Object-Oriented Concurrent Reflective Architectures [*]

Satoshi Matsuoka Takuo Watanabe Yuuji Ichisugi
Akinori Yonezawa
Department of Information Science, The University of Tokyo[†]

Abstract

Reflection provides the abilities to reason and alter the dynamic behavior of computation from within the language framework. It is a practical scheme that offers a new perspective in constructing a malleable, large-scale system such as programming languages, operating systems, and window systems. Reflection is more beneficial in object-oriented concurrent/distributed computing, where the complexity of the system is much greater compared to sequential computing; this has been demonstrated through recent works in *Object-Oriented Concurrent Reflective (OOCR)* architectures. From a structural perspective, we can categorize OOCR architectures into *individual-based*, *group-wide*, and their hybrid (*hybrid group*). Through the introductive categorization of various OOCR architectures based on our previous and current work, we explore the characteristics as well as benefits and limitations of each.

1 Introduction to Reflection and Reflective Systems

Reflection is the process of reasoning about and acting upon the system itself[23, 15, 34]. In a conventional computing system, the *subject matter* of computation is external to the system, i.e., the computation is performed on data that represent or model entities that are external to the system. A *reflective system* takes a step further by allowing the subject matter be the system itself, i.e., computation can be performed on the system itself in a uniform way[15]. A reflective system thus 'opens up' the system by providing an appropriate abstraction on the internal (often implementational) detail of itself to the user program. Contrary to the misconception that 'reflection' is some difficult-to-understand, not-too-useful philosophical jargon, it is a practical scheme that offers a new perspective in constructing a malleable, large-scale system such as programming languages[14], operating systems[32], and window systems[22]. One can make an analogy of reflection to

---

[*] Presented at the Workshop on Object-Based Concurrent Programming, Geneve, Switzerland, July, 1991. (Proceedings to be published as Mario Tokoro et.al. eds., a volume in Lecture Notes in Computer Science, Springer-Verlag, 1992)

[†] Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-3812-2111 (overseas +81-3-3812-2111) ex. 4108. E-mail: {matsu,takuo,ichisugi,yonezawa}@is.s.u-tokyo.ac.jp

inheritance, which was derived from AI and incarnated into object-oriented programming languages, and now serves important roles in methodologies for system construction of user interfaces, operating systems, and database systems. We believe that, in the same manner, reflection can offer another clear architectural perspective, especially for the dynamic aspects of the system where inheritance is known to be somewhat ineffective.

To expose, or *reify* its internals, a reflective system contains data that represents or models the metalevel structural and computational aspects of itself within itself. Such data is said to be *reified*, and must be dynamically self-accessible and self-modifiable by the user program. Furthermore, when the user modifies such data, the modification must be 'reflected' to the actual computational state of the user program. This requirement is termed as *causal-connection*, and is satisfied by all reflective systems. When the requirement is satisfied, the meta-level data that provides the abstraction is called the *Causally-Connected Self-Representation(s) (CCSR)*. For practical purposes, the meta-relation is often that of implementation. In such systems, causal connection is automatically guaranteed. Although modifying the implementation may seem like a dangerous endeavor, it has been shown that modification can be confined to the local portion of the user program by some appropriate *Metaobject Protocol*, a scheme pioneered in CLOS[14].

Another requirement for reflective systems is that self-access and self-modification must be possible in a way that is little different from the base-level programming. In other words, appropriate abstraction of the internal program state must be provided by the system so that the user can perform the meta-programming of the system in a clean, concise manner that does not deviate significantly from 'normal' programming. For example, in 3-Lisp[23], one can program reflective computation merely by defining *reflective procedures* whose means of definition differ little from that of normal procedures.

2 Reflection in Object-Oriented Concurrent Systems — The Benefits

The benefits of *computational reflection* in sequential languages have been identified by many researchers: Previously, programming language functionalities such as debugging facilities, interfaces to external world, and exception handling were added to languages in an ad hoc and inflexible manner. In contrast, languages based on reflective execution models are powerful in providing linguistic mechanisms to use the above functionalities in a uniform and flexible manner via clean access of the abstraction on how the system is implemented[15, 14].

We claim that reflection is more beneficial in concurrent computing, for a concurrent system embodies multitudes of aspects that do not arise in sequential computing: scheduling, communication, and load-balancing, among numerous others. Those aspects are still considered to be mostly outside the scope of programming languages, and their control is only available in a fixed, ad-hoc fashion, with little possibility for user extensibility. This is a not favorable, since concurrent architectures are much more complex as compared to sequential ones, and the system must be *open* to structured dynamic extensions and/or modifications for adapting to new problems and environments.

Here, as were pointed out in [34] and [24], constructing a concurrent system with a reflective architecture could be beneficial to encompass such aspects within the programming language framework for the reasons mentioned above. The dynamic optimization is especially effective in a concurrent system because (1) as mentioned, the reflective architecture allows the programmer to control the computational aspects of concurrent system dynamically[1], and (2) both the reflective computation and the base-level computation can be performed in parallel[34, 17, 29].

3 Object-Oriented Concurrent Reflective (OOCR) Architectures

With the aforementioned observations and principles, our research proposed the use of reflection in the framework of *Object-Oriented Concurrent Programming (OOCP)*, with various *Object-Oriented Concurrent Reflective (OOCR)* architectures: languages ABCL/R[29] and X0/R[18] were based on the *individual-based architecture*, ACT/R[30] was based on the *group-wide architecture*[2], and the evolution of ABCL/R, the language ABCL/R2[17], was based on the *hybrid group architecture*. Each language differed in what meta-lingual features can be realized through reflection, i.e., what features particular to concurrent OO computing, such as object monitoring and migration, could be cleanly encompassed within the programming language framework. There, the evolution of the languages can be understood for the difference in the characteristic of their reflective architectures.

Although there have been works that discusses various aspects of reflection and classifies metalevel/reflective architectures from different viewpoints[16, 28, 6, 24], so far to our knowledge no work has existed that generally discusses reflection in the context of concurrent/distributed object-oriented languages. Nor, has there been work that classifies architectures dependent on characteristics that are *particular to object-oriented concurrency*, i.e., not becoming manifest in sequential object-oriented languages. It is our premise that the need for such a discussion and classification is becoming omnipresent — for as we argued above, the power of reflection can be best utilized in concurrent computation. Furthermore, we claim that in order for reflection to be put to practical use in concurrent/distributed systems, we need to analyze and evaluate the merits and weaknesses of each, and synthesize a new architecture suitable for practical use.

First, the two aspects of CCSR in OO languages are the (1) *structural aspect*, indicating how objects or group of objects in the base-level and the meta-level are constructed and related, and the (2) *computational aspect*, indicating how meta-level objects represent the computation of the base-level objects[6]. Altogether, appropriate construction of the *meta-level architecture* dictates what structural and computational aspects can be reflected and operated upon — this we believe is more prevalent in concurrent systems, for factors such as scheduling must be given appropriate meta-level abstractions in order for them to be effectively and efficiently accessed from the base-level.

The OOCR architectures can be categorized from the structural aspect, chiefly dependent on whether the architecture has the notion of *metaobjects*, which Pattie Maes introduced for sequential object-oriented language 3-KRS in [15]. It is often argued that providing explicit metaobjects in the architecture is conceptually natural with respect to

---

[1] One of the interesting applications we will note is to change the scheduling policy of object execution dynamically [3] in distributed simulation based on the Time Warp Scheme[13].

[2] The original idea of the group-wide architecture was mentioned in the last sections of [29] and [34]

the uniformity principle, "everything is an object." But, what do we really mean by a 'metaobject'? It is pointed out that there has been great confusion on the usage of the term[7] — we first attempt to provide conceptual distinction in the terminologies.

A *meta-level object* is any object that resides in the meta-level of the object-level computation. *Metaobjects* are objects that reflects the structural and computational aspect of a *particular* base-level object; thus, a metaobject is a meta-level object, while the converse is not necessarily so. The relationship between the metaobjects and the base-level object they represent could either be one-to-one or many-to-one, i.e., there could be multiple metaobjects collectively 'representing' a single base-level object[10, 21]. The metaobjects must be unique to a single object in the base-level, that is, they are not shared.

There are several important criteria that metaobjects must satisfy that distinguish it from other meta-level objects:

1. *Identifiability of Metaobjects* — a base-level object must be able to identify its metaobjects intrinsically.
2. *Uniqueness of Metaobject Denotation* — metaobjects must be able to intrinsically and uniquely identify the base-level object it denotes.
3. *Causal Connection* — a base-level object and its metaobjects must be *causally connected.*
4. *Governing of Computation* — the metaobject must dynamically govern the computation of the object it denotes or represents.

By these criteria, the 3-KRS and ABCL/R 'metaobjects' are metaobjects as we describe later on, whereas the CLOS 'metaobjects' are meta-level objects in our terminology. Class objects in class-based languages such as ObjVLisp[4] are not metaobjects either, for (1) the denotation is not direct and/or not unique, and (2) classes are not "meta" in a computational sense[6], that is, they do not actually govern the dynamic aspect of object computation.

Given the distinction of metaobjects and meta-level objects, a reflective object-oriented system can basically be constructed in two ways:

- **Individual-Based Architecture:** Each object in the system has its own metaobject(s) which govern(s) its computation. By 'individual-based' we mean that an individual object is the unit of base-level computation that has a meaningful CCSR at the meta-level.
- **Group-wide Architecture:** In this architecture, a group of meta-level objects comprises the meta-level. The behavior of an object is not governed by metaobjects; rather, the collective behavior of a *group* of objects is represented as the coordinated actions of a group of meta-level objects, which comprise the *meta-group.* By 'group-wide' we mean that the entire object group is the unit of base-level computation that has a meaningful CCSR at the meta-level (as a meta-group); thus, there are *NO* intrinsic meta-relationships between a base-level object and a particular metaobject at the meta-level.

- **Hybrid Group Architecture:** The hybrid group architecture combines the individual-based architecture and group-wide architecture. Each object has its own metaobject(s), while the coordinated actions of a group of objects, such as scheduling, are controlled by meta-level objects that comprise the meta-group. Both the individual object and the group have meaningful CCSR at the meta-level.

For sequential OO-languages, the representative of the individual-based architecture is 3-KRS, and the one representative of the group-wide architecture is CLOS. Below, we will give examples of each architectures for object-oriented concurrent languages in our research.

3.1 The Individual-Based Architecture — ABCL/R

An example of this architecture is ABCL/R[29]. Each object has its own unique metaobject, $\uparrow x$ that can be accessed with a special form `[meta` x`]`. Conversely, given a metaobject $\uparrow x$, `[den` $\uparrow x$`]` denotes the object it represents. The correspondence is 1-to-1, i.e., `[meta [den` $\uparrow x$`]]` $\overset{\text{def}}{=} \uparrow x$ and `[den [meta` x`]]` $\overset{\text{def}}{=} x$. The structural aspects of x — a set of state variables, a set of scripts, a local evaluator, and a message queue — are part of state variables of $\uparrow x$. A metaobject has its own metaobject $\upuparrows x$ as well and so on, conceptually forming an infinite tower of metaobjects (Figure 1).

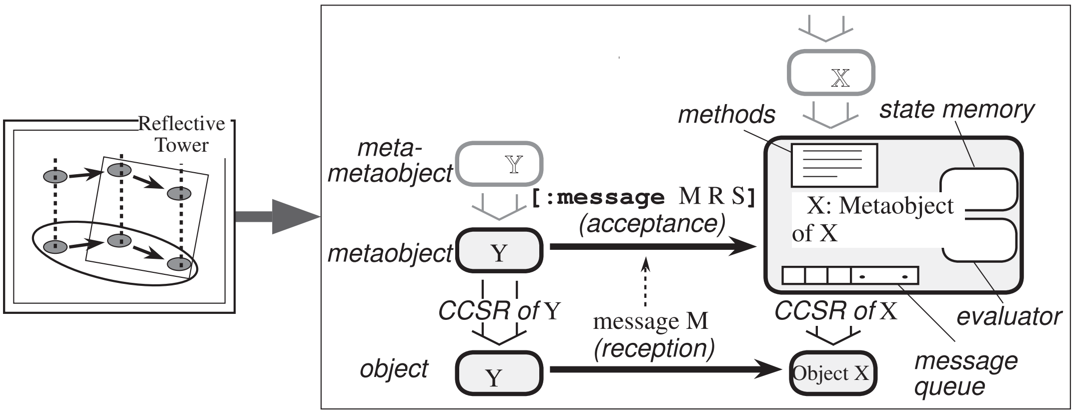

Figure 1: The Individual-Based Architecture in ABCL/R

The *arrival* of a message M at object x is represented as an *acceptance* of the message `[:message` M R S `]` at $\uparrow x$, where R and S are the *reply destination object* and the *sender object*, respectively. The interpretation of the script of x by $\uparrow x$ is carefully designed so that the concurrent activities allowed within an object in the computational model are preserved. Causal connection and total governing are guaranteed as $\uparrow x$ implements x. Figure 1 illustrates this. As are most individual-based architectures, reflective computation by an object in ABCL/R is basically performed by sending messages to its metaobject (or any other metaobjects in the tower); alternatively, the metaobject can immediately identify and send messages to its denotation.

Note that the individual-based architecture is independent from the issue of inter-level concurrency. As stated earlier, In a sequential OO-reflective architecture, there is only a single computation thread in the tower of metaobjects. This thread performs the interpretation of a certain level, and a reflective operation causes a 'shift' of this level. By contrast, in ABCL/R, there is internal concurrency within a single object as defined by the computational model of ABCL/1[33]: message reception/queuing versus the execution of the user script. The interpretation of x by $\uparrow x$ is designed so that the former is performed as a meta-level task while the latter is performed as a base-level task. Concurrency between the tasks is then preserved, because ABCL/R supports inter-level concurrency between the base-level object and its metaobject.

Other examples of individual-based architecture are Rosette[27], Merling III[5], Tanaka's actor language[3][26], and X0/R. The distinction here is the relationship between the object and its metaobject — whether the meta-object has a separate identity from the object it denotes:

- **'Unified' Identity:** when the meta relationship is that of total implementation as in ABCL/R. There is actually only one unique object in the tower; whether to regard it as an object or its metaobject (or metametaobject, ...ad infinitum) depends on from which level the object is being 'viewed' in the computation.

- **Separate Identity:** the metaobject has its own separate identity. Metaobjects in 3-KRS, Merling III, and X0/R are of this kind. In a class-based architecture, a metaobject is often an instance of class `MetaObject` or its subclasses.

- **Partially Self-Reifying Identity:** when there are no explicit metaobjects, but the snapshot of object state can be reified. We can consider the reified structure to be obtained from the implicit metaobject. T-Actor is an example of this.

The advantage of the separate identity over the first two is that the metaobject could be interchanged dynamically in order to change the basic behavior of the object, such as message reception. Although unrestricted alteration of metaobjects could be catastrophic to the integrity of computation, it nevertheless can be a powerful tool if employed in a controlled manner[18]. Dynamic alteration is more difficult for the first two, although possible for the unified identity by changing the behavior of the metaobject through its meta-metaobject. One major drawback of the separate identity particular to concurrency is the possible cost for maintaining causal connection. This becomes prevalent if the only way for the objects to communicate is via messages, because there is a natural delay for the transmission to occur. Then, the object and its metaobject must explicitly synchronize their activities in order to maintain the causal connection.

3.2 The Group-wide Reflective Architecture — ACT/R

ACT/R[30] is an Actor-based language that supports the notion of groups under this architecture. It is based on Gul Agha's Actor formalism[1] which models the state of *system* S of Actors as its *configuration*, C (a pair consisting of a set of Actors and a set of *tasks* in the system), and defines transitional semantics between the configurations, each transition being the computation of a task. Now, given a configuration C, We define the

---

[3] Unfortunately, this language does not have an official name; henceforth, we will refer to it as *T-Actor* for convenience.

metaconfiguration $\uparrow C$ as being a metalevel representation of a configuration by a system of Actors. A task $< t, m, k >$ in C where t is the *tag* of the task, m is the *target mail address*, and k is the *message value*, is represented in the metaconfiguration as a *meta-task* $< u, m_\theta,$ `[:task` $\uparrow t \uparrow m \uparrow k$`]` $>$, where u is the task tag of the meta-task, m_θ is the mail address of the *task handler Actor* θ, and $\uparrow$ is the *handle function*[4]. Actors in the base-level have their specific information stored at the meta-level *database entry Actors*, which are accessed via the meta-level *database Actor* δ^S. The framework is conceptually illustrated in Figure 2. We formally prove that $\uparrow S$ is a correct representation of S by showing that the transitions in $\uparrow S$ faithfully represent transitions in S. The difficulty is that the atomicity of a transition in the base level no longer holds at the meta-level; we overcome this with a technique called *normalization*. For details, see [30].

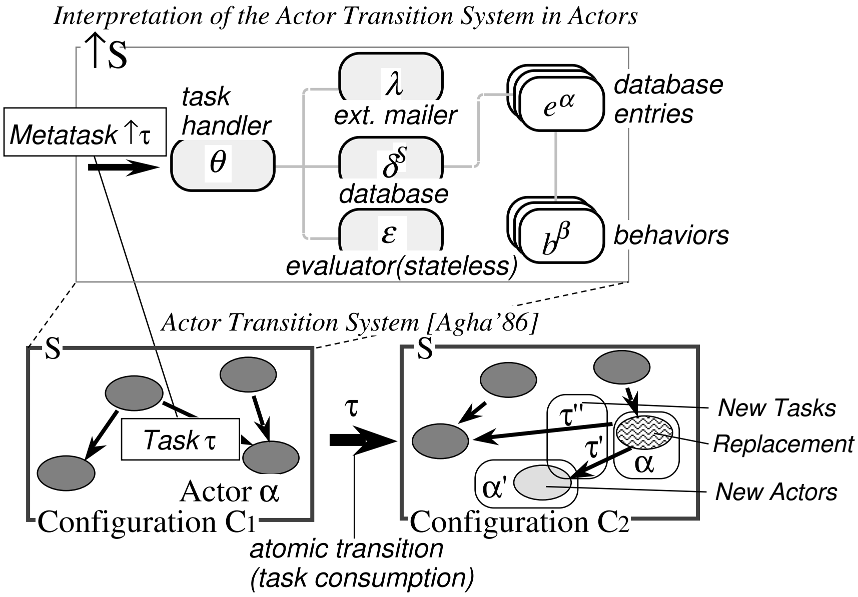

Figure 2: The Group-Wide Architecture in ACT/R

The essence of ACT/R is the lack of metaobjects, for the behavior of a single Actor is realized at the meta-level by coordinated action of multiple meta-level Actors. All reflective operations are performed solely via message sends, which are interpreted at the meta-level concurrently with interpretations of Actors in the base level. We claim that there are no metaobjects in ACT/R, although one could argue that the database entry Actors are so, for there is one entry Actor for each handle of the base-level Actors. They do not, however, satisfy our criteria, for (1) the denotation is not direct, and (2) the database entry Actor is not solely responsible for the computation of the corresponding Actor in the base level, e.g., the message reception is the responsibility of the task handler Actor.

Another example of the group-wide architecture, is the Muse distributed operating system[32, 31], although it has some features of the hybrid group architecture we discuss below. In Muse, the meta-level objects in the 'meta-space' 'support' the activity of the

---

[4] Not to be confused with the metaobject notation in ABCL/R.

objects of the base-level. There are specific meta-level objects responsible for various operating system tasks, such as message delivery, scheduling, memory management, etc. The reflective operation is performed by communicating with the meta-space via communication ports called *reflectors*. Muse is not a programming language, nor has an underlying monolithic base language, as is true for Genera[25] (which is Lisp-based). The system is nevertheless reflective, in that Muse has its own object model that can be realized by multiple object-oriented programming languages. Because of its language independence (being a general-purpose operating system), features that are highly language dependent, e.g., environments, are not handled in the Muse object model. However, Muse does support primitives for constructing some types of language features that are essential for the manipulation of the Muse object system, such as classes.

3.3 The Hybrid Group Architecture — ABCL/R2

In real-life concurrent language architectures, various system resources such as computational power, communication, storage, I/O, etc., are naturally distributed, and their availabilities are limited; computational power, for example, is limited by the number of CPUs in the system. For this reason, coordinated resource management is a fundamental problem in any concurrent architecture, but usually become little manifest at the language level. We made attempts to linguistically (namely, at the programming language level) formulate reflective features including resource management with previous OOCR architectures, and discovered the limitations of the architectures on their manageability of finite resources and their coordinated management. The primary reason was that such coordinations require the following abstractions to exist simultaneously in the meta-level: (1) (meta-level) object groups representing global computation, (2) object identity in the meta-level for the control of individual objects, and (3) proper abstraction of limited or bounded computational resources.

To solve this problem, we proposed the notion *hybrid group architecture* and the language ABCL/R2 based on this architecture[17]. By 'hybrid' we mean that the architecture embodies the architectural properties of both the individual-based and the group-wide architecture. Each object has its own (tower of) metaobject in the same manner as ABCL/R, and it also belongs to an ACT/R-like group. At the base-level, the structure of a group is flat in the sense that there are no base-level member objects that perform tasks specific to the group. Rather, analogous to the group-wide reflection, the structure and the computation of a group is explicitly defined at the meta- and higher levels of the group, by the objects called the *group kernel objects*. The group performs management of computational resources, called *group shared resources* by coordinating among the metaobjects of the members and the group kernel objects.

Groups can be created dynamically with the group creation forms. The creation process of a group is not intrinsic, but is given a concrete metacircular definition with ABCL/R2. As a result, not only that we have the tower of metaobjects, but we also have the tower of *meta-groups* as in ACT/R.

The key features of the language ABCL/R2 are as follows[17]:

- Heterogeneous Object Groups and Group Shared Resources
- Non-reifying Objects (objects that do not have metaobjects)

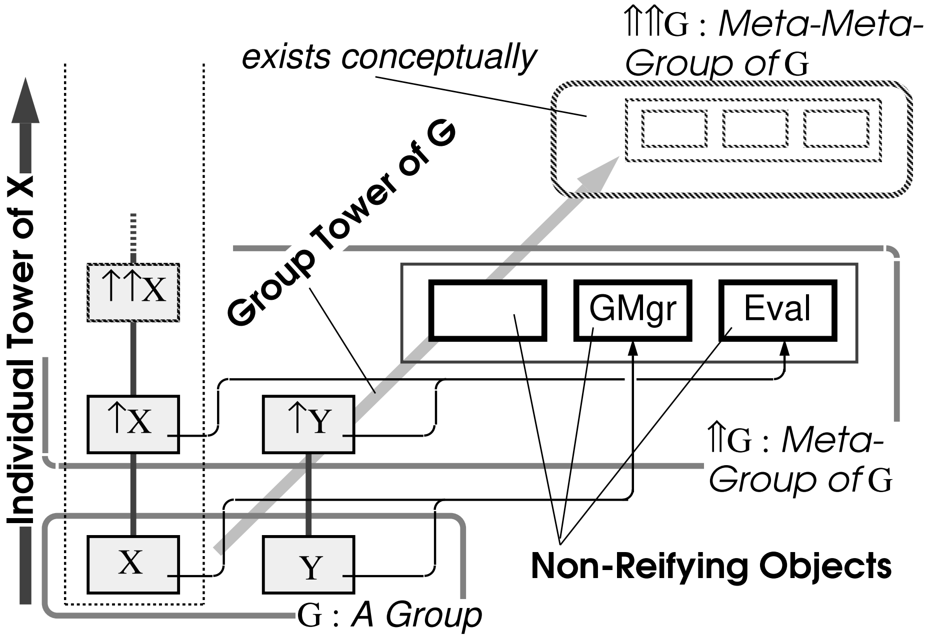

Figure 3: The Hybrid Group Architecture in ABCL/R2

- Meta-groups and Individual/Group Reflective Towers

The essential point is that these features are not ad-hoc, but are defined within the framework of our reflective computational model. As a result, reflective features of ABCL/R2 allow coordinated computational resource management to be effectively modeled and efficiently controlled — for example, the scheduling problem of the Time Warp algorithm[13] used in parallel discrete event simulations can be elegantly solved. Figure 3 is the pictorial illustration of the reflective architecture of ABCL/R2.

The hybrid group architecture also contributes feedback to the conceptual side of OOCR architectures and object groups by (1) showing that (heterogeneous) object groups are not ad-hoc concepts but can be defined constructively and lucidly, and (2) identifying that hybrid group architectures embody *two* reflective towers, instead of one: the *individual tower* that mainly determines the structure of the object, and the *group tower* that mainly determines the computation.

Our work on ABCL/R2 is still ongoing — the actual implementation of the compiler raises various interesting issues regarding optimizations. Also, there are still many aspects of concurrent computation we have not addressed in ABCL/R2. For example, there is a difficulty in the management of two distinct resources exhibiting collaborative behaviors; or, how to model the behavior of group of objects actively competing for the computational resource to solve a problem as a whole.

4 Future Directions — Towards Stronger Theoretical and Practical Foundations

Recently, there have been several interesting works on OOCR architectures other than the ones we have mentioned so far. These include AL-1[11, 12], OO||[2], and Actra[19, 20]. Although they could be classified as being an instance of the above architectures, each has interesting and unique architectural aspects that were not covered in this paper. For example, AL-1 is basically individual-based architecture; the novel aspect of AL-1, however, is that the CCSR of a single object can be 'viewed' in several ways, depending on how much the user wants to expose the underlying implementation of the object. This framework, called *multi-model reflection framework* in [12], allows the user to work with varying degrees of reflection by not restricting the user to a single model of CCSR.

Still, much more work needs to be done in order for reflection to be used generally in constructing malleable architectures. Figure 4 is a road map describing the relationships between our previous and current work, plus our future directions in search for a better OOCR architecture. Our current research topics include:

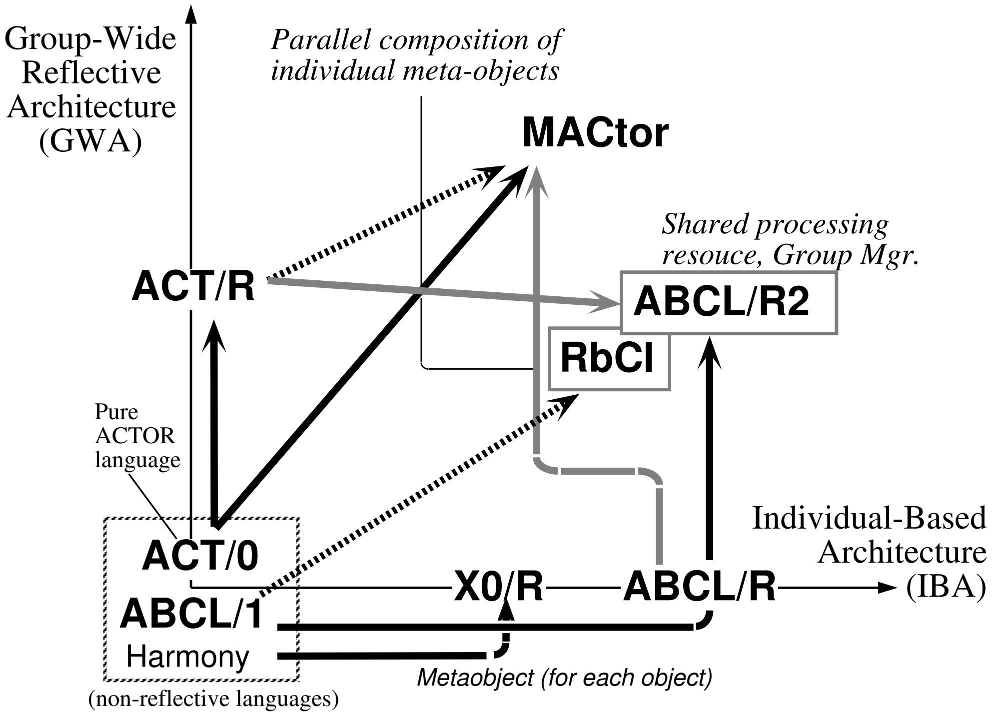

Figure 4: The Road Map of Our Previous and Current Works, Plus Future Directions

- To lay a more theoretical foundation for the hybrid group architecture — rather than giving direct formalization of ABCL/R2, we are currently taking an alternative approach by again using Actor transition system as the base computational model. The work is currently called *MACTOR*, and has a different formulation from ACT/R, more in line with ABCL/R2.

- To have an effective parallel implementation of ABCL/R2 — We are in the process of (1) constructing a prototype called Micro-ABCL/R2, which runs pseudo-parallel on conventional workstations, and (2) a full implementation on OMRON LUNA-88K, which has a shared-memory architecture and runs Mach. There are various technical intricacies, especially on how to efficiently and safely break the metacircularity in the compiler, as mentioned above.
- To construct and apply reflective system to real-world applications in concurrent/distributed systems — In parallel to the development of ABCL/R2, we are designing a language called RbCl (Reflection-based Concurrent language). The non-reflective features of RbCl are similar to that of ABCL/1; it is intended to run on a network of workstations and also on multicomputers such as iPSC/860.

We will briefly overview our latest language, RbCl, in the next section.

4.1 RbCl — A Hybrid Group Architecture Language for Practical Distributed Computing

The main motivation of this research is to develop a practically useful concurrent programming system that is easy to customize and become adaptable. We propose our object-oriented concurrent reflective language RbCl (Reflection-based Concurrent language)[10]. The goals of the RbCl system are as follows:

1. The system should be adaptable to various environments that differ in the architecture of parallel computers, and in the characteristics of applications. The system should also be able to evolve according to dynamic changes of execution environments.
2. The system should be used as a platform for various experiments on parallel languages to test linguistic mechanisms and implementation schemes.
3. The programmer should be able to access external environments (e.g., window systems, database systems, etc.) in a uniform manner.
4. The system should be efficient enough for a variety of parallel and distributed applications.

The requirements 1 and 2 are realized by the reflective computation facilities provided by RbCl. To make our language easy to program and practically useful, one of our goals of this research is to provide a *disciplined* scheme for the usage of reflective facilities that is analogous to the metaobject protocol (MOP) of CLOS[14].

The reflective architecture of RbCl is a practical incarnation of the hybrid group architecture. Each unit of metalevel representation models (or represents) a group of base-level objects, and such a group corresponds to the group of objects running on a single processor node. The RbCl metalevel architecture models the actual hardware configuration of a distributed computation system. Basically, the metalevel architecture consists of metalevel objects that form a metacircular definition (or, implementation) of the execution mechanism of base-level objects, just as ABCL/R2. Namely, the execution mechanism

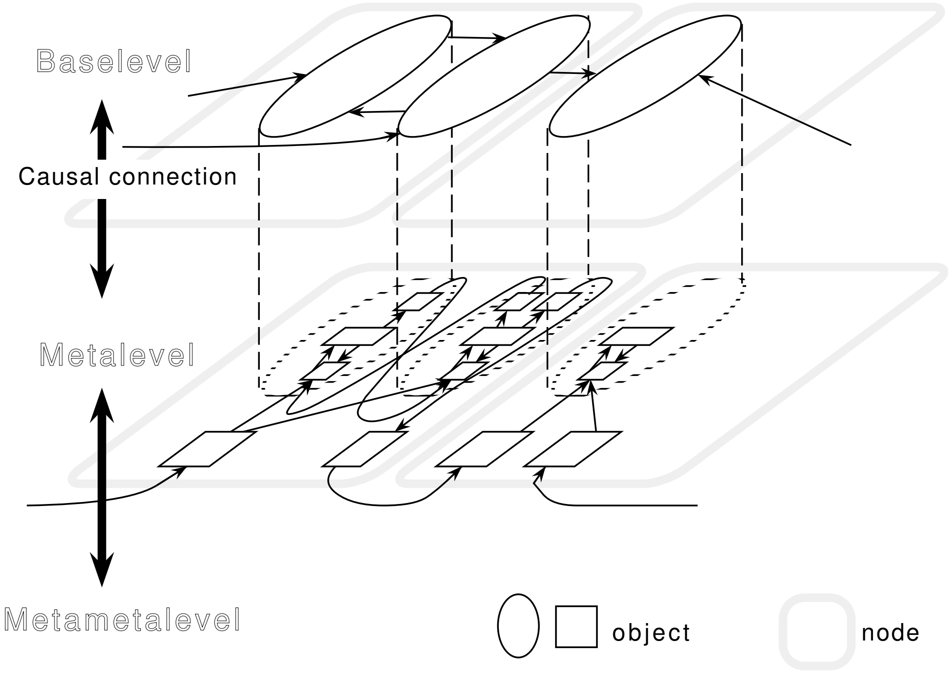

Figure 5: Overview of the RbCl Architecture

of a group of objects is represented by a (unit) group of meta-level objects. Reflective computation in RbCl is realized by the interaction between the base-level objects and the metalevel objects via message passing as in [29, 30, 17] (Figure 5). Our *metaobject protocol* (RbCl-MOP) to be designed provides a set of rules to customize and modify/extend metalevel objects in a disciplined manner.

One of the interesting characteristics of RbCl is the *linguistic symbiosis* with other object-oriented languages, namely C++, which does not offer reflective capabilities by itself. C++ objects and RbCl objects may coexist and communicate with each other within the RbCl system. Communication is transparent to each language due to the encapsulation property of objects: A RbCl object can communicate to a C++ object via the RbCl message passing protocol, which to a C++ object corresponds to a virtual function invocation. The converse is also possible, i.e., message passing expression in a C++ object to a RbCl object would correspond to a *now type* (i.e., RPC-like) message passing in RbCl.

The notion of symbiosis is different from simple "external language interface" schemes of previous languages. Neither C++ objects nor RbCl objects require little or no special programming to do the interlingual communication, because they cannot basically distinguish whether the receiver is a C++ object or a RbCl object in their message transmission. This is more difficult to realize than it seems, because the message passing semantics of the two languages are totally different. For example, a RbCl object could send an asynchronous message to a C++ object, which involves creation of a new thread of activity.

The linguistic symbiosis in RbCl is realized with the metalevel architecture of RbCl. For example, the forking of the thread of activity is programmed in the meta-level as an coordinating action of the RbCl meta-level objects. In fact, both RbCl objects and C++ objects interdependently comprises the meta-level architecture of RbCl itself, allowing (1) efficient implementation of the RbCl language and also (2) enables the use of external systems (e.g., window systems and database system, etc.) via the C++ objects acting as an interface. Furthermore, linguistic symbiosis allows the programmer or the compiler to replace a RbCl object with an object coded in C++ with the same behavior.

Implementation of symbiosis in the meta-level introduces circular interdependency in the system, since it is the meta-level architecture of RbCl that enables the symbiosis of RbCl objects and C++ objects in the first place. We can show that this circularity is not a vicious one, however. In fact, the initial construction of the default RbCl meta-level system is designed so that objects communicate with each other only via the now type message passing. Together with the fact that all meta-level objects are initially C++ objects of the same behavior, this allows all message passings between objects to be directly realized with direct virtual function call for high efficiency. Only when a RbCl object is introduced into the meta-system, the lazy instantiation of meta-meta-level support for interlingual communication in the meta-level become necessary.

5 Conclusion

We have categorized and reviewed several OOCR architectures, mainly from our past and present works. Research in OOCR architectures has only yet to reveal only a portion of their power so far, and the possibilities they offer in construction of a clean and malleable systems are being actively investigated in many places. The past two ACM OOPSLA workshops on Reflection and Metalevel Architectures in Object-Oriented Programming[8, 9] have collected nearly 30 papers per year, and each paper therein addressed numerous issues and ideas of object-oriented reflection with respect to programming language design, system architecture, theory, and implementation.

In conclusion, we feel that works in OOCR languages should have strong emphasis on the architectural issues, for it is not the language itself but the language architecture that would contribute the most in solving the problems in practice.

References

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] Peter C. Bahrs. On Reflection in Object-Oriented Heterogeneous Environments for Concurrent Processing. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[3] Christopher Burdorf and Jed Marti. Non-Preemptive Time Warp Scheduling Algorithm. *Operating Systems Review*, 24(2):7–18, April 1990.

[4] Pierre Cointe. Metaclasses are first class. In *Proceedings of OOPSLA'87*, volume 22, pages 156–167. SIGPLAN Notices, ACM Press, October 1987.

[5] Jacques Ferber. Conceptual reflection and Actor languages. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 177–193. North-Holland, 1988.

[6] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of OOPSLA'89*, volume 24, pages 317–326. SIGPLAN Notices, ACM Press, October 1989.

[7] Brian Foote. Object-oriented reflective metalevel architectures: Pyrite or panacea? In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[8] Mamdouh Ibrahim, editor. *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[9] Mamdouh Ibrahim, editor. *Proceedings of the OOPSLA'91Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[10] Yuuji Ichisugi, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. An object-oriented concurrent reflective architecture for distributed computing environments. In *Proc. 29th Annual Allerton Conference on Communication, Control and Computing*, Allerton, Illinois, October 1991. (To Appear).

[11] Yutaka Ishikawa. Reflection facilities and realistic programming. *SIGPLAN Notices*, 26(8):101–110, August 1991.

[12] Yutaka Ishikawa and Hideaki Okamura. A New Reflective Architecture: AL-1 Approach. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[13] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[14] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, Massachusetts, 1991.

[15] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, volume 22, pages 147–155. SIGPLAN Notices, ACM Press, October 1987.

[16] Pattie Maes. Issues in computational reflection. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. North-Holland, 1988.

[17] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 231–250. Springer-Verlag, 1991.

[18] Satoshi Matsuoka and Akinori Yonezawa. Metalevel solution to inheritance anomaly in concurrent object-oriented languages. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[19] Jeff McAffer. A unified distributed simulation system. In *Proceedings of the 1990 Winter Simulation Conference*, pages 415–422, New Orleans, LA, December 1990.

[20] Jeff McAffer. Variably Asynchronous, Reflective Tasks in Smalltalk. In *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1991.

[21] Tatsuo Minohara and Mario Tokoro. Multiple meta-objects support an object. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1990.

[22] Ramana Rao. Implementational reflection in Silica. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 251–267. Springer-Verlag, July 1991.

[23] Brian C. Smith. Reflection and semantics in Lisp. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984.

[24] Brian C. Smith. What do you mean, meta? In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[25] Symbolics Inc. *Genera User's Guide*, 1990.

[26] Tomoyuki Tanaka. Actor-based reflection without meta-objects. Technical Report RT-0047, IBM Research, Tokyo Reserach Laboratory, August 1990.

[27] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with Enabled-Sets. In *Proceedings of OOPSLA'89*, volume 24, pages 103–112. SIGPLAN Notices, ACM Press, October 1989.

[28] Frank van Harmlen. A classification of meta-level architectures. In Abramson and Rogers, editors, *Meta-Programming in Logic Programming*, chapter 5, pages 103–122. The MIT Press, 1989.

[29] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88*, volume 23, pages 306–315. SIGPLAN Notices, ACM Press, September 1988. (Revised version is in [33]).

[30] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands*, May 1990. (Also in number 489 in Lecture Notes in Computer Science. Springer-Verlag, 1991).

[31] Yasuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. Reflective object management in the Muse operating system. In *Proceedings of the IEEE International Workshop on Object-Orientation in Operating Systems*, pages 16–23. IEEE Computer Society Press, 1991.

[32] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of ECOOP'89*, pages 89–106. Cambridge University Press, 1989.

[33] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* Computer Systems Series. The MIT Press, 1990.

[34] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 50–54. SIGPLAN Notices, ACM Press, April 1989.