object) will be able to define a local type database as an extension of the global one. New types can then be first tested as local types, and then added to the global type database. The next step will be to have a distributed type manager. Possible approaches are to either have a centralized type manager serving the network, or distinct type managers for each environment.

Dynamic linking of libraries is a feature that once incorporated will better support dynamic inheritance and local type databases.

The language itself needs to be re-designed and extended. Exception handling is clearly essential, and must be integrated with the type model. Variant types and the run-time type-checking construct (`check`) should be replaced by a mechanism supporting dynamic inheritance. (Dynamic inheritance would provide a means for expanding the interface to an object without violating encapsulation.)

We need to use the prototype to build concurrent and distributed applications to see how well Hybrid addresses these areas. The implementation effort also provides us with a very clear indication of the language constructs that are costly, and those whose semantics were poorly conceived. In particular, Hybrid's approach to dynamic binding makes it almost impossible for the compiler to generate efficient code for finding and executing methods. A choice between static and dynamic binding must be offered, since the proportion of cases where the latter is really needed is quite limited. This experience will contribute to the re-design, with the goal of achieving a more simple and expressive language without requiring the programmer to sacrifice efficiency.

There are several other worthwhile directions to pursue. On the practical side, it is clear that without a suitable programming environment, object-oriented languages offer only marginal advantages over traditional programming languages. Object design tools, large, reusable software bases, intelligent browser, and prototyping, debugging and monitoring tools are all needed to help object-oriented languages deliver the promise of faster, more reliable, open application development.

Finally, on the more theoretical side, we need computational models to help us describe and understand the semantics of concurrent and object-oriented languages.

10 Conclusions

The implementation of Hybrid finally required a great deal more effort than was originally estimated (about four times the expected effort and time). The reasons for this were mainly: (1) Hybrid was a new language rather than an extension of an existing, implemented language, (2) the semantics of Hybrid was not formally defined, and contained many inconsistencies, (3) the run-time functionality

(concurrency, persistence, etc.) had to be build from scratch, (4) some time and effort was wasted because the original implementation plan proved unrealistic, and had to be changed in mid-stream.

The overwhelming lesson from the point of view of language design was that exception-handling must be a fundamental part of the language if we are to make sense of strong-typing in a distributed environment of active objects. Time and again, when semantic difficulties were encountered, it was apparent that exceptions were needed to notify objects when unexpected events occurred during execution, and that the exceptions that could be raised must be part of an object's interface.

Some of our problems would have gone away if more time had been spent on the language design, but there were many issues that would not have been uncovered without an early prototype. The feedback from the implementation effort will be invaluable for ironing out language design problems, and will also provide a forum for testing whether our ideas were valid or not. One of the surprises was that the reusability mechanisms, the strong-typing, and the concurrency constructs were not as orthogonal as was originally thought. Since their semantics were defined independently, the interference was not discovered until the implementation had started.

Another surprise was that the natural way to implement Hybrid's message passing active objects (within a single UNIX process), was to model them as passive resources shared by lightweight processes. This is less surprising if we consider that message-passing in Hybrid is intentionally patterned after remote procedure calls, so that threads of control can be easily mapped to lightweight processes. It will be interesting to see what happens to this mapping if we consider implementing Hybrid on a machine with a highly parallel architecture ...

11 Appendix A: Language

11.1 Names in Hybrid

Names of types, variables, and operations in Hybrid are composed of letters and digits; the first character must be a letter. Upper and lower case letters are distinguished. The names can be of arbitrary length. Keywords like `type, of, array, variant` etc. are reserved.

Numbers are decimal integers with or without a minus (-) sign.

Strings are enclosed between double quotes and can contain any printable ascii character. Double quotes within the string should be escaped with a backslash (i.e., `\"`).

11.2 Operators

Operator names in Hybrid are made up from the following set of characters

```
* / % + - ^ | < = > \ $ & ~ ?
```

plus the assign operator `:=` . They can be of any length.

Operators are recognized as *assign* operators when their last character is an equal sign (`=`) .

Boolean relation operators have `?` as the final character.

The single character operators `* / %` are recognized as *priority* operators.

Operations in Hybrid are parsed from left to right, with the priority operations parsed first.

11.3 Grammar

typeDef :: **type** typeName
indexTypeParamNames$_{OPT}$ typeParamNames$_{OPT}$:
typeSpecOrConstr ;
privatePart$_{OPT}$

typeName :: *identifier*

indexTypeParamNames :: [paramType$_{LIST}$]

typeParamNames :: **of** paramTypeArgs

paramType :: typeName constraint$_{OPT}$

constraint :: :< typeName

paramTypeArgs :: paramType
| (paramType$_{LIST}$)

typeSpecOrConstr :: typeConstr
| typeSpec

typeConstr :: **enum** { constName$_{LIST}$ }
| **record** { varPubDec ; varPubDec$_{SEQ}$ }
| **abstract** { varPubDec$_{SEQ}$ operationStub* }

typeRef :: typeName indexTypeParamsBind$_{OPT}$ typeParamsBind$_{OPT}$

typeSpec :: typeRef
| rangeEnumVal .. rangeEnumVal
| **oid of** typeName
| **array** indexTypeParam typeParam
| **variant** { typeName variantCase+ }
| **inherits** { typeName parentType+ }

indexTypeParamsBind :: [argType$_{LIST}$]

indexTypeParam :: [typeSpec]

typeParam :: **of** typeRef

typeParamsBind :: **of** argTypes

argTypes :: argType

| (argType$_{LIST}$)

argType :: *identifier*

varPubDec :: **var** varPub$_{LIST}$: typeName

varPub :: *identifier*

constName :: *identifier*

rangeEnumVal :: *integerVal*

variantCase :: **or** typeName

parentType :: **and** typeName

operationStub :: operationTypeDec : argTypes$_{OPT}$ returnDec$_{OPT}$;
delayDec$_{OPT}$

returnDec :: → argTypes$_{OPT}$
| → **var** argType

delayDec :: **uses** typeName ;

operationTypeDec :: **mutable** operationDec
| operationDec

operationDec :: **prefix** prefix
| **infix** infix
| operationNameDec
| **reflex** operationNameDec

operationNameDec :: *identifier*

prefix :: *priorityOp*
| *operator*

infix :: *priorityOp*
| *relationalOp*
| *operator*
| *assignmentOp*

privatePart :: **private** { privDec$_{SEQ}$ operation* }
| **private** ‘{ cCode ‘}

cCode :: *Ctoken*
| cCode *Ctoken*

privDec :: insVarDec
    | valuesDec

insVarDec :: **var** insVar$_{LIST}$: typeName indexTypeParamsBind$_{OPT}$
        typeParamsBind$_{OPT}$

insVar :: varName init$_{OPT}$

dec :: varDec
    | valuesDec

varDec :: **var** var$_{LIST}$: typeName indexTypeParams$_{OPT}$ typeParams$_{OPT}$

var :: varName init$_{OPT}$

init :: ← constExpr

varName :: *identifier*

constExpr :: element

indexTypeParams :: [typeName$_{LIST}$]

typeParams :: **of** typeArgs

typeArgs :: typeName
    | (typeName$_{LIST}$)

valuesDec :: **values** valuesName$_{LIST}$

operation :: operationSpec compoundStmt

operationSpec :: operationType : args$_{OPT}$ returnDec$_{OPT}$; delayDef$_{OPT}$

operationType :: **mutable** operationDef
    | operationDef

operationDef :: **prefix** prefix
    | **infix** infix
    | operationNameDec
    | **reflex** operationNameDec

args :: ( )
    | (argDec argDec$_{SEQ}$)

argDec :: varName$_{LIST}$: typeName

delayDef :: **uses** varName ;

stmt :: ;
| ‘{ cCode ‘}
| expr ;
| compoundStmt
| **check** (varName :? typeName) compoundStmt elsePart$_{OPT}$
| **if** (expr) compoundStmt elsePart$_{OPT}$
| **block** labelPart$_{OPT}$ { dec$_{SEQ}$ stmt* }
| **loop** labelPart$_{OPT}$ { dec$_{SEQ}$ stmt* }
| **break** labelPart$_{OPT}$;
| **continue** labelPart$_{OPT}$;
| **switch** element { casePart+ defaultPart$_{OPT}$ }
| **return** element$_{OPT}$;
| **end** ;

compoundStmt :: { dec$_{SEQ}$ stmt* }

elsePart :: **else** compoundStmt

labelPart :: : *identifier*

casePart :: **case** enumCase compoundStmt

defaultPart :: **default** compoundStmt

enumCase :: enumVal
| enumVal .. enumVal

enumVal :: *integerVal*
| constName

expr :: term$_{LIST}$
| *‘Cexpr‘*

term :: binary
| binary ← term
| binary *assignmentOp* term

binary :: priority
| binary *operator* priority

priority :: unary
| priority *priorityOp* unary
| priority *relationalOp* unary

unary :: primary typeCast$_{OPT}$
| *priorityOp* unary
| *operator* unary

typeCast :: : typeName

primary :: element
| primary [expr]
| procedureCall
| operationCall (expr_{OPT})
| operationCallReflex (expr_{OPT})
| instanceVar
| **export** (typeName)
| delegateOp

delegateOp :: **delegate** (binary *assignmentOp* term)
| **delegate** (binary *operator* priority)
| **delegate** (operationCall (expr_{OPT}))

procedureCall :: operationName (expr_{OPT})

operationCall :: primary . operationName

operationCallReflex :: primary ! operationName

instanceVar :: primary . instanceVarName

element :: const
| constOrVarNameOrTypeName
| oidCall constOrVarNameOrTypeName
| (expr)
| oidCall (expr)

oidCall :: @

const :: *integerVal*
| *stringVal*

constOrVarNameOrTypeName :: *identifier*

instanceVarName :: *identifier*

operationName :: *identifier*

12 Appendix B: Bounded Buffer Example

A simple Hybrid example utilizing *delay queues* is a *bounded buffer*.

In our example the bounded buffer has three methods: one for initializing, one for writing data and one for reading data. The last two methods use delay queues for synchronization.

```
#
# The buffer
#

type boundedBuffer : abstract {
    put : string -> ; uses delay ;
    get : -> string ; uses delay ;
    init : -> ;
} ;

private {
    var putDelay, getDelay : delay ;    # the delay queues
    var putIdx , getIdx : integer ;     # position in the buffer.
    var buffer : strArray ;             # the buffer area

    init : -> ;
    {
        putDelay.open() ;           # allow writing of data
        getDelay.close() ;          # nothing to read
        putIdx := buffer.lower() ;
        getIdx := buffer.lower() ;
    }

    get : -> string ; uses getDelay ;
    {
        var nextPoss : integer ;
        var newString : string ;

        # wrap around the bounds of the buffer
        nextPoss := getIdx + 1 ;
        if (nextPoss >? buffer.upper() ) {
            nextPoss := buffer.lower() ;
        }

        newString := buffer[getIdx] ;    # read the data

        getIdx := nextPoss ;
        if (getIdx =? putIdx ) {   # close the reading queue if
            getDelay.close() ;     # all data have been read
        }
        putDelay.open() ;          # open the queue for writing
        return(newString) ;
    }

    put : (newString : string ) -> ; uses putDelay ;
```

```
    {
        var nextPoss : integer ;

        # wrap around the bounds of the buffer
        nextPoss := putIdx + 1 ;
        if (nextPoss >? buffer.upper() ) {
            nextPoss := buffer.lower()  ;
        }

        buffer[putIdx] := newString ;
        getDelay.open() ;           # open the reading queue

        putIdx := nextPoss ;
        if (putIdx =? getIdx ) {    # close the writing queue if
            putDelay.close() ;      # the buffer is full
        }
        return ;
    }

}
```

The buffer is used by a *producer* and a *consumer*, who will respectively write and read information. Examples of a producer and a consumer follows:

```
#
# The producer
#

type producer : abstract {
    init : (oidBB)  -> ;
    reflex put : -> ;
};

private {
    var oidBuffer : oidBB ;

    init : (bufOid : oidBB)  -> ;        # set the buffer's oid
    {
        oidBuffer <- bufOid ;
        return  ;
    }
    reflex put : -> ;
    {
        var line : string ;

        ....
        @oidBuffer.put(line) ;        # store the data in the buffer
```

```
        ....

        return ;
    }
}


#
# The consumer.
#
type consumer :
abstract {
    init : (oidBB)  -> ;
    reflex get : -> ;
};

private {
    var oidBuffer : oidBB ;

    init :  (bufOid : oidBB)  -> ;        # set the buffer oid
    {
        oidBuffer <- bufOid ;
        return  ;
    }
    reflex get : -> ;
    {
        var lineOut : string ;

        ....
        lineOut := @oidBuffer.get() ;    # read data from the buffer
        ....
        return ;
    }
}
```

The bounded buffer has two delay queues: one for writing and one for reading. Upon initialization, the *write* queue is opened (i.e., the clients can write), and the *read* queue is closed (i.e., there is nothing to be read).

When a client has data to write into the buffer, it will call the *put* method via an oid call. If the *write* queue is closed, the call will block until the queue is opened. If the queue is open, then method will be executed. The data will be stored and the *read* queue will be opened. Then if this last write fills the buffer, the *write* queue will be closed, thus blocking all write requests until empty slots are again available. Similarly, read requests will be blocked if no data are available, will open the *write* queue after reading the data, and will close the it read queue if the buffer is empty.

References

[Hoare 1974] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", CACM, vol. 17, no. 10, pp. 549−557, Oct 1974.

[Holt 1983] R.C. Holt, *Concurrent Euclid, the UNIX system, and TUNIS*, Addison-Wesley, 1983.

[Johnson 1975] S.C. Johnson, "Yacc: Yet Another Compiler Compiler", Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[Lesk and Schmidt 1975] M.E. Lesk and E. Schmidt, "Lex − A Lexical Analyzer Generator", Computer Science Technical Report #39, Bell Laboratories, Murray Hill, NJ, 1975.

[Nierstrasz, et al. 1983] O.M. Nierstrasz, J. Mooney and K.J. Twaites, "Using Objects to Implement Office Procedures", Proceedings of the Canadian Information Processing Society Conference, pp. 65−73, Ottawa, May 1983.

[Nierstrasz 1985] O.M. Nierstrasz, "An Object−Oriented System", in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 167−190, Springer Verlag, Heidelberg, 1985.

[Nierstrasz 1987a] O.M. Nierstrasz, "Hybrid −− A Language for Programming with Active Objects", in *Objects and Things*, ed. D.C. Tsichritzis, pp. 15−42, Centre Universitaire d'Informatique, University of Geneva, March 1987.

[Nierstrasz 1987b] O.M. Nierstrasz, "Triggering Active Objects", in *Objects and Things*, ed. D.C. Tsichritzis, pp. 43−78, Centre Universitaire d'Informatique, University of Geneva, March 1987.

[Nierstrasz 1987c] O.M. Nierstrasz, "Active Objects in Hybrid", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243−253, Dec 1987.

[Swinehart, et al. 1986] D. Swinehart, P. Zwellweger and R. Beach, "A Structural View of the Cedar Programming Environment", ACM TOPLAS, vol. 8, no. 4, pp. 419−490, Oct 1986.