References

[1] Rober E. Filman and Daniel P. Friedman. *Coordinated Computing.* McGraw-Hill Book Company, 1984.

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

[5] B. Mishra and E.M. Clarke. Automatic and hierarchical verification of asynchronous circuits using temporal logic. Technical Report CMU-CS-83-155, Dept. of Computer Science, Carnegie-Mellon Univ., September 1983.

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

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

[8] 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.

[9] 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.

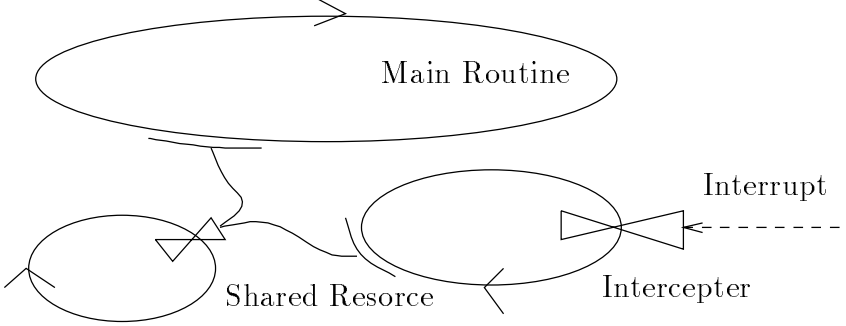

Figure 19: Interrupt

kind of synchronization is necessary. In this thread diagram, these synchronization are clearly visible.

As we saw in examples, communications may have its own states in detailed modeling. Generally speaking, every components in a thread diagram has a state. In a meta level description, these states can be visible and can explicitly be scheduled.

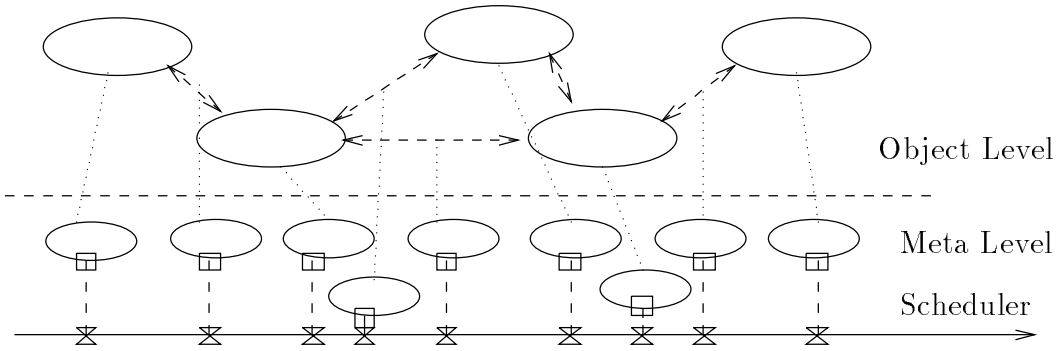

Figure 20: Reflective hierarchy

7 Conclusion

In this paper, we discussed the usefulness of the multiple-thread computation in objects. At first we started with combination of meta level computation and a group of tightly coupled objects. To describe complex communication among them, we introduce thread diagram. Using the thread diagram, a pattern of communication among these set of thread become visible. If we think each synchronization type as a message passing, again we have a single thread representation of objects. Now our long trip from single thread objects is closed.

The thread diagram is much more simple and convenient for specification or analysis, since it has finite synchronization state. It is also hierarchical representation. An object consists of a set of threads and a thread can consist of another set of threads.

of expressiveness. Difference is as follows: Petri-net has a data driven visibility but thread diagram has a control driven visibility. In other words, thread diagram is a control flow graph with synchronization.

In a concurrent program specification, it is good to start from high level description, and then to do incremental refinement. For example, consider a simple disk driver. A disk driver communicates with a disk drive using some communication. The most simple and

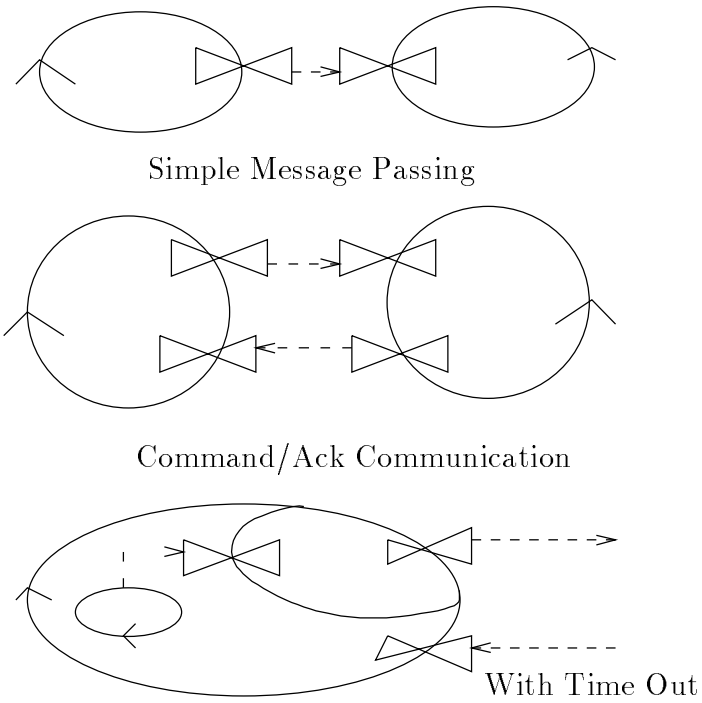

Figure 18: Communication Abstraction

abstract way is direct rendezvous and exchanging information at once. In the next level, these rendezvous are separated into directed communications: command and acknowledge. In more detailed model, command or acknowledge may fail, so we need to introduce time out mechanism. To implement time out a separate process is necessary. In this small example we omit detail of time out generator. It does not even specify the type of synchronization.

6.1 Example descriptions of Parallel Reflection

Previous examples are rather classical examples and they only use object level description. However real usefulness of thread is reflective computation. An interrupt is a good example of reflective procedure. To describe an interrupt, object level is not enough, because it must change object level computation. To describe meta level computation, an object is divided into more detailed parts. Here we use a heap (a local storage of object) explicitly (Fig.19). An interrupt acceptor is running separately, so it accepts interrupt without interfere the main routine for a while. However to access main routines local states, some

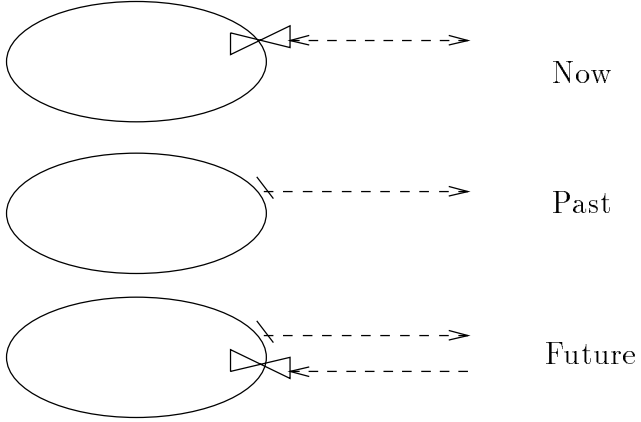

Figure 16: Now, Past and Future

our diagram. (Fig.16). In the thread diagram, no information on local state is used. So all the synchronization information is visible. In this Past example, a wait and pass type synchronization prevent duplicate sends. It actually violates the fact that Past type can send messages in unbounded way. The thread diagram cannot describe unbounded synchronization.

To show the exclusion explicitly, a non-deterministic choice can be used. Here we show both thread diagram representation and Petri-net representation. In both representations,

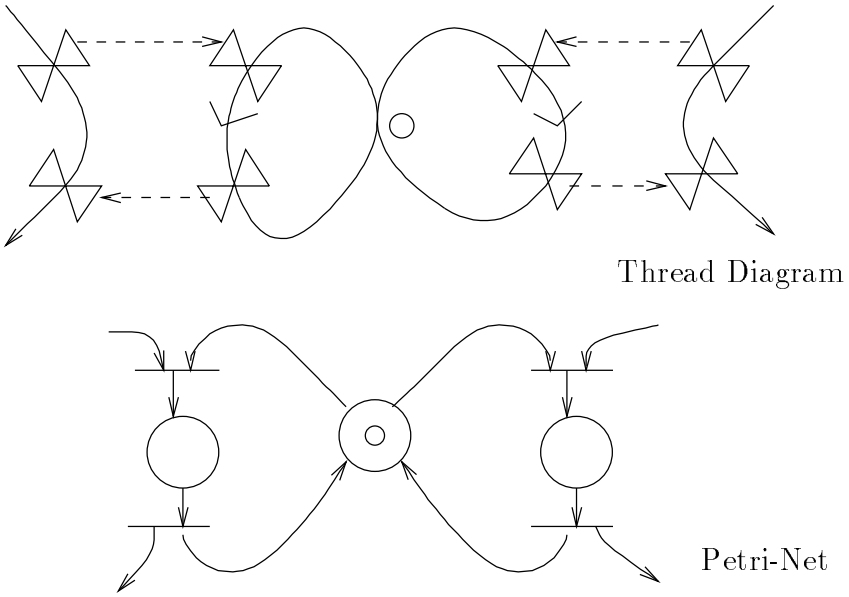

Figure 17: Exclusion

a position of token makes a state of exclusive resources. These two implementations are equivalent from the view point of the non-deterministic finite state machine. There is a good correspondence between Petri-net and thread diagram. Since bounded Petri-net is a finite state machine, thread diagram is equivalent to bounded Petri-net from a viewpoint

tional which checks token's state in a branch or a synchronization primitive (Fig.15). In the case of the branch, only the path which satisfies condition is used. In the case of the wait, if its condition is false, token passes without any effect of waiting. Conditional is a part of synchronization type.

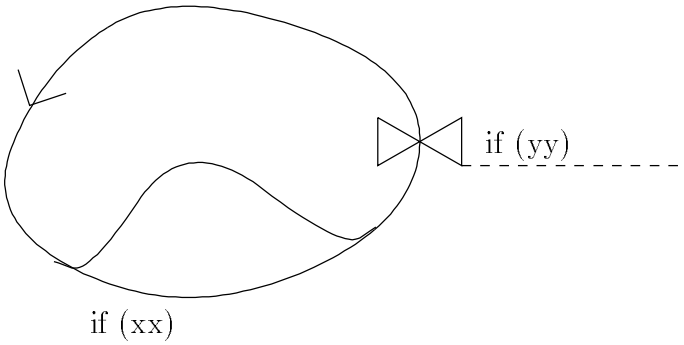

Figure 15: Conditional

5.5 Synchronization State

In this thread diagram, it is possible to extract synchronization state only. This is possible because we carefully avoid to add states to communication lines. If we ignore the information on local token and direction of connection, only synchronization state is remain. In this abstraction, a direction of rendezvous is just a constraint in a token initialization. The positions of tokens fully represent all the synchronization state. Using process wide clock time i, synchronization state S for thread diagram $P\{p, p', p'', ...p^{(n)}\}$ is as follows:

$$S_i = \{p_i, {p'}_i, {p''}_i, ...p_i^{(n)}\}$$

where p_i means position token in thread. Since all the local position p is finite, S is also finite.

In this sense, expressiveness of thread diagram is equivalent to non-deterministic finite state machine. For example, a mapping from a map $G \rightarrow G'$ to a truth value set $\{T, F\}$ is a transition matrix of this thread diagram. A state G is a dead lock state when it does not have mapping from G to T in transition matrix. Various verification schemes based on finite state machines can be applied here, for example, temporal logic verification [5].

In the next section, using various examples, abstraction methods in thread diagram is discussed.

6 Examples

In concurrent object oriented language, a simple remote procedure call type message passing is not enough. For example, ABCL[9] supports three kinds of message passing: Now, Past and Future. Followings are example implementations of Now, Past and Future using

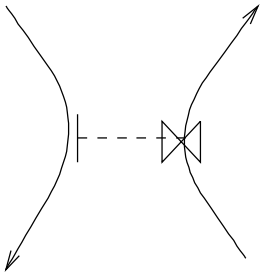

Figure 12: Wait and Pass

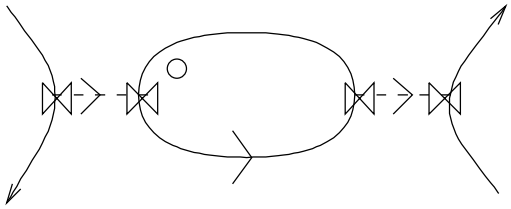

Figure 13: Implementation of Wait and Pass

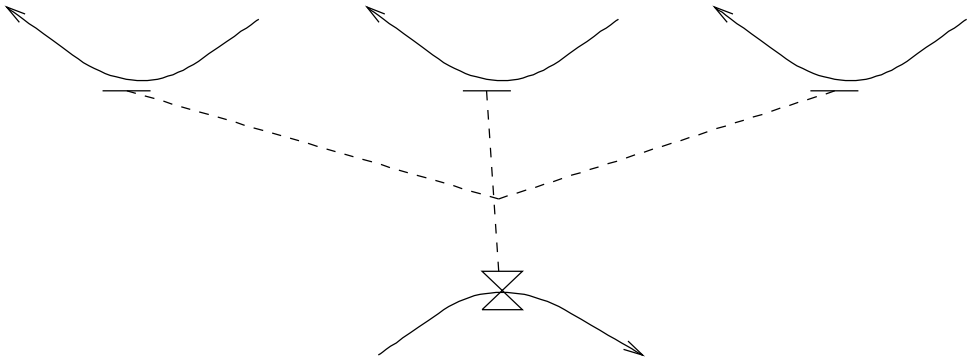

Figure 14: Merge

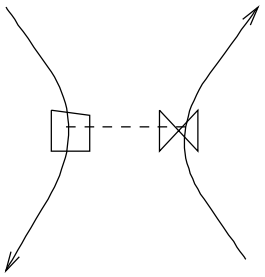

Figure 9: Gate

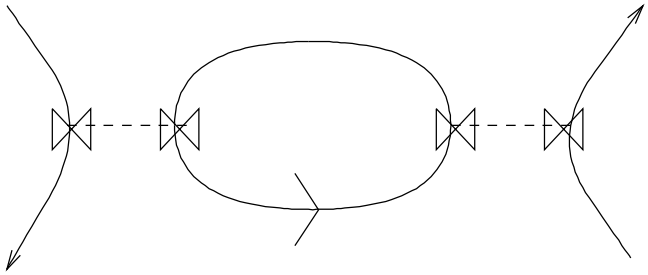

Figure 10: More Complex Communication

To indicate such kind of asymmetry, an arrow is add to the communication link (Fig.11). There is a special syntax for one buffer unidirectional link, since that is very useful and

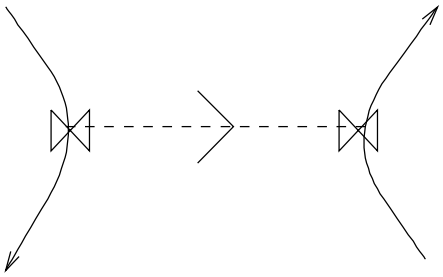

Figure 11: Wait and Wait with Direction

popular one in OO people. This link has one state and it must be properly initialized. If a token points empty position of the buffer, the other synchronization point can be passed once. It leaves some information on the buffer. Later on the other side of buffer, it will be picked up. A wait and pass link creates natural communication direction.

Merge is a syntax sugar on communication line. In Fig.14, three threads send information in random. A $\bowtie$ represents a set of three nondeterministic rendezvous.

Sometimes, an object requires detailed control of synchronization. Using explicit message passing to its meta object solve this problem. However, it is possible to write condi-

It is possible to make rendezvous with more than two node. $\bowtie$ is in *close* state when in coming token will not cause movement of waiting token, on the other hand, it is in *open* state when incoming token will cause movement of them. This does not mean that a connection has state. This state is determined by tokens not connection itself.

Each simple move p_i to p_{i+1}, or execution of $\bowtie$ rule are deterministic move. To represent non-deterministic move, a simple branch is used. A sequence of $< s_{j_0} s_{j_1} ... s_{j_n} >$ represent an arc of a thread. An arc may have a multiple-way branch $< \{< s_{j_0} s_{j_1} ... s_{j_n} >, < s_{k_0} s_{k_1} ... s_{k_m} > \} >$. The most simple non-deterministic branch is $< s_1 \{s_2, s_3\} >$ (Fig.7).

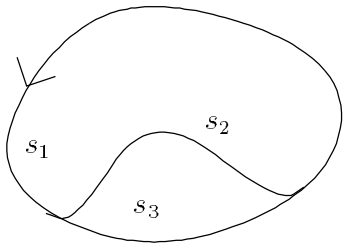

Figure 7: Non Deterministic Path

It is possible to combine non-deterministic choice and rendezvous. It automatically selects possible transition(Fig.8). If rendezvous point is closed state, it will go another path. This is used as a multiple-way rendezvous. In this way, it is possible to preempt rendezvous wait.

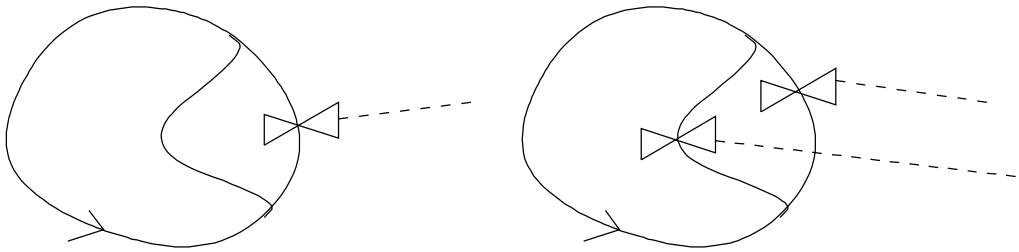

Figure 8: Non Deterministic Select

A gate $\square$ is another kind of synchronization types. It is used with $\bowtie$ types and it works like inverse of $\bowtie$. If any one of other $\bowtie$ is active (but closed), this $\square$ will be open, otherwise it is closed. A $\square$ does not affect other $\bowtie$, so if all other $\bowtie$ become active, all waiting tokens will move and $\square$ will be closed again (Fig.9). This is useful for creating scheduler.

To make more complex communications, we can combine primitives above. Fig.10 is an example of communication with single slot bounded buffer. The communication line has one state to represent empty. N-buffer communication is easily implemented by series of one-buffer communication. The above bounded buffered communication example above is bidirectional. $\bowtie$ communication is a kind of exchange functions[1]. We also need to describe unidirectional communication for example a message passing or remote procedure call. These asymmetries are coming from causality or asymmetry of exchanging information.

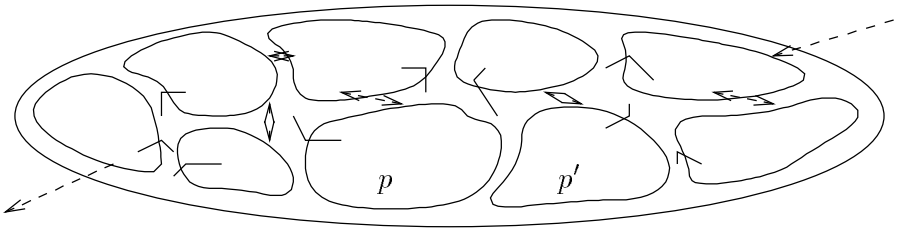

Figure 5: Multiple-Threads Object

The threads are communicating each other. The process P communicates with the outside using normal asynchronous message passing. We cannot determine global clock here. The message queue is accessed by some thread segment one at a time. In this sense some of threads have special side effect to outside. In the next section, communication primitives among threads and its semantics are discussed.

5.4 Communication Primitives

Tokens will run over rings according to synchronization rules. If no communication connection, at token simply proceed the segment from its tail to head. The most simple communication type in thread diagram is rendezvous $\bowtie$. $\bowtie$ represents synchronization point in a thread, which is connected to other synchronization points by a dashed line. A connection is represented by a sequence of pair of synchronization point (tail of segment) and its type.

$$<< s_{j_0}^{(i_0)} t_0 >< s_{j_1}^{(i_1)} t_1 > ... < s_{j_n}^{(i_n)} t_n >>$$

where s_j^i means j th segment of thread $s^{(i)}$ and t_k is type of communication such as $\bowtie$.

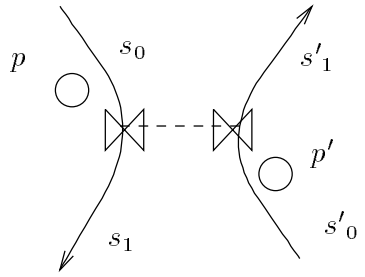

Figure 6: Wait and Wait

If tokens are all available in each $\bowtie$ synchronization points, all tokens proceed at once. If not, tokens will stay at $\bowtie$ point until all tokens available. We call a $\bowtie$ with staying token *active*. For example, in a connection $<< s_0\downarrow \bowtie >< s'_0\downarrow \bowtie >>$ between threads p and p', a state $p_0 = s_0\downarrow, p'_1 = s'_0\downarrow$ causes a transition:

$$p_1 = s_0\uparrow, p'_1 = s'_0\uparrow$$

debugging. If an execution diagram is easily understood, it also can be used for synthesizing multiple threads program.

5.3 The Thread Diagram

A thread in a stable stage is a repeated activity. To represent repeating we use a directed ring p. p can be divide into segments of state s_i, where i is an index of n segments in p : $< s_0 s_1 ... s_n >$. There is exactly one token p in a ring, so we need not distinguish token and ring. A ring is called p, if it has token p.

s_i is a directed segment of a ring, $s_i\uparrow$ means head of s_i, and $s_i\downarrow$ means tail of s_i. Usually we use tail of segment for token positioning and a down arrow may omit. Fig.4 has two segments s_0, s_1 and $s_0\uparrow = s_1\downarrow, s_1\uparrow = s_0\downarrow$. Adding to its position p usually carries local states $L(p)$, value of instance variable, of the thread.

Since the token p is running discrete segment, local time of thread is an integer and it can be defined easily. When a token proceed one segment, local time is incremented by an arbitrary finite positive number. It starts from 0. A local time increase monotonicly but not uniformly, later we use this freedom for making a clock over multi-thread. A local state of token is indexed with local time l. $L(p_l)$ is the last local state of token p at local time l. p_l itself means a position of token in a ring. In this case, p_l is always in position $s_0\downarrow$ or $s_1\downarrow$.

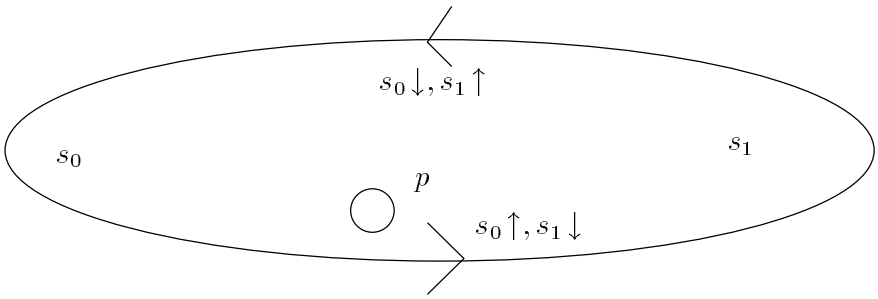

Figure 4: Simple Thread

A process P may contain a set of threads (Fig.5). In Muse system, some of the threads are meta objects and some others are collapsed objects.

$$P \equiv \{p, p', p'', ...p^{(n)}\}$$

Synchronous communications determine a partial order on a finite set of local clocks. Synchronous means that a communication between two thread will occur at the same time. We can make a full ordered over P, and adjust all local clocks to the order using successive positive integer. This is a process wide clock i. In each clock, one step of token movement is occurred in some thread in P.

A state of process P_i is a set of the thread states where i is process wide clock of P.

$$P_i \equiv \{p_i, {p'}_i, {p''}_i, ...p_i^{(n)}\}$$

threads description.

5.2 Requirements

At first, we do not want to describe full detail of process communications, on the contrary, concise and abstracted description. To make problems easy, execution of parallel programs are divided into two stages. From the operating system view, when an application program is running, it does not create processes continuously. So there are two stages:

stable stage: Fixed number of processes are running in a cooperating way.

reconfiguration stage: New processes and new connection between these processes are created.

The stable stage may last infinitely but reconfiguration stage may not. If reconfiguration stage will last very long, it cannot run efficiently, since cost of process creation is usually highest one. We want to concentrate on stable stage. Here is a list of requirements.

- a model for stable processes, it does not change configuration.
- a model for fixed number of processes.
- a finite model for decidability.
- visible synchronization states and invisible object states.

Some of the nondeterminism comes from synchronization mechanism. It creates a state of synchronization which is independent from states of original object. It is necessary to separate the states of synchronization from the states of the object. CCS has large expressiveness, but it does not distinguish communication, synchronization and the states of the object.

Concurrent program run in a non-deterministic way, so description must cope with non-determinism. Non-determinism is a hard part of concurrent programming, so it is helpful to see it in a visible way.

From the software engineering view point, compositability is important. If two diagrams cannot connect each other directly, there is no such compositability. If we use the state diagram, a composition of two independent parallel execution makes completely different diagram. From this point of view, Petri-Net has a good nature.

It is required that our diagram can apply to practical programming languages. Even sequential programming language such as C can run in a distributed way using operating system functions. The most useful nature of object oriented programming is its support for abstraction. An abstraction can be thought as an encapsulation. The encapsulation mechanism for complex communication will be useful.

The model should provide the ability for users to examine various propositions such as dead lock detection, liveness or reachability.

The diagram should also used for a system analysis. For example, detecting critical path, estimating performance or response time. It is also usable for algorithmic program

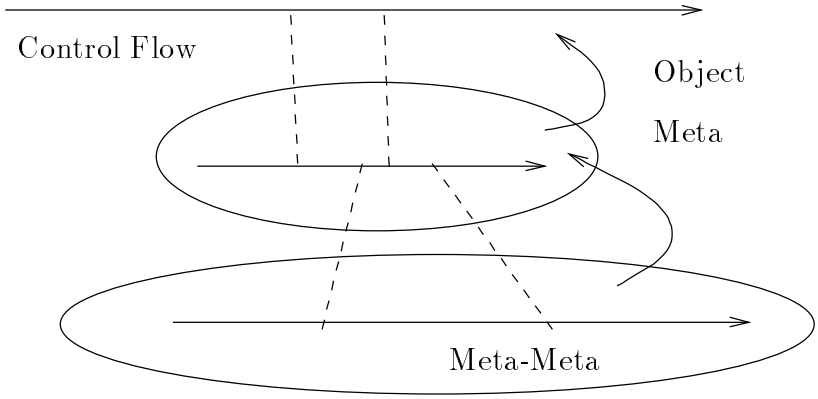

Figure 3: Concurrent Execution of Meta and Object

5 Object with Multiple-Thread

Basically an object in Muse is a process having only one activity. But as we mentioned before, parallel reflection and collapsing combines several activities in a tightly coupled way. From one point of view, they are still independent objects communicating each other by message passing. But their message passing will have another characteristics, which is a combination of intra-object data accessing and synchronization. A message passing will become more efficient and fast using this character, because accessing and synchronization can be directly implemented. In other words, a package of meta objects and collapsing objects is created, and their activities will be a set of thread.

A multi-thread makes it possible to reduce expensive message passing, but at the cost of managements of the threads and inter-thread communication. All these managements are meta level computation. Problem is how to program a multi-thread object. Object oriented programming cannot be used directly here. An abstraction of synchronization relationship is necessary here. Here we show an abstraction technique using thread diagram.

5.1 Synchronization Abstraction

The Muse Operating System runs objects written in various programming languages. Some of them are sequential languages, and the others are parallel programming languages. Communication between these languages and Muse operating system is message passing, unlike in classical operating system, these are using kernel traps. There are another kind of interaction in Muse: Parallel Reflection. Representations of these interactions and activities of the processes are proposed.

Various kinds of models or diagrams for parallel computations are already discussed in many places, for example CCS[4], Petri-Net[6] or Temporal Logic[3].

The thread diagram we introduce here is designed for multiple-threads object. One principle of designing our diagram is decidability for various kind of queries. The other principle is practical usefulness: fitness to the current programming language and visibility. Before describing detail of thread diagram, let us show basic requirements of multiple-

in the meta level. Since object level computation and the meta level computation cannot run simulteanously in a true sense, there is an error of replication between meta level data structure and object level data structure. Such an error is synchronized in an ad-hoc way by causal relationship (Fig.2). This is also a part of definition of causal relationship. Various

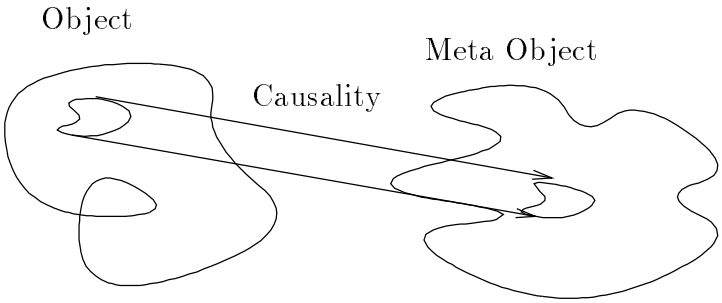

Figure 2: Replication Base Reflection

errors may occur in the procedure of the replication:

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

synchronization error: In an interval between two synchronization, both reflected data in object level or meta level may be changed, then it may generate an replication error.

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

In spite of these errors, we still make this type of reflection useful if we can define the operational semantics clearly. It contains error from the view point of true reflective causal relationship, but if these errors are well defined, we can still use them in a consistent way.

We establish a separation between meta level description and object level description here, because reified data in object level is only defined in terms of object level. If an object level language contains concurrent execution, the access or synchronization procedure of reified data structure can be represented in the object level programming construct. If an object level language does not support concurrent execution, the synchronization procedure is modification of object level data structure from outside of the language. In this case, the reified data is actually a procedure which has states. The description of reflected data is separated from meta-interpreter, however in order to determine a semantics of reflection we still need a meta-interpreter.

In this scheme, there are no suspension in reflective procedure (Fig.3). Object level program will change meta level data structure as an object level data structure. These changes are automatically incorporated into meta object with some delay, via replication mechanism. Conversely meta object can change object level program or data.

In Parallel Reflection, an object, its meta object and replication mechanism (casual connection) are running as threads in one object.

4.1 Classical Reflection Scheme

In the classical reflection schemes, reflection is implemented in a sequential way. In a sequential implementation, there are three important requirements:

- Meta circular interpreter description,
- Control passing mechanism over meta levels,
- Passing data between two different meta levels.

We can define useful operations beyond normal program execution, which include catch/throw operation, error handling, signal/interrupt and so on. In Smith [7], level shifting are syntactically defined as ↑ or lambda expression with *reflect* keyword. Operations on different level are represented by explicit description of continuation treatment such as *normalize*. In Friedman's approach [2], communication between meta level and object level is represented with two basic operation *reification* and *reflection*.

In reification and reflection, the control flow of the object level program is transferred to meta level (Fig.1). In other words, the execution of object level is suspended. Between the reification and the reflection, any mutation of reified data in object level is not allowed, since it causes the error of reflection.

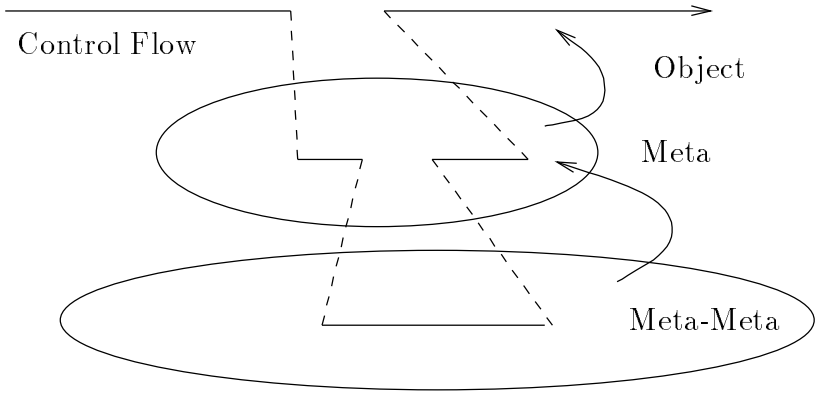

Figure 1: Classical Reflection

4.2 Replication Base Reflection

Replication is a copy of original object with automatic equivalence maintenance. Our new reflection scheme features replication as a communication between meta object and normal object.

To initiate reflection, a new data structure is created. This data structure is not an instance of meta-level data, and it is defined in terms of object level description. A causal relationship between a data and some data in meta level is established as a mechanism of reflection i.e. replication mechanism. This causal relationship is operationally defined

levels of reflective hierarchy. They are a roughly corresponding to the hierarchy of the operating system implementation.

Object Level: Application programs or normal objects are running on this level.

Meta Level: Kernel dependent local state of an object such as memory segment table, context structure or scheduler are in this level as meta objects. These are hardware independent kernel parts.

Meta-Meta Level: Hardware dependent parts and operating system kernel are in this level. Meta level of this level is this level itself, so no further meta level.

3 Collapsing

An important feature of Muse operating system is collapsing. Before defining collapsing, here we define some important words in Muse.

object A unit of abstraction for programs, an object has a unit of code and data

process A unit of code and data with activities in objects from kernel's point of view

activity A current of executions for a virtual processor

thread each activity in a process

group object Representative object of a set of objects

Objects in Muse are concurrent objects. One object is corresponding to a process in normal case. They have separate activities, separate memory spaces, separate scheduling structures and separate code and data each other. A process is a special case of group object, which is a set of objects for a management unit in Operating System kernel.

Several objects can be combined to share some of those for attaining higher performance such as sharing codes, sharing memory space or combined scheduling. We call such a unit a collapsed object. This procedure is the collapsing. Collapsing makes multiple threads in a process. This paper shows a frame work of detailed communication among multiple threads.

Collapsing does not create multiple threads within an object, but in Muse multiple threads are also required with in an object, because of the reflection in Muse. In next section we discuss Parallel Reflection in Muse.

4 Parallel Reflection

In this paper, accessing a method of a meta object is performed in a way different from the 3-Lisp's approach. There are a lot of concurrent objects in Muse system. This forces reflective accesses in parallel, i.e. some meta objects are accessed simultaneously by several different objects. A meta object may be called while it's normal level object is running, such as an interrupt.

Thread Diagram

Shinji Kono, Masaki Yamada 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

Nowadays, every computer is connected to some kind of networks. Computers perform parallel processing. Every application should run in parallel or in a distributed way for higher speed and availability. Already various communication primitives have been proposed to implement parallel computations, but they are very complicated.

What we are concerned with here, is the abstraction of synchronization. Fine grain parallelism are not considered here, because in such kind of systems, it is impossible to pay attention to each synchronization. So here we consider coarse-grain parallelism.

What is the difficulty in parallel and distributed computation? There are several activities running on some shared resources, so that some kind of control over them is necessary. We want to find a way to manage parallel processing in a good way like structured programming or object oriented programming. However, even in object oriented programming, there are no way of abstracting parallel process managements. In this paper, we use reflective hierarchy and thread diagram for synchronization abstraction.

2 Muse - A Reflective Operating System

The Muse Operating System is an object-oriented distributed operating system with meta object concept. An object has several meta objects which represent its computation. Since a meta object also has its meta objects, the tower of meta level hierarchy is naturally created. A meta object of an object is the object itself in some sense, so accessing to its meta object is called reflection. Thus, meta level hierarchy is called reflective hierarchy.

Reflective hierarchy is naturally infinite, however, direct implementation of such infinite tower is not possible nor desirable. In Muse system, unlike another reflective systems such as 3-Lisp [7] or meta object system [8], reflection is used as the frame work of operating system and, therefore it is not fully reflective system. For example, Muse has only three