[2] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. *Conf. Record of the 1984 ACM Symp. on Lisp and Functional Programming*, pp. 348–355, 1984.

[3] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Implementation of temporal logic programming language Tokio. In *Logic Programming '85*, number LNCS-221. Springer-Verlag, 1985. Lecture Notes in Computer Science.

[4] R. A. Kowalski. Predicate logic as a programming language. In *Information Processing 74*, 1974.

[5] Pattie Maes. COMPUTATIONAL REFLECTION. Technical Report TR-87-2, VUB AI-LAB, 1987.

[6] Pattie Maes and Daniele Nardi, editors. *META LEVEL ARCHITECTURE AND REFLECTION*. North-Holland, 1988.

[7] George Milne and Robin Milner. Concurrent process and their syntax. *J. ACM*, Vol. 26, No. 2,, April 1979.

[8] James Lyle PETERSON. *Petri Net Theory and the modeling of systems*. Prentice-Hall, Inc., 1981.

[9] Brian C. Smith. Reflection and semantics in lisp. Technical Report CSLI-84-8, Center for the Study of Language and Information, 1984.

[10] Linda R. Walmer and Mary R. Thompson. A Mach Tutorial. Technical report, Department of Computer Science, Carnegie-Mellon University, August 1987.

[11] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1988*. ACM, September 1988. also appeared in SIGPLAN NOTICES, Vol.23, No.11.

[12] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of European Conference on Object-Oriented Programming in 1989*, 1989.

[13] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modeling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

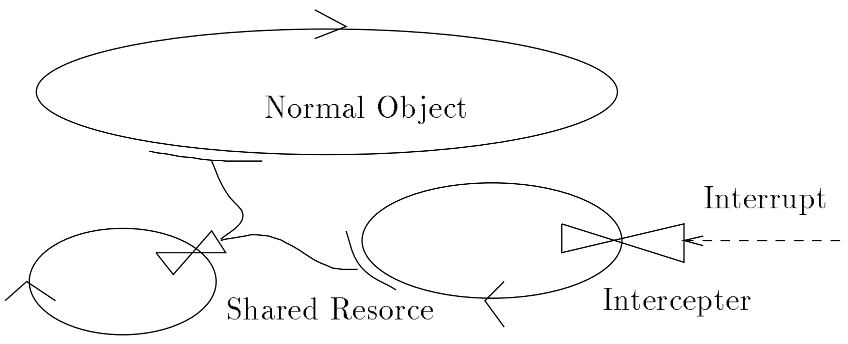

Figure 10: Interrupt

The components can be explicitly scheduled because their states are visible in the detailed description. In this figure, the synchronization points represented by squares are gates to control the

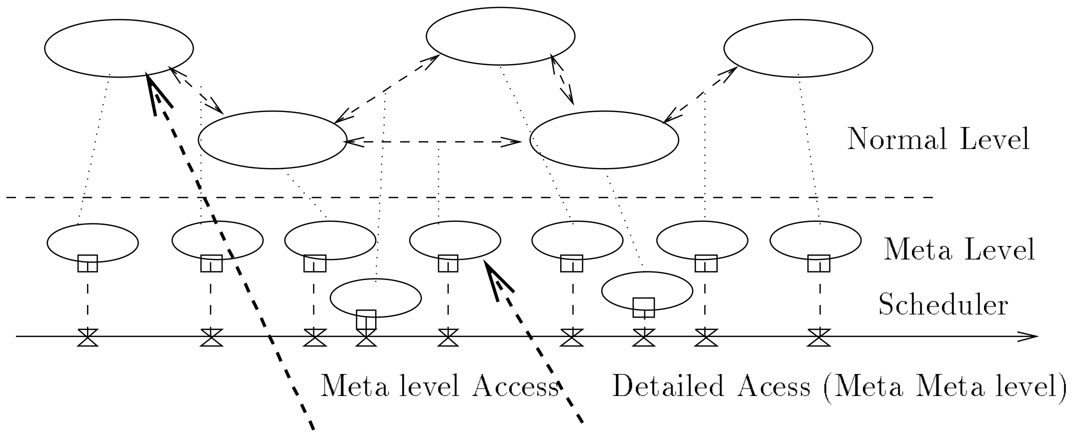

Figure 11: Scheduler

passage of tokens. The scheduler can be controlled directly by changing the represented structure with a reflective operation.

9 Conclusion

We presented some problems that arise in concurrent reflective programs. Such problems could not be solved elegantly with the use of meta-interpreters. The approach presented here, called replication-based reflection, does not use meta-interpreters. Without meta-interpreters, reflection can be described in any language. Thus it allows, for example, to control pieces of program coded in Fortran from others written in Smalltalk.

The approach is expected to have good results in implementing reflective operating systems. It is being designed for Muse operating system[12] and will be provided as an interfacing library of reflective operations to control the operating system and concurrent programs.

References

[1] G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

object change or how many messages the object send to its acquaintances. Such description is ad-hoc and impractical. In the next chapter, we will discuss how to establish fully represented reflection.

8 Fully Represented Reflection

If reflection is fully represented, we need no meta-interpreter to describe how reflection works. Generally it is impossible to provide fully represented reflection with static syntax. This is because reflection changes the computation dynamically. From the viewpoint of replication-based reflection, fully represented reflection is supported by detailed replication of the system's structure. In this case, a caught context should be modeled by the local state of the object and the trace of the descendant messages, as illustrated in (Fig.9). Modification of this replication fully specifies the semantics of reflection.

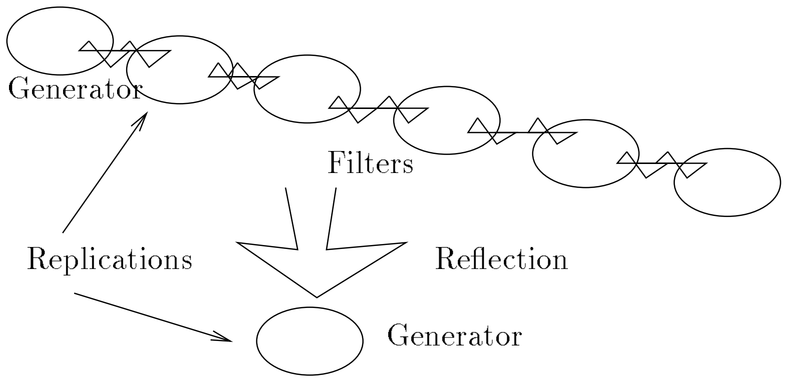

Figure 9: Fully Represented Reflection

Fig.9 shows a *thread diagram.* A thread diagram describes the activities and communications of objects with multiple threads. An activity is represented by a token on a fixed finite closed path (circle). Synchronization points and communications are represented by small "butterflies" that link those circles. A token contains the local states of an object. A thread diagram is not designed to describe all the details of computation but it is powerful enough to describe the controls of concurrent execution.

Various kinds of models or diagrams for parallel computation have been introduced in the literature (e.g., CCS[7], Petri-Net[8] and Temporal Logic[3]). A thread diagram is a finite representation of a fixed number of processes. It has the expressive power equivalent to non-deterministic finite automaton.

An interrupt is a good example of reflective operation. Interrupts cannot be described in the normal level alone because they can change computation in that level. The meta level computation of an interrupt consists of the detailed parts of the execution of the object (Fig.10. Here we use a heap (the local storage of an object) explicitly. Synchronization represented by a line is a buffered communication.

The meta-object that accepts an interrupt runs independently with the interrupted object. Some kind of synchronization is necessary when the meta-object accesses the local states of the normal level objects. The synchronization points are explicitly shown in the thread diagram.

Fig.11 shows one more level detailed thread diagram. Communication can have its own states in the detailed diagram. Generally speaking, every component in a thread diagram can have a state.

```
[object genartor
       (state [filter:=[filter−class <== [:new]]]
             [n :=1 ])
       (script
             (=> [:generate]
                   (reflect (state [c := (catch)])
                         [filter <= [catch:
                            [object (script
                               (=> [:throw last]
                                  (print last)
                                  (terminate catch)
                            ))]]])
                   (loop [filter <= [:check (incf n)]]))))]

[object filter−class
       (script
             (=> [:new]
             ![object filter
                   (state next−filter)
                   (reflect (state [catch]))
                   (script
                         (reflect (=> [catch: c])
                            [catch := c]
                            [next−filter <= [catch: c]])
                         (=> [:check n]
                            (output <== [:print n])
                            (next−filter := [filter−class <== [:new]])
                            (reflect
                            (if (> n 100) (catch <= [:throw n])))
                            (wait−for−loop
                               (=> [:check m]
                                  (unless (zerop (mod m n))
                                     [next−filter <= [:check m]])))))])))]
```

Figure 8: Catch and Throw

7 Unsafe Reflection

In the case of unsafe reflection, the meaning of reflective program P_R is the meaning of the interpretation m of $P + R$:

$$\mathsf{M}(P_R) = \mathsf{M}(m(P + R)).$$

In this case, the semantics of reflection depends on the meta-interpreter. In this sense, replication-based reflection (which uses no meta-interpreter) is partial.

However, reflection usually changes only a part of the program's semantics. If $\mathsf{M}(P_R)$ is a composition of the reflection part $\mathsf{M}(P_R)_R$ and the original part $\mathsf{M}(P_R)_P$:

$$\mathsf{M}(P_R) = \mathsf{M}(P_R)_P + \mathsf{M}(P_R)_R.$$

$\mathsf{M}(P_R)_P$ is a subset of $\mathsf{M}(P)$. If $\mathsf{M}(P_R)_R$ is determined only by R, the meta-interpreter m is not necessary. In this case, reflection is *fully represented*. Otherwise, the reflection is *partially represented*. An example of partially represented reflection is *catch/throw* (Fig.7).

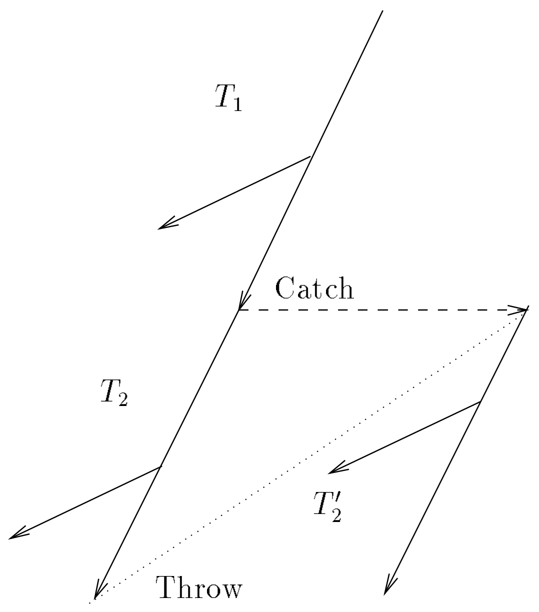

Figure 7: Traces of Catch and Throw

There are three execution traces here. A context is caught after the execution of T_1 and is thrown after the execution of T_2. Then a new execution T_2' starts.

Unlike sequential environment, a concurrent object creates many activities. To recover a context is not an easy task because all the created activities must be reclaimed. Since the object is encapsulated, a context caught consists of the object's execution only, but the later throw affects many descendant activities. Implementing *catch/throw* with a meta-circular interpreter is a very difficult task.

In the example shown in Fig.8, only a finite number of primes are generated. In this case, $\mathsf{M}(P_R)_P$ is a finite subset of $\mathsf{M}(P)$, and $\mathsf{M}(P_R)_R$ is an empty set. If another program is executed after the throw, the trace of this execution depends on $\mathsf{M}(P_R)_R$.

$\mathsf{M}(P_R)_R$ depends on the implementation of *catch/throw*. Reflection is thus partially represented. The reflection would be fully represented if we wrote all the possible side effects of *catch/throw* within the description of *catch*. The representation should then contain how local states of the

```
[object genartor
    (state [filter:=[filter−class <== [:new]]]
          [n :=1 ])
    (reflect
        (state [priority := 1])
        (script (=> [:priority p] [priority := (/ 1 p)])))
    (script
        (=> [:generate]
            (reflect [filter <= [priority: (+ p 1)]])
            (loop [filter <= [:check (incf n)]]))))]

[object filter−class
    (script
        (=> [:new]
        ![object filter
            (state next−filter)
                (reflect
                    (state [priority := 1])
                    (script (=> [:priority p] [priority := (/ 1 p)])))
                (script
                    (=> [:check n]
                      (output <== [:print n])
                      (next−filter := [filter−class <== [:new]])
                      (reflect [next−filter <= [priority: (+ p 1)]])
                      (wait−for−loop
                        (=> [:check m]
                          (unless (zerop (mod m n))
                            [next−filter <= [:check m]])))))]))]
```

Figure 6: Safe Reflection

6 Safe Reflection

First, we separate reflective computation from normal computation syntactically.

$$P_R = P + R$$

Here, P_R is the reflective program of P. R is the additional syntactical reflective part. Here we use ABCL/1 [13] as the base language and extend it with constructs for representing reflective operations. For example,

$$(\mathbf{reflect}(\text{incf}i))$$

is a reflective statement.

The statement below is a *projection* of the reflective program.

$$P = P_R - R$$

Projection can be realized by defining **reflect** statement as a null function.

Besides its syntax, the semantics of a reflective program must also be separate the normal part and the reflection part. Such separation is similar to separation of control and logic in logic-programming [4]. $\mathsf{M}(P)$ is the meaning function of a program P. $\mathsf{M}(P)$ is an instance of a program execution. In concurrent languages, it represents the history of the states of the objects and the history of messages. Not all those histories can be fully specified in ABCL/1 because of its non-deterministic execution. For simplicity, we assume a deterministic program semantics.

Let us consider a program P that generates prime numbers (Fig.6). For simplicity, we ignore some messages that is not relevant to understanding the problem. $\mathsf{M}(P)$, the semantics of P, can be expressed by the output which is an infinite set of prime numbers $\{2, 3, 5, 7, 11, 13, 17, ...\}$.

In parallel reflection, only the reified replication is visible in program P. We use R as a description of the replication in program P. A reflective program P_R is a composition of P and R. (The operator $+$ expresses composition.)

$$P_R = P + R$$

If the original semantics is not affected by reflection, we call it a *safe reflection*. The operators $+$ and $-$, for composition and projection respectively, are also used here. $\mathsf{M}(P) + \mathsf{M}(R)$ is a union of the trace of states and messages in both the reflection part and in the normal part. A reflection R is safe if

$$\mathsf{M}(P_R) = \mathsf{M}(P) + \mathsf{M}(R).$$

This means that the messages in the reflection part does not interfere with those in the normal level.

An example of useful safe reflection is control of scheduling of objects (Fig.6). Assigning a fixed (fair) scheduling priority to each object in program P is a reflective operation. In this case, $\mathsf{M}(R)$ consists of the set of assigned priorities and the messages required to setup these priorities. The meaning of program P is not affected by the assignment but efficiency is. The effect in efficiency cannot be seen in the program's semantics.

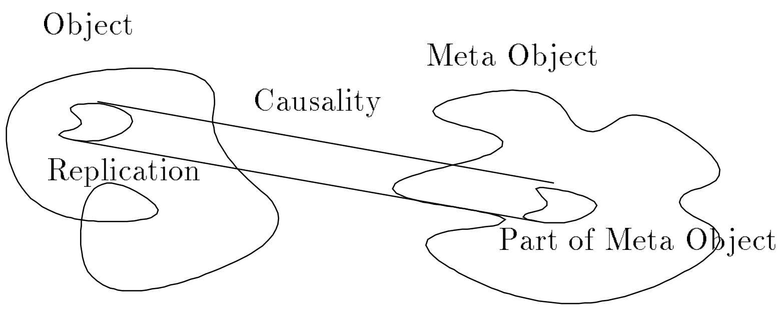

Figure 4: Replication Base Reflection

We must specify the operational semantics of the approach clearly in order to make it useful in spite of those errors. Replication errors are considered from the viewpoint of true reflective causal connection, but if they are well defined we can still use reflection in a consistent way.

The separation between meta level description and object level description is clear in this approach because the reified data in the object level can access only objects in the normal level. If the language that describes the normal level computation supports multi-thread execution, the access or synchronization procedure of reified data can be represented in the normal level programming construct. If the language that describes the normal level computation does not support multi-thread execution, the synchronization procedure can be seen as a modification of the normal level data structure from outside of the language. In this case, the reified data is actually a procedure which has states. The description of the reflected data is separated from the meta-interpreter. However, usually, in order to determine a semantics of reflection we still need a meta-interpreter.

In this scheme, the reflective procedure is not suspended (Fig.5). A normal level program will change its meta level data structure as a normal level data structure. These changes are automatically reflected into the meta-object after some delay, via replication. Conversely, a meta-object can change a normal level program or data. In parallel reflection, an object, its meta-object

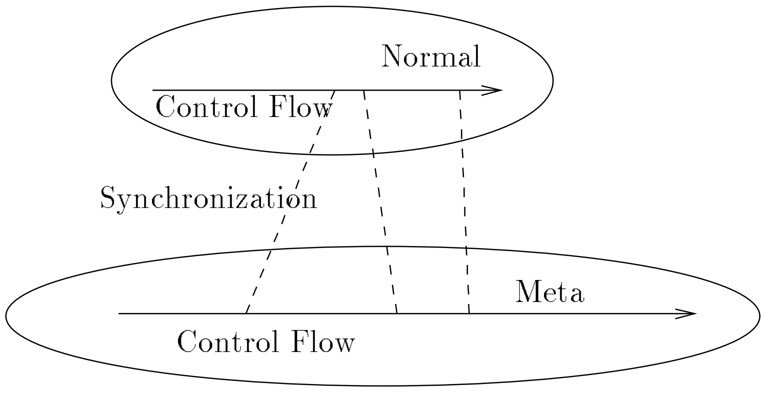

Figure 5: Concurrent Execution of Meta-Objects

and the replication mechanism (casual connection) run as threads within one object.

in the normal level and that in the meta-level are different, we cannot expect that the description of concurrency in the former be also the description of concurrency that actually happens in the latter. We must separate those two descriptions. The meta-circular interpreter approach does not allow such separation.

Secure reflection is necessary.

In the meta-circular-interpreter approach every part of the system can be reflected and touched. Reflection in that approach is overly free. Such freeness cannot be tolerated in systems that require strong security, as in operating systems. In a sense, this is a separation between description and actual computation.

Original program and reflective program must be clearly separated.

We can extend the power of a language by adding reflective facilities into it. Reflective facilities are important, but the original semantics of the program is still the most important. The reflective part of a program is less portable than the normal part. Consider the example of a debugger. The debugger essentially performs reflective modifications into the original program but such modifications should not change the original semantics. Meta-interpreter approach is weak here, since detailed meta-interpreter and its modification are complex and the effects of the reflection cannot be separated.

5 Replication-Based Reflection

The separation of meta level computation and normal level computation is blurred meta-objects are accessed from the normal level. A normal level object should not touch a meta level object directly. In this section we will introduce a new reflection scheme, which we call *replication-based reflection*. In this scheme normal objects do not interact directly with meta objects.

We allow non-strict reflection and use *replication* instead of reified meta-object. Replication is a copy of an object with automatic equivalence maintenance (Fig.4). In our communication between meta-object and normal object is replaced by replication. Inter-level communication must be treated separately from normal level message passing; it must be implemented using meta level message passing. In order to keep a clear separation between the levels of computation, we do not allow normal level messages to be sent to meta-objects.

When a reflective statement is executed, first a new data structure is created. This data structure is a normal level description rather than a meta-level one. The causal connection between a normal level object and the corresponding replication is operationally defined in the meta level. Various errors related to maintenance of replicated data may occur:

abstraction error: A reflected data structure in the object level is the abstraction of a meta level data structure, so it only represents a part of the meta level information.

synchronization error: Both the reflected data in the object level and the meta-object may be changed between two synchronization points, thus generating a replication error.

definition error: Since the synchronization procedure is defined in the meta level in an ad-hoc way, it is prone to error.

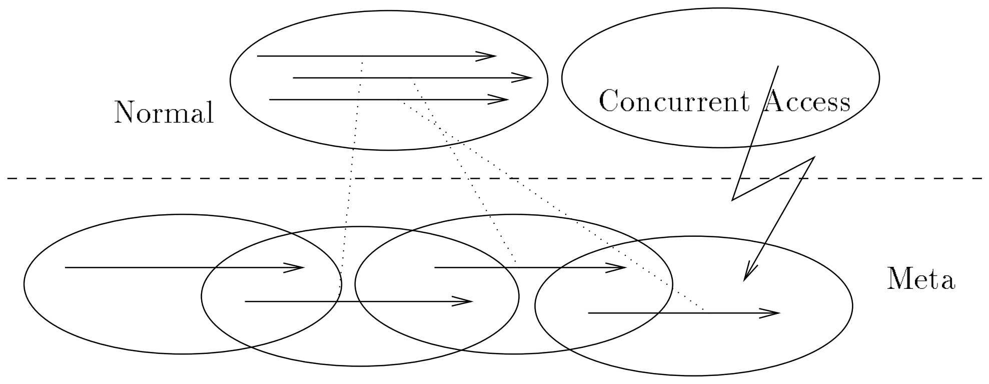

Figure 2: Multiple Thread from Parallel Reflection

Another important issue in parallel reflection comes from concurrency between normal objects and meta-objects. Consider a reified data. Since computation of objects proceeds concurrently, we must assure that the reified data (e.g., a message queue) is not changed asynchronously by the meta-objects. This requires some synchronization. Without synchronization, normal level objects cannot access reified data in a consistent way. (This problem does not arise in sequential reflection.) Introducing synchronization raises another problem related to causality: we cannot assure that meta-level operations be immediately reflected into the normal level. (Fig.3).

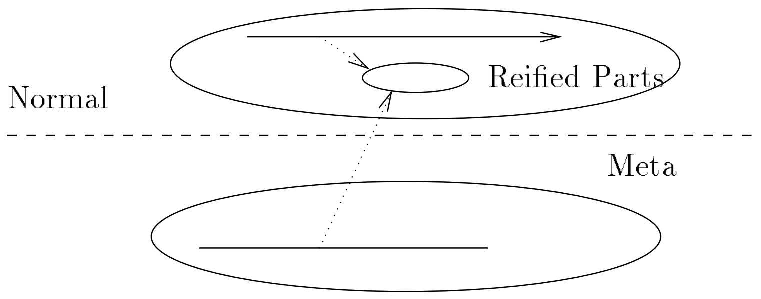

Figure 3: Concurrent Access to Reified Objects

The above issues are common to concurrent languages that support reflection. In a concurrent programming language, the unit of concurrency is fixed, but once we introduce reflective capability into the language, additional finer grain level of concurrency is required and this level cannot be controlled from the normal level.

4 Manageable and Well-Separated Reflection

Synchronization in the normal level must be separated from those in the meta-level.

The problems described in the previous section are products of the confusion between normal level objects and meta level objects. For example, we can send a message to another message which is an object in the meta level. Those two messages are of different levels and might not be confused. In the same manner, concurrency in the normal level and concurrency in the meta level must be distinguished. The causal connection between different levels of computation must include the relationship between the granularity of objects of those levels. If the granularity of concurrency

- Meta-circular interpreter description,
- Control passing mechanism through levels of computation (normal computation to meta-level computation, and so forth),
- Passing data between different levels.

In Smith's approach[9], operations on different levels are represented by explicit description of continuation (e.g., *normalize*). In Friedman's approach [2], communication between levels is represented with two basic operations: *reification* and *reflection*. In the former control is transferred from the normal level to the meta level and in the latter control is transferred from the meta-level to the normal level (Fig.1). Computation in the normal level is suspended after control is passed to the meta level. Changes to reified data is not allowed in the normal level because they cause errors.

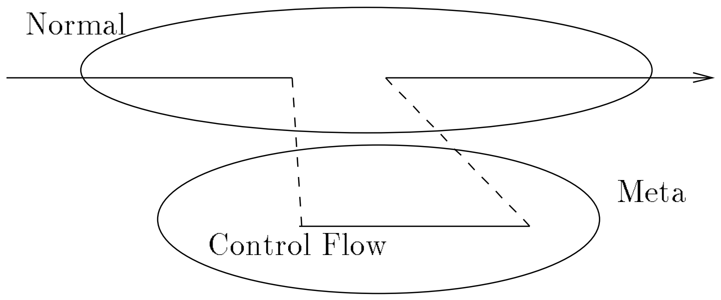

Figure 1: Sequential Reflection

3 Parallel Reflection

How things change in a concurrent computational environment consisting of a lot of running objects? Quite naturally, a computation is described by several concurrent objects called meta-objects[11]. There are three different concurrencies:

- concurrency among normal objects,
- concurrency among normal objects and meta-objects,
- concurrency among meta-objects.

As the result of those concurrencies, the meta-objects are accessed in parallel. For instance, while a meta-object is taking care of a normal object, another meta-object can accept interrupts.

Reflection in a concurrent object system requires multiple threads within an object.(Fig.2).

Since reflection changes the behavior of the meta-objects and meta-objects run in parallel, reflection happens in a multi-thread way. In this sense, normal objects can be seen as having multiple-threads.

Parallel reflection cannot be strict.

Parallel Reflection

Shinji Kono and Mario Tokoro

e-mail:kono@csl.sony.co.jp
Sony Computer Science Laboratory Inc.
3-14-13, Higashigotanda, Shinagawa-ku, Tokyo 141, Japan

February 13, 1991

1 Introduction

A manageable and separated reflection scheme is presented here. Reflection using meta-circular interpreter worked well in sequential programming. But in parallel programming, granularity of meta level computation is different from that of normal level computation, and this difference can cause errors in synchronization between normal level and meta level. Here we strictly separate meta level computation from normal level computation, syntactically and semantically, in order to solve this problem. We use "replication-based reflection", reflection without meta-interpreter, to achieve this separation.

2 Sequential Reflection

A programming language is a tool to describe computation. Since it uses fixed syntax and execution semantics, it usually has restrictions. For example, in Lisp, computation is applied to atoms and lists only, and not to their textual representations. Reflection is a method to overcome this restriction. Reflection allows manipulation of relationship between symbols and names. Computational reflective system is defined as " a computational system which is about itself in a causally connected way "[5, 6]. It is a non ad-hoc way for extending systems. Reflection is different from recursion. While recursion allows access to program in normal computation, reflection allows access to the system itself. Reflection is also different from partial evaluation of meta-interpreters. While partial evaluation does not extend the program language itself, reflection does.

In this paper, we focus on parallel reflection: reflective systems in concurrent or parallel programming environment. Throughout this paper, objects are units of concurrency (like Agha's actors[1]) rather than units of program (modules), or Smalltalk objects. More precisely, an object consists of several threads (activities, like Mach's threads[10]) and encapsulated data. The data is shared by the threads and accessed in a consistent way. The simplest way of keeping consistency is applying single thread restriction to objects.

In most computational reflection schemes, causality is automatically satisfied using a meta-circular interpreter [9]. In such schemes, reflection is sequential because the meta-interpreter has only one thread. Those schemes have three important parts: