O3: A Rational Construction of an Open Object-Oriented Language

Mihai Barbuceanu and Gheorghe Ghiculete

Expert Systems Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA

Abstract: O3 is an object-oriented language developed with the explicit aim of providing a rational framework for extensive language customization. This framework goes as far as to support a different view of programming. According to this view, programming in O3 is a two-stage process: the first is a language construction process in which users design the structure and behaviour of the sorts of objects they need and the second is the normal language use process in which the custom designed language is being applied to model applications. O3 is implemented as layered reflective architecture clearly separating the kernel, interface and application levels. Reflective, self-modifying behaviour is acquired by dynamic, yet efficient dispatching to kernel operation protocols which programmers can use to create and integrate various language features. Experience in building O3 versions supporting the development and integration of significant languages and tools with stringent requirements is reported.

Keywords: object-oriented languages, language extension, language construction,reflective languages, knowledge representation.

Gheorghe Ghiculete graduated from the Polytechnical Institute of Bucharest in 1986. His work experience covers software for digital IC CAD (at the "Microelectronica" Co., Bucharest, 1986- 1987) and software development for the DIALISP Lisp Machine (The Polytechnical Institute of Bucharest, Functional Electronics Lab., 1987-1989). Since autumn 1989 he joined - as a research fellow - the Expert Systems Laboratory at the Research Institute for Informatics in Bucharest, working in knowledge representation and processing.
His research interests include knowledge representation, object-oriented systems, constraint programming, expert systems, symbolic calculus and Lisp architectures.

1. Introduction

Programming language constructs can be classified according to two features: usability and reusability. The usability of a construct can be defined only in relation to a given application or application domain and depends on the degree of match between the construct and the application. For example, string processing constructs are usable for text editing applications. Usable constructs are thus application specific and reflect features of the application. Reusability, on the other hand, depends on the ease of using a construct in many different application types. An if-then-else construct is for example highly reusable. For that reason it is also a very abstract construct. Usability and reusability are contradictory features. The more usable a construct, the more specific and hence non-reusable it is, and viceversa.

Programming languages make various compromises between these features. Specialized or problem-specific languages have usable constructs for their designated application types. General-purpose languages offer reusable constructs for a large class of domains.

Ideally, a language ought to offer both types of constructs in order to be at once useful for many domains and suitable for each domain in part. This is however, not possible unless the language is extensible in a way allowing users to create or customize specific constructs and to get rid of constructs they do not need.

Object-oriented languages do offer such a capability to an important extent as they allow users to define and combine objects with specific structure and behaviour. This kind of an extensibility offered by "classic" OO languages like C+ + or SmallTalk, is nevertheless not enough. The kind of extensibility we need goes further than what the languages allow, touching deeper aspects related to the behaviour of "primitive" services like inheritance, message-passing or demonization or to the implementation of the internal data structures of the language.

As an example of the extent of flexibility we really need, consider the case of building a modern

hybrid programming environment integrating distinct paradigms such as objects, rules, constraints and graphic interfaces. We have for long been involved in the development of such environments [Barbuceanu, Trausan and Molnar 87] and always needed customize the intimate structure and behaviour of the basic services of the OO implementation language we used in order to obtain clear and efficient implementations. For example, we often needed objects which could efficiently store a large number of slots, or objects which would only accept a predefined collection of slots, or objects which would enforce a minimum and maximum number of slot fillers.

Such requirements may in general be either simulated with existing machinery or simply avoided. However, when simulating such mechanisms one has to pay for the overhead ensued by unused, partly used or inadequately used features, while avoidance leads to incomplete or otherwise inadequate implementations. Even worse, when many such "simulations" accumulate over time, the system grows in ways hard to understand and modify. That is why the best is always being able to create the right construct for the task at hand.

O3 is an OO language developed in the Expert Systems Research Laboratory of the Research Institute for Informatics with the explicit aim at providing a rational framework for extensive language customization. This framework goes as far as to support a specific programming paradigm. According to this paradigm, programming in O3 becomes a two-stage process: the first is a language construction process in which users design the structure and behaviour of the sorts of objects they need and the second is the normal language use process in which the custom designed language is being applied to model applications. O3 is now used as the underlying base or carrier for developing and integrating expert system languages and tools implemented in our laboratory.

2. O3 Architecture

The major goal of O3 is offering a uniform framework for systematically developing specialized language mechanisms and OO language versions which can be consistently integrated into and used in a single AI oriented OO environment.

The language evolution mechanisms provided by O3 can control both the internal data structures for representing objects - e.g. the data structures used for implementing slots - and the behaviour of the basic language operations, including e.g. object creation, slot inheritance, method activation, demon invocation, etc.

An O3 environment can any time contain any number of objects with distinct implementations and distinct behaviours of their basic operations. All of them are manipulated through a uniform functional interface allowing programmers to uniformly operate upon internally distinct objects. For example, an O3 environment can simultaneously work with objects whose slots are implemented as lists, hash tables or records, or with objects which inherit in totally distinct manners.

Unlike other attempts to attain this goal [Filman 87], we tried to come with a rational language design yielding this flexibility rather than attempting to bend an existing language to fit these needs.

The major lines of this design are briefly reviewed now. First, O3 has three major software layers depicted in Figure 1. At the first layer - the O3 kernel - there are the basic O3 operations and internal data structures. At the second layer, there is a functional interface programmers can use. At the third layer there are application programs written in terms of the functional interface. Each layer can only use the functions provided by the layer beneath it.

To customize the language, the O3 programmer can use special mechanisms which modify the first layer. Once made, these modifications are automatically applied by the functional interface which stays as such. This is possible due to the following arrangement. First, the functional interface makes no assumption on the internal representation of objects. Interface functions perform only type checking of arguments and dispatch to internal (first layer) operations. Second, internal operations are organized in protocols and dynamically linked to the O3 objects they apply to. This is illustrated in Figure 2.

According to this organization, each O3 object, whatever its implementation, has a pointer to a

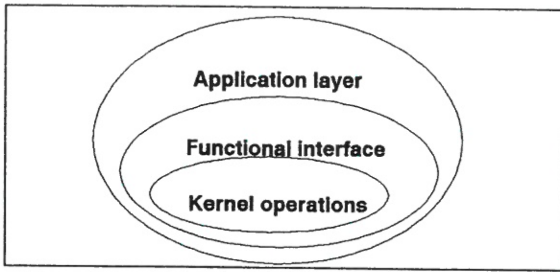

Figure 1.O3 Layers.

special object named its LanguageObject. The LanguageObject contains for each internal operation applicable to the object a Common LISP function which implements it. An O3 environment can have any number of Language Object-s.

Consider now that the interface performs operation Op on object O. The interface will actually take the following actions as part of this invocation.

1. Check the syntactic correctness of the operation application.

2. Determine which kernel operations are required to execute the requested operation.

3. Invoke the functions associated with the requested kernel operations. They are retrieved from the LanguageObject linked to the object O.

4. Assemble the result of the Op operation from the results produced by the kernel functions.

Defining as many kernel operation protocols as needed and associating them with objects allow programmers to create structural and behavioural diversity which can be uniformly handled.

3. The Functional Interface

O3 has five sorts of entities: worlds, objects, slots, values and operation protocols or language objects. An O3 world is a collection of objects

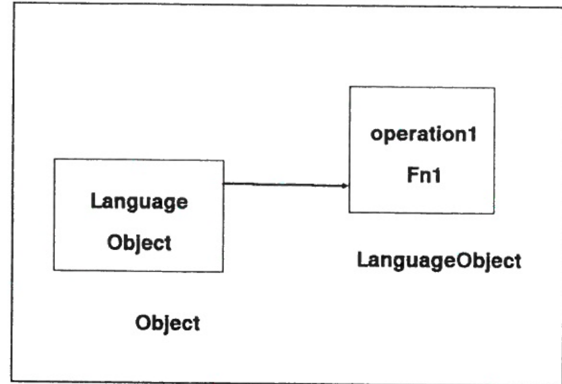

Figure 2. Dynamically Linking Operation Protocols to Objects

which can be manipulated by special operations. There exists an inheritance relation among worlds along which objects are inherited. Worlds offer an efficient storage mechanism for objects as an object is stored only once in its definition world and inherited without copying. Object modification in a world is marked locally, thus avoiding copying and allowing local versions to be created. Worlds are useful in search problems and can be backed up by a truth maintenance system. O3 provides an assumption based TMS [De Kleer 86], similarly to KEE [Filman 88, Fickes and Kehler 85], but this is outside the scope of this paper. As far as the other O3 entities are concerned, operation protocols are encoded in LanguageObjects while slots and values have the usual meaning in frame or object-oriented systems.

The functional interface provides three operations for each entity type: one creation operation, one query operation and one deletion operation. Table 1 shows all these operations. Each interface operation has a number of formal parameters for which actual values must be provided as well as a number of parameters for which default values exist. These defaults can be set by the programmer

Table 1: Operations of the Functional Interface

Entity	Creation	Deletion	Query
World	world	noWorld	world?
Object	object	noObject	object?

in different manners. O3 extensions can introduce new parameters.

Here are some examples concerning the use of the functional interface.

1. (world 'w :LanguageObject '*worldLanguage* :delete t :result 'node)

Creates a new world w, deletes any other world with the same name and returns the "node" created for the world, that is the O3 data structure created for the w world. The :LanguageObject parameter gives the special object storing the kernel operations which will be used upon this world. If omitted, an O3 supplied default will be attached.

2. (object 'o :world 'w :LanguageObject '*ObjectLanguage*)

Creates an object o in world w. If an object o already exists, it will be deleted or an error will be generated depending on the default value of the :delete option. The returned result is the default value of the :result option. As above, the :LanguageObject parameter gives the special object storing the kernel operations which will be used upon this object. If omitted, an O3 supplied default will be used.

3. (slot 's :object 'o :world 'w :result 'name)

Creates slot s of object o in world w and returns the name of the slot. The :object an :world parameters can be defaulted to the top values of two systems or user managed stacks.

4. (value 12 :slot 's :object 'o :world 'w :result 'node)

This puts 12 as a value of slot s of object o in world w. The result is the "node" - the O3 data structure - of the value (provided the used O3 version creates such nodes, otherwise an error is reported).

5. (object 'o

:world 'w :result 'slots

(s1 ('(12 13) :multiple t))

(s2 ("xxx"))

(s3 ('(a b c))))

This is a compact notation describing the creation of the o object in the w world. Three slots of o are also created, s1 s2 and s3 and values are also given to them. The :multiple option associated with a value indicates multiple values. Thus, slot s1 will receive two values, 12 and 13 and not the value (12 13). Slot s3 on the other hand has one value, (a b c). The :result option makes the expression return the data structure holding the slots of the object. In general this is a list, but other data structures are also possible, such as hash tables.

6. (noslot 's2 :object 'o :result 'values)

Deletes the s2 slot from object o in the current (default) world and returns the list of values of s2.

7. (novalue 13 :slot 's1 :object 'o

:value-delete-key 'k1

:value-delete-test 't1)

Removes value 13 from the values of slot s1 of object o. The :value-delete-key and :value-delete-test options supply the key and test functions used to remove a value from a sequence. This actually applies the Common LISP :key and :test mechanism to removing slot values.

8. (object 'o :options '(:slot-delete-key k1 :slot-delete-test t1))

This illustrates the use of the :options parameter. As shown in the previous examples, O3 has many options (:delete, :multiple, :result, :value-delete-key, etc.) to direct its operation. Options values can be supplied as actual parameter values (see the above example), but default values can also be specified for each object and world in part. This is done by the :options option at entity creation time, as illustrated. Option values given at operation innovation time override any existing defaults.

In general, the O3 interface operations permit the following several levels of default values, in their increasing order of precedence: (i) defaults fixed by implementation of all options, (ii)defaults specified at entity creation time, (iii)defaults specified at operation invocation time.

9. (object? 'o :result 'slots)

Returns the data structure holding the slots of o if o is an object and nil otherwise. (object? 'o) is an idiom for checking if o is an O3 object.

10. (options? 'o :slot-delete-test)

Returns the value of the :slot-delete-test option of object o.

11. (slot? 's :object 'o :inherit t :result 'node)

Returns the node of the s slot of object o. The inheritance operation is activated for retrieving the s slot (:inherit is usually t).

12. (execute 's :object 'o :control 'progn)

Executes any methods found as values in the s slot combining them with the progn combiner and returns the resulting value. This is the O3 way of activating methods.

4. The Prototype/Clone Mechanism

O3 is a classless language where an object can play at the same time the role of "class" for some objects and the role of "instance" of another object. This is achieved through a prototype/clone mechanism which allows an object to be copied for creating other objects. Copies can then be modified locally or get copied themselves for creating other objects.

Suppose O is an object. Then a clone (copy) O1 of it can be created using the assertion service as

(object 'O1 :prototype 'O).

This creates a new object O1 and links it with O through the prototype link. O is linked to O1 through a clone link. In this moment all queries concerning O1 are handled by first trying to find the requested information locally in O1 and, if not possible, by forwarding the query to O. Thus, O1 has all features of O except those locally overridden or removed. This mechanism allows one to simulate all features of a class/instance organization and define many other features.

An important use of this organization is in connection with the worlds system. In presenting it, we will first describe the most important features of worlds in O3.

An O3 world W can be created using the interface assertion service as, for example,

(world 'W :superworlds '(W1 W2)).

We will have W1 and W2 as superworlds inheriting objects from them. The object inheritance algorithm is of the "shortestpath" type, being identical to the default O3 algorithm for slot inheritance. When an object is created, the world mentioned at its creation time is considered its world of origin. Objects are inherited by subworlds of their world of origin. When an object is modified in its world of origin, all subworlds will inherit the modification. When an object is modified in a world where it was inherited, it is cloned in that world so that its definition in the superworlds of the cloning world (including the world of origin) is unaffected but subworlds of the cloning world inherit the modified version. This is implemented with the prototype/clone mechanism discussed.

5. O3 Kernel Operations

O3 objects and worlds point to special LanguageObject objects which hold kernel functions implementing O3 kernel operations. They are dynamically retrieved and executed by the previously discussed interface operations. The collection of kernel operations associated with objects forms the object protocol while the collection of operations associated with worlds forms the world protocol. The two protocols are stored in special objects named LanguageObjects.

Table 2 shows the object protocol and Table 3 the world protocol.

Creating an operation protocol is done by creating a LanguageObject and specifying the operations for which implementation functions are provided. Operations not explicitly mentioned are handled by the default implementation functions defined in Tables 2 and 3. For example, to create an object protocol which constructs and uses a hash table to store slots, the following LanguageObject can be created using the O3 service for creating LanguageObject-s:

(ObjectLanguage *ObjectLanguageHT*

:assert 'ObjectAssertHT

:slotquery 'SlotQueryHT

:slotassert 'SlotAssertHT

:slotdestroy 'SlotDestroyHT

:slotupdate 'SlotUpdateHT

:valuedestroy 'ValueDestroyHT).

The above expression defines the language object named *ObjectLanguageHT* which contains special functions implementing the mentioned kernel operations. Once created, this language object can be used to create objects behaving according to this protocol. To create such an object, one simply has to set the value of the :LanguageObject parameter to the desired language object as in the following example:

(object 'O :LanguageObject *Object LanguageHT*).

To create, remove and query language objects O3 provides a set of interface operations shown in Table 4.

6. Dispatching Kernel Operations

6.1. The Dispatch Mechanism

Functional interface operations call kernel operations through a dispatching mechanism which has two major features. First, it is dynamic in that the kernel functions implementing the kernel operations are retrieved at execution time. This allows dynamic modification of the set of kernel functions. Second, the dispatching process is carried out by a kernel meta-operation, ApplyOp, which - as any other operation - can be tailored by the programmer.

Interface operation call:

(object 'O :LanguageObject *Object LanguageHT*

:delete t

:result 'slots).

Dispatch sequence inside the interface operation :

(apply (ObjectLanguage-ApplyOp Language Object)

(ObjectLanguage-Assert LanguageObject)

name

parameters)

where *name* is bound to Oand *parameters* is bound to (:delete t :result slots).

In this example, LanguageObject is bound to the used language object, *ObjectLanguageHT*, name is the *name* of the new object and *parameters* is the list of parameters of the object assertion interface operation. The dispatch sequence takes the ApplyOp operation from the language object

Table 2: The Object Protocol

Interface operation	Kernel implementation fn	Purpose
Assert	ObjectAssert	Creates a new object
Destroy	ObjectDestroy	Destroys an object
ApplyOp	ApplyD	Meta operation for applying an operation
SendMessage	SendMessage	Activates methods

Table 3: The World Protocol

Interface operation	Kernel implementation fn	Purpose
Assert	WorldAssert	Creates a world
Destroy	WorldDestroy	Destroys a world

and applies it to a list of arguments containing the Assert operation also retrieved from the language object and the rest of arguments supplied to the object creation interface function. The ApplyOp operation is free to do whatever it desires with its arguments. In principle, the default O3 ApplyOp (named ApplyD) checks and applies demons and then applies the kernel operation (in the example assert).

Table 5 shows the dispatch sequences of the O3 interface operations and of some kernel operations which must also dispatch.

As concerns the overhead introduced by the dynamic dispatch sequence, we have measured it by comparing the standard O3 version against a specially configurated O3 system where dynamic dispatch has been eliminated (that is interface services directly called the kernel services with no language object looking for the kernel service). Tests run on both versions revealed the standard version as being only 4-5% slower. The careful implementation of the dispatch sequence and of the data structures for language objects accounts for this. On designing this, the extensibility costs have been saved considerably.

6.2. Issues of Optimization

Dispatching is a connection between the interface level and the kernel level. Kernel operations have to use services of the interface level to do their job, thus creating the reverse connection. This second connection is a source of inefficiency if achieved by directly calling interface services. Suppose a kernel operation needs have a list of slots of an object O. Issuing a request of the form

(object? 'O :result 'slots)

will yield the needed result, but will involve a lot of processing such as arguments checking, dispatching to the ObjectQuery kernel service, looking for demons, etc. An efficient way of acquiring the requested information is doing something like

(object-slots O).

Knowledge of implementing data structures of objects is nonetheless required. One can see that if kernel services are to make as few assumptions as possible about the rest of the system, one must adopt inefficient solutions, while if efficiency is planned the modularity and independence of components must be reduced.

The general solution is to start with a modular organization (few assumptions) and to optimize it manually or automatically. We have manually optimized our used versions. An intermediate solution to avoiding such optimizations would be to add a set of data structure accessors to Language Object-s to be used by all services in the language.

Table 4: Operations for Language Objects

Operation	Purpose
ObjectLanguage	Creates a language object for object operations
ObjectLanguage?	Queries a language object(for object operations)

7. Slot Inheritance

O3 provides a flexible slot inheritance capability allowing any relation to be used as an inheritance path and practically unlimited inherited value combination.

7.1. Inheritance Sources

An inheritance source is any slot S of an object O which has O3 objects as values and an attached ToInherit option. The value of this option must be a function able to return a sorted list of objects, called the InheritanceList. The objects will be those from which slots will be inherited in O

Table 5: Dispatch Sequences

Operation	Dispatch sequence
execute	ApplyOp + SendMessage
object	ApplyOp + Assert (object)
noObject	ApplyOp + Destroy(object)
slot	ApplyOp + SlotAssert
	ApplyOp + SlotDestroy

through the S relation. For illustration, consider the following definition of the ISA slot as an inheritance source:

(object 'O

(ISA '(O1 O2) :multiple t :ToInherit 'Shortest Path)).

In this example, object O inherits through slot ISA from O1 and O2. The ShortestPath function (supplied by the O3 kernel and used as default value for ToInherit) returns the transitive closure of objects reachable through ISA relations from O1 and O2, ordered according to a shortest path criterion also used in CLOS and other languages.

Users can define any number of inheritance sources. Inheritance comes from the SlotInherit kernel operation which implicitly assumes ISA as an inheritance source with ShortestPath inheritance.

7.2. Combining Inherited with Asserted Values

Suppose we interrogate slot S1 about its values:

(slot? 'S1 :object 'O :result 'values).

Provided that O has inheritance sources, they can contribute values to S1. Of course, this is possible even though S1 has already been asserted at the moment the above query takes place. In this case, the SlotInherit operation allows programmers to define the following aspects of inheritance.

1. The inheritance sources thereby values can be contributed to S1. This specification is made using the InheritThru option as in the following case:

(object 'O

(ISA '(O1 O2))

(S1 :InheritThru '(ISA))).

In this case, values for S1 can derive from the objects in the InheritanceList computed for the ISA slot. These values can be combined with the locally stored values in S1 in order to create a new set of values for S1.

2. Combining and storing inherited values. These aspects are specified using the IValuesCombine and IValuesMemo options as in the following example:

(object 'O

(ISA '(O1 O2))

(S1 :options (:InheritThru (ISA)

:IValuesCombine Max

:IValuesMemo Store))).

In this example, the inheritance service builds a list of all inherited values for S1, with the local values of S1 added to it and sends the list to the function Max. This function will return the maximum of the received arguments which will become the new value of slot S1. As option IValuesMemo has value Store, the newly computed value will be stored in S1 overriding old values therein. Instead of storing, the new values can be recomputed each time.

3. Inheritance of options. Slots can besides values inherit and combine options. This is specified as above, using the IOptionsCombine and IOptionsMemo options.

7.3. Inheriting Non-asserted Slots

When a slot S1 is inherited by an object which has no S1 slot, inheritance options must be specified in the query expression as follows:

(slot? 'S1 :object 'O

:inherit t

:InheritThru '(ISA)

:IValuesCombine 'Max

:IValuesMemo 'Store).

Note the :inherit option whose non-nil value triggers inheritance. Slot 'S1 is automatically asserted in O if inheritance is successful. Special values of the :inherit option determine inheriting only values, only options or both values and options.

8. Methods

An O3 method is a function appearing among the values of a slot. Hence, methods are treated like any other values (e.g. asserted, searched for, inherited, replaced, etc.). Methods are only treated differently when they are activated. Different ways of activating methods are shown next.

8.1. Activating a value-method

Let O be the following object

(object 'O

(S1 ('(12 f1 13) :multiple t)))

and let f1 be a function defined in the LISP environment. Function f1 can be activated as an effect of the following query expression:

(value? 'f1 :slot 'S1 :object 'O

:apply t

:args '(a b)

:replace t

:result 'result).

In this query, the :apply option indicates that if a function is retrieved, it must be applied. The :args option contains the actual arguments of the invocation and the :replace option specifies that the f1 value will be replaced in S1 by the result of activating f1 with the given arguments (which implies that next time f1 may not be found in S1). Finally, the :result option specifies that the query must return whatever f1 returns (but any other :result may be specified as well).

8.2. Activating All Methods in a Slot

The above mechanism can be extended to activate all methods found in a slot. For illustration consider the following object:

(object 'O

(S1 ('(f1 x f2 y) :multiple t

:options (:control 'and

:replace nil)))).

Suppose f1 and f2 are LISP functions and x and y global variables. The execute interface service can be used to activate all methods from slot S1 as in the following example:

(execute 'S1 :object 'O :control 'or :args '(11 12)).

The activation proceeds as follows:

1. Retrieve all methods in S1 and replace them by calls in the identified functions using :args as actual arguments. In the example producing the list is the result

((f1 11 12) x (f2 11 12) y).

2. Put in the first position of the above list the specified :control. Priority option values from actual calls yield the list

(or (f1 11 12) x (f2 11 12) y).

3. Execute the expression thus constructed and save its value.

The uniform treating of methods as values fits well in the inheritance scheme as inheritance can build lists of inherited/combined methods which can then be activated in various ways in the illustrated manner.

9. Demons

O3 demons are LISP functions automatically executable before, after or instead of any kernel operation. Demons are activated by the kernel meta-operation ApplyOp. The possibility for demonizing any O3 kernel operation (except ApplyOp) makes O3 demonization more general than in any language we are knowledgeable about. The possibility of using demons to dynamically replace any operation brings new flexibility to the system.

9.1. Specifying Demons

Demons can be either defined at object, slot or world creation time using the options mechanism, or specified among the arguments of interface operations, at the time such operations are used.

There exist three kinds of demons. Before demons are executed before a given kernel operation. After demons are executed after a given kernel operation. When demons are executed in place of a given kernel operation.

To specify demons at object, slot or world creation time the following options can be used:

(object 'O :options '(:demons t

:before-objectassert d1

:after-objectassert d2)).

The :demons option allows firing demons. Demons are specified by options whose name is formed by a prefix indicating the type of the demon (before-, after- or when-) and a suffix indicating the name of the demonized kernel operation. Thus, in the above example d1 is fired before the ObjectAssert operation and d2 after it.

To specify demons at interface operation use time, demons are similarly specified among the arguments of the interface operation. For example,

(slot? 'S :object 'O :when-slotinherit 'd3)

will activate demon d3 instead of the SlotInherit operation.

9.2. Activating Demons

Any demon receives the following arguments: (i) the object, slot or world on which the demonized operation is executed(the <node> below), (ii) the name of the demonized operation (e.g. SlotInherit) and (iii) the list of arguments of the demonized operation.

Demons are activated by applying the demon function to the above three arguments according to the scheme:

(apply <demon> <node> <op> <op-args>).

Thus, the demon function must have two required parameters - the object,slot or world and the operation - while the rest can be organized according to the expectations of the demonized operations. Also, this allows "when" demons to execute the demonized operation by another apply, according to the scheme:

(apply <op> <op-args>).

In this manner, "when" demons can control in any way the execution of the demonized operation.

9.3. Combining Demons

One can specify several demons of a given type by listing all demons under the associated option. In this case demons must be combined. The current implementation of ApplyOp does this in the following manner. "Before" and "after" demons are executed in the declared sequence. "When" demons are combined through a more complex procedure involving two steps.

In the first step a unique expression is built. For example, assuming that the declaration is

:when-slotinherit (d1 d2 d3),

the constructed expression will be:

(d3 <node> d2 (<node> d1 (<node> <op> <op-args>)))

which is a recursive application of the activation expression for a single demon. In the second step, the above expression is evaluated. In the example, this takes place as follows:

(i) first execute d3 with <op> = d2 and <op-args> = (<node> d1 ...)) (ii) now d3 can execute its operation (activating d2) by doing

(apply <op> <op-args>)

which will evaluate

(d2 <node> d1 (<node> <op> <op-args>))

which in turn activates d2 with

<op> = d1 and <op-args> = (<node> <op> <op-args>), a.s.o.

This mechanism delegates to a demon full control over the execution of the previous operation or demon. This may be used in many ways such as using demons to execute code sequences before or after the demonized operation. In the example, assuming that all demons contain such sequences, the global evaluation will effect into the following:

<before sequence d3>

<before sequence d2>

<before sequence d1>

<demonized operation>

<after sequence d1>

<after sequence d2>

<after sequence d3>

10. Creating O3 Languages

O3 was designed to explicitly support the view of programming as language construction plus language use. Language construction is supported at different levels in O3, from simple parameterizations to proper construction of new language mechanisms. Opportunities of customization through parameterization have already been illustrated. They consist in setting values for options either at entity creation time (with lower priority) or at service request time (with higher priority). This is useful due to the large range of options available. In this section, we will present the opportunities for constructing new language mechanisms and for integrating them in order to create new O3 languages.

10.1. Language Flavours

An O3 language flavour is a subset of kernel operations implementing a language feature. This subset can be placed in a dedicated language object. For example, a useful language feature has objects which store their slots in a hash table. This feature can be implemented by customizing several O3 kernel operations. These new operations can then be placed in a new language object - the flavour - which can be loaded in any O3 system needing this capability. In this particular case, the O3- HASHTABLE flavour currently used is the following:

```
(ObjectLanguage *ObjectLanguageHT*
:assert 'ObjectAssertHT
:slotquery 'SlotQueryHT
:slotassert 'SlotAssertHT
:slotdestroy 'SlotDestroyHT
:slotupdate 'SlotUpdateHT
:valuedestroy 'ValueDestroyHT).
```

Currently we build O3 flavours manually. The process requires identification of the kernel operations to be modified - identification through inspection - modification of these operations and the construction of the new language object. In the above case there were six kernel operations to be modified, the modifications affecting about 2-3 lines of code in each of them. For these reasons, creating the above flavour took less than two hours of work.

Other useful flavours include O3-RECORD and O3-VALUE-NODES. O3- RECORD stores slots as a record with static fields. The objects of this kind have a fixed (predefined) number of slots declared through a special option at object creation time. For this reason declared slots are automatically asserted and programmers cannot delete them or assert others. O3-VALUE-NODES is a flavour allowing objects to create a special data structure (a node) for each slot value. This may waste space, but is needed e.g. when meta-information must be associated with values. One such case is when using an assumption-based truth maintenance system [DeKleer 86]. In this case the value must be labeled with sets of assumptions maintained by the ATMS.

The use of flavours in an O3 system is straightforward. One simply has to load the files containing the requested kernel functions and the definition of the flavour. Then, any object created with a flavour as its :LanguageObject parameter value will behave according to the flavour.

10.2 Some Existing O3 Languages

In this section we tackle the design of two specialized O3 languages. In both cases the goal of the design is to provide the best support for implementing a specialized representation language.

The first problem is that of implementing an O3 language supporting a term-subsumption representation server. Term- subsumption languages (TSL-s) (such as KLONE [Brachman and Schmolze 85]) are well-defined frame languages allowing the description of concepts and roles by means of a finite set of descriptors whose semantics is axiomatically defined. A TSL- based server is an implementation of a TSL used as a general representational service in a large range of applications.

The major requirements for implementing a TSL can be stated as follows:

1. A large number (thousands) of concepts, roles and instances to be accommodated.

2. Fast access to concepts, roles and instances.

3. Usage of worlds and assumption-based TMS to allow non- monotonic reasoning.

The simplest solution that comes to the mind is to implement each concept, role and instance as an O3 object. This idea is misleading because very few features of O3 objects are actually needed and unused features will induce great overhead. For example, TSL concepts inherit in a very specific manner so that O3 inheritance would rather be completely avoided than tried to be adapted. There is no real need for methods, options, demons as far as TSL entities are concerned. As O3 objects occupy a rather large space, accommodating many of them may create storage space problems.

Because of these arguments, TSL entities are better encoded with customized data structures. On the other hand, some of the O3 features are still needed. These include the worlds mechanism and the ATMS which would provide a ready-made environment for producing alternative versions, supporting retractions and non- monotonicity. These features would have to be preserved after all.

We have adopted the solution of using the O3-HASHTABLE flavour for creating hash-table objects where TSL entities would be stored as slots. This ensures fast access. The concepts, roles and instances are coded as special data structures. The O3-NODE-VALUES flavour is also used to allow labelling by the ATMS. Furthermore, in order to get higher speed, we remove the standard ApplyOp service (removing the possibilities of demonization as well) for hash- table objects and customize SlotInherit to support only a primitive inheritance scheme.

The second problem to be discussed is that of designing an O3 language supporting the implementation of a constraint propagation system. Constraints are defined as predicates among given variables whose domains can be continuous or discrete sets. A set of values for all variables is a solution to the constraint satisfaction problem iff it satisfies all constraints. The constraint satisfaction process propagates values locally (within a constraint) and globally (among constraints which share variables) in order to reach a solution [Davis 87], [Sussman and Steele 80].

The requirements for a large number of constraints and fast access apply here as well. As in this constraint language users can define specific local propagation mechanisms, constraint propagation is usefully carried out by activating the attached methods. Constraints are both generic and instantiated and methods must be inherited from the generic constraints by the instantiated ones. These elements suggest that constraints should be implemented as customized objects providing method attachment and inheritance along a simple "instance-of" relation.

The solution will be to use the O3-RECORD flavour for creating objects with a fixed number of slots to model constraints. The sort of inheritance needed is supplied by the prototype-clone relation so the SlotInherit service may be dispensed with. Hash-table objects are again used as databases for storing constraints. Using a different (logical) TMS system dismisses the thought of node values. As in the previous example, many other smaller customizations are applicable to make the implementation better suited (e.g. placing caches handled by kernel operations in database objects).

Both examples are implemented creating and importing specific language flavours. Both implementations can co-exist in the same O3 environment together with the standard O3 version and possibly other versions, with the programmer using the same interface to all of them.

11. Comparisons

O3 uses a reflective architecture as a solution to the problem of creating a language able to provide the right programming constructs to each given problem. Similar motivations urge an effort to integrate extensive customization facilities in the KEE language [Filman 87]. This project tried to add such capabilities to an existing language. It relied on programmatic manipulations of the implementation in order to include some dispatch sequences which would apply alternative mechanisms. Being an afterthought rather than an initial design objective, the result is, we think, less clear than ours.

Another language whose motivations are not far from ours is 3-KRS [Maes 87]. The solution here is to make extensive use of meta- objects handling all service requests addressed to objects. This solution may function at several "meta" levels. We use a more restrictive form of language objects at

one level which guarantees efficiency and does not prevent flexibility. An explicit common interface level is more helpful than the general message passing access style of 3-KRS. Unfortunately, many of the examples in [Maes 87] are referring to mundane capabilities which can be implemented in many languages (e.g. tracing, stepping, etc.) rather than concentrated on harder issues. The basic concepts on reflective computation have been discussed in [Smith 82].

Not few researchers did attempt to build reflective languages as a way of improving expressive power in knowledge representation. RLL [Greiner and Lenat 80] is an earlier attempt based on a clever, flexible but lower level dispatching mechanism. Besides, while being used to make programming languages better suited to various task demands, reflective computation has also been applied to build more powerful problem-solving systems. The SOAR [Laird, Newell and Rosenbloom 87] problem-solving architecture uses reflection to build layered problem-solving strategies. Systems like CYC [Guha and Lenat 90] or FOL [Weyhrauch 80] duplicate the represented knowledge at two levels. The first is a clear predicate language to be used by the programmer while the second is a scruffy procedural language for efficient implementation. Reification makes the implicit procedural mechanisms explicit, allowing communication between these levels.

12. Conclusions

Ideally, programming languages should provide constructs which are both usable (for specific application domains) and reusable (for a large range of application domains). In order to provide both sorts of constructs, languages must be open to extension and customization. As the language aspects which have to be modified affect intimate language mechanisms, a rational language design having extensibility as an explicit purpose is needed in the first place.

We have proposed such a design based on two main ideas:

(i) a clear decomposition of the language into kernel, interface and application layers,

(ii) an economical way of dynamically linking objects to distinct and modifiable protocols of kernel operations.

With this architecture the language is causally connected to itself in an efficient manner and can integrate any number of "language flavours" implementing specific language features. The interface layer ensures uniform access to any language entity or service ensuring peaceful co-existence in spite of the differences. The standard O3 version provides default kernel services which can be overridden/modified in particular configurations. Default services for inheritance, method invocation and demon specification are provided in a very general form which opens up many opportunities for customization.

We have illustrated the use of the O3 language construction capabilities for implementing two significant knowledge representation languages with stringent demands on the implementation environment. O3 is now the carrier language for the knowledge system tools currently being built in our laboratory.

REFERENCES

BARBUCEANU, M., **Knowledge Based Development of Reusable and Evolutionary Software**, Proceedings of the Sixth International Workshop on Expert Systems and Their Applications, Avignon, France, 1986.

BARBUCEANU, M., TRAUSAN-MATU, S. and MOLNAR, B., **Integrating Declarative Programming Styles and Tools in a Structured Object AI Environment**, Proceedings of IJCAI-87, Milan, Italy, 1987.

BRACHMAN, R.J. and SCHMOLZE, J.G., **An Overview of the KLONE Knowledge Representation System**, COGNITIVE SCIENCE, 9(2), 1985.

DAVIS, E., **Constraint Propagation with Interval Labels**, ARTIFICIAL INTELLIGENCE, 32, 1987, pp. 281-331.

DE KLEER, J., **An Assumption-Based TMS**, ARTIFICIAL INTELLIGENCE, 28, 1986, pp. 127-162.

FICKES, R. and KEHLER, T., **The Role of Frame Based Representations in Reasoning**, COMMUNICATIONS OF THE ACM, 28, 1985, pp. 904-920.

FILMAN, R., **Reasoning with Worlds and Truth Maintenance in a Knowledge-Based Programming Environment**, COMMUNICATIONS OF THE ACM, Vol. 31, No.4, 1988, pp. 382-401.

FILMAN, R., **Retrofitting Objects**, Proceedings of OOPSLA-87, ACM Sigplan Notices, 1987, pp. 342-352.

GREINER, R. and LENAT, D., **RLL - A Representation Language Language**, Proceedings of AAAI-80, also Report HPP-80-9, Stanford Heuristic Programming Project, Stanford, CA., 1980.

KEENE, S., **Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS**, ADDISON-WESLEY PUBLISHING COMPANY, Reading, MA., 1989.

LAIRD, J.E., NEWELL, A. and ROSENBLOOM, P.S., **SOAR: An Architecture for General Intelligence**, ARTIFICIAL INTELLIGENCE, 33, 1987, pp. 1-64.

LENAT, D.B., GUHA, R.V., PITMANN, K., PRATT, D. and SHEPHERD, M., **CYC: Towards Programs with Common Sense**, COMMUNICATIONS OF THE ACM, 33, No. 8, 1990.

MAES, P., **Concepts and Experiments in Computational Reflection,** Proceedings of OOPSLA-87, ACM Sigplan Notices, 1987, pp. 147- 155.

SMITH, B., **Reflection and Semantics in a Procedural Language**, MIT, Lab. for Computer Science, Technical Report 272, Cambridge, MA., 1992.

STEFIK, M. and BOBROW, D., **Object Oriented Programming: Themes and Variations**, AI MAGAZINE, Winter 1986, 6 (4), pp. 40-62.

SUSSMAN, G.J. and STEELE, G.L., **Constraints: A Language for Expressing Almost Hierarchical Descriptions**, ARTIFICIAL INTELLIGENCE, 14, 1, 1980, pp. 1-39.

WEGENER, P., **Dimensions of Object Oriented Language Design**, Proceedings of OOPSLA-87, ACM Sigplan Notices, 1987.

WEYHRAUCH, R., **Prolegomena to a Theory of Mechanized Formal Reasoning**, ARTIFICIAL INTELLIGENCE, 13, Nos. 1,2, 1980.