RbCl: A Reflective Concurrent Language without a Run-time Kernel

Yuuji Ichisugi, Satoshi Matsuoka, Akinori Yonezawa
Department of Information Science, The University of Tokyo
{*ichisugi,matsu,yonezawa*}*@is.s.u-tokyo.ac.jp*

Abstract

We propose a reflective object-oriented concurrent language RbCl which has no run-time kernel. That is to say, all behaviors of RbCl except for what is restricted by the operating system and hardware can be modified/extended by the user. RbCl runs efficiently in a distributed environment and is intended for practical use. The execution of a RbCl program is performed by a *metasystem* that consists of *metalevel object*s. All the features of RbCl including concurrent execution, inter-node communication, and even reflective facilities themselves are realized by the metalevel objects. Important metalevel objects are called *system object*s, that are registered in *system object table*s. The user can change the behavior of the metasystem by replacing elements of system object tables with user-defined objects. RbCl also provides a novel feature called *linguistic symbiosis* with metalevel objects. All the metalevel objects in the initial RbCl metasystem are actually C++ objects, but linguistic symbiosis enables the user to manipulate metalevel C++ objects just as ordinary RbCl objects. Even reflective schemes and facilities themselves are realized by system objects that can be modified/extended by the user. Therefore, debugging of reflective programs and experiments on reflective schemes and facilities can be expressed and performed within the RbCl language framework.

1 Introduction

Reflection is a scheme that realizes highly flexible and malleable systems. A reflective system can manipulate data called *Causally-Connected Self Representation*(CCSR)[3] that represents the current state of its own computation. In a reflective system, the user program can manipulate its CCSR to change the computation of itself. The user can define new language features or change the representation of data structures of the language within the same language framework. Therefore, we believe that reflective systems are extremely useful as a platform for experimenting new language facilities and implementation techniques. Previous reflective systems, however, have the following problems:

- All previous reflective language systems are implemented over their *run-time kernel*s, that cannot be manipulated by the user. For example, the reflective tower of 3-Lisp[8][6] is implemented by code of hundreds of lines which acts as the run-time kernel. Although the semantics of the language may be altered using the reflective capabilities, the behavior of the run-time kernel cannot be changed by the user. This is a serious problem for the language users who are keen to efficiency: for example, a user may want to tune the language system to adapt to a specific application to improve its efficiency. The existence of run-time kernel, however, will restrict such modifications. Also, the underlying scheme of reflection, which is implemented by the run-time kernel, cannot be modified/extended by the user within the language framework.

- In many reflective systems, CCSRs that can be manipulated by the user does not expose the entire aspects of system implementation; rather, the CCSRs are abstracted so that the user can easily manipulate the system behavior. This restricts what aspects of computation the user can manipulate: for example, if the representation of garbage collection mechanisms is not included in the CCSR, the user cannot change the garbage collection scheme.

- In most reflective systems, the *reflective tower* observed by the user actually does not exist; rather, the run-time kernel makes the system act as if there is a infinite reflective tower. Because the reflective tower is far from the actual implementation, the user must understand the behavior of the run-time kernel to predict the amount of the CPU power and memory used by reflective programs. Furthermore, complex implementation techniques are required for efficient implementation in spite of their relatively simple metacircular definitions. This may cause various implementation problems for large systems (e.g., whether or not the semantics of the reflective tower is properly preserved).

In this paper, we propose a new reflective architecture and implementation techniques that alleviate the above problems. Our language RbCl (*Reflection Based Concurrent Language*) is an object-oriented concurrent language with a reflective architecture, and runs efficiently in a distributed environment. Specifically, our RbCl system has the following characteristics:

- A simple mechanism called *system object table* is introduced to remove the fixed run-time kernel at the level of the implementation language. In other words, all the run-time routines comprising the language system can be replaced by the user-defined ones. In the case of RbCl, the implementation language is C++; the user, therefore, can redefine the entire C++ program code to change the behavior of the system up to the restriction imposed by the operating system and hardware. Every possible run-time facility can be provided as libraries or applications written in RbCl, not as a dialect of RbCl.

- A novel facility called *linguistic symbiosis with implementation language* is introduced to make the CCSR completely be in accordance with the actual implementation. The linguistic symbiosis enables the user to manipulate an object of the implementation language in the same manner as an ordinary RbCl object. All language facilities initially realized by C++ objects, including concurrent execution, inter-node communication, program code management, memory management, etc., can be subject to modifications by the user. Even the reflective scheme themselves can be modified/extended by the user. Therefore, RbCl is highly useful as a platform for experimenting new language facilities and implementation techniques. Furthermore, the linguistic symbiosis allows efficient implementation of the reflective system itself.

- In RbCl, the reflective tower can be regarded as actually existing; that is to say, the reflective tower of RbCl is *the infinite tower of the direct implementation*. However, the *metasystem*, which is a system that realizes the execution of a system on each level, is created in a lazy manner. This creation of metasystems can be achieved without using the run-time kernel, by using the characteristic of *direct implementation*. Because the behavior of the reflective tower is completely defined by the CCSR of RbCl, the user can easily predict the efficiency of the reflective programs and can also modify/extend the behavior of the entire reflective tower.

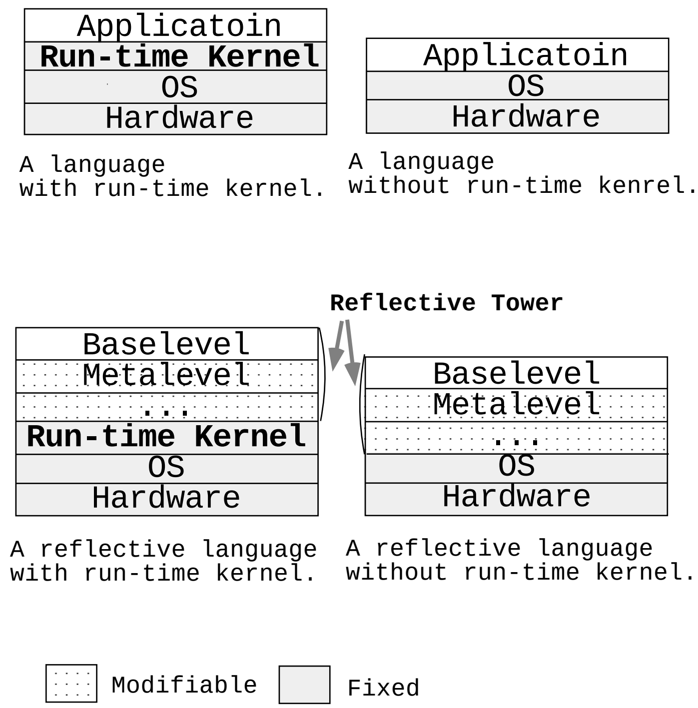

Figure 1: Language systems with/without run-time kernels.

- The metasystem of RbCl (that is, the CCSR of RbCl system as mentioned above) is designed based on an *object-oriented* and *layered* architecture, so that the user can easily modify/extend its behavior in an encapsulated manner.

- The reflective facilities of RbCl are implemented using only simple mechanisms of the implementation language: the only special mechanism required is the coroutine mechanism. The implementation techniques employed by RbCl can be easily applied to a wide range of language systems.

Since RbCl is a kernel-less system, all the features described here can be altered if the user desires so. We refer to the language initially provided for the user as *plain RbCl* to distinguish it from the modified/extended RbCl. Although this paper describes the characteristics of the plain RbCl in the strict sense of the word, we simply use the name "RbCl" except for the cases where we need to make the distinction.

The remainder of this paper is structured as follows. In Section 2, we describe the characteristics and benefits of kernel-less systems. Section 3 explains the facilities and the implementation of RbCl. In Section 4, we present examples that change and extend the reflective schemes themselves. Section 5 compares our system with related work. Finally, we summarize our work in Section 6.

2 Kernel-less system

RbCl is a *kernel-less system*. In this section, we describe the characteristics and benefits of kernel-less systems.

If a system written in a programming language L has a facility that enables the user to replace every part of its program code by a user-defined one, we say that the system is a *kernel-less system on language L.*

The following systems are examples of kernel-less systems: when a system's entire machine language instruction code is located in mutable store, and the system has a facility for modifying the values of contents of arbitrary addresses, the system is a kernel-less system on the machine language. The window system on a lisp machine is a kernel-less system on lisp because all the lisp functions defining the behavior of the system can be redefined by the user. A more elaborate example is as follows: the behavior of a reflective language is often defined by a *reflective tower*, that is, an infinite tower of metacircular interpreters. More specifically (e.g., in 3-Lisp), the behavior of the language at level n is defined by the interpreter at level n+1. The user can replace the entire interpreter code at level n using reflective computation at level n+1. Therefore, any system at level n is a kernel-less system on the language defined at level n+1. This is one reason why the reflective system is so flexible. However, since the entire 3-Lisp system must be implemented by another language for efficient implementation, the program code implementing the behavior of the reflective tower can never be modified by the user of the 3-Lisp system, so the 3-Lisp system is not kernel-less as it has a kernel running on the implementation language.

Most of the traditional language systems such as Lisp, Prolog, Smalltalk, etc., have run-time kernels to support their high level facilities. By contrast, machine language programs written in an assembler or C(++), etc., require no run-time kernel support.

RbCl is a kernel-less language system on C++, or, we can say "on machine language" because C++ has an ability to incorporate assembler programs, and C++ programs are compiled to machine language programs. This kernel-less system is realized by a simple mechanism called *system object table*s as described in Section 3.3. As a result, the user can redefine the entire C++ program code to change the behavior of the system up to the restriction by the operating system and hardware (Fig. 1). Thus, concurrent execution, inter-node communication, program code management, memory management, and even reflective schemes and facilities can be modified/extended by the user. In usual language systems, the trade-offs between various characteristics such as efficiency, flexibility, programmability, safety, portability, etc., are achieved only by the system implementer. In RbCl, the balance of the trade-offs between all such characteristics can also be changed by the language user.

A problem with kernel-less systems, such as a machine language program in mutable store (as described above), is that it is often extremely difficult and dangerous to change the behavior of the system dynamically: as a result, they cannot be used for practical purposes. In RbCl, its reflective facilities and linguistic symbiosis enable the user to easily modify/extend behavior of the system thanks to the encapsulation provided by the object-oriented nature of RbCl.

3 Design and Implementation of the RbCl Metasystem

In this section, we describe the main facilities provided by RbCl and the design and implementation of the RbCl metasystem.

3.1 Characteristics of RbCl Objects

The basic features of RbCl are similar to that of ABCL/1[12] except for its reflective facilities. Each object is the unit of concurrency and scripts/methods are executed in a usual, sequential imperative manner (no internal concurrency in a stateful object is allowed). Currently, we use a subset of

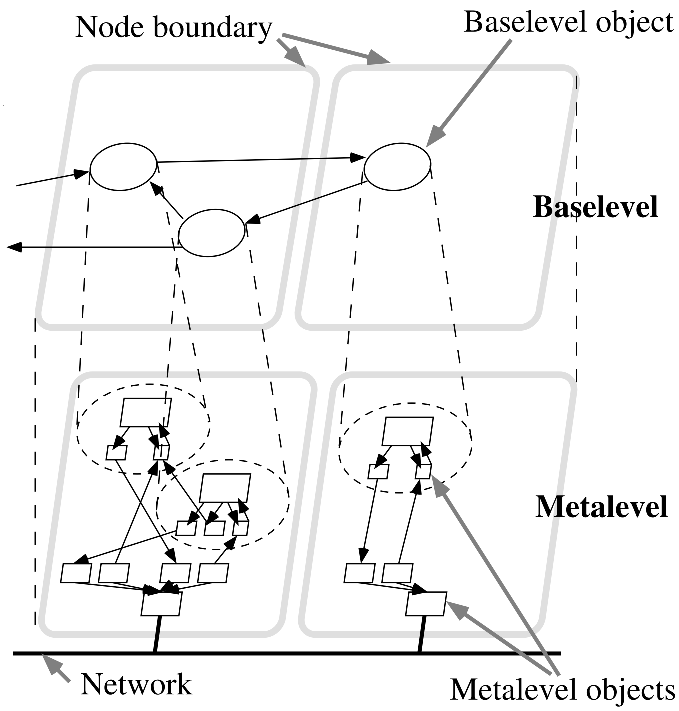

Figure 2: The baselevel and the metalevel in RbCl.

Common Lisp to describe the sequential behavior of each object. Objects are dynamically created and they interact with each other only by sending messages. RbCl provides both synchronous and asynchronous types of message sending.

3.2 Linguistic Symbiosis

RbCl provides a novel facility called *linguistic symbiosis* with C++ objects. It hides the implementation gap between RbCl objects and C++ objects by allowing transparent inter-communication within the same memory space. A RbCl object can regard a C++ object as a RbCl object, and conversely, a C++ object can regard a RbCl object as a C++ object. The user need not be conscious of the difference of these languages. When communicating between C++ and RbCl, each uses its own communication protocols: a RbCl object communicates with a C++ object by the RbCl message passing protocol, and a C++ object communicates with a RbCl object via C++ virtual function invocation. The implementation scheme of the linguistic symbiosis is described in Section 3.6. As described in Section 3.4, the linguistic symbiosis plays an important role in implementing the infinite reflective tower with finite computing resources without any run-time kernel.

3.3 Metasystem

The client RbCl program resides in the *baselevel*. The *metasystem* is a *metalevel* system that realizes the execution of the baselevel RbCl program (Fig. 2). The metasystem consists of *metalevel objects*. Likewise the execution of the metalevel objects is realized by the meta meta system and so on ad infinitum, forming a *reflective tower*.

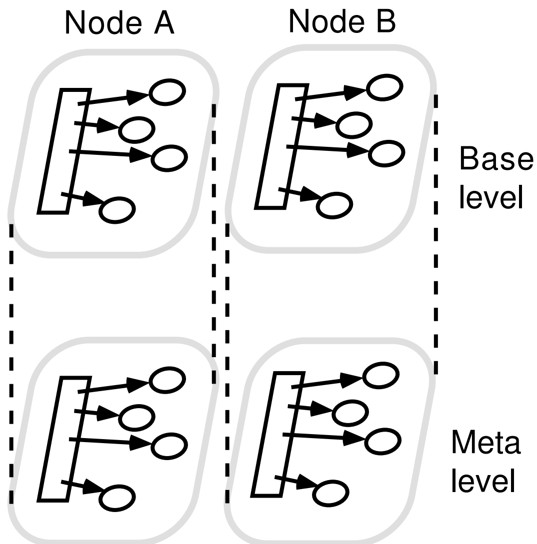

Figure 3: Each level of each node has its own system object table.

The RbCl system consists of *node*s which are units of resource sharing such as CPU power and memory. Each node has its own reflective tower. If a node is implemented on a shared memory architecture machine, multiple threads may run simultaneously within the node. Otherwise, the metasystem is executed sequentially; in this case, concurrent execution of the baselevel objects becomes pseudo-parallel. In programming the baselevel, the user need not be aware of the distributed nature of the architecture. However, the design of the metalevel takes node boundaries into account. In the metasystem of the plain RbCl, there are no inter-node references between metalevel objects. The inter-node communication between baselevel objects is realized at the metalevel using system calls provided by the operating system.

Each level of each node has its own *system object table* (Fig. 3), which plays a crucial role in realizing the reflective facilities of RbCl. A system object table realizes a name space of *system object*s. System objects are important metalevel objects that determine the basic behavior of the baselevel, and are recorded in the system object table of the metalevel. For example, the metalevel objects that determine the global behavior of the baselevel, such as schedulers, active queues and the network daemons, are all system objects. The *generator*s are also system objects that generate metalevel objects such as parts of baselevel objects, or primitive data element such as `cons` cells, etc. System objects are referenced to by name from other objects within the same node and level. The user can change the behavior of the baselevel on a node by replacing elements of the system object table of the metalevel on the node.

Although the plain RbCl metasystem consists of only C++ objects, the behavior of each object is carefully designed so as to be independent of C++ language features as much as possible. For example, global variables and global functions of C++ are not used; rather, system objects are employed for encapsulating global data and operations. C++ object creation constructs such as `new class_foo()` are not used directly; instead, generators are used for the purpose. For instance, when a system object named `cons_generator` receives a message, it generates a metalevel object that represents a `cons` cell.

The user can replace arbitrary system objects with user-defined ones, that may be either C++ objects or RbCl objects (Fig. 4). The user need not be conscious of the differences between these

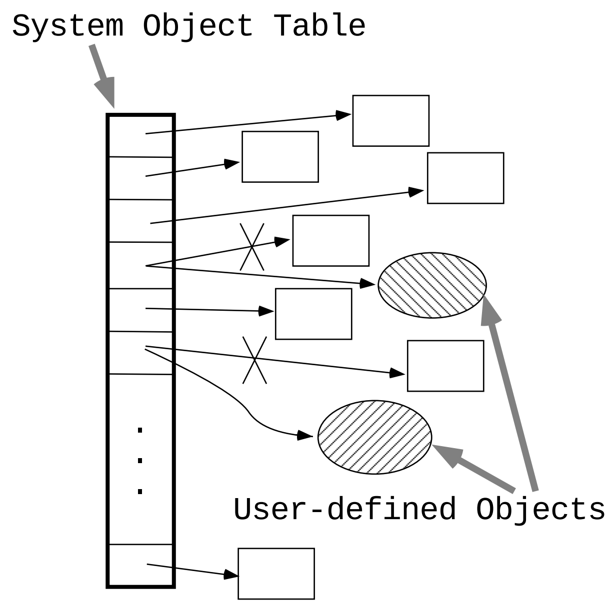

Figure 4: The user can replace all system objects.

languages thanks to the linguistic symbiosis.

The RbCl metasystem is constructed in a layered manner. In other words, the metasystem consists of several layers corresponding to the degree of abstraction. Program modules that depend on the hardware architecture are in the lower layers, and program modules that depend on specific features are in the higher layers. The layered architecture enhances portability and extensibility of RbCl.

3.4 The Infinite Tower of Direct Implementation

Without run-time kernels, the previous reflective systems would be impossible to implement efficiently with finite computing resources. A run-time kernel is usually necessary to make the system act as if an infinite reflective tower exists. In contrast, a RbCl system, which embodies a tower, is implemented with finite computing resources without a run-time kernel. This is achieved by employing the linguistic symbiosis with C++ objects, that can be executed without a run-time kernel. The reflective tower of the plain RbCl is *the infinite tower of the direct implementation.* In this section, we describe the reflective tower of RbCl.

In reflective systems, baselevel entities do not exist at the metalevel in the strict sense of the word; that is to say, there exist only the metalevel entities that *represent* the baselevel entities. For example, imagine a lisp interpreter written in C. Although there are `cons` cells in lisp, there are none in C; rather there are C structures that represent the lisp `cons` cells. In the same way, there are no C structures at the machine language level, but rather there is storage whose contents represent the C structures. We can go far as to say that, there is no storage at the hardware level, but rather, there are electronic entities that represent the storage. When a lisp program is running, all the levels are active at the same time, but usually one pays attention to one level at a time to understand the behavior of the system.

Let us pay attention to the C level interpreting lisp. Generally, there are two methods for implementing a language facility over another language system: *direct implementation* and *explicit*

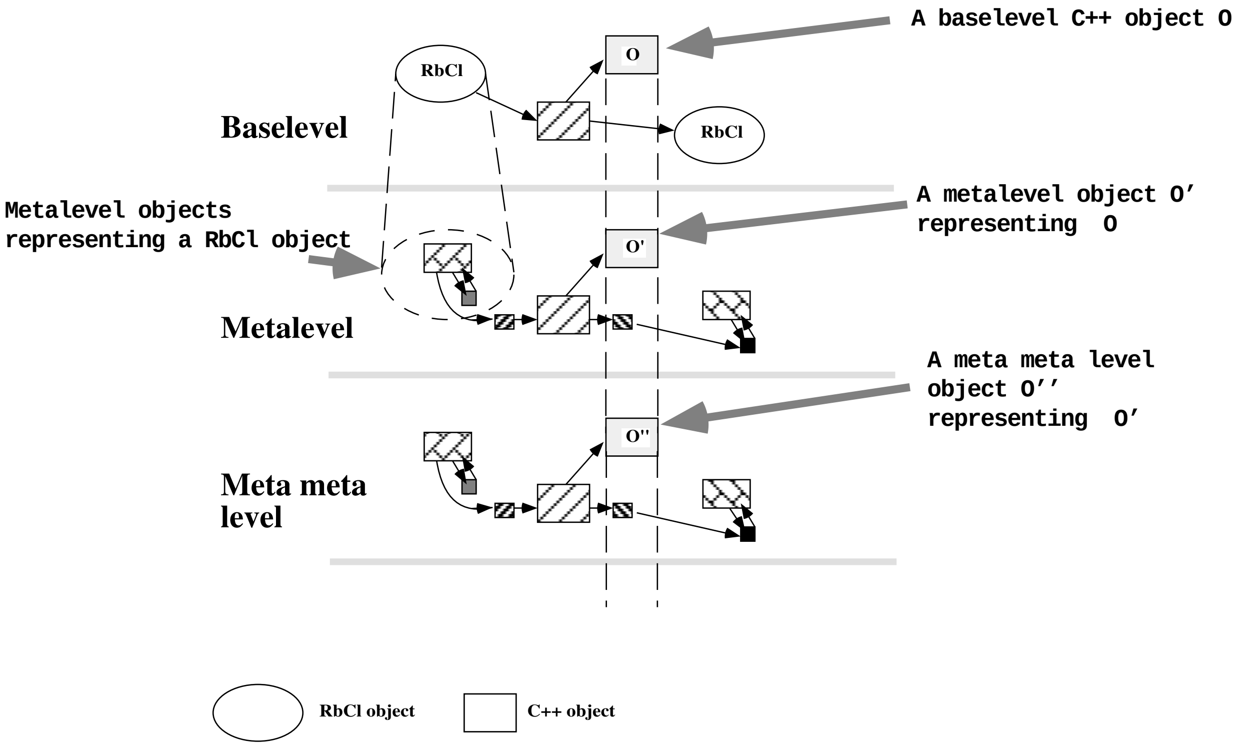

Figure 5: The infinite tower of the direct implementation.

implementation [6]. For example, when the '+' operation of lisp is implemented by using only the '+' operation of the C language, we say that "the '+' operation is directly implemented." In this case, the semantics of the '+' operation of lisp fully depends on that of C. On the other hand, if the meaning of the '+' operation is defined explicitly using the usual primitive recursion scheme, we say that "the '+' operation is explicitly implemented."

Analogously, let us pay attention to the metalevel of RbCl that implements the baselevel of RbCl. RbCl provides linguistic symbiosis that enables C++ objects to be executed as baselevel objects. In the plain RbCl metasystem, the execution of the baselevel C++ objects are directly implemented. This is achieved as follows(Fig. 5): a baselevel C++ object O is represented by a single metalevel object O' (in contrast to a baselevel RbCl object which is represented by multiple metalevel objects). O' has instance variables and methods identical to those of O. Therefore, a message sending to O at the baselevel can be simply represented by a message sending to O' at the metalevel.

Let us pay attention to the meta meta level of RbCl that implements the metalevel of RbCl. There are meta meta level C++ objects that represent metalevel C++ objects. For example, the metalevel object O' is represented as a meta meta level object O" that has instance variables and methods identical to those of O'. Because the plain RbCl metasystem consists only of C++ objects, the computation state of the meta meta level is strictly identical to that of the metalevel. In this way, the meta meta level of the plain RbCl can be regarded as actually existing and directly implementing the execution of the metalevel C++ objects. (Fig. 5 illustrates this whereby each C++ object is represented by a meta level C++ object that has the same hatch pattern.)

The meta meta level itself is also regarded as directly implemented by the meta meta meta level and this tower of direct implementation continues on infinitely. In this manner, the reflective tower of RbCl is realized with finite computing resources.

According to this view, arbitrary (non-reflective) systems could be regarded as being implemented by the infinite tower of direct implementation. However, such view is usually meaningless because the user of the such systems cannot manipulate the lower levels. On the other hand, the reflective facilities of RbCl enable the user to manipulate the arbitrary levels if the user desires. The meta meta system that interprets RbCl objects at the metalevel is explicitly generated by a metasystem when a user creates a RbCl object at the metalevel for the first time. Therefore, the infinite tower of direct implementation of RbCl has the same power as the reflective towers of other reflective systems. How the meta meta system is generated by the metasystem is described in Section 3.7.

3.5 Level Shifting by Level Managers

The linguistic symbiosis enables C++ objects to be executed at the baselevel. As explained in Section 3.3, each level has its own system object table that realizes a name space of system objects. Consequently, the baselevel C++ objects must be executed in the baselevel's name space. The management of the name space is conducted by system objects called *level manager*s. Each level of each node has its own level manager. The level manager of the metalevel performs *level shifting*, that is, switching the current name space of the system objects between the baselevel and the metalevel. The level manager of the meta meta level similarly performs the level shifting between the metalevel and the meta meta level. The linguistic symbiosis and all the reflective facilities such as creating metalevel RbCl objects are implemented using the primitive level shifting capability provided by the level managers.

Note that the level manager only switches the name spaces. Wherever the name space of the metalevel is shifted, the baselevel is still executed by the metasystem and the metalevel is still executed by the meta meta system and so on.

The level shifting mechanism is implemented as follows: a C++ object refers to a C++ global variable to learn the appropriate system object table that represents the current name space. C++ objects access a system object as follows:

```
system_object_table[symbol_id]
```

`System_object_table` is a C++ global variable which is a pointer to an array of C++ objects. This array represents the system object table. `Symbol_id` is a small integer that represents the global name of a system object. The level manager changes the value of the C++ global variable `system_object_table` to an appropriate system object table when it receives messages `:shift-to-meta` or `:shift-to-base`.

Level managers are system objects, and thus can also be replaced with user-defined objects. Therefore, programs that require changes to the reflective facilities of the RbCl itself, such as debugging of reflective programs, or performing experiments on reflective facilities, can be expressed within the RbCl language framework. In Section 4 we will explain this in detail.

3.6 Implementation of the Linguistic Symbiosis

In order to realize the linguistic symbiosis, the implementation gap between RbCl and C++ is absorbed by metalevel objects called *intermediate pointer*s (Fig. 6) that perform (1) conversion of message passing protocols between the two languages, and (2) level shifting by sending messages to the level manager of the metalevel. Each reference between baselevel RbCl objects and baselevel C++ objects is represented by an intermediate pointer object at the metalevel.

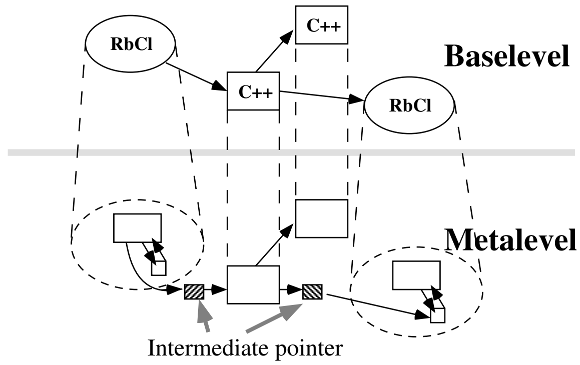

Figure 6: Implementation of the linguistic symbiosis.

An intermediate pointer also manages the coroutine facility using the thread library of the C language. All baselevel RbCl objects are interpreted at the metalevel on one thread, and baselevel C++ objects are directly implemented by metalevel C++ objects that run at the metalevel on the other threads. When a baselevel RbCl object sends a message to a baselevel C++ object, the intermediate pointer at the metalevel switches the active thread to a new thread where the corresponding metalevel C++ object runs. If control is passed back to a baselevel RbCl object, the interpreter thread at the metalevel is reactivated. Message passing between C++ objects is performed efficiently with an ordinary virtual function invocation without using intermediate pointers.

3.7 Generating the Meta Meta System

Many reflective systems implement lazy creation of metasystems within the run-time kernel. The RbCl metasystem can generate the meta meta system within the RbCl language framework — that is, it does not require a run-time kernel to do so. This is achieved as follows: first, let us define an object to be *primitive* if it is implemented only by direct implementation. For example, C++ objects are primitive objects while interpreted RbCl objects are not. As explained in Section 3.4, when a primitive object runs at the $(\text{meta})^n$ level, the same primitive object actually runs at the $(\text{meta})^{n+1}$ level. For example, when a metalevel primitive object A creates another metalevel primitive object B and sends a message to B, the corresponding meta meta level object A' actually creates a corresponding meta meta level object B' and sends a message to B'.

The meta meta system can be generated by the metasystem using this characteristic of primitive objects. To briefly summarize, the only thing needed to be done is to generate a new metasystem that consists of only primitive objects.

The meta meta system is generated by the metasystem in the following way:

1. Create an array of objects that represents the system object table of the meta meta system.

2. Shallow copy all the elements from the `default-system-object-table` to the system object table of the meta meta system. `Default-system-object-table` is a system object that contains the system objects of the plain RbCl metasystem that are shared by all the levels. Most system objects such as generators have no internal states, so they can be shared.

3. Create additional objects and register them to the system object table. These objects are system objects that cannot to be shared by all the levels — examples are a level manager and

a scheduler.

4. Initialize the created system objects as if the execution of the metalevel were directly implemented by the meta meta system.

The generated meta meta system provides all the facilities provided by the plain RbCl metasystem. When a RbCl object is created at the meta meta level, the meta meta meta system will be generated in the same way.

4 Examples of Modifying the Reflective Schemes

4.1 Replacing level managers

The reflective facilities of RbCl itself can be modified/extended or even completely changed by the user by using reflection. If the user wants to print messages to the console whenever the current level shifts between the baselevel and the metalevel, the user can replace the default level manager with a user-defined level manager which prints out a message whenever the level shifts. This is achieved by the following RbCl program.

```
(metalevel-exec
 (setf (G level-manager)  a-user-defined-level-manager))
```

`Metalevel-exec` is a macro form to execute expressions in the metalevel environment. `(G level-manager)` denotes a system object named `level-manager`.

The above example only changes the level manager at the metalevel. In addition to this, the user can change all the level managers of the metasystems created in a lazy manner. As described in Section 3.7, new metasystems are created using the elements of `default-system-object-table`. Level managers of the new metasystems are created by the system object named `level-manager-generator`. The user can replace this element of `default-system-object-table` as follows:

```
(metalevel-exec
 [[(G default-system-object-table)
   <== [:aset (symbol-id 'level-manager-generator)
                a-user-defined-level-manager-generator]]])
```

After executing this code, when a new metasystem (the meta meta system, the meta meta meta system, etc.) is generated, a level manager is generated using the user-defined level manager generator.

There is one caveat in this example: when the user replaces the `level-manager-generator` with an interpreted RbCl object, in practice it will never be used because the new meta meta system will have already been generated using the old value of `default-system-object-table` when the user creates the interpreted RbCl object. To avoid this situation, in the next example, we introduce the (meta)$^\omega$ level which enables the user to replace the elements of the `default-system-object-table` with interpreted RbCl objects.

4.2 The (Meta)$^\omega$ Level

The behaviors of 3-Lisp programs are defined by the infinite reflective tower of meta-circular interpreters. In practice, for efficient execution, the reflective tower of 3-Lisp is realized by a run-time

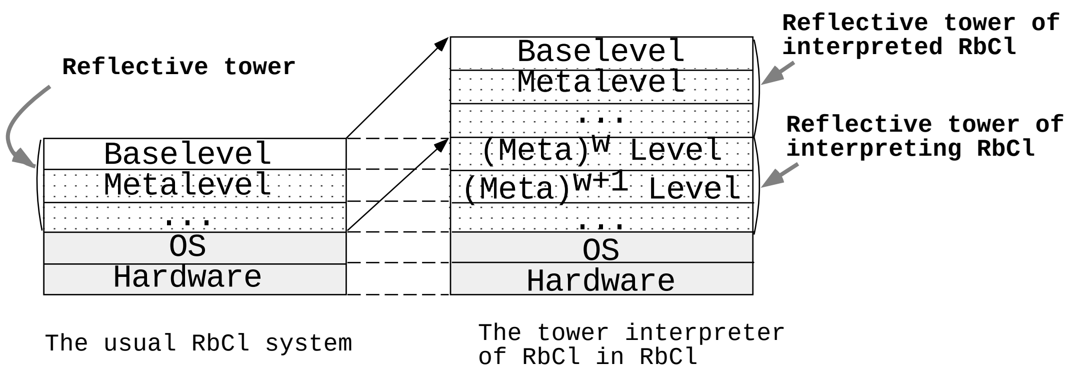

Figure 7: An interpreted reflective tower and (meta)$^\omega$ level.

interpreter kernel written in another language L. We refer to such an interpreter as the *tower interpreter of 3-Lisp in L*. A tower interpreter of 3-Lisp can also be written in 3-Lisp itself (without using its reflective facilities)[6]. This system would be the tower interpreter of 3-Lisp in 3-Lisp.

In the same way, we can write a tower interpreter of RbCl in RbCl. As explained in Section 3.4, the reflective tower of the plain RbCl is implemented on the C++ language, so the plain RbCl metasystem is a tower interpreter of RbCl in C++. Obviously we can use the RbCl language itself to write a tower interpreter of RbCl: we call this system the tower interpreter of RbCl in RbCl (Fig. 7). The reader should not confuse two RbCl systems, the *interpreted RbCl* and the *interpreting RbCl*. The interpreted RbCl is implemented by the tower interpreter, and the tower interpreter is only an application program of the interpreting RbCl. The baselevel of the interpreting RbCl may be called *the (meta)$^\omega$ level* of the interpreted RbCl, because it implements the entire reflective tower of the interpreted RbCl. The programming at the (meta)$^\omega$ level allows, for example, debugging of reflective programs, experiments on reflective facilities using RbCl, rather than using other low level programming languages such as C++. New reflective systems — for example, a multi-user system supporting each user to have his/her own reflective tower — could be efficiently realized within the RbCl framework.

Fig. 8 is an example of tower interpreter in RbCl. `Obj-code` is the program code of the start-up object [1]. `(G metasystem-generator)` denotes a system object that generates a new metasystem by the scheme described in Section 3.7. `Metasystem-generator` returns the level manager of the generated metasystem when it receives a `[:call]` message. We can implement a complete tower interpreter in such a few program steps because `metasystem-generator` has an ability to create a complete metasystem explicitly.

It is likewise possible in other reflective language systems to write the tower interpreter of itself. However, it would require hundreds of lines of program code to implement the run-time kernel explicitly. Furthermore, the execution of interpreted reflective tower would be much slower than the original one, and the user must change the program code of run-time kernel (in an ad hoc way) to experiment with new reflective facilities. In contrast, the execution speed of programs in the interpreted RbCl would be as fast as the interpreting RbCl, because the entire reflective tower would be directly implemented by the (meta)$^\omega$ level. Furthermore, the user could replace some elements of `(G default-system-object-table)` with (meta)$^\omega$ level RbCl objects to modify the behavior of the entire reflective tower. So, for example, the user could use RbCl to implement a debugger of

[1] The RbCl metasystem represents program code of RbCl objects as metalevel objects.

```
(defun Tower-in-RbCl (obj-code)
  ;; This level is the (meta)^(omega) level
  ;; interpreted by the (meta)^(omega+1) level.
  (let (level-manager
        obj)
    ;; Generate a new metasystem.
    (setq level-manager
          [(G metasystem-generator) <== [:call]])
    ;; Shift to the generated level.
    [level-manager <== [:shift-to-meta]]
    ;; Now, this level is the new metalevel
    ;;still interpreted by the (meta)^(omega+1) level.
    (setq obj [(G make-local-object) <== [:call obj-code]])
    [(G active-queue) <== [:enqueue obj]]
    ;; Start interpretation of the new baselevel.
    [(G scheduler) <== [:call]]
    ))
```

Figure 8: A tower interpreter in RbCl.

reflective programs rather than using other low level programming languages such as C++.

5 Related Work

An important difference between RbCl and the other reflective systems is that RbCl is a kernel-less system. Furthermore, there are many differences between RbCl and the other reflective systems. In this section, we compare the reflective facilities provided by other systems to those of RbCl.

The CCSR of 3-Lisp[8][6] is only the *current expression*, the *continuation* and the *environment*, and do not include detailed information of the system. In RbCl, detailed CCSR is provided by system objects. The user can, therefore, modify/extend the behavior of the system with precision. The modification can be easily done thanks to the *object-oriented* and *layered* architecture of the RbCl metasystem.

In 3-KRS[3] and ABCL/R[9], all the objects are defined by its metaobjects. The user can change the behavior of an object by modifying its metaobject. A metaobject itself is also an object, so it has another metaobject, and this chain continues to infinity comprising multiple *individual towers*[5]. A plain RbCl baselevel object is defined by the corresponding metalevel C++ objects. The user cannot change the definition of a C++ object, but that is not necessary because all the metalevel objects have an opportunity to be modified/extended by replacing their generators. This design choice of RbCl makes it possible to implement the system efficiently and naturally.

Muse OS[10, 11] is a reflective object-oriented operating system while RbCl is a reflective language system. Like RbCl, Muse OS runs efficiently in distributed environments, and the user can modify/extend almost all parts of the system. However, there are the following clear differences from RbCl, mainly stemming from the nature of their respective origins:

- A small kernel called *MuseCore* manages the primitives of reflection. The behavior of the MuseCore cannot be manipulated by the user. In this sense, Muse is not a kernel-less system (although Muse allows some customization of reflective behaviors by subclassing new *reflector* classes).

- Communication between the baselevel Muse objects is achieved by the objects at the metalevel, always requiring kernel traps. In RbCl, communication between the baselevel C++ objects are efficiently performed with normal virtual function invocations. Therefore, no level shifting takes place.

- Muse objects at the baselevel are represented as raw data at the metalevel. In RbCl, all the baselevel objects are represented as metalevel objects to which messages can be directly sent from the metalevel, as is with ABCL/R.

CLOS[2] is an object-oriented system that provides metaobject facilities to modify/extend the behavior and implementation of objects. The metaclass of a class is actually an object which creates the class. The user can modify/extend the system by specifying customized metaclasses. Since CLOS has only one name space, it is difficult to change the system's global behavior without losing the system's consistency. In RbCl, modification of the baselevel does not affect the behavior of the metalevel, so it is easy to make experiments on the language facilities.

ABCL/R2[4] is a reflective object-oriented concurrent language designed to manipulate computational resources such as computing power. The metalevel of ABCL/R2 is constructed by concurrent objects and their scheduling policy is not defined exactly by its CCSR. In RbCl, the scheduling policy within each node is defined by the metalevel, and therefore the user can control the system behavior more precisely.

AL/1 [1] is an object-oriented concurrent language. AL/1 is based on the *multi-model* reflection framework. AL/1 provides multiple CCSRs that are suitable for various purposes in modifying the behavior of the interpreter, resource management, etc. Although RbCl provides only one CCSR, it embodies all the modules implementing concurrent execution, inter-node communication, and even reflective facilities themselves. Therefore, the user can modify/extend all the behaviors of RbCl.

Rose proposes a metaobject protocol for dynamic dispatch[7] that is efficient, powerful and language independent. However, Rose's metaobject protocol only supports the dynamic dispatch mechanism, and does not support other facilities such as concurrent execution as RbCl does.

6 Conclusion

RbCl realizes the reflective tower efficiently with finite computing resources without a run-time kernel. This is achieved by employing a simple mechanism called *system object table*s and a novel facility called *linguistic symbiosis* with the C++ objects.

Because RbCl is a kernel-less system, the user can change the behavior of the system up to the restriction imposed by the operating system and hardware within the RbCl framework. This implies that every possible run-time facility can be provided as libraries or applications written in RbCl, not as a dialect of RbCl.

The system object table of the metalevel on a node is the CCSR of the baselevel on the node. The system objects that realize the basic behavior of the baselevel are the elements of in the system object table of the metalevel. The benefits of the system object tables are as follows:

1. The system object table of the metalevel is the mechanism that makes the RbCl metasystem a kernel-less system. All the system objects are indirectly accessed through the system object table. Therefore, the user can change the behavior of the metasystem by replacing elements of

the system object table with user-defined objects. The overhead of this indirection is negligible because access to a system object can be achieved in only a few instruction steps.

2. The system object tables represent the name spaces of system objects that are equivalently accessed by various languages such as C++ and RbCl.

3. Level shifting, that is, switching the current name space of the system objects, can be efficiently implemented.

The benefits of the linguistic symbiosis with the C++ objects are as follows:

1. An efficient reflective system can be easily implemented. The metasystem can be constructed only by the C++ objects that are efficiently executed, and the user can manipulate the metalevel C++ objects just as RbCl objects.

2. The linguistic symbiosis also serves as a foreign language interface to the C(++) language that enables the user to directly use the system calls provided by the operating system.

It should be noted that the implementation techniques used by RbCl are easily applicable to other systems. The reflective facilities of RbCl are implemented using only simple mechanisms of the implementation language: the system object table is only a simple array of objects, and the linguistic symbiosis requires only the coroutine facility for its implementation.

Acknowledgments

The authors wish to thank to Takuo Watanabe and Hidehiko Masuhara for numerous helpful discussions.

References

[1] Ishikawa, Y., "Reflection Facilities and Realistic Programming," SIGPLAN Notices, Vol.26, No.8, Aug.1991, pp.101-110.

[2] Kiczales, G., des Rivières, J. & Bobrow, D. G., The Art of Metaobject Protocol, MIT Press, 1991.

[3] Maes, P., "Concepts and Experiments in Computational Reflection", In Proc. OOPSLA '87, ACM, pp. 147-155, 1987.

[4] Matsuoka, S., Watanabe, T. & Yonezawa, A., "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming", in Proc. ECOOP '91, pp. 231–250, Lecture Notes in Computer Science, 512, Springer, 1991.

[5] Matsuoka, S., Watanabe, T., Ichisugi, Y.& Yonezawa, A., "Object-Oriented Concurrent Reflective Architectures," In Proc. of ECOOP Workshop on Object-Based Concurrent Programming, Geneve, Switzerland, July, 1991, to appear in a LNCS, 1992.

[6] Rivières, J., Smith, B. C., "The Implementation of Procedurally Reflective Languages," Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, 1984.

[7] Rose, J., "A Minimal Metaobject Protocol for Dynamic Dispatch", In Proc. of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, October, 1991.

[8] Smith, B. C., "Reflection and Semantics in Lisp", In Conference Record of ACM POPL '84, pp. 23-35, 1984.

[9] Watanabe, T. & Yonezawa, A., "Reflection in an Object-Oriented Concurrent Language", In Proc. ACM OOPSLA '88, pp. 306-315, 1988, (revised version in [12]).

[10] Yokote,Y.,Teraoka,F.,and Tokoro,M., "A Reflective Architecture for an Object-Oriented Distributed Operating System," Proceedings of European Conference on Object-Oriented Programming, July, 1989.

[11] Yokote,Y., "The Muse Reflective Operating System: The Concept and Its Implementation", to appear in Proc. of OOPSLA'92, ACM, October, 1992.

[12] Yonezawa, A.(Ed.), ABCL: An Object-Oriented Concurrent System, MIT Press, 1990.