Hybrid – A Language for Programming with Active Objects[1]

O.M. Nierstrasz

Abstract

Hybrid is an object-oriented programming language in which objects are the active entities. Active objects in Hybrid are both concurrent and persistent, thus unifying the notion of an "object" with that of processes and files. Hybrid introduces the concept of an *activity* as a means for controlling the interactions between active objects. The language provides constructs that allow one to restrict or relax this control in a fairly simple way. In particular, mechanisms for delaying and "delegating" activities are provided. Furthermore, Hybrid is designed so as to provide powerful constructs for reusing code in a way that is consistent with strong-typing.

1 Introduction.

Previous work in object-oriented programming has emphasized mechanisms for building new object types and re-using existing code, such as inheritance, overloading, and parameterization. To a lesser extent, strong-typing has been considered, and issues of concurrency and distribution have been virtually ignored. In Hybrid, we attempt to provide a fully object-oriented language that is strongly-typed, and extends the notion of object-orientation to "active objects". Active objects are persistent, and when they are independent they may execute concurrently. As such, they unify the concept of an object with those of processes and files.

The motivation for our work in designing Hybrid is to provide what we believe are better paradigms for software which will make it easier to produce correctly-functioning applications more quickly. It particular, we address three fundamental issues:

1. Reusability: code that can be easily tailored or reused reduces the effort needed to develop software.
2. Reliability: strong-typing and clearly defined boundaries and interfaces between modules helps in the development of reliable and maintainable software.
3. Concurrency and distribution: development of concurrent and distributed software suffers from a lack of paradigms and programming constructs that are both powerful and flexible.

We believe that the object-oriented approach is appropriate for addressing all three of these issues in a clean and consistent manner.

This is in marked contrast to traditional object-oriented languages and systems, such as Smalltalk [Gold83] and Lisp with flavors [Wein81 Moon86], which address neither strong-typing nor concurrency.

Hybrid starts with the object-oriented approach as it is manifested in various systems existing today, and it adapts it to a model in which objects are both strongly-typed and highly concurrent. Objects are structured, and may contain other objects. A small set of constructors are available for structuring objects and defining object "types".

A unit of concurrency is called a *domain* and is comparable to a process. Each domain contains a single top-level "root" object and any number of sub-objects. Independent objects (in different domains) may execute concurrently. An *activity* is defined as a thread of control, which may pass between domains when independent objects communicate. Activities may be scheduled by objects through the use of *delay queues*.

[1]In *Objects and Things*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, March 1987, pp. 15-42.

Objects may switch between several activities by *delegating* calls to other objects. The notion of *subactivities* is introduced to more easily manage sets of related activities. Longer term mutual exclusion and atomicity are provided by a transaction mechanism.

Hybrid is an experimental language which has already undergone several major changes [Nier85a Nier85b]. This document describes its current incarnation with respect to a prototype system presently being implemented. We expect future versions of the language to be consistent with the one described here.

In the next section we shall provide an overview of the Hybrid model before continuing with details of the programming language.

2 Overview.

The goal in the design of Hybrid is to provide a programming language based on a small number of concepts that support reusable code, reliability through well-defined interfaces, and concurrency and distribution. To this end we have produced a model that starts with what we see as the fundamental concepts in the object-oriented approach, and adds strong-typing, persistence and active objects. In this section we shall give an overview of our model before we continue with details of the language.

There is a wide range of languages and systems each of which claims to some extent or another to be "object-oriented". Rather than propose our own definition, we suggest that an examination of these systems shows that those that can be called "fully object-oriented" must exhibit the following properties [Nier86 Nyga86 Stef85 Stoy84]:

1. *Data abstraction*: objects are characterized by their behaviour (interface), not their implementation (realization).
2. *Instantiation*: there may be many object instances of the same "type" (i.e., with the same interface).
3. *Independence*: an object's state is not directly accessible to other objects; one interacts with an object by "passing it a message", and the object itself decides how to respond.
4. *Inheritance*: object types are related to other types, with subtypes "inheriting" the behaviour of their parent types.
5. *Homogeneity*: everything is an object, in particular, types are objects, and new types may be added dynamically.

Reusability of code is thus acheived primarily through the following four mechanisms:

1. *Structuring*: Object types can be re-used when declaring instance variables of new object types.
2. *Parameterization*: "Generic" object types (like arrays) may take other types as parameters.
3. *Inheritance*: A new object type may inherit the representation and realization of one or more parent types.
4. *Tailoring*: Applications that use objects of an existing type continue to be valid for objects of a new equivalent type (i.e., with the same interface but possibly different realization).

We believe that this approach is sound, but that it does not go far enough. We extend the approach in Hybrid by taking strong-typing to be fundamental, and by considering independent objects as concurrent, active entities. Furthermore, objects in Hybrid are persistent, thus eliminating the need for a notion of "files", and simplifying some of the concurrency control issues.

An *object* in Hybrid is an active entity that supports a collection of operations, called its *interface*. Objects are persistent, and so may change state when they perform an operation. The *type* of an object is a specification of the object's interface and of the set of values that the object may assume as side-effects of operations. A *subtype* is a refinement of a parent type such that all instances (i.e., values) of the subtype are also instances of the parent type. This refinement may be accomplished by either restricting the set of values that objects may assume, or by expanding the operations visible in their interface.

A "program" in Hybrid is a type definition, which consists of a new type name with optional parameters, a type specification which determines the type's interface, and an optional *private* part, which contains the type's realization. A type specification may make use of any number of existing types, such as integers, strings, etc., and the type constructors provided by the language, such as array, record, etc. If a type specification is *complete*, that is, the specification itself determines a realization, then the type definition requires no private part. For example, one could define a new type *age* as being equivalent to *integer*.

Alternatively, the constructor *abstract* allows one to define a new type in terms of an interface alone, without giving a realization. To create instances of abstract types, or of types constructed from abstract types, one must complete the type definition by giving a realization for the type's operations. In the Smalltalk literature, one would speak of providing the "methods" for operations in a realization. An abstract type is comparable to the notion of an object "class". It is just one of the type constructors available in Hybrid.

Strong-typing in this context means that objects are typed, and may only assume values of their type, or of a subtype. The ability to add types to a running system, however, means that one must also be able to determine the type of an object at run-time. Some mechanisms for run-time type-checking are therefore provided in Hybrid (others may be added in future).

Object instances are structured, and so may *directly contain* other objects in their variables. We call these the "children" of a "parent" object. An object also (indirectly) contains "sub-objects", which are its children and (recursively) the sub-objects of its children.

A *root* object is one that has no parent. A root object together with all its sub-objects is called a *domain*. A domain is comparable to a process in that at most one of its objects may be active at any time, though control is transferred in a straightforward way. We say that two objects that share the same root are *dependent*, since they are in the same domain and can therefore not execute concurrently. *Independent* objects are in different domains and may potentially execute in parallel.

Domains are necessary in order to have a well-defined granularity of concurrency. The programmer, of course, is free to decide where the boundaries between domains will occur.

An object "environment" is a collection of related domains under the control of a single "object manager". The object managers support transparent communication between objects in different environments. They may also support the ability for objects to move between environments. An environment can be thought of as a meta-domain with the object manager as its root object.

The paradigm for all communication between objects is the remote procedure call. This may be accomplished by message-passing, yet this need not be the case in all implementations. An object may manipulate a dependent object using "real" procedure calls, for example. Message-passing is therefore not visible in the Hybrid object model, and no explicit primitives for sending or receiving messages are provided in the language. We believe that the higher-level constructs provided in our model are easier to program with, and entail no loss of generality. By way of comparison, there is no "goto" amongst the control-flow statements in the language.

An *activity* is a thread of control determined by *call* and *return* messages. An activity may be thought of as a token being passed between objects, since it is characterized by executing in only one object at a time. An activity moves from one domain to another when an object in the one invokes an operation of an object in the other.

The notion of an activity suggests the following definitions: An object is *idle* if it not taking part in any activity, otherwise it is *busy*. A busy object may be *active* or "executing", or it may be *blocked* for the result of the invocation of an operation of some other object. If one object in a domain is busy, then the entire

domain is busy. Consequently an object may be idle only if all its dependent objects are also idle.

When a domain is blocked, it can only accept requests related to the activity it is blocked on. Normally this would be the awaited *return* message, but it could also be a recursive *call*. Requests from other activities continue to be delayed until the domain is idle again. This provides a default level of mutual exclusion (effectively synchronous message-passing with recursion allowed) to prevent objects from being seen in an inconsistent state while a call to an operation is pending.

A new thread of control can be introduced by creating a new activity. The thread of control originates in a "top-operation" called a *reflex*. Reflexes, unlike operations, do not return to their caller, and the object that invokes one consequently does not have to wait.

The activity mechanism we have outlined can be either strengthened or relaxed, depending on the degree of mutual exclusion or concurrency desired. Hybrid provides a transaction mechanism which causes a set of statements within a certain scope to be considered as *atomic*. This means that operations invoked on objects within that scope cause those objects to continue to wait in a *ready* state until the transaction either commits or aborts. If the transaction aborts, the object must return to its former state. Until the transaction completes, other activities attempting to communicate with the ready objects are delayed. Transactions may also be nested, as they are in Argus [Lisk83a].

Concurrent subactivities can be created within the *coblock* or *coloop* constructs using the *activity* statement. Switching between concurrent activities or between subactivities may only take place when an activity leaves a domain or terminates, or when a *delegated call* takes place. This has the effect that the calling domain does not block, but instead becomes idle, though remembering its execution state so that it may resume that activity when the delegated call eventually returns. This is especially useful when the calling object is not undergoing a state change, and hence mutual exclusion is not an issue.

Finally, objects may decide to *delay* activities invoking operations whose "services" are temporarily unavailable. Used in conjunction with reflexes and delegation, the delay mechanism can be used to capture event-driven behaviour such as triggering and constraints.

The rest of this paper describes the details of the programming language.

3 Type specifications.

Programming in Hybrid consists of defining new types. When a type definition is complete, objects which are to be instances of that type may be created and thus enter the environment.

A type definition consists of a new type name, some optional parameters, and a type specification. The type specification determines the visible interface of objects of the type being defined. If the type specification is incomplete, that is, does not provide a default realization for the operations of the new type, these may follow in the *private* part of the definition. This is the case, for example, when the *abstract* type constructor is used to specify a type. A discussion of realizations follows in section 4.

A type specification is built up from type constructors, existing types, and the parameters of the defined type. A number of types are given as "basic", and are always assumed to be available. Among these are: *boolean*, *integer*, *character* and *string*. The type constructors in Hybrid enable one to specify enumerated types, arrays, records, variant records, abstract types, abstract subtypes (through inheritance), and object identifiers.

The simplest type specifications use no constructors, and simply rename an existing type. A complete type definition, for example, would be:

type age : integer ;

This declaration defines a new type *age*, which is completely specified by the basic type *integer*. Though the

two types are equivalent, values of the one type are considered as distinct unless they are cast to the other type, or the equivalence is explicitly stated. Type-casting will be discussed in the next section.

Enumerated types are specified by providing an ordered list of names:

```
type size :
    enum {
        smallest ,
        small ,
        normal ,
        big ,
        biggest
    } ;
```

Enumerated types have the operations ".succ" (successor) and ".pred" (predecessor) defined on their instances. The types themselves (as objects) have the operations ".first" and ".last" defined. The predecessor of the first element and the successor of the last element are both *nil*. (In fact, this should raise an exception since the values returned here are not defined. An exception mechanism is planned for future versions of Hybrid.) The *nil* value is defined for all types, and is the default value for uninitialized variables.

A range is a subtype of an enumerated type:

```
type teen : 13 .. 19 ;

type mediumSize : small .. big ;
```

Also note that *integer* can be considered as:

```
type integer : smallestInteger .. biggestInteger ;
```

where *smallestInteger* and *biggestInteger* are the smallest and largest integers actually supported by the language (or a particular implementation).

Object identifiers provide a mechanism for indirectly identifying and communicating with an object. An object identifier uniquely identifies any independent or dependent object in any known environment:

```
type docOid : oid of document ;
```

Note the use of a type parameter in declaring oids. Newly-defined types may also take parameters, as we shall see shortly

Records are a familiar notion. The components of a record are "visible", and are accessed in Hybrid by following the name of the object by a "dot" and the name of the component. The components of a record are, in a sense, the "operations" for instances of that type:

```
type fraction :
    record {
        var numerator , denominator : integer ;
    } ;
```

Arrays, like oids, have type parameters, which are used to identify the type of the objects contained in

the array. In addition, arrays require an index parameter:

```
type mailbox : array [1..20] of message ;
```

Index parameters are a form of type parameter available for types that support an index operator. The index parameters of an array must be enumerated types. Strings, for example, would not be appropriate for indexing arrays in Hybrid (but they might be for a programmer-defined *table* object).

The variant constructor provides a mechanism for specifying a parent type as the union of several subtypes:

```
type rational :
    variant {
        integer or
        fraction
    } ;
```

When an instance of a variant is used, however, one must determine which subtype the instance belongs to in order to correctly apply the appropriate operations. Hybrid has a type-checking statement for this purpose. It will be described in section 4.2 on statements.

Abstract types are similar to records, except that their visible components may be operations, in addition to variables. In an abstract type specification, only the operation "stubs" are given. The stubs include a declaration of the operation name, the argument types, and the return value types. The realization of the operations are either unspecified or hidden in the type definition's private part. The internal variables of a realization of an abstract type are typically hidden, though they may also be made visible, as with records. Fractions, defined as abstract types, might look like:

```
type fraction :
    abstract {
        infix := : fraction → fraction ;
        infix + : fraction → fraction ;
        infix − : fraction → fraction ;
        infix ∗ : fraction → fraction ;
        infix / : fraction → fraction ;
    } ;
```

In this example, no realization is given for the listed operations. The operation name may be an identifier or an operator. Either infix or prefix operators may be defined. Furthermore, one may define an index operation (as for arrays), and an *init* operation, for initializing newly-created objects. Operations may be declared as "reflexes", if their invocation is to result in the creation of a new activity.

Note that no formal specification is given for operations (other than what may be provided in comments by the programmer). We are investigating the extent to which it may be useful to incorporate some formal specifications of objects into Hybrid.

Multiple inheritance can now be understood in the following way: Recall that a subtype is a refinement of a parent type, in the sense that every instance of the subtype is also an instance of the parent type. In Hybrid, we consider an abstract type to be a specification of the objects which have realizations for *at least* the given set of operations, and possibly more. An abstract type that specifies additional operations is therefore a subtype. In general, an abstract subtype inherits the operations of two or more parent types.

```
type ownedDocument :
    inherits {
        ownedItem and
        document
    } ;
```

Instances of *ownedDocument* support the operations of *ownedItem* as well as those of *document*. Every instance of *ownedDocument* is therefore an instance of *ownedItem*, and so the set of instances of *ownedDocument* is a subset of the instances of *ownedItem*. *ownedDocument* is a subtype of both *ownedItem* and of *document*, and the set of its instances is the intersection of those of its parent types.

Unlike the *variant* construct which specifies a new parent type as a union of subtypes, the *inherits* construct specifies a new subtype as the intersection of several abstract parent types. Instances of an abstract subtype always have all of the properties of their parent types, whereas instances of a variant have the properties of one of the subtypes.

A type may be defined so as to require index parameters or type parameters. The parameter names may then be used within the type specification that follows. All of the type names used within the specification must be bound to either existing types, or to parameters of the type being defined:

```
type matrix [xtype, ytype] of stuff :
    array [xtype] of
        array [ytype] of stuff ;
```

It is possible to constrain type parameters to a be subtype of a given parent type:

```
type intArray [indexType] of (intRange :< integer) :
    array [indexType] of intRange ;
```

Instances of the type *intArray* may only contain values belonging to some subtype (i.e., subrange) of integers.

4 Realizations.

A realization for an abstract type may be given in the optional *private* part of a type definition. The private part consists of a number of variable (and other) declarations, and realizations for the type's operations. As a simple example, consider the following type definition:

```
type linkedListElement of elementType :
    abstract {
        next : → oid of linkedListElement
            of elementType ;
        contents : → elementType ;
    } ;
```

```
private # for linkedListElement of elementType
{
    # Hidden variables ...
    var item : elementType ;
    var nextItem : oid of linkedListElement
        of elementType ;

    # Realizations of operations ...
    next : → oid of linkedListElement
        of elementType ;
    {
        return (nextItem) ;
    }

    contents : → elementType ;
    {
        return (item) ;
    }
} # end of linkedListElement of elementType
```

This example declares two hidden variables, and provides realizations for two operations. Note that comments may appear following a number sign ("#").

A variable declaration consists of the keyword *var*, a list of optionally initialized variables, a colon, and a type specification. List elements are separated by commas. A constant declaration is identical, except the keyword *const* is required, and initialization is compulsory. The assignment operator ":=" is used to initialize constants and variables to constant expressions:

```
const min:=10, size:=50, max:=(min+size) ;
```

One may "undeclare" variables, constants and operations inherited from the private part of a parent type. An "undeclaration" consists of the keyword *drop*, and a list of names of items to drop:

```
drop min, size, max ;
```

In addition to variable and constant declarations and "undeclarations", Hybrid also provides a "type-casting declaration". The *cast* declaration is used to enable automatic type-casting between equivalent types, or from subtypes to parent types. Hybrid conservatively assumes (with few exceptions) that objects whose types are not literally identical are type-incompatible. One must either explicitly cast objects to equivalent or parent types, or enable automatic casting by declaring a type relationship. Casting from a parent type to a subtype requires a run-time type-check, to be discussed in the subsection on statements. Equivalence between two types, or sequences of types, is indicated by the "::" relation, and subtyping by the ":<" relation:

```
cast fraction :: (integer, integer);
cast teen :< integer ;
```

Automatic casting is applied only to arguments of an operation, not the object to which the operation is being applied (called the *target*). The type-casting mechanism is well-defined, since it requires only a

comparison of the type of an argument to the type expected for the operation being invoked. Target objects must always be explicitly cast, since it is not in general possible to determine from the name of an operation and the argument type what type is intended for the target.

Realizations for operations repeat the operation stub, binding any arguments to variable names, and are followed by a compound statement consisting of the code to be executed when the operation is invoked. A *delay* queue may be specified for the operation, if calls to the operation are to be conditionally accepted. Delay queues are discussed in the subsection on triggering.

The rest of this section is divided into three subsections dealing respectively with expressions, statements and triggering.

4.1 Expressions.

Expressions in Hybrid are used for invoking operations of objects, and for structuring and manipulating values. Every expression has a value, and is therefore typed. Every subexpression of an expression identifies either a *target* of an operation to be invoked, or an *argument* of some operation.

The simplest expressions are variables and constants. Predefined constants include integers, strings, and the boolean values *true* and *false*. Names may also be bound to constant expressions using the *const* declaration described earlier. In addition to declared variables, there are also a number of standard "pseudo-variables" such as *self* and *caller* (both of type *oid*) which objects have access to.

Values may be bundled together using commas and parentheses. Complex values constructed in this way are used when invoking operations that require multiple arguments. Named operations are invoked using the familiar "dot" notation:

```
stack.push(5,6)
```

An expression identifying a target object (in this case, a variable name) is followed by a period, the name of an operation, and an expression identifying the argument values. A local operation may be invoked using either the pseudo-variable *self* as a target, or by leaving out the target and the period altogether (as though invoking a procedure).

Reflexes are "top" operations that create a new activity. They are invoked using an exclamation mark instead of a period:

```
mailbox ! purge
```

When an object invokes a reflex, a new activity is created. The new activity will proceed when the called object is available, as with other operations. The caller, however, does not wait, but proceeds independently.

A new object is created by invoking the *create* operation of the corresponding type object. If the new object is to be initialized, the initialization arguments must be given to the *create* operation. The value returned is an oid of the given type:

```
var d : oid of document ;
d := document.create(args) ;
```

This has the side-effect of instantiating a new dependent object, that is, in the same domain as that of the calling object. If the new object is to be independent, the *export* construct must be used:

```
d := export(document.create(args)) ;
```

In this case, the object created will be the root object of a new domain.

In addition to named operations and reflexes, one may define an *index* operation, and prefix and infix operators. The index operation is invoked by following the recipient by the bracketed argument expression:

```
item := table[key]
```

Operators can also be defined which are made up of a fixed alphabet of operator characters. There is a simple set of rules that allow all expressions to be parsed uniformly, regardless of the operators that are actually defined for the targets of the expression. This approach is also used for operator-overloading in C++ [Stro86a] and Objective C [Cox86].

The prefix operators have higher priority than infix operators. They are invoked in the obvious way, preceding the target expression (which is the sole "argument" of a prefix operator). Infix operators follow their target, and are in turn followed by the argument expression. Any infix operator terminating with a "?" is assumed to be a relational operator, and should yield a boolean result. The infix operators "*", "/" and "%" together with the relational operators are given higher priority than other infix operators. Finally, any operator terminating with an "=" is assumed to be an assignment operator, is parsed right-to-left instead of left-to-right, and is given lowest priority. As a consequence of these conventions, expressions such as:

```
n := a * ++b + c
```

are parsed in a fairly natural fashion. In this case, as:

```
(n := ((a * (++b)) + c))
```

We also adopt the convention that object identifiers inherit the operators of the referenced type, each preceded with a "@". "@" itself is a prefix operator yielding the value of the referenced object. This allows us to disambiguate operators with the same name, such as ":=". Consider the example:

```
var d1, d2, d3 : oid of document ;

# Make d1 identify the same document as does d2.
d1 := d2 ;

# Copy the contents of the document identified by d3
# to the one identified by d1.
d1 @:= @d3 ;
```

Any expression may be typecast by following it with a colon and the specification of an equivalent type:

```
# We assume the earlier definition of fractions
# as a record of two integers.
var x : fraction ;
var n, m : integer ;

x := (n,m) : fraction ; # explicit casting
```

Implicit casting could have been used to accomplish the same thing by including the declaration:

```
cast fraction :: (integer, integer) ;
```

since the := operation of the target x expects an argument of type *fraction*.

For a precise definition of the way expressions are parsed refer to the grammar given in the appendix.

4.2 Statements.

Statements in Hybrid, unlike expressions, have no value or type. They are used to evaluate expressions and to control the flow of execution. The simplest statement in Hybrid is the *null* statement, consisting of a single semi-colon. An expression followed by a semi-colon is a statement which causes that expression to be evaluated. The type of such an expression should be *nil*, that is, all values generated in the evaluation of the expression should be consumed.

A compound statement consists of an opening brace, a sequence of declarations, a sequence of statements, and a closing brace. (All statements in Hybrid terminate either with a semi-colon or with a closing brace.)

The *check* statement is used to perform a run-time type-check. It determines whether the current value of a variable is an instance of a subtype of that variable's declared type. If the condition is satisfied, then the variable is temporarily redeclared to be of that subtype. The check statement consists of the keyword *check*, a type-checking expression, and a compound statement. There is an optional *else* clause to handle failure of the type-checking expression. The type-checking expression consists of a variable name, the type-checking operator ":?", and a type specification:

```
var n : integer ;
var a : array [1..10] of integer ;
# some code ...
check (n :? 1..10) {
    a[n] := 0 ;
}
else {
    # complain ...
}
```

The variable n is redeclared for the body of the compound statement following the type-checking expression, provided the type-check succeeds. This is necessary to allow it to be used to index the array a. In case of failure, the *else* clause is executed, and the variable n is not redeclared. A type-check is similarly required for meaningful manipulation of variables of variant types.

If statements consist of the keyword *if*, a boolean expression, and a compound statement. There is an optional *else* clause.

```
if (n =? m) {
    n := m+1 ;
}
```

Block statements and *loop* statements consist of the keyword *block* or *loop*, an optional label, and a compound statement. They are similar, except that blocks are by default executed once, whereas loops are repeated. The *break* and *continue* statements can be used to jump to the beginning or the end of a block or loop. One may break or continue several nesting levels by indicating the label of the block or loop to break or continue, but one may not break or continue a block or loop that one is not currently inside. The following examples are equivalent:

```
loop {
    # body ...
    if (done)
        break ;
}
loop : a {
    # body ...
    if (done)
        break : a ;
}
block : a {
    # body ...
    if (~done)
        continue : a ;
}
```

The *switch* statement has do-first semantics. It consists of the keyword *switch*, the name of a variable or an expression in parentheses, and, within enclosing braces, a sequence of *case* clauses. There is an optional *default* clause. Each case consists of the keyword *case*, a value or a range, and a compound statement. There is some similarity to the *check* statement, but the switch expression may be an arbitrary expression instead of just a variable, and so there is no type re-declaration in the cases.

```
var n : 1..10 ;
var a : array[1..10] ;
n := 1 ;
a[n] := 0 ;
loop {
    switch (n) {
        case 1..9 {
            n += 1 ;
            a[n] := 0 ;
        }
        default {
            break ; # out of loop ...
        }
    }
}
```

The *return* and *forward* statements include an expression in parentheses which agrees with the type of the

declared return value of the operation. The forward statement causes the target of the contained expression to return its result to the original caller (i.e., of the operation containing the forward statement) rather than to the forwarding object. This is useful for "administrator" objects whose function is to allocate jobs to "worker" objects [Gent81]:

```
forward ( worker.operation(job) ) ;
```

After the job is forwarded, the worker will return directly to the original caller, and the administrator is immediately free to handle other requests.

The *coblock* and *coloop* constructs generate a scope within which concurrent subactivities may be created. The *break* and *continue* statements apply as with *block* and *loop*. A subactivity is created using the *activity* statement. When a subactivity is created, the parent is suspended, and the child starts to run. Only when the child terminates or executes a delegated call will control be switched back to the parent. Delegation is therefore necessary to acheive interleaved execution and parallelism. When the parent activity executes the *break* statement or reaches the end of a *coblock*, it waits till all of its children have terminated.

In the following example, an object has a "weight" which is the sum of the weights of two independent "parts". To discover its own weight at any time, it can request the weights of each its parts and wait for the two results in parallel:

```
coblock {
    activity {
        wx := delegate(xOid.weight) ;
    }
    activity {
        wy := delegate(yOid.weight) ;
    }
} # parent waits here
myWeight := wx + wy ;
```

Transactions may be specified using the *atomic* statement. It begins with the keyword *atomic* and is optionally labelled. There is an optional abort handler clause beginning with the keyword *onabort*. All objects taking part in a transactions are obliged to delay calls from other activities until the transaction either aborts or commits. All objects must be capable of returning to the state they had when they joined the transaction. Nested transactions are supported. One may abort or commit a transaction with the statements *abort* and *commit*, optionally providing the label of the nesting level to abort or commit. A two-phase commit protocol is assumed. A full discussion of the transaction mechanism is beyond the scope of this paper, as is the issue of deadlock prevention. These issues are treated in [Nier87].

4.3 Triggering.

An activity is *triggered* when it is initiated, or when it is resumed after being suspended or *delayed*. The most basic form of triggering in Hybrid, then, is the creation of a new activity through the invocation of a reflex. An activity created in this way will be completely independent of the activity that created it, and the two will proceed in parallel.

Somewhat more interesting is the case of activities which, once initiated, are delayed for some reason and later resumed. A simple form of triggering results from the non-interference of activities. When an object in one domain calls an operation of an independent object in a busy domain, then the call is delayed. When the domain becomes idle, delayed calls may be triggered (one at a time).

This idea can be generalized by allowing an object to have conditional delay queues on its operations

and reflexes. We will show how this notion can be used to implement various kinds of triggering

A delay queue is declared as a variable of type *delay*. An operation may then be declared as *using* that queue. Delay queues have the operations *open* and *close*. When a delay queue is open, operations that use that queue may be invoked. Otherwise they are delayed. Various operations may share the same queue.

The simplest triggering example is that of acquiring a resource, and is comparable to the way in which an activity acquires the services of an active object:

```
type resource :
    abstract {
        acquire : → ;
        release : → ;
    } ;

private
{
    var available : delay ; # initially open
    var holder : oid of object ;

    acquire : → ;
        uses available ;
    {
        holder := caller ;
        available.close ;
    }

    release : → ;
    {
        if ( caller =? holder ) {
            available.open ;
            holder := nil ;
        }
    }
}
```

Requests to acquire the resource are honoured if the resource is available. Otherwise they are delayed, and later triggered when the resource is released.

Delay queues have some similarities to waits and signals in monitors, with some important differences:

- Domains (collections of dependent objects) are active, whereas monitors are passive.
- Closing a delay queue does not cause the current activity to be suspended. One may only delay future activities, not the current one, as with *wait*.
- Opening a delay queue triggers delayed activities only when the current activity releases the object. There is no immediate transfer of control, as with *signal*.

Delay queues have a few rather strong restrictions: The only information the message-handler can use to determine when to delay a message is the name of the operation being invoked. It cannot, for example, examine the oid of the caller, or the list of arguments being passed, since one would have to accept the call and start executing the operation to do so. Furthermore, one is limited to a fixed number of delay queues, at most one per operation. It is not possible to attach multiple queues to an operation. Despite these

restrictions, it is possible to achieve similar effects by forwarding calls to other objects that delay messages on command, as we shall show.

Consider, for example, a "clock" object (any object with a countable number of ordered states) that keeps track of its own internal state. An object may sample this state, or ask to be triggered when a particular state has occurred. In the latter case, the clock first checks if that state has passed. If it has, it returns immediately. If not, it forwards the request to a trigger object that is specialized in waiting for that *particular* state. The clock object maintains a table of trigger objects, one for each awaited state. When a request comes to be triggered on a previously unawaited state, the clock object creates a new trigger object, and adds it to its table. The trigger objects are created with a closed delay queue which can only be opened on instruction from the clock. Whenever the clock "ticks" it wakes up the corresponding trigger object, and then destroys it, or reuses it to wait for a different state.

Another interesting case is that of objects interested in a constraint involving several other objects. The object may not wish to be "busy" while it is waiting for notification from the other objects. A simple example is a line object that is constrained to be connected to two point objects. The line must be free to redraw itself on request at any time, and also be ready to move automatically when one of the attached points is moved.

As was the case with concurrent subactivities, *delegation* is required to prevent the constraint activity from blocking the line object. A delegated call to an operation that will later be triggered frees an object to switch to other activities. When the delegated call returns, execution may continue from the point where the call was made (when the domain is idle and free to accept the return message).

To handle this situation, the line object must be equipped with "notifier" activities that do the actual waiting. The notifier activities are initiated by a *constraint* reflex of the line object that calls the triggering operation of the point object using the *delegate* construct:

```
reflex constraint : (p:point) → ;
{
    # await notification of update
    delegate(p.awaitUpdate)) ;
    # handle it
    self.handleUpdate() ;
    # start a new notifier activity
    self!constraint(p) ;
}
```

The delegation of the trigger condition frees the line object to switch its attention to other activities (such as requests to the *redraw* operation). The *awaitUpdate* operation of the point object uses a delay queue in a fashion similar to that of the clock object. Whenever a point is moved, it wakes up the delayed notifier activity, and the constraint reflex of the line object continues executing. The reflex can then take care of managing the constraint. The last act of the constraint reflex can be to re-invoke itself. Variations of this theme are possible depending on the kinds of constraints one wishes to enforce. The issue of triggering is discussed in greater detail in [Nier87].

5 Implementation.

A prototype Hybrid system is currently being implemented on Sun workstations running Unix[2] 4.2 BSD. The compiler, written using the Yacc and Lex compiler-writing tools, will translate Hybrid type definitions into parse trees that will be directly interpreted at run-time. (They can also be used to generate executable code in another target language in future versions of the system.) A Hybrid environment will consist of a

[2]Unix is a trademark of AT&T Bell Laboratories

single Unix process managing a collection of active objects.

The object manager in each environment handles message-passing, "process" switching between domains, and stable storage. Objects reside in a "workspace" in virtual memory. The workspace can be atomically updated onto stable storage. For the prototype implementation, an update of stable storage requires writing the entire workspace to disk. Objects which have been idle or blocked for a long time may become "dormant" and be retired to a secondary workspace on disk.

Scheduling of objects is initially planned to be simple first-come, first-served, with objects being switched only when they wait or go idle. Other forms of scheduling will be investigated in detail in the future, for example, based on real-time requirements of running objects.

Switching between running objects requires the ability to save and restore the run-time stacks of active objects. This can be done within a Unix process either by simulating the object stacks in the heap, as would be the case for interpreted objects, or by using an implementation of co-routines that save portions of the real stack into the heap. The latter approach may be used to switch objects whose realization is not given in Hybrid (i.e., using pre-existing software).

Objects in one environment may communicate with those in other environments. Object managers will handle communications using the BSD "socket" mechanism. Unix libraries and system calls will be made available through Hybrid interfaces. Existing code can thus be mapped into a Hybrid system.

Many parts of the prototype have already been implemented, including workspace management, stable storage, simple scheduling, co-routines for C, and a Yacc/Lex parser.

6 Concluding remarks.

Hybrid is a design for a fully object-oriented programming language in which objects themselves are the active entities. A type specifies a set of values, together with operations over that set. An object is viewed as a variable which may assume values belonging to a type, and which may change state according to the operations of the type.

Hybrid is strongly-typed, and is equipped with type-checking constructs for recognizing when values of two types are compatible. New types can be defined flexibly using type constructors and existing types. Overloading of operation names is allowed, as is multiple inheritance over abstract types. Reusability of code is supported through instance variables, parameterization, inheritance and tailoring.

Hybrid unifies processes, files and objects into the notion of persistent "active objects". Groups of objects are organized into process-like entities called "domains". The notion of an "activity" is introduced as a mechanism for avoiding interference between objects by capturing a single-thread flow-of-control. New activities are created by invoking "reflex" operations. Delay queues are introduced as a mechanism for delaying and resuming running activities. Forwarding, delegation and subactivities are presented as ways of enabling flexible interleaving of activities.

A number of important issues are not dealt with in the version of Hybrid presented here. Some of these are: exceptions for operations, signals, or "high-priority operations", and security. We believe that it will be possible to integrate solutions to these problems into Hybrid in a way that is consistent with, and perhaps even exploits, the object-oriented approach. We are also exploring the idea of developing a usable definition of type that captures some of the semantics of operations, and is consistent with the notion of type inheritance.

7 Acknowledgements.

The work described in this paper was carried out with the assistance of a postdoctoral fellowship granted by the Natural Sciences and Engineering Research Council of Canada. Its support is gratefully acknowledged.

8 Appendix.

A grammar for the Hybrid language follows in extended BNF. It is derived from the grammar specified using the *YACC* compiler-writing tool provided by Unix.

Non-terminals are indicated in non-emphasized type. Keywords are shown in bold. Literals (such as parentheses, commas, etc.) are given in single quotartion marks. Other tokens and terminals are indicated by names in italic.

An operator begins with any number of "@" signs, followed by a sequence of operator characters:

$* / \% + - \wedge | < = > \backslash \$ \& \sim ?$

An assignment operator is an operator terminating in an equals sign. A relational operator is one terminating in a question mark. Priority operators are "∗", "/" and "%".

Constant, variable, pseudo-variable and operation names are identifiers consisting of an alphabetic character followed by any number of alphanumerics.

typeDef : **type** typeName
[indexTypeParamNames] [typeParamNames]
[**init** typeArgs] ':'
typeSpec ';'
[**private** '{'
{ dec ';' }
[operationSpec compoundStmt]
'}']

typeName : newTypeName | *basicTypeName*

newTypeName : *identifier*

indexTypeParamNames : '[' { constrainedType ',' } constrainedType ']'

typeParamNames : **of** constrainedType
| **of** '(' { constrainedType ',' } constrainedType ')'

constrainedType : newTypeName [':<' typeSpec]

typeArgs : typeSpec | '(' [{ typeSpec ',' } typeSpec] ')'

typeSpec : *basicTypeName*
| **enum** '{' { constName ',' } constName '}'
| enumVal '..' enumVal
| **oid of** typeArgs
| **array** indexTypeParams **of** typeArgs
| **record** '{' { varDec ';' }+ '}'
| **variant** '{' { typeSpec **or** } typeSpec '}'
| **inherits** '{' { typeSpec **and** }+ typeSpec '}'
| **abstract** '{' { varDec ';' }
[operationStub] '}'
| newTypeName [indexTypeParams] [**of** typeArgs]

indexTypeParams : '[' { *indexType* ',' } *indexType* ']'

constName : *identifier* | boolean

enumVal : *integer* | constName

operationStub : operationDec ':' [typeArgs] ['→' [typeArgs]] ';'

operationSpec
    : operationDec ‘:’
        [‘(’ [{ argDec ‘;’ } argDec] ‘)’] [‘→’ [typeArgs]] ‘;’
        [**uses** varName ‘;’]
operationDec : **init**
    | **prefix** prefix
    | **infix** infix
    | **index**
    | **reflex** operationName
    | operationName
prefix : *priorityOperator* | *operator*
infix : *priorityOperator*
    | *relationalOperator*
    | *operator*
    | *assignmentOperator*
    | ‘:=’
argDec : { varName ‘,’ } varName ‘:’ typeSpec
dec : varDec
    | constDec
    | castDec
    | unDec
varDec : **var** { varInit ‘,’ } varInit ‘:’ typeSpec
varInit : varName [‘:=’ constExpr]
varName : *identifier*
constExpr : element
constDec : **const** { constInit ‘,’ } constInit ‘:’ typeSpec
constInit : constName ‘:=’ constExpr
castDec : **cast** { castSpec ‘,’ } castSpec
castSpec : typeArgs ‘::’ typeArgs
    | typeArgs ‘:<’ typeArgs
unDec : **drop** { dropItem ‘,’ } dropItem
dropItem : *identifier* | **all**
stmt : ‘;’
    | expr ‘;’
    | compoundStmt
    | **check** ‘(’ varName ‘:?’ typeSpec ‘)’ compoundStmt
        [**else** compoundStmt]
    | **if** ‘(’ expr ‘)’ compoundStmt
        [**else** compoundStmt]
    | **block** [labelPart] compoundStmt
    | **loop** [labelPart] compoundStmt
    | **break** [labelPart] ‘;’
    | **continue** [labelPart] ‘;’

| **switch** element ‘{’
{ **case** enumCase compoundStmt }+
[**default** compoundStmt] ‘}’
| **coblock** [labelPart] compoundStmt
| **coloop** [labelPart] compoundStmt
| **activity** compoundStmt
| **return** [‘(’ [expr] ‘)’] ‘;’
| **forward** ‘(’ expr ‘)’ ‘;’
| **end** ‘;’
| **atomic** [labelPart] compoundStmt
[**onabort** compoundStmt]
| **abort** [labelPart] ‘;’
| **commit** [labelPart] ‘;’

compoundStmt : ‘{’ { dec ‘;’ } { stmt } ‘}’

enumCase : [enumVal ‘..’] enumVal

labelPart : ‘:’ *identifier*

expr : term | expr ‘,’ term

term : binary
| binary ‘:=’ term
| binary *assignmentOperator* term

binary : priority
| binary *operator* priority

priority : unary
| priority *priorityOperator* unary
| priority *relationalOperator* unary

unary : primary [typeCast]
| *priorityOperator* unary
| *operator* unary

typeCast : ‘:’ typeSpec

primary : element
| primary ‘[’ expr ‘]’
| callOperation
| varOperation ‘(’ [expr] ‘)’
| varOperation
| **export** ‘(’ expr ‘)’
| **delegate** ‘(’ expr ‘)’

element : const
| constOrVarName
| pseudoVar
| ‘(’ expr ‘)’

const : *integer* | *string* | **nil**

constOrVarName : *identifier* | boolean

boolean : **true** | **false**

pseudoVar : **self** | **caller**

varOperation : primary ‘.’ operationName
| primary ‘!’ operationName

callOperation : operationName ‘(’ [expr] ‘)’

operationName : *identifier*

References

[Cox86] B.J. Cox, *Object Oriented Programming – An Evolutionary Approach* , Addison-Wesley, Reading, Mass., 1986.

[Gent81] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software – Practice and Experience, vol. 11, pp. 435-466, 1981.

[Gold83] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation* , Addison-Wesley, May 1983.

[Hoar74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", CACM, vol. 17, no. 10, pp. 549-557, Oct 1974.

[Lisk83a] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM TOPLAS, vol. 5, no. 3, pp. 381-404, July 1983.

[Moon86] D.A. Moon, "Object-Oriented Programming with Flavors", ACM SIGPLAN Notices, vol. 21, no. 11, pp. 1-8, Nov 1986.

[Nier85a] O.M. Nierstrasz, "An Object-Oriented System", in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 167-190, Springer Verlag, Heidelberg, 1985.

[Nier85b] O.M. Nierstrasz, "Hybrid: A Unified Object-Oriented System", IEEE Database Engineering, vol. 8, no. 4, pp. 49-57, Dec 1985.

[Nier86] O.M. Nierstrasz, "What is the 'Object' in Object-oriented Programming?", Proceedings of the CERN School of Computing, Renesse, The Netherlands, Sept 1986.

[Nier87] O.M. Nierstrasz, "Triggering Active Objects", in *Objects and Things*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, March 1987.

[Nyga86] K. Nygaard, "Basic Concepts in Object Oriented Programming", ACM SIGPLAN Notices, vol. 21, no. 10, pp. 128-132, Oct 1986.

[Stef85] M. Stefik and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations", The AI Magazine, Dec 1985.

[Stoy84] H. Stoyan, "What is an 'Object-Oriented' Programming Language?", Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences, 1984.

[Stro86a] B. Stroustrup, *The C++ Programming Language* , Addison-Wesley, Reading, Mass., 1986.

[Wein81] D. Weinreb and D. Moon, *The Lisp Machine Manual* , Symbolics Inc., 1981.