p6: A State Abstraction-Based Parallel Object-Oriented Language

Research Report No. 96-02

by

Yasushi KUNO and Atsuo OHKI [1]

February 1996

[1]Graduate School of Systems Management, The University of Tsukuba, Tokyo, 3-29-1, Otsuka, Bunkyo-ku, Tokyo 112 JAPAN

Abstract

We have been proposing concept of "abstract state," in which internal states of encapsulated objects are made visible as a part of objects' external interface. In this paper, an idea of synchronization mechanism based on abstract states is described, along with design/implementation of parallel object-oriented language "p6." Because abstract states are part of objects' interface and observable/understandable from outside, abstract state-based synchronization is superior to guards and other existing synchronization scheme in that selective message sending and multi-object synchronization can be described, and codes are more comprehensive. An implementation of p6 on a shared-memory multiprocessor and its evaluation is also described, with the result that efficient implementation of multiple-object synchronization is possible.

Chapter 1

Introduction

In object-oriented programming languages, objects' internal states are hidden inside and protected from external access. The term "encapsulation" or "information hiding" are used to denote this property, and it greatly eases program design because objects' internal states can be designed independent of objects' users (client objects).

However, in some cases it is desirable to expose objects' state information to their clients. We have proposed a mechanism called "state abstraction," in which such information can be used from clients in a controlled fashon[1]. In this report, application of state abstraction to parallel programming languages and its initial evaluation are described. In chapter 2, the idea of state abstraction is briefly introduced. In chapter 3, application of state abstraction to parallel languages is discussed. In chapter 4, design of p6, a parallel object-oriented programming language, which incorporates state abstraction-based synchronization, is explained. In chapter 5, implementation and initial evaluation results of p6 language on a SMP (symmetric multiprocessor) are described. Finally in chapter 6, discussion and summary are presented.

Chapter 2

Objects and Their Abstract States

2.1 Pros and Cons of Information Hiding

In object-oriented programming languages (OOPL), objects' state information is stored in their instance (or state-) variables, and those variables are not directly accessible from outside. This kind of protection is usually refereed as "encapsulation." Objects' external interface solely consists of information about their methods (operations). The results of encapsulation are:

- Information hiding — Internal data structure of objects can be designed/changed without affecting client code (outside code that make use of these objects).
- Polymorphism — Various objects can be used interchangeably as long as their external interface remains compatible.

However, not all of the methods included in the objects' interface is callable at any moment. For example, in case of a "stack" object, its "pop" method is not available when the stack is empty. Similarly, in case of bounded buffer object, "put" method cannot proceed when the buffer is full. In short, availability of methods are dependent of objects' internal states. When using classical OOPL, this problem has been approached in the following way:

A. Those methods that cannot be applied in the current state notify failure, by using status flag or exception.

B. Prepare additional method to examine objects' current internal state.

The former strategy is applicable only when the caller need to know simple availability (yes/no) information. For example, imagine the situation that two buffers are available and we want to put the data in either of them, but want to avoid one whose remaining capacity is less than 50% while the other retains more capacity. [1]

The latter strategy does not have this problem, but cannot be used in parallel programming language because obtained state information can be obsoleted at any moment when other objects concurrently change the state of the object.

2.2 The Idea of Abstract State

The authors' opinion is that the problems stated above are direct consequence of information hiding, in which objects' state information is perfectly shielded from external access. However, some essence of objects' state information should be exposed in a controlled fashion. In fact, both of two strategies listed in the above section exposes some part of objects' state (by using flags/exceptions or additional methods).

However, exposing internal state variables as is will be undesirable because it breaks major benefits of encapsulation stated above. Thus, our approach is to enumerate "abstract" distinguishable states of an object (just like enumeration types in Pascal language), and make only this value observable from outside the object. We call this framework "state abstraction," and enumerated values noted above as "abstract states."

In case of the bounded buffer object, its abstract states will be as follows (we enclose abstract state names in braces):

```
{empty, mid, full}
```

In case whether more than 50% capacity remains is an issue, we can change this to "`{empty, midlow, midhigh, full}`." As to application of state abstraction in sequential programming language, see [1].

[1] We can attain this behavior by preparing put method that fails when remaining capacity is less than 50%, but this is not very smart.

Chapter 3

Parallel Object-Oriented Languages and Abstract States

3.1 Parallel Languages and Synchronization

In parallel programming, we need some synchronization mechanism that controls orderings of multiple concurrent execution activities. There are several reasons for the need of synchronization mechanism; one major use of it is to delay execution until some specific method becomes runnable (e.g., to delay "put" method while a bounded buffer object is in "full" state).

In pure actor model, the unit (scope, extent) of mutual exclusion is single actor, and all messages (method invocation requests) are processed on first-come first-served (FCFS) basis. This model is very simple and clear, but to make a practical programming language, selective message processing (just as in processing "get" only when the buffer is non-empty) will be crucial (in pure actor model, this kind of action is accomplished through use of multiple separate actors, which is all right for a computational model but awkward for a practical programming language).

The consequence is that the majority of parallel object-oriented programming languages are equipped with various feature-rich selective message receive mechanisms, such as guards, enabled sets, synchronizers, wait/signal (like Hoare's monitors), explicit message receive, meta-methods, and so on. Note that function of these mechanisms are to delay processing of some (once received) messages, and the message sender cannot perceive this delay. Thus, the sender cannot change their action according to whether the message being

sent will be processed immediately or not.

3.2 State Abstraction-Based Synchronization

Above consideration led us to the idea of state abstraction-based synchronization, in which synchronization actions are described using abstract state. In this framework, we attach abstract state constraints on each argument in a method header:

```
put = method({empty,mid}b:bbuf ...)
```

The above code indicates that "put" method can execute only when the first argument (receiver) object, which is of `bbuf` type, must be in either `empty` or `mid` abstract state. If this condition does not hold when the message is being sent, message sending itself is delayed until the condition becomes satisfied (through the execution of other — "get" in this case — method).

This framework is somewhat similar to "enabled set" approach, but differs in the following points:

(1) Abstract state information is included in objects' external interface and is observable from outside.

(2) Multiple arguments of a method can have state constraints, leading to multiple-object synchronization.

As for (1), we might use this information to accomplish selective message sending. See the following piece of code:

```
select b1!put(...) or b2!put(...) end
```

This code put an item to either of two buffers which is not full, but execution is delayed if both were full.

The point (2) leads to very powerful and easy-to-use synchronization specification. For example, following code could be used to solve the infamous dining philosopher problem:

```
pick = method({down}f1:fork, {down}f2:fork)
```

Above method header specifies that the method `pick` is accepted only when two folk `f1` and `f2` are both in `down` state.

The framework of state abstraction has some similarities with Milner's process calculus [6], in which multiple agents are connected through their ports, and computation is modeled as message exchanges plus each agents' state changes.

In process calculus, agents are associated with their limited set of states, and events (= message send/receive and internal actions) triggers state transition. States in process calculus correspond to observable difference in agents' behavior; even when internal structure of the object is a complex one (with multiple sub-agents), these complex internal states do not expose to outside as long as they are not distinguishable from the object's observable behavior. Thus, states in process calculus is largely equivalent to abstract states in our framework.

Chapter 4

p6 — COOPL with State Abstraction

4.1 Design Criteria for p6 Language

To show practicality and usefulness of the idea presented in the previous chapter, we have designed and implemented a concurrent object-oriented programming language called "p6," whose only synchronization mechanism is state abstraction-based synchronization. Here is the design criteria for p6:

a. p6 should be class-based concurrent object-oriented language, because (1) declaration of abstract states will naturally fit into class declaration, and (2) class inheritance with state partitioning will be interesting research topic as a next stage.

b. p6's goal is provision of minimal test-bed for state abstraction-based synchronization, so complex language features such as dynamic dispatch, inheritance, type parameters (genericity), and separate compilation are left out.

c. p6 have only one message type in order to keep language simple and compact. However, various message styles can be expressed by combination of this base message type and support objects.

4.2 p6's Message Facility

In p6, message sending is expressed in the following syntax:

obj`!`*method*`(`*arg*`,`$\cdots$`)`
var `:=` *obj*`!`*method*`(`*arg*`,`$\cdots$`)`

The first line sends message (with selector *method*) to *obj*, and does not receive a reply value. The second line is likewise, but receive reply value and stores it into *var*. p6 is a strongly typed language; each expression has a unique type determined at compile-time. Let us express this type of an expression *exp* as *type*[*exp*]. Then, above two lines are considered equivalent to the following lines:

typeof[*obj*]`!`*method*`(`*obj*`,` *arg*`,`$\cdots$`)`
var `:=` *typeof*[*obj*]`!`*method*`(`*obj*`,` *arg*`,`$\cdots$`)`

Thus, method selection in p6 relies only on the type of the first argument (receiver object). [1]

As in process calculus, p6's message is synchronous, in that execution of message sender is delayed when abstract state constraints associated with the method are not satisfied. This also holds in the case where the sender does not receive reply from the receiver object. However, once the synchronization constraint is satisfied, both object execute in parallel (fig. 1 left). In the case of message with reply, the sender is suspended until a reply message arrives (fig. 1 right). In both cases, it is assured that the abstract state constraints have been satisfied at least once before the receiver resumes execution (however, those states might already have been changed by another object or the receiver itself).

Note that `reply` statement does not terminate the receiver's method execution; receiver continues until end of the method body is reached or it explicitly executes `exit` statement. For the convenience of the programmers, p6 also have `return` statement, which is semantically equivalent to `reply` followed by `exit`.

p6 does not have asynchronous messages and future messages, but they can be implemented by introducing relay objects. In the case of asynchronous

[1] Currently, p6 does not have dynamic dispatch functionality. In the future, we are planning to incorporate dynamic dispatch through type generator `any[`T`]`, as in Misty[2] language proposed by one of the author.

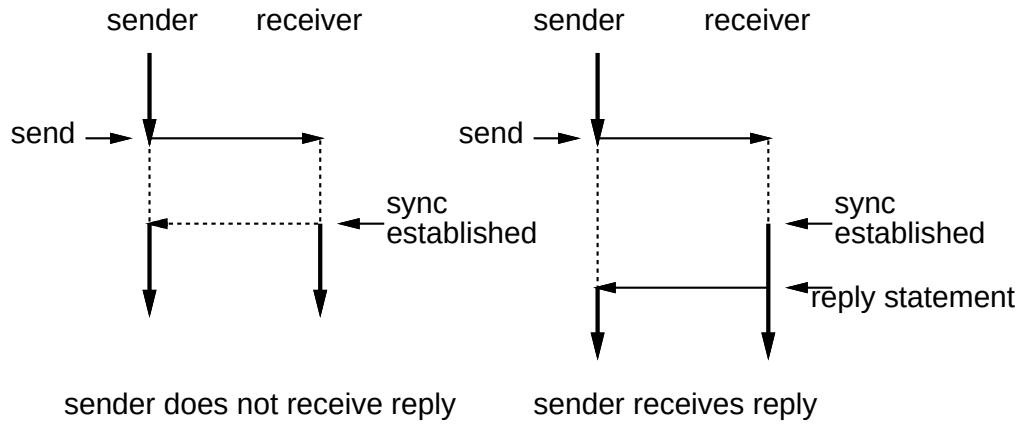

Figure 4.1: Message Sends in p6

(past mode) message (see fig. 4.2), the relay object simply passes arguments handed by the sender unmodified to the receiver, and as the relay does not have abstract state constraints, the sender continues execution without delay. The relay itself must wait until abstract state constraints of the receiver method become satisfied.

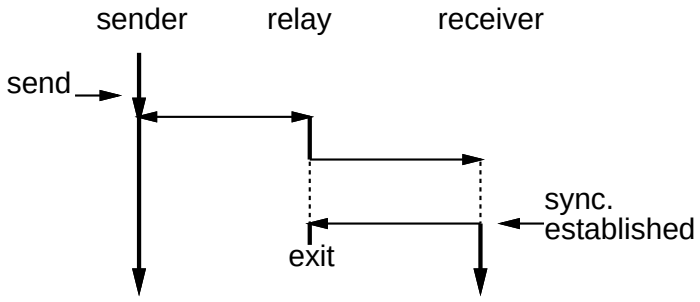

Figure 4.2: Implementation of Past Message using Relay

When the sender wants to send asynchronously and also wants to receive reply (future message), the relay object waits until the receiver returns the reply message and stores it. On the other hand, when the sender want to use reply value, it sends "touch" message to the relay. If the reply has yet to been stored in the relay, invocation of this "touch" method is delayed (see

fig. 4.3). Finally when the reply value is available, "touch" method returns the value to the original sender.

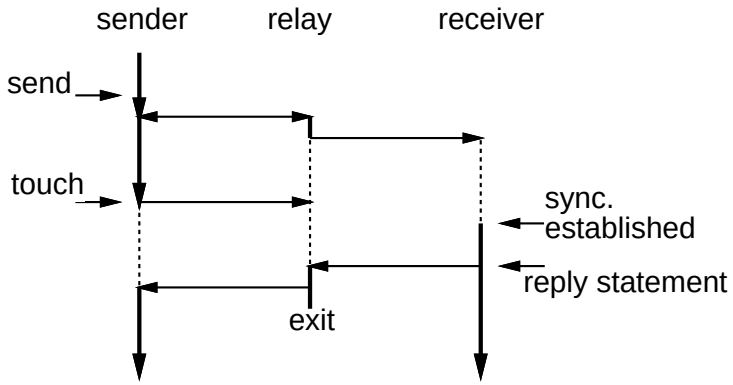

Figure 4.3: Implementation of Future Message using Relay

4.3 Example 1: Bounded Buffer

To demonstrate basic syntax and functionalities of p6, an example of bounded buffer class is presented. The syntax of p6 is modeled after CLU[3] and Misty[2].

```
aint = array[int]                                                    %1
bbuf = class { full, empty, mid }                                    %2
  slot a:aint, size:int, c:int, i:int, o:int                         %3
  new = method(n:int) replies(bbuf{empty})                           %4
    return(bbuf$[a:aint!new(n), size:n, c:0, i:0, o:0]!{empty})      %5
  end new
  put = method({empty,mid}b:bbuf{mid,full}, i:int)                   %6
    b.i := (b.i + 1) // b.size                                       %7
    b.c := b.c + 1
    b.a[b.i] := i
    if b.c = b.size then b!{full} else b!{mid} end
  end put
  get = method({full,mid}b:bbuf{mid,empty}) replies(int)             %8
    b.o := (b.o + 1) // b.size                                       %9
```

```
    b.c := b.c - 1
    v:int := b.a[b.o]
    if b.c = 0 then b!{empty} else b!{mid} end
    return(v)
  end get
end bbuf
```

1. Define `aint` as an "array of integer" type.

2. Define `bbuf` to be a class (and hence a type). A `bbuf` object is in one of three abstract states declared here.

3. A `bbuf` object's internal representation is an implicit record with five state valiable: `a`, `size`, `c`, `i` and `o` (the record has additional hidden field to store abstract state value and mutex data structures). The above variables represent buffer array, buffer size, number of items currently stored, input pointer and output pointer respectively (circular buffer algorithm is used).

4. `new` method takes an integer argument and returns `bbuf` object in abstract state `empty`. The integer argument specifies size of the created `bbuf`.

5. The body of `new` allocates internal record with appropriate field values, and set its abstract state to `empty` before returning.

6. `put` method can proceed only when its first argument, which must be a `bbuf` have either state `emtpy` or `mid` (otherwise the method call is delayed), and expedted to set the `bbuf`'s state to either `mid` or `full`.

7. The body of `put` advances input pointer and stores the data to the prace where input pointer point to, increments item count, compares the value against `size` and sets state to either `full` or `mid` according to the result of comparison.

8. `get` method can proceed only when its first argument, which must be a `bbuf`, is in abstract state `mid` or `full`, and is expected to set the `bbuf`'s state to either `mid` or `empty`.

9. The body of `get` advances output pointer and extracts value from where it points to, decrements item count, compares the value against zero, and sets state to either `mid` or `empty` according to the result of comparison.

Next, we present an example of transferring data using `bbuf` presented above. Below is the class `worker` that generate/consume specified number of data item:

```
worker = class
  produce = method(n:int, b:bbuf)
    i:int := 1
    while i < n do b!put(i); i := i + 1 end
    b!put(0)
  end produce
  consume = method(b:bbuf)
    while b!get() ~= 0 do end
  end consume
end worker
```

Program starts from method `startup` contained in the class `main`:

```
main = class
  startup = method()
    b:bbuf := bbuf!new(100)
    worker!produce(10000, b)
    worker!consume(b)
  end startup
end main
```

4.4 Example 2: Reader/Writer Problem

As an example that requires more complicated synchronization, we present reader/writer problem here. To avoid starvation, when a writer requests write lock, all readers requesting read lock henceafter will be delayed until the writer's request is granted.

```
lock = class { free, reading, rtow, writing }
  slot r:int, w:bool
```

```
    new = method() replies(lock{free})
      reply(lock$[r:0, w:false]!{free})
    end new
    readlock = method({free,reading}r:{reading})
      r.r := r.r + 1; r!{reading}
    end readlock
    readrelease = method({reading,rtow}r:lock{reading,rtow,free})
      r.r := r.r - 1
      if r.r = 0 then r!{free}
      elif r.w then r!{rtow}
      else r!{reading} end
    end readrelease
    writelock = method({free,reading,rtow}r:lock{writing})
      if r.r = 0 then
        r!{writing}
      else
        r.w := true; r!{rtow}; r!waittowrite()
      end
    end writelock
    waittowrite = method({free}r:lock{writing})
      r!{writing}
    end waittowrite
    writerelease = method({writing}r:lock{free})
      r.w := false; r!{free}
    end writerelease
  end lock
```

A lock has four abstract state, namely: `free`, `reading`, `writing`, and `rtow`. When the lock is `free`, both `readlock` method and `writelock` method can be invoked, and changes the lock's state to `reading` and `writing` correspondingly.

In `reading` state, additional `readlock` can be called, but in `writing`, further `writelock` is delayed. When `writerelease` is called, (as the number of writer holding the lock must be one) the lock becomes `free` immediately. However, in the case of `readrelease`, the lock becomes `free` only when all the reader holding a lock released one.

Moreover, when `writelock` is called in `reading` state, the lock's state changes to `rtow`, and `writelock` method invokes `waittowrite` internally to

delay itself until the state becomes `free`. State transition diagram of a `lock` object is presented in fig. 4.4)。

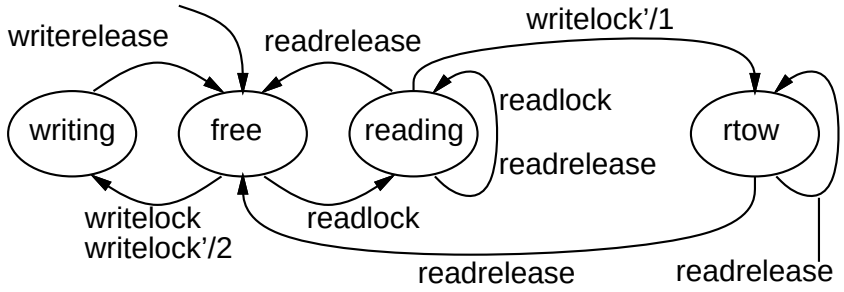

Figure 4.4: State Transition Diaglam of `lock` Object

In this solution, when a writer calls `writelock` in `reading` state, the method suspends untill the lock becomes `free` and then have to hunt for lock with any possible competitor at that moment (no assurance for obtaining the lock next). To correct this situation, it only need to add another new state (say `wturn`), and exclude readers from obtaining the lock while in this state, as follows:

```
rwlock = class { free, reading, wreg, wturn, writing }
  slot r:int, w:int
  new = method() replies(rwlock{free})
    reply(rwlock$[r:0, w:0]!{free})
  end new
  readlock = method({free,reading}r:rwlock{reading})
    r.r := r.r + 1; r!{reading}
  end readlock
  readrelease = method({reading,wreg}r:rwlock{reading,wreg,free})
    r.r := r.r - 1
    if r.w > 0 then
      if r.r = 0 then r!{wturn} else r!{wreg} end
    else
      if r.r = 0 then r!{free} else r!{reading} end
    end
  end readrelease
  writelock = method({free,reading,wreg}r: rwlock{writing})
```

```
    r.w := r.w + 1
    if r.r = 0 then r!{writing} else r!{wreg}; r!waittowrite() end
  end writelock
  waittowrite = method({wturn}r:rwlock{writing})
    r!{writing}
  end waittowrite
  writerelease = method({writing}r:rwlock{free,wturn})
    r.w := r.w - 1
    if r.w = 0 then r!{free} else r!{wturn} end
  end writerelease
end rwlock
```

In general, programming based on abstract state description is straitforward and comprehensive even when state transition become complicated.

4.5 Multi-Object Synchronization

Next we present the infamous "dining philosopher" example. First, only functionality required for"fork" objects is to record its states, thus `fork` object have no instance variables:

```
fork = class { up, down }
  new = method(n:int) replies(fork{down})
    reply(fork$[]!{down})
  end new
  pick = method({down}f1:fork{up}, {down}f2:fork{up})
    f1!{up}; f2!{up}
  end pick
  release = method({up}f:fork{down})
    f!{down}
  end release
end fork
```

Note that `pick` method can proceed only when both arguments (which are `fork` objects) are in `down` state, and their states are changed to `up`. Next, "philosopher" class is as follows:

```
phil = class
  life = method(n:int, f1:fork, f2:fork)
```

```
      while true do
        % thinking...
        fork!pick(f1, f2)
        % eating...
        f1!release(); f2!release()
      end
    end life
  end phil
```

As shown above, multi-object synchronization is valuable tool to write simple, comprehensive and straitforward parallel programs; in case of dining philosopher problem, thinkin in term of "pick up at once" is the most natural and straightforward way of solving the problem.

Chapter 5

p6 Implementation

5.1 Implementing State Abstraction-Based Synchronization

The most basic functionality of state abstraction-based synchronization is to delay method invocation until specified abstract state condition become satisfied. Conceptiall, the delay occurs when the message is being sent. However, there are three alternatives to implement the behavior stated above:

(a) The sender waits until the condition become satisfied.

(b) Insert some intermediate entity (such as message router) between the sender and the receiver, and this intermediate entity handls synchronization.

(c) The receiver performs synchronization before actual computation starts.

In design (a), variable behavior with respect to delay occurence (such as selective message sending) is easily implemented. On the other hand, when all code necessary for synchronizaiton actions are emitted at each message expression, code size will be increased significantly.

The design (b) will be simpler because all synchronization can be handled by intermediate entity, but the fact itself might lead to single performance bottleneck.

In design (c), code size will be moderate without creating single bottleneck, but selective message sending will require extra information exchange between the sender and the receiver.

Merits and demerits noted above have different performance impact when implementation environment changes. For example, on distributed memory systems, methods are normally executed by the CPU that holds the object in its memory, so explicit inter-CPU communication is required when message is sent between objet that are held by different CPU. This means that to decrease message frequency and/or size is crucial to efficiency.

On the other hand, on shared memory systems, every objects are equally accessible from every CPU, and methods are not tied to any specific CPU. Thus, the sender and the receiver can execute in single CPU and simple/efficient procedure calling may substitute more expensive message queueing and scheduling.

5.2 Shared-Memory Implementation of p6

We have implemented p6 on 4CPU SparcServer 1000, which is a shared memory multiprocessor. Our main goal was to obtain lowest possible message sending overhead, so the idea noted above (use call/return instead of message queueing) was extensively used. Below, we explain major design decisions.

We think that the number of abstract states per one object type (class) will not be so large, because programmers enumerates each of them explicitly. Thus, we represent state as single 32-bit word, with each bit corresponding to one state. This representation allows efficient examination of synchronization condition in p6 (e.g. is the argument in one of listed states?), using single bitwise "and" operation against precalculated bit mask. This test is repeated for each of the argument that has associated abstract state constraint.

However, this test must be perfomed atimically without intervention of other objects (of course). To assure atomicity, there are two design choices:

(a) Each execution entity (an object or a threads) locks the relevant object and gain exclusive access.

(b) Limit execution entity that does the testing and avoid race.

Scheme (a) will be more efficient when frequency of race condition is relatively low, because each object simply acquire locks (which are mostly free) and make judgement themselves. However, when the race becomes frequent, repeated lock request will degrade overall system performance.

On the contray, scheme (b) in which synchronization tests are solely performed by single "scheduler thread," each object must communicate to the

scheduler every time they need synchronization and will result in higher overhead, but problem on higher race frequency will be alleviated.

Under above consideration, we have chosen the following scheme:

- Each method invocation (with synchronization) obtains locks on relevant arguments one by one and test is made at the place of invocation — scheme (a). Here, to avoid deadlocks, relevant argument list is first sorted by their logical address and locks are obtaind in increasing address order.

- When all objects are successfully locked and all conditions are satisfied, all locks are released and direct procedure call is made to the receiver's method.

- When any of lock attempts fails, or any of abstract-state condition is not satisfied, all locks are released and current context along with invocation information are stored in a memory data structure, and the data structure is handed to the scheduler thread.

With this mixed scheme, invocations that have low possibility of races are efficiently handled with simple spin-lock and procedure invocation, while race or suspension is handled by the scheduler one-by-one basis without exsessive spin-lock overheads.

5.3 Basic Configuration of p6/SPARC

p6/SPARC is a compiler and runtime systems for p6 that run on SparcServer 1000(50MHz SuperSparc+ $\times$ 4CPU). Its overall structure is presented in fig. 5.1.

p6 source is translated to SPARC assembly code by the compiler, and assembler code is assembled and linked with the runtime library, resulting in SPARC executable.

The runtime library includes standard library class codes and support code that serve as start-up driver, scheduler, etc. The runtime library is written in C. For muti-thread operation, Solaris 2.3 thread library is called for. Execution framework of p6/SPARC is presented in fig. 5.2.

The structure of objects is shown in fig. 5.3. Top of an object is always the "mutex" area, which is 24-byte data structure used for Solaris 2.3 thread

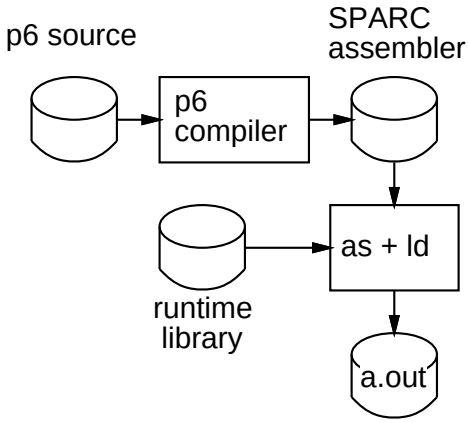

Figure 5.1: Configuration of p6/SPARC System

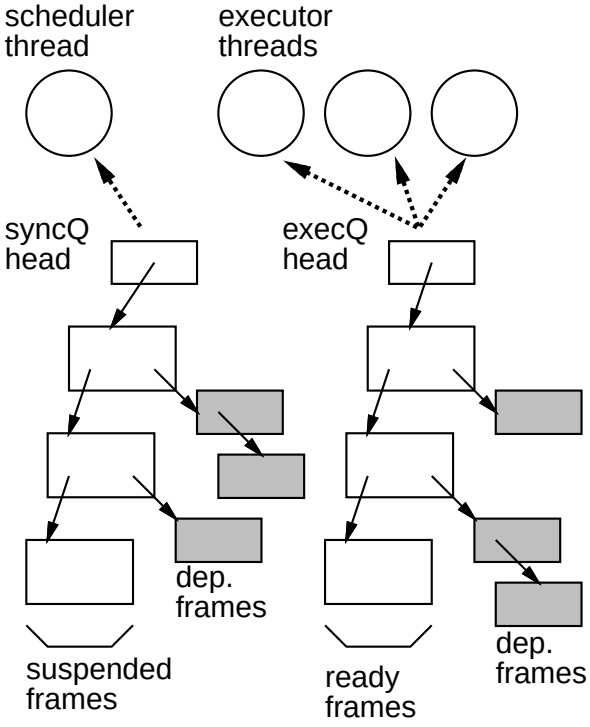

Figure 5.2: Runtime Configuration of p6/SPARC System

library's spin-lock routines. This area was layed at the top in order to avoid offset calculation overhead frequent lock/unlock operation.

Below the "mutex" area is the abstract state word, which is 32-bit, as noted above. Only one bit is set to "1" for each object. Following area is used to store instance variables, whose layout is determined in class-by-class basis.

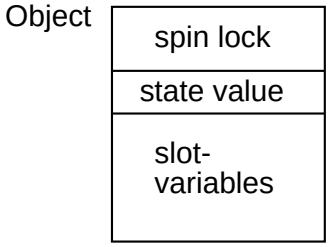

Figure 5.3: Object Structure of p6/SPARC

When a method is executing, state related to the execution (namely arguments, local variables, scratch variables, etc.) are all held by CPU registers. Usage of SPARC resisters is presented in table 5.1. Ordinary stack frames are allocated but not explicitly used. SPARC CPU has register window functionality that automatically store/restore CPU resister to/from stack frames (with help of OS supplied handlers), so our code does not have any explicit stack access.

Every executing method is holding its current state in the CPU registers, but when execution must be suspended, these states are stored to a memory block called "frame" (distinct from SPARC stack frames noted above). In addition to the register save area, this frame block includes mutex area (like objects) and other housekeeping information fields. Suspended frames are put into queues called synchronization queue (syncQ) and execution queue (execQ). The former contains frames that are waiting for abstarct state synchronizatin constraints to be satisfied. The latter contains ready-to-execute frames.

The runtime environment contains single scheduler thread and zero or more executor threads. The scheduler scans the syncQ at some timings [1], and move those frames whose synchronization conditions satisfied to the

[1]In the current implementation, it scans all frames every time any objest state change is reported; we are looking for more efficient implementation.

Table 5.1: Register Usage of p6/SPARC

regs.	usage
g0-g6	scratch regs.
g7	resv. for system
o0-o5	call params.
o0	ret. value
o1	ret. status
o6	stack pointer
o7	link register
l0	frame register
l1-l7	local vars.
i0-i5	args. + local vars.
i6	frame pointer
i7	ret. addr.

execQ. Executor threads repeatedly extract ready frame from the execQ, load the CPU registers, and resume execution. The numbers of executor thread varies according to the length of the execQ, with specified upper bound (to avoid resource starvation due to too many number of threads in the system).

Every frame can have associated (dependent) frames. They are either ready-waiting frame that must wait for execution of the former frame, or reply-waiting frame that must wait for reply value from the former frame. When a frame is moved to the execQ, its ready-waiting frame are also moved to the execQ at the same time [2]. A reply frame is moreved to execQ when the former frame put out reply.

5.4 The Implementation of Send/Suspend/Reply

Basic framework is as presented above, but actual emitted code is more complex. In this section, behavior of actual code is presented in detail. It is somewhat similar to StackThreads[8] in that a message sending is implemented

[2]It is possible that ready-waiting frame once again have its ready-waiting frame (and so forth). They are all moved to the execQ on the situation above.

as a procedure call when possible, but abstract state-based synchronization and multiple thread control for shared memory multiprocessores are added, so detail is different.

Firstly, message sending code is as follows:

(1) Store arguments to argument registers(o0∼o5).

(2) Call receiver objects' method using standard "jump subroutine" instruction.

(3) Upon return, examine status register (o1) and chose one of following three processing.

(3a) o1 = 0: Normal return. When there is reply value, the value is held by o0 register. Continue execution normally.

(3b) o1 = −1: Suspension due to unsatisfied synchronization condition. The register o0 contains the address of the suspended frame block. Create this method's suspension frame, lock the former frame, and register this frame to the former frame as either ready-waiting frame (in case that does not require reply value) or reply-waiting frame (in case that do require reply value), and put the former frame to the syncQ [3]. Finally, store all state information to the frame block, put this frame's address to o0, put −1 to o1, and return.

(3c) o1 = 1: Called method started execution but suspended afterwards. The register o0 contains the address of suspended frame block, and the frame is locked. If reply value is not needed, simply unlock the frame and continue execution. If reply is needed, prepare this method's frame, link it as the former frame's reply frame, lock this method's frame, unlock the former frame and suspend execution (store all state information to the frame block, put this frame's address to o0, put −1 to o1, and return).

Processing at the receiver object's method entry is as follows:

(1) When the method does not have any abstract state constraints, do nothing (proceed as an ordinary procedure call).

[3]Locking is necessary because once the former frame is in the syncQ, scheduler thread might access the former frame and link to the current frame at any moment.

(2) Otherwise, load synchronization-relevant arguments and associated masks onto sequence of paired registers, and sort these pairs to address increasing order. [4]

(3) For each pair, lock the argument and test for condition using and instruction.

Below here, two cases occur:

(4a) All conditions are satisfied: release all locks and continue execution.

(4b) Any failed lock attempt, or unsatisfied condition: release all locks and store current context to frame block, put the address of the frame block to o0, put -1 to o1 and return.

Finally, action of `return` and `reply` is explained. `return` simply put reply value to o0, zero to 01 and return. `reply` is likewise, but create current contexts' suspend frame and put the frame to the readyQ before returning (thus sender's remaining action always gets priority).

5.5 Performance Evaluation

To evaluate efficiency of method invocation, we have measured timings of basic operations in p6/SPARC. The measurement was done by repeating method calls (with empty body) in a loop. The result is presented in table 5.2. In this measurement, method invocation without synchronization becomes plain procedure call and is very fast (0.2μsec). Compared to this, method call with synchronization is three times slower but is still good (0.6μsec). However, such method invariably requires abstract state setting operation within its method body, which is rather slow (2.2μsec). Most of this time is the overhead to wake up scheduler thread, and actually is a conditional variable "signal" call of Solaris 2.3 thread library. This part of implementation is under review. In case of unsatisfied condition, suspension processing/scanning frame by the scheduler/resume processing all requires approx. $10\sim30\mu$sec. Eliminating this overhead is also future consideration.

[4] The number of synchronization-relevant arguments will be small, so the number being N, $\frac{N(N-1)}{2}$ serieses of "compare and swap if necessary" operation sequence are generated.

Table 5.2: Basic Timings of p6/SPARC Operation Calls

Operation	Time
Method Call without Synchronization	0.2μsec.
Method Call with Synchronization	0.6μsec.
(Setting Abstract States)	2.2μsec.
Suspension	28.2μsec.
Scheduling	14.7μsec.
Resume from Suspension	21.3μsec.

5.6 Implementation of Other Features

This section describes planned design of currently unimplemented features. As for selective message sending described in section 3.2, following implementation is possible:

- The sender actually try to execute message sending for each branch, and collect the suspended frame (returned due to unsatisfied abstract state constraints) in a list.
- In case any of message sending was successfully started, discard all frames and continue executing the successful branch.
- In case all of message sending was delayed, hand collected frame list to the scheduler for later processing.
- The scheduler periodically scan the list for any frame to become ready for execution, and resume its execution while discarding all other frames. Additionally, the scheduler notifies the sender about which branch was actually chosen.

Another problem lies with respect to substitution of message sending with more efficient procedure calling; when the sender does not use reply value, the sender needs not wait for the receiver method to complete. However, when the procedure calling was used, the sender cannot resume execution until the receiver returns. This can cause problem when the receiver's method require long period to complete. To solve the problem, following strategy is considered:

- When calling a method and the sender is not using reply value, a timer is started.
- When the timer expires while the called method is running, force suspension of the method and store its context to the frame block.
- Put the frame block to the execQ and resume sender's execution.

Thus, the sender and receiver can execute parallel after the timer expires.

Chapter 6

Discussion

6.1 Merits and Demerits of State Abstraction

In the field of software design methodology and software specification, concept of abstract states is not new. However, in these domains, abstract states are only used on design documents and actual code does not have such data while the software is running. In our proposal, programming language explicitly have syntax to represent abstract states, and each objects' states are actually stored in objects' storage. As a consequence, explicit representation of design decisions on the program code become possible. This also allows run-time state checking when necessary.

Some programming systems with algebraic specification introduces logical specification of ADTs, and correspondence between actual (concrete) code and this specification is maintained through theorem proving techniques. Such system will assure accurate handling of each objects' states. On the other hand, the method proposed in this paper uses finite number of abstract states, and correspondence between objects' internal (concrete) state and abstract state is maintained by the implementer. It might increase implementer's labor, but the clients (programmers that make use of the objects) need only know about the objects' abstract states. Moreover, our intuition is that implementers are already aware of each objects' abstract states, so making this information explicit in the language will clarify the code and be beneficial even to the implementer.

Next, considering state abstraction-based synchronization, the scheme

might look similar to ordinary guard expression because method invocation is delayed at the top of the method. However, as the example of reader/writer problem indicates, method suspension after some computation is also possible by the use of internal method invocation (with synchronization).

Moreover, in the case of guards, as its expression directly refers object's internal state, so the expression itself is not very meaningful to the clients of the object. On the other hand, abstract states are designed as a part of objects' external interface and are meaningful to clients.

The method proposed in this paper allows multiple (and heterogeneous) object synchronization as a primitive operation. This makes description of some kind of problems (such as dining philosopher) easy. Let us note that, popular solution that limit number of philosopher requesting a fork (or already eating) up to four is quite specific of topology (circular) and numbers (five) of philosophers and forks. On the contrary, our method can easily describe arbitrarily (for example, three fork per each philosopher and random topology) configuration.

6.2 Related Works

In the area of concurrent object-oriented programming languages, many research are done with respect to reuse of code through inheritance. Among them are the method that use enabled set[4][5]. This set values (set of method names that can be accepted at some moment) can be considered as an another representation of abstract states. Moreover, Matsuoka et. al.[7] are using these set values actually as states and describes transition between these states.

However, in these works, the main focus is to enable/disable each methods with inheritance frameworks. Features such as multiple object synchronization is not provided either.

In ActorSpace[9] model, each object may have associated "attribute" values, and synchronization condition can refer to those attributes. These "attributes" can act as as abstract states. In fact, dining philosopher example presented in [9] is essentially same as those presented in this paper. However, main focus of ActorSpace model is to separate objects' synchronization condition from objects themselves, and attributes are not regarded as something related to objects' internal state.

Regular Type[10] is a work to formalize method enabling/disabling using

state transition, thus it have some similarity with the method proposed in this paper. However, the work does not speak about implementation in actual language, and neither handles multiple object synchronization.

6.3 Summary and Future Works

In this paper we have proposed state abstraction-based synchronization, and also presented an object-oriented programming language p6, whose sole synchronization scheme is state abstraction-based one. The design of a p6 implementation on shared-memory multiprocessor and its performance is also reported. Our experience is that state-abstraction based synchronization is simple, powerful and comprehensive, and can be implemented efficiently.

Future works include investigation of theorycal background of this synchronization scheme, relation with inheritance mechanism, and experience on development of larger parallel systems in this framework.

Bibliography

[1] Yasushi Kuno: State Abstraction — Yet Another Possibilities of Modules, IPSJ PRO 14-4, 1993.

[2] Yasushi Kuno: Misty — An Object-Oriented Programming Language with Multiple Inheritance and Strict Type Checking, in JSSST ed., Advances in Software Science and Technology, vol. 3, pp. 109-125, Iwanami Shoten and Academic Press, 1991.

[3] Barbara Liskov, John Guttag: Abstraction and Specification in Program Development, MIT Press, 1986.

[4] Chris Tomlinson, Vineet Singh: Inheritance and Synchronization with Enabled-Sets, Proc. OOPSLA'89, pp. 103-112, 1989.

[5] Dennis G. Kafura, Keung Hae Lee: Inheritance in Actor based concurrent object-oriented languages, Proc. ECOOP'89, pp. 131-145, 1989.

[6] Robin Milner: Communication and Concurrency, Prentice Hall, 260p, 1989.

[7] Satoshi Matusoka, Kenjiro Taura, Akinori Yonezawa: Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, Proc. OOPSLA'93, pp. 109-126, 1993.

[8] Akinori Yonezawa, Satoshi Matsuoka, Masahiro Yasugi, Kenjiro Taura: Implementing Concurrent Object-Oriented Languages on Multicomputers, IEEE Parallel and Distributed Technorogy, vol. 1, no. 2, pp. 49-61, 1993.

[9] Gul Agha, Svend Frϕlund, Woo Young Kim, Rajendra Panwar, Anna Patterson, Daniel Sturman: Abstraction and Modularity Mechanisims

for Concurrent Computing, IEEE Parallel and Distributed Technology, vol. 1, no. 2, pp. 3-14, 1993.

[10] Oscar Niestraz, Regular Types for Active Objects, Proc. OOPSLA'93, pp. 1-15, 1993.