Hybrid Update

Dimitri Konstantas

University of Geneva
Centre Universitaire d’Informatique
24 rue General-Dufour
CH-1211 Geneva 4

tel: +41 22 705.76.64
fax: +41 22 705.77.80
e-mail: dimitri@cui.unige.ch

Abstract

Cell, a model for strongly Distributed Object Based systems is discussed. Its components, the nucleus and the membrane, are presented and their characteristics are described. The notions of trading and type transparency in the context of the Cell model are described and issues related to their design and implementation are presented.

Appeared in:
Hybrid Update

Dimitri Konstantas

Abstract

In order to improve the usability of the first prototype implementation of the Hybrid language we have introduced a number of changes to both the language and the system. This way features that were vaguely or not at all mentioned in the original language design were added, bugs were corrected and better run time facilities were introduced. The modifications and extensions include the introduction of versioning, type operations, a revised abstract type specification, dynamic loading and more portable run-time support system.

1 Introduction

Hybrid [1] is a strongly typed, concurrent, object oriented language for programming with active objects. The prototype implementation of the language [2] incorporated most of the language features and served as a vehicle for an evaluation of the language ideas [3]. By using the Hybrid system we identified a number of deficiencies and problems. These were corrected with the introduction of a number modifications and extensions to both the language and the run time system which we report in this paper.

The modifications and extensions introduced into the Hybrid system fall into two areas, namely the language design and the run time system. One important feature that had been mentioned in the original language design but which was not implemented due to a lack of precise semantics and specifications was the support of versions. In order to introduce version support in the Hybrid language we revised the abstract type specifications and introduced type operations. This way we allowed a consistent introduction of versions and version control within the language.

A major problem with the Hybrid run time system was that in order to introduce new types the system had to be stopped and recompiled or relinked. This made programming with Hybrid a painful and time-consuming task. For this reason dynamic compilation and loading of types was introduced into the run time system and a set of types was provided allowing access to the compiler and loader via both a window interface and programmatic interface.

The run time system of the first prototype was based on a custom-made light weight process support library. This proved to be difficult to port to a new architecture and it was replaced by the SUN light weight process library. As a result the run-time performance was improved and porting to SPARC architecture became a matter of recompilation. Nevertheless a side effect of this modification was that the support for persistence became weaker.

In the rest of this paper we describe the modifications and extensions to the Hybrid language and run time system. In section 2 we present the language modifications, in section 3 we describe the run time modifications and in section 4 we present our conclusions. Finally in the annex we give a description of the compiler and system types.
2 Language modifications and extensions

2.1 Abstract types

In the original Hybrid specifications a type is introduced by either specifying only its abstract part or both its abstract and private parts. In the first case the type is an abstract type with no implementation provided. When one wanted to introduce an implementation for an abstract type, the complete type definition, that is both abstract and private parts, was required. We found this way of providing an implementation to be problematic. The reason being that there was no way to distinguish the definition of new types from the definition of a new implementation for an abstract type. This is a very important issue when large collections of types are involved. Since it is unlikely that the programmer will know all the names of the existing types he may accidentally use a name of an existing abstract type when defining a new type. In this case an ambiguity will arise whether this is a new type definition or an implementation to be bound to the existing abstract type. The compiler can of course resolve the ambiguity with a signature comparison, but nothing restricts the two types from having the same signature. The situation becomes even more complicated due to possible typing errors that the programmer may introduce.

In order to resolve the above problems we modified the type specifications so that the implementations of abstract types are provided without the abstract part. The type name is sufficient to specify for which abstract type the implementation is to be bound. This way the compiler can easily distinguish between implementations of abstract types and definitions of new types. Thus the definition of an abstract type and an implementation provided at a later time will be:

```plaintext
# Abstract type
#
type A: abstract {
    op1 : (...) -> integer ;
    ...
} ;

# Implementation for abstract type A
#
# type A: private {
#    op1 : (...) -> integer ;
#    {
#        ....
#    }
#    ...
} ;
```

When both the abstract and private parts are provided in the same type definition, the type is always considered to be a new type definition.

For the better support of the abstract type specifications we have introduced a minor syntactic modification in the type definition syntax. Specifically, the semicolon at the end of the abstract part of the original specification was moved to the end of the type definition. This way the completion of a type definition is better marked. As result of the above modifications the syntax for a type definition has become:
Null type definitions, that is without abstract and private parts, are rejected by the compiler.

2.2 Type Operations

One of the requirements we set for the version support was the ability to instantiate an object of a specific type using a specific version. In the original Hybrid language objects were instantiated in the current execution domain by their declaration according to rigid default rules or in a new domain with the export statement. These instantiation methods were inadequate for our requirements where we wanted to instantiate objects in the current execution domain with flexible rules. For this reason we introduced type operations that allowed us to instantiate objects in the current execution domain at any moment and not only by declaration. Type operations were not introduced at the syntactic level of the language but at the semantic level. This way the grammar of the language was not modified. The syntax of a type operation is the same as that of an operation call:

```
primary :: operationCall '(' [expression] ')'  
operationCall :: primary '.' operationName
```

Where `primary` is the type name, `operationName` is the type operation and `expression` defines possible arguments of the type operation. For this implementation we have defined three type operations valid for all types:

- **new** which creates a new object of the type in the current execution domain and returns the handle to its instance. Objects created with the new operation can be bound directly to variables of the corresponding type. Note that the previously bound object is cleared (garbage collected) before the binding.

  ```
  var x : X ;
  x <- X.new (...);
  ```

- **create** which creates a new object of the given type and an oid object which points to it, both in the current domain, and returns the handle to the oid object.

  ```
  var y : oid of Y ;
  y <- Y.create (...);
  ```

- **export** creates a new object of the given type in a new domain and an oid object pointing to it in the current domain and returns the handle to the oid object.

  ```
  var z : oid of Z ;
  z <- Z.export (...);
  ```

All three type operations take one argument, a string, which specifies the version name (see section 2.3).

With the above syntax of type operations an ambiguity can appear when we have a variable with the same name as a type and with an operation named new, create or export taking an argument of the same type as the type operation. For example if we have the type
type someType : abstract { ... } ;
type XYZ : abstract {
  create : (...) -> ... ;
} ... ;

and we have a program

var someType : XYZ ;
...
someType.create(...) ;

the compiler being unable to resolve the ambiguity of the call someType.create(...) will consider it as an error and will terminate the compilation. This way the program will not be able to be compiled. The solution is to change the name of the variable so that the ambiguity is eliminated. Another solution would had been to disallow the names of \textit{new}, \textit{create} and \textit{export} as operation names of abstract types, but we found it too restrictive.

Due to the introduction of type operations the \textit{export} statement became obsolete and was therefore dropped from the language.

2.3 Versions’ support

Our major requirement for the version support was to have different suites of versions for a type. That is, we needed named versions. A named version is a version linked to a specific name. Each version has a name and, when inserted into the system, it replaces the existing version with the same name. The user can create instances of the type using the implementation provided by a specific named version. This way we can have different implementations for the same type. A special named version is the one with a null name, which we call the \textit{anonymous} version. It is the default version used for the instantiation of objects when no version name has been defined.

In our design for version support we considered two alternatives. First, to introduce the version mechanisms outside of the language, in the type manager for example, and second, to introduce them inside the language. Both alternatives had their advantages and disadvantages. The most important advantage of the first approach was that it did not require any language extensions or modifications. A specialized (system) type would had been introduced for the version handling. The disadvantage of the approach was that we could not handle, in a simple way, named versions. The name of a version could not be included in the type definition. As a result we would need to introduce some kind of preprocessing schema for the identification of the version name and the storage of the implementation in the right version suite. Furthermore, instantiation of objects using a specific version would have had to be done via an untyped operation of the version handling object. This was undesirable since it created a hole in the strong typing of the language.

The second approach, introducing the version control inside the language, had the disadvantage of introducing a new construct in the language but it allowed easier handling of named versions, the inclusion of the version name in the type definition and, most important, it preserved the strong typing of the language. For these reasons we decided to implement the version control inside the language.
The private part of a type defines an implementation version. Thus we introduced the version name as an optional argument of the private part definition:

```
typeDef :: type typeName
           [indexTypeParamNames] [typeParamNames]
           ':' [typeSpec]
           [private [versionName] '{'...'}']
           ';'  
```

A missing version name implies the anonymous version. This way new versions for abstract types can be introduced as

```
type XYZ : private newVersionName { ... } ;
```

The new version, once compiled, will become the default one from which new objects of the version suite are instantiated. Nevertheless old versions are not destroyed, since there might still exist objects that are using them. The system however guarantees that any new instantiation will use the latest version available.

One problem with the above schema was that we could not control if a named version suite existed or not. Specifically a version intended to be the first of a new version suite, could have been introduced and actually replace an existing version, because the name suite was already defined. In order to resolve this problem we decided to introduce a second keyword for the definition of the private part. We introduced the keyword `version` which can be used instead of the keyword `private`.

```
type XYZ : version newVersion { ... } ;
```

The difference between the `version` and `private` definitions is that the use of `private` means that the version name suite cannot already exist. The version introduces a new version suite and does not replace any existing version. In contrast the use of `version` means that the version suite may or may not exist. The version can thus replace a possibly existing one with the same name. Note that the use of `private` and `version` definitions have the same effect when a new type is defined (that is when both abstract and private parts are present in the definition).

As mentioned in the previous section the three type operations take one argument which is a variable of type `string` holding the name of the desired version. The string can be null in which case the anonymous version is used. If however there is no defined anonymous version then the latest version of all named versions is used. For example we can have

```
var a : oid of Atype ;
var versionName : string ;
versionName := "myVersion" ;
a <- Atype.create(versionName) ;
```

This will instantiate an object of type `Atype` using the latest version with name `myVersion` and an oid of `Atype` which will be pointing to the instance. The existence of a version with the given name is checked at run time. It is a run time error if a version with that name does not exist when the type operation is executed.
3 Run time system modifications and extensions

The most important run time system modifications, which are described in this section, were the introduction of dynamic compilation and loading of new types, the replacement of the run time support system, and the introduction of unbind instances. In addition we introduced a basic type port that provides an interface to UNIX files and sockets, and revised the implementation of the display type.

3.1 Dynamic compilation and loading

A major deficiency of the first Hybrid prototype was that the compiler was independent from the run time system. This way the system had to be stopped so that new implementations could be loaded. In order to resolve this problem we used the GNU Dynamic Loader package [4][5] which allows the dynamic loading and unloading of libraries. Furthermore we introduced the system type compiler which encapsulates the Hybrid compiler and allows the dynamic compilation and loading of new implementations. A type providing a “user friendly” interface to the compiler was also introduced (see Annex II for the description of the compiler type).

New implementations are introduced to the Hybrid compiler inside UNIX files. The user can specify the directory where the Hybrid sources are to be found, the directory where the generated C code is to be stored and the directory for the object code of the new implementation. If any of the paths is not defined, default values are used. The file names used for the C code and the object code are defined by the compiler according to the type and version names so that file name uniqueness is guaranteed. At the end of the compilation a status variable is available to the user indicating the status of the compilation. The information given indicates if the compilation succeeded or not and, in the latter case, what was the reason for the failure (syntax errors, semantic errors etc.). There is no restriction on how many instances of the compiler can exist. The user or users can compile and load more than one type at the same time.

One of the problems with strongly typed languages is that only existing types can be used when a program is written. That means that even if we introduce a new type dynamically at run time we cannot create instances of it from existing objects (which have no knowledge of the new type). The only way to resolve the problem and allow the instantiation of new types is by breaking the strong typing of the system. For this reason we introduced the type system. This is a special type which provides information for the run time system and can create instances of types outside the strong typing of the language (see Annex I for the description of the system type). The name of the type to be instantiated is provided as a string. The system object will instantiate the type in a new domain and will return an oid pointing to it. This oid is used by the system object to control the instantiated object. It is expected that the new type has a reflex operation named init which will be called after the type has been instantiated with arguments specified by the user.

3.2 Run time support

The run time support package of the first Hybrid prototype provided a custom made implementation of light weight processes. Its major advantage was that it allowed us to have full control over the execution threads. This way we were able to include the execution threads in the per-
sistent workspace. Nevertheless its disadvantage was that it was not 100% portable and it was incompatible with the dynamic loader. For these reasons we replaced it with the SUN LWP light weight process package [6].

The use of the LWP allowed us to more easily port the Hybrid system onto a SPARC platform and improved the overall performance of the system. On the other hand we lost control of the execution threads and we were not able to include them in the persistent workspace. This way the dynamic persistence of the first version of the Hybrid system was lost, since active threads could not be saved.

3.3 Unbound instances

Except with the use of the type operations, new instances are created during execution when dynamic variables are encountered.

```{var}
var dynamic : Atype ;
...
dynamic.someOperation(...) ;
```

When the execution point reaches the dynamic variable definition a new object will be instantiated. However there is a question of which version to use for the new instance. Since more than one named versions can exist, there is no obvious candidate. We thus decided that the anonymous version will be used. If however there is no anonymous version then the object is instantiated as unbound. That is, an instance of the basic type unbound is created adapted for the specific type so that a call to any operation of an unbound object generates a run time error. This way objects that use abstract types for some of their variables can be instantiated, even if there are no implementations for the abstract types. Nevertheless variables can be bound to an instance of a specific implementation with the new type operation:

```{var}
var dynamic : Atype ;
dynamic <- Atype.new("someVersion") ;
dynamic.someOperation(...) ;
```

4 Conclusions and future directions

The extensions and modifications of the Hybrid language and system improved the usability of the prototype and allowed us to use it for further research in a different direction [7]. Nevertheless the Hybrid environment is far from being even a primitive programming environment. The complete lack of system type libraries and support for run time exceptions prevent the full use of the language features. The Hybrid system needs to be populated with types that will provide an interface to the underlying UNIX environment and allow a better control of its run time environment. In addition, some of the new features should be better studied and revised so that they can be better integrated. For example the introduction of versioning introduced a problem to the inheritance semantics. That is, it is not clear which implementation version should be used when inheriting from an abstract type, nor there is a way of specifying it.

Presently we are using the Hybrid prototype vehicle for the further research in the direction of distributed systems and interoperability. This way the Hybrid language and the system are
evaluated for their suitability in application programming. Furthermore, because the same application will be implemented in another object oriented language, we will be able to have a comparative evaluation of the Hybrid language.

**Annex I. The System Type**

```kotlin
type system : abstract {  
    getNoOfTypes : -> integer  
    getTypeNo : (integer) -> string  
    checkType : (string) -> boolean  
    startType : (string, integer, string) -> boolean  
    getOpIndex : (string, string) -> integer  
    getNumOfArgs : (string, string) -> integer  
    getArgTName : (string, string, integer) -> string  
}
```

Operations’ description:

**getNoOfTypes:**
- **Arguments:** (nil)
- **Returned value:** Integer : Total number of defined types.

**getTypeNo:**
- **Arguments:** 1. Integer : Number of type in the type manager.
- **Returned value:** String : Type name of the requested type.

**checkType:**
- **Arguments:** 1. String : Type Name.
- **Returned value:** Boolean : TRUE if type exists, FALSE if type does not exist.

**startType:**
- **Arguments:** 1. String : Type Name to instantiate
  2. Integer : Integer argument to be passed to the init operation
- **Returned value:** Boolean : TRUE if success, FALSE if failed.

**getOpIndex:**
- **Arguments:** 1. String : Type Name
  2. String : Operation Name.
- **Returned value:** Integer : The operation number of the operation for the type.

**getNumOfArgs:**
- **Arguments:** 1. String : Type Name
  2. String : Operation Name.
- **Returned value:** Integer : Number of arguments of the operation.

**getArgTName:**
- **Arguments:** 1. String : Type Name
  2. String : Operation Name
  3. Integer : Argument Number.
- **Returned value:** String : Type name of the argument.
Annex II. The Compiler Type.

```typescript
type compiler : abstract {
    compile : (string) -> integer ;
    load : -> boolean ;
    setParameter : (integer, string) -> ;
    getParameter : (integer) -> string ;
}
```

Operations’ description:

**compile:**
- Arguments: 1. String : File name where the type is defined.
- Returned value: Integer.
  - 0 : Compilation Completed, new code loaded.
  - 1 : Internal Compiler Error.
  - 2 : Compiler called with illegal arguments
  - 3 : Compilation aborted due to syntax errors
  - 4 : Compilation aborted due to semantic errors
  - -1: Compilation aborted due to internal compiler limitations.

**load:**
- Arguments: (nil)
- Return value: Boolean
  - TRUE : loading completed
  - FALSE : loading failed

**setParameter:**
- Arguments: 1. Integer : Parameter to set
  - 2. String : Parameter value
- Returned value: (nil)

**getParameter:**
- Arguments: 1. Integer : Parameter for which the value is requested.
- Returned value: String : Parameter value

**Compiler Parameters:** (all strings)
- 1. Hybrid Source directory.
- 2. Hybrid Source file name
- 3. Type Name (read only)
- 4. C translation directory
- 5. C translation file name (read only)
- 6. Compiled C code library directory
- 7. Object code library name
- 8. Version Name (read only)
- 9. Version suite serial number (read only)
- 10. Operations’ loading routine (read only)
- 11. Flag indicating if relink of executable is needed (YES, NO) (read only)
- 12. Version ID in the Type Manager (read only)
References


