Solving Inheritance Anomaly Problems by State Abstraction-Based Synchronization

Yasushi Kuno
Graduate School of Systems Management — Univ. of Tsukuba, Tokyo
3-29-1, Otsuka, Bunkyo-ku, Tokyo 112-0012 Japan.

ABSTRACT. "Abstract state" is a programming language facility that makes the internal state information of abstract data types available from outside, in a controlled fashion. Abstract state is also a useful and efficient tool for specifying synchronization for parallel object-oriented languages. In this paper, we introduce interface- and implementation- inheritance to state abstraction-based parallel language and analyze its descriptive power. As a result, we could obtain (1) clean, comprehensive language design, which is (2) efficiently implementable, (3) avoiding typical inheritance anomaly problems. Additionally, in our scheme (4) type system reflects difference in objects' synchronization behavior, which affects objects' substitutability.
KEYWORDS: Concurrent OO Language, Inheritance Anomaly, Abstract State.

1 Introduction

Inheritance in object-oriented programming languages (OOPLs) is an indispensable functionality for both orderly program structures and code reuse. The same statement also holds for parallel object-oriented languages. However, Matsuoka and Yonezawa[MAT 93] have pointed out the existence of inheritance anomaly problems that prohibit reuse of code containing synchronization actions. Many researchers have proposed language mechanisms to alleviate the problem, but the inheritance of synchronization code is still not widely used. In our opinion, the reason is that simple, clean and efficiently-implementable language designs have not been achieved yet.

We have been proposing a concept of "state abstraction," in which the internal states of objects are made visible from outside in an abstract and controlled fashion. We have also proposed the use of these client-visible states, called "abstract states," as a primitive synchronization mechanism for parallel object-oriented languages[KUN 97].

In this paper, we focus on the interaction between the inheritance mechanism and state abstraction-based synchronization. The major contribution of this paper is as follows: (1) With state abstraction-based synchronization, clean and comprehensive language designs that allow inheritance of synchronization

description are possible. (2) Implementation of state abstraction-based synchronization can be very efficient. (3) State abstraction-based synchronization can solve typical inheritance anomaly problems. (4) In state abstraction-based synchronization, the synchronization behavior of an object becomes part of the object's external interface.

The last point is the most important, because objects that are not substitutable (due to difference in synchronization behavior) should not be declared compatible by the language's type system.

An outline of the rest of this paper is as follows: In section 2, we introduce state abstraction and state abstraction-based synchronization. In section 3, we focus on design decisions to integrate inheritance into state abstraction-based parallel languages. In section 4, we discuss how the design described in section 3 can solve typical inheritance anomaly problems. In section 5, we turn to the issue of implementation and show that state abstraction-based synchronization with inheritance can be compiled to efficient codes. Finally in section 6 and 7, comparisons to related works, a discussion and conclusion are presented.

2 State Abstraction and Synchronization

2.1 State Abstraction

In OOPLs, the state of each object is represented as a set of instance variable values. Due to encapsulation, these values cannot be seen from outside. However, there are many occasions in which the information of an object's internal state is absolutely necessary. For example, an empty stack cannot be popped, or a full bounded buffer cannot be put.

Traditionally, these situations were handled by the programmers in one of the following ways: (1) Make some of the instance variables accessible from outside. (2) Prepare "polling" methods, e.g. "is_empty" for stack, that return internal state information. (3) Try-and-error scheme using error flag variable or (preferably) exception mechanism.

For modern programming practice, (1) is unacceptable because the object's encapsulation is broken. (2) seems reasonable, but is unsuitable for parallel languages because the object's state might change between the polling method calls and actual object access. (3) does not have this problem, but flag tests or exception handlers clutter up the code — exception should be used for really "exceptional" cases, not for normal lines of operation.

As a solution to the problems, we have introduced the "abstract state" as a language feature. For each object, their associated abstract states (similar to Pascal's enumeration types) are declared. For example, a bounded buffer object will be in one of the `{ full, mid, empty }` abstract states.

Abstract state values are set in the objects' method bodies, and can be examined from outside. Note that those values are proper abstraction of internal state values. For example, in the ring-buffer implementation of bounded buffers, the `put` method will set the abstract state to `full` when its input

pointer becomes equal to the output pointer (after advancing the former).

2.2 State Abstraction-Based Synchronization

Now we turn to the issue of parallel language. In parallel programming, concurrent activities require synchronization. We have noticed that most of the uses of synchronization can be described as "delaying an action until the object reaches some specific abstract state."

For example, the putting to a full buffer must be delayed until the buffer becomes `mid` or `empty`, and the getting from an empty buffer must by delayed until the buffer becomes `full` or `mid` [1] . From this observation, it will be natural to treat the method itself as "delayed action," and specify the required condition (set of abstract states) for each method argument, including the receiver object itself [2] .

When the condition is established, the object's abstract state becomes "undefined" and prevents further state match until the method body re-establishes the new state. Thus, state abstraction-based synchronization handles both conditional and non-conditional (mutex) synchronization in a unified manner.

To demonstrate the practicality of our approach, we have designed and implemented a language called p6[KUN 97]. p6's syntax is modeled after Misty[KUN 91] and CLU, but the object model is closer to C++ or Java in that every object is an "implicit record" whose components are instance variables. Hereafter, we will use p6 as an example.

In p6, the bounded buffer's `put` method will be described as follows:

```
put = method({empty,mid}b:bbuf{mid,full}, val:int)
  b.ipt := (b.ipt+1)//b.size; b.cnt := b.cnt+1; b.arr[b.ipt] := val
  if b.cnt = b.size then b!{full} else b!{mid} end  % set new state
end put
```

The method header declares that the `put` method can proceed only when the state of the receiver object `b` is `{ empty, mid }`. It also says that the state of `b` becomes `{ mid, full }` when the method is executed; we plan to use this information for checking validity in the future. Note that the buffer is guaranteed to be non-full when the method actually starts, making the non-full test unnecessary.

The complete p6 code for the bounded buffer is included in the appendix. Further examples, including dining philosopher problem and readers/writers problem, can be found in [KUN 97] [3] .

---

[1] In our model, method calls (message sendings) are synchronous ("now" type); "past" or "future" messages can be implemented using intermediate objects.

[2] As a result, multi-object synchronization can be specified in the same framework.

[3] For example, readers/writers problem with writer preference can naturally be expressed by transition to "new reader prohibited" abstract state.

2.3 Comparison to Other Synchronization Scheme

In this section, we compare the state abstraction-based synchronization against monitors, guards, and accept sets. We have chosen them because they are popular and widely adopted in recent programming languages.

Monitor is one of the most popular synchronization mechanisms; recent languages as Java choose monitor as its built-in synchronization scheme. Monitor can express mutual exclusion in a simple and safe manner. However, when it comes to conditional synchronization (as required in bounded buffers), monitor is as unstructured as gotos; suspension occurs in the middle of the code, and one must throughly follow the code to understand the objects' synchronization behavior. In the case of state abstraction-based synchronization, mutual exclusion and conditional synchronization are specified in the same framework, suspension occurs only at method entry, and synchronization description in the method headers and bodies are more structured.

Guards are as popular as monitors, but are more "declarative" in that they specify synchronization condition as boolean expression. However, there are some difficulties in when and how to evaluate the expression. In naive implementation, the expression need be evaluated at each method invocation attempt and receiver's state change. Moreover, evaluation must be done in the context of callee (receiving object) with mutual exclusion, leading to additional overhead. Yet another problem of guards are that they are predicates over objects' internal representation and are incomprehensive to the clients. Meyer[MEY 93] has pointed out this problem and proposed to use only public, boolean-valued methods as guard expressions, but this leads to more evaluation overheads. In the case of state abstraction-based synchronization, states are simple scalar values, can be examined efficiently without side effects, and their values are available as a part of the objects' public interface.

Accept sets[KAU 89] (sets of method names that can be accepted) and enabled sets[TOM 89] (accept sets that are of first class data type values) are very close to state abstraction in that each accept set value can be mapped to one abstract state. However, set notation cannot handle certain kinds of inheritance anomalies — see section 4. As the last note, there exist similarities between the process calculus[MIL 89] and state abstraction-based synchronization: (1) Agent's client-visible states are abstraction of its internal (more complex) states. (2) Interaction between agents (i.e. method invocation) is possible only when their states match [4] .

[4] In p6, while the receiver's state is represented as an abstract state, the sender's state is represented as the current execution point. We are also working with more complete, "symmetric" models[KUN 96].

3 State Abstraction-Based Synchronization and Inheritance

3.1 Goals and Functionalities of Inheritance

Before proceeding to p6i — an inheritance-enabled version of p6 —, some consideration of the goals and mechanisms of inheritance are required.

In a traditional sense, inheritance consists of the following functionalities:

- Inherit a set of instance variables and their types.
- Inherit method implementations.

These are sometimes called "implementation inheritance," whose goals are:

- Code reuse — write only "different" portion of the new class.
- Code mixin — mix up additional functionality to the base class.

There is another kind of inheritance called "interface inheritance" (or "subtyping"), whose functionality is:

- Inherit method names and signatures.

Goals of interface inheritance are:

- Avoid redundant interface declaration for similar classes.
- Provide basis for generic handling of similar objects.

Older OOPLs such as Smalltalk-80 do not distinguish between implementation and interface inheritance, while newer OOPLs such as Java allow programmers to control them more or less separately [5] .

3.2 Parallel OOPLs and Inheritance

For parallel object-oriented programming languages, a problem called "inheritance anomaly" is known[MAT 93]. Inheritance anomaly means:

- Methods that contain synchronization actions are difficult to inherit — i.e. cannot be shared "as is" among the base class and its subclasses.

The above statement is apparently an "implementation side" view, so we prefer to define the same problem in "interface side" terms. We named this "substitution anomaly:"

- Objects that have compatible interface cannot be substituted with each other because they behave differently with respect to synchronization.

[5] In Java, implementation inheritance forces interface inheritance, but not vice versa.

Note that the substitution anomaly is a slightly wider concept because the definition also applies to objects that do not share implementation. And this is more relevant because dynamic method dispatch, which relies heavily on substitutability, is the heart of object-oriented programming.

The above argument suggests the following principle: if two objects' synchronization behavior differs, they should not be made substitutable by the language's type system.

Some researches are aimed toward definition of subtypes based on its behavioral substitutability. Among these, Liskov and Wings[LIS 94] are not addressing synchronization [6] . America[AME 90] has introduced notion of properties that are used to capture behavioral side of an object, including synchronization. However, synchronization and other behaviors (such as Last-in, First out) are not clearly separated. Nierstrasz[NIE 93] has proposed regular types, in which each object have its own finite state-transition that controls the availabilities of methods, with subtyping facility. However, his work is not tied to the actual programming language so far.

Our approach roughly corresponds to Nierstrasz's work in that every object have finite (abstract-) state and transitions (via method invocation), but in our work actual transitions are explicitly specified by the programmer. A more theorical approach would be based on the equivalence between concurrent processes, as in the process calculus[MIL 89], but we are aimed toward practical/implementable programming languages, just as in [AME 90].

In the following section, we will first show some examples and then explain our subtyping rules.

3.3 Interface Inheritance with State Abstraction-Based Synchronization

Hereafter, we will use p6i as the basis for our examples and discussion. As suggested in section 3.1, p6i has a separate module for interfaces and implementations. Below is the interface module for a bounded buffer object:

```
bbuf = interface { full, mid, empty }
  new = method(n:int) replies(self{empty})
  put = method({empty,mid}b:self{mid,full}, i:int)
  get = method({full,mid}b:self{mid,empty}) replies(int)
end bbuf
```

The type identifier `self` denotes the interface type being defined (`bbuf` in this case).

Next, we can use the interface inheritance to extend the base interface in one of three ways: (1) The set of abstract states can be modified. (2) Method signatures can be modified. (3) New methods can be added.

[6] As their approach is based on ADT's pre- and post-condition, guard-based synchronization can easily be added to their proposal. However, drawbacks of guards noted in the previous section remain intact.

In the case of (1) and (2), only those modifications that do not break substitutability are allowed (described later).

As an example, we can add the `peek` method to examine the next value to be obtained:

```
bbuf_peek = interface extends bbuf
  peek = method({full,mid}b:self{-}) replies(int)
end bbuf_peek
```

Now that the new interface has four methods, namely: `new`, `get`, `put` and `peek` (`{-}` in the argument list indicates that this method does not change the state of the argument object). Abstract states are inherited as is in this case.

When abstract states are to be modified in the new interface, only two cases are allowed: (1) One state in the base interface may be split into two or more states in the new interface. (2) Some of the states in the base interface may be removed.

These restrictions are required to maintain substitutability between the base and new interfaces.

For example, we can create a variant of `bbuf` in which clients can see if half of its space is in use or not:

```
bbuf_hl = interface { empty, low, high, full }
    extends bbuf { mid } becomes { low, high }
end bbuf_hl
```

With an object which implements this interface type, `get` can proceed when the state is `{ low, high, full }`. Additionally, when such objects are used as a `bbuf` object, both `hight` and `low` are treated as `mid`. These mappings can automatically be handled by the compiler and the runtime.

Or alternatively, we can create an unbounded version of the buffer and make it a subtype of `bbuf`:

```
bbuf_u = interface { empty, mid }
    extends bbuf { full } removed
end bbuf_u
```

It may seem odd that the removal of states does not break substitutability, but it is all right because existence of an object with any specific state depends on the programmer anyway. This case corresponds to "constrained subtypes" in Liskov and Wing[LIS 94].

In the above example, we know that unbounded buffers are usable in place of bounded ones in most cases; as for the rare cases (e.g. a code that first fills up a buffer and then passes it to someone), we consider them as logical (design) errors. However, we prohibit removal if it causes abstract state constraint to become empty in certain methods (see below).

Finally, the interface inheritance and substitutability rules for p6i are as follows:

1. An interface consists of an abstract state set and zero or more method signatures.

2. An interface may have one or more superinterface. If no superinterface is specified, the interface is by default a subinterface of built-in interface `object` [7] .
3. A subinterface inherits abstract state set and method signatures from its superinterface. When inheriting method signatures, the special type name `self` denotes the subinterface's type.
4. In the case of two or more superinterfaces (multiple inheritance), the abstract state set of the subinterface becomes the cartesian product of the superinterfaces' state sets, whose elements are a dot-concatenated series of superinterfaces' state names. As for method signatures, see 8.
5. A subinterface may split and/or remove some of its superinterfaces' state. In the case of removal, the removed state virtually exists, but cannot be referred anymore. Removals that cause certain abstract state constraints to become empty (even with possible overriding) are not allowed.
6. A subinterface may add or override method signatures. In case of overriding, the number of arguments and return values (actually 0 or 1) must be the same, argument types must be equal to or more general (i.e. belong to superinterface) than inherited ones, and return types must be equal to or more specific (i.e. belong to subinterfaces) than inherited ones — standard contravariance/covariance rules[AME 90]. `self` for the first argument is an exception, because the first argument (the receiver) is used for dynamic method dispatch.
7. When inheriting method signatures, in abstract state conditions attached to arguments and return values, preconditions (written before each argument) must be equal to or narrower (smaller set) than inherited ones, and postconditions (written after each arguments or the return type) must be equal to or wider (larger set) than inherited ones — analog to standard contravariance/covariance rules[LIS 94].
8. If two or more superinterfaces have a method with the same name, the number of arguments and return values must be identical. Moreover, if all arguments/return types and abstract state conditions are not identical on those, the subinterface must override the method signature according to the rules stated above.

In p6i, the type and interface have 1-to-1 correspondence (no genericity or parameterized interface yet). Thus, type substitutability can be defined as follows:

- Type A can substitute type B if the corresponding interface A' is (indirect-) subinterface of B'.

Intuitively, this roughly corresponds as B having more state distinction and narrower synchronization condition (extended functionality in Liskov and Wing's

[7] `object` has an abstract state set that has one unnamed state, which is split into subinterfaces as necessary.

term), or more constraints (constrained subtypes in Liskov and Wing's term), compared to A. Sound theorical bases for the above rules are yet to be investigated, however.

3.4 Implementation Inheritance with State Abstraction-Based Synchronization

In p6i, object implementation is coded as class module. Every class module must implement one or more interfaces.

```
bbuf_basei = class implements bbuf
  slot arr:aint, size, cnt, ipt, opt:int
  new = ...   % same as in
  get = ...   % p6; see appendix
  put = ...   % for the complete listing.
end bbuf_basei
```

By creating subclass, we can inherit the base class's implementation. Note that the subclass need not implement the same interfaces as its base class; interface inheritance and implementation inheritance are totally independent in p6i.

The subclass can modify its base class implementation in one of the following manner: (1) Subclass may have additional instance variables. (2) Subclass may add, override or modify method definitions.

Note that method signatures can be modified as long as the subclass correctly implements the designated interface(s).

Below is an example of a class that implements `bbuf_peek` interface in the previous section:

```
bbuf_peeki = class implements bbuf_peek extends bbuf_basei
  peek = method({full,mid}b:self{-}) replies(int)
    reply(b.arr[b.opt])
  end peek
end bbuf_basei
```

The `peek` need not set the state of `b` explicitly; it is automatically restored to the original value upon method exit (effect of "`{-}`" in the argument list).

As a next quesiton, when the set of abstract states are changed in a new interface, how much of its corresponding implementation needs to be changed? It may seem that most of the code must be modified because state values must explicitly be set in the method bodies. However, the following observation can alleviate the burden: (1) Some of the methods may not be related to the changed states and can be left intact. (2) Even when the relevant state was actually removed or split, in many cases the programmer can designate an alternative state to use, leaving the method code unchanged. (3) Finally, when we need to determine the new state dynamically, the operation can be performed after the method execution has completed.

(3) comes from the fact that abstract states are some projection of the object's internal states, and thus can be computed functionally on demand.

To support this operation, we introduced "after daemon" methods (in CLOS term) into p6i. Using this scheme, the implementation of `bbuf_hl` example can be written as follows:

```
bbuf_hli = class implements bbuf_hl extends bbuf_basei
  after get{mid}, put{mid} use hilow
  hilow = method({mid}b:self{high,low})
    if b.cnt>=b.size/2 then b!{high} else b!{low} end
  end hilow
end bbuf_hli
```

Line 2 is the heart of this scheme. `bbuf_hl` interface has split the `mid` state into `high` and `low`, so the behavior of `get` and `put` should be modified accordingly. However, this modification is only required after `get` and `put` have successfully been completed and the object's state is `mid`. Only on such occasions, method `hilow` is invoked as an after daemon and the correct state is re-established. Although `hilow` is a private method and is not defined in the interface, public methods can also serve as after daemons when appropriate.

When multiple levels of state splitting has occurred, after daemons are invoked in a cascade, in base class-to-subclass orderings. This is similar to the ordinary method combination (as in CLOS), but differs in that the necessity of invocation is tested at each level.

One might guess that a daemon method is indifferent to a normal method, and thus no special mechanisms are needed. However, this is not the case for the following reasons: (1) As explained in section 2.2, the setting of state values allows other concurrent activities to access those values (and hence the object itself). Thus, ordinary cascaded method calls are insufficient to prevent interference from other activities. With the daemon method, the state setting period is extended to include the daemon method bodies (by the compiler), preventing interference. (2) Even without the above argument, calling extra state setting code explicitly is tedious and fallible task for the programmer.

Another criticism might be that referencing to the superclass variable leads to the breakage of the encapsulation. However, this is indifferent from the subclass variable access problems; we were able to use private/protected distinction as in C++, but we wanted to keep the language simple.

4 Solution to Inheritance Anomaly Examples

4.1 Typical Inheritance Anomaly Examples

As noted in section 1, inheritance anomaly is the major obstacle against the code reuse of parallel object-oriented languages. In this section, we follow the classification described by Matsuoka and Yonezawa[MAT 93], and present solutions for their example problems. We have chosen these examples because they all differ in the way they prohibit synchronization code reuse.

4.2 State Partitioning Anomaly

State partitioning anomaly occurs when some of the accepted states have to be partitioned in a subclass. This is not a problem for guard-based synchronization, but accept set-based synchronization suffers because all codes that designate partitioned state need modification.

`bbuf_hilo` in the previous section was such a case; in [MAT 93] example of `get2` — a variant of `get` that obtains two values at once — is shown. Below is the solution in p6i:

```
bbuf2 = interface { empty, one, mid, full }
    extends bbuf { mid } becomes { one, mid }
  get2 = method({mid}b:self{mid,one,empty}) replies(int,int)
end bbuf2

bbuf2i = class implements bbuf2 extends bbuf_basei
    after get{mid}, put{mid} use setstate
  get2 = method({mid}b:self{mid,one,empty}) replies(int,int)
    b.opt := (b.opt+1)//b.size; v1:int := b.arr[b.opt]
    b.opt := (b.opt+1)//b.size; v2:int := b.arr[b.opt]
    b.cnt := b.cnt-2; b!setstate(); reply(v1, v2)
  end get2
  setstate = method(b:self)
    if b.cnt = b.size then b!{full}
    elseif b.cnt = 0 then b!{empty}
    elseif b.cnt = 1 then b!{one}
    else                  b!{mid} end
  end setstate
end bbuf2i
```

As a result, state partitioning anomalies can naturally be handled by abstract state-splitting and after daemons.

4.3 History Sensitive Anomaly

History sensitive anomaly occurs when the acceptance of some methods depends only on historical traces of method invocations. In the case of guards, historical information needs be recorded in the additional instance variable, and all affected guards need be modified to refer to this variable. In the case of accept sets, all codes that designate affected next sets must be modified to reflect historical conditions.

In [MAT 93], example of `gget` — identical to `get` except that it cannot be invoked after `put` — is shown. Below is the solution in p6i:

```
bbufh = interface {empty,mid,midput,full,fullput}
    extends bbuf { mid } becomes { mid, midput }
                 { full } becomes { full, fullput }
  gget = method({mid,full}b:self{mid,empty}) replies(int)
end bbufh
```

```
bbufhi = class implements bbufh extends bbuf_basei
  after get{mid} use {mid}, put{mid} use {midput},
        put{full} use {fullput}
  gget = method({mid,full}b:self{mid,empty}) replies(int)
    reply(b!get())
  end gget
end bbufhi
```

This time, as the modified next state is uniquely determined for each method, we only need to inform the compiler about the assignment — case (2) in section 3.4. No daemon method is necessary.

The above solution is possible because state abstraction names each state uniqely and thus allows description of simple mapping (designation of next state). It might seem possible to add similar mappings to accept (or enabled) sets, but as they are "sets," two abstract states with identical acceptable methods (as in `high` and `low` in the previous section) are indistinguishable and cannot be handled.

4.4 State Modification Anomaly

State modification anomaly occurs when subclass introduces orthogonally restricting functionalities. We prefer to use the term "mixin anomaly" because orthogonal functionalities are often represented as mixins. Enabled sets can handle mixin anomalies by saving/restoring accept set values. Guards do not have this capability.

In [MAT 93], an example of lockable `bbuf` is shown. Our solution uses `lock` interface and its implementation as mixin:

```
lock = interface { free, locked }
  new = method() replies(self{free})
  lock = method({free}l:self{locked})
  free = method({locked}l:self{free})
end lock
locki = class implements lock
  new = method() replies(self{free}) reply(self$[]!{free}) end new
  lock = method({free}l:self{locked}) l!{locked} end lock
  free = method({locked}l:self{free}) l!{free} end free
end locki
```

Note that the implementation has no instance variable and acts as abstract state placeholder. Next, we create mixed interface as follows:

```
bbufl = interface extends bbuf and lock
  new = method(n:int) replies(self{empty.free})
  put = method({empty.free,mid.free}b:self{mid.free,full.free},i:int)
  get = method({full.free,mid.free}b:self{mid.free,empty.free})
        replies(int)
end bbufl
```

In p6i, the interface can have multiple base interface (multiple inheritance), and the abstract state set of resulting interface becomes the production of base interface state sets, whose elements are dot-concatenated sequences of base interface states.

Here, signatures of `put` and `get` were overridden so that these operations are called only when the lock is `free`. However, this change only restricts the allowable states, so previous implementations of `put` and `get` are conforming to this new interface and thus can be reused.

Implementation (class) modules are limited to single inheritance, but we can hold the `lock` object in a state variable and delegate appropriate operations:

```
bbufli = class implements bbufl extends bbuf_base
    slot l:lock
    delegate lock, free to l
  new = method(n:int) replies(self{empty.free})
    b:self := super(n); b.l := lock!new(); reply(b!{empty.free})
  end new
end bbufli
```

Thus, the only method that needs redefinition is `new`; all other methods are reused intact. As an explanation, the above code without the `delegate` clause will be as follows:

```
bbufli = class implements bbufl extends bbuf_base
    slot l:lock
  new = method(n:int) replies(self{empty.free})
    b:self := super(n); b.l := lock!new(); reply(b!{empty.free})
  end new
  lock = method({*.free}b:self{-.locked}) b.l!{-.locked} end lock
  free = method({*.locked}b:self{-.free}) b.l!{-.free} end free
end bbufli
```

Note that `lock`/`free` leaves the {`empty,mid,full`} component intact (because they are orthogonal) in this case; they can be designated to specific values or daemons can be used as explained previously.

Apparently, "state explosion" will occur with only several number of mixins. The problem is alleviated by wildcards ("`*`") as shown above, or by splitting only some designated states (in case of non-orthgonal splitting), but we are looking for better solution.

5 Implementation Issue

To show that efficient implementation of state abstraction-based synchronization can be constructed, we briefly describe design and performance of p6/SPARC[KUN 97], a shared-memory multiprocessor implementation of p6 language. The p6i implementation is currently under development, but its runtime design and performance will be similar to that of p6.

p6/SPARC currently runs on SparcServer/1000 (4×50MHz SuperSparc+). The compiler accepts p6 source and emits SPARC assembly code. The generated code is assembled and linked with the runtime library, including Solaris 2.x thread library. The runtime framework of p6/SPARC is shown in Figure 1. A scheduler thread handles the resumption of the suspended method entry, while several executor threads runs actual program codes in parallel.

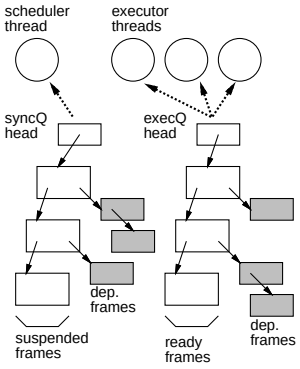

Figure 1: Runtime framework of p6/SPARC

The state abstraction-based synchronization is handled by the generated codes at the top of the method entry. Both the current state and condition are represented as bit masks, so state matching can be tested efficiently with single "and" operation. To prevent interference by other CPUs, each argument is locked and then tested for state match. To prevent deadlocks, locks are acquired in the increasing memory address order. If all matches succeed, locks are released and the method body is executed.

When the match fails and suspension occurs, the object's execution context is saved in the memory area (frame), queued in the SynqQ, and the control is returned to the caller. As the caller must also be synchronized, cascaded suspension occurs. These "dependent" frames are linked to the suspended frame. This scheme is similar to StackThreads proposed by Taura et. al.[TAU 94], with additional mutex controls to guard from interference by other CPUs.

When a state setting code is executed, the code also wakes up the (possibly) sleeping scheduler. When woken up, the scheduler scans SyncQ and moves ready frames (and its dependent frames) to ReadyQ.

Table 1 shows the measured overhead for some primitives. The performance of method calls with abstract state synchronization is very good, while the changing of abstract state is relatively slow, because a condition variable must be set to wake up the (potentially) sleeping scheduler. Note that the overhead is due mainly to Solaris 2.x thread library's condition variable implementation; we are looking for an alternative that does not rely on this slow implementaiton.

Table 1: Overheads of p6/SPARC Primitives

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.

p6i can be implemented using almost identical framework, with additional indirection for VFT (vitrual function table) access. One issue in p6i is representation of abstract state and state mask information, because the same object must behave as several interface types. For example, abstract state of `bbuf` shown in section 3.3 can be expressed in three bits:

```
100:full, 010:mid, 001:empty
```

However, if `bbuf_hl` was introduced, four bits should be used instead:

```
1000:full, 0110:mid, 0100:high, 0010:low 0001:empty
```

With this assignment, the state mask for the superinterface (e.g. {`full,mid`} = `1110`) correctly captures states belonging to subinterfaces (e.g. {`low`} = `0010`). However, the assignment cannot be determined until all interfaces used for the program is collected.

Thus, we are currently planning for linker-assigned bit masks, dynamically switching between efficient, word-sized bit vector (when possible) and more general, slower representation. For mixins, the "slower" version further has two alternatives, namely: (1) expand cartesian product to long (in-memory) bit vectors, or (2) simply combine multiple bit masks and perform additional computation on condition checks. The appropriate choice requires further investigation and experiments.

Another p6i issue is after daemons. We are planning to copy inherited method bodies that require after daemons, and embed the bodies of daemons in place of the state setting code. This is possible because the state names in the state setting code are compile-time constant. Also note that scheduler wakeup is required only once after the final state has determined — thus, the nesting of the subclass has no impact on wakeup overhead noted previously.

6 Comparison to Related Works

The inheritance anomaly problem has attracted many researchers' interests, and numerous "solutions" have been proposed. In this section we compare

these proposals with our approach, grouped by base synchronization scheme they have adopted.

Many parallel object-oriented programming languages are based on explicit message receiption (or bodies), but little attempt has been done to introduce inheritance to those langauges because bodies have to be rewritten when synchronization condition changes. Among these, Caromel[CAR 93] has proposed first-classing relevant facilities (messages, guards, ...), with dynamic and incremental modification of their combination. This scheme is quite flexible, but actual modification of existing bodies may not be so easy. The scheme also might incur too large an overhead for fine-grained parallelism.

Other languages are based on guards, but simple guards cannot deal with history-only sensitivities, as suggested in the previous section. Some researchers have supplemented guards with additional mechanisms[NEU 91][MCH 94]. In our opinion, they are more or less hybrid approach and makes the language more complex. Aside from these, Frølund made guard expression incrementally modifiable[FRO 96], but this does not handle history-only sensitivities. Additionaly, for all of these scheme, the weak points of guards suggested in section 2.3 remain intact.

The third category is based on accept sets[KAU 89][TOM 89][AND 92]. Additionally, Ishikawa[ISH 92] has suggested use of after demon as described in section 3.4. However, the accept set-based scheme still do not handle the state partitioning described in section 4.2 satisfactorily, as Matsuoka and Yonezawa described[MAT 93].

The fourth category is concurrency annotation[LOH 93][BAQ 95]. In this scheme, the sequential code and its concurrency behavior, described as annotation, are made distinct. When the grain of annotation is made small, the amount of code that requires modification upon synchronization condition change becomes smaller. However, modification is still required for all affected methods, making the scheme unsatisfactory.

The fifth approach, proposed by Matsuoka and Yonezawa[MAT 93] is a hybrid one, and can be summarised as follows: (1) For each object, prepare a special method that calculats the next accept set (set of method names that can be executed). (2) This special method can be overridden or extended for a subclass using normal inheritance mechanism. (3) The actual form of this special method is either a synchronizer (maps guard expressions to accept sets), or a transition specification (choose accept sets based on state transition). (4) Operations on accept sets are restricted to compile-time expression, allowing efficient implementation.

This approach has the following strong points: (1) Synchronization handling is separated from ordinary methods, allowing single special method to be used for many ordinary methods (and for different objects through inheritance). (2) Frequently used transition patterns (state push/pop, apply once, etc.) are collected and supplied as library features.

On the other hand, it has the following weak points: (1) To understand the behavior of an object, one must read the special method and ordinary method

in parallel, leading to some awkwardness. (2) Library approach seems more or less ad hoc. (3) When guards are used, they have the same drawbacks described in section 2.3.

The sixth and last category is Meseguer's Maude language[MES 93] and its successor[LEC 96]. In Maude, objects' internal state values and their transitions are described using concurrent rewriting logic. The behavior of subclass objects is defined in the same framework with additions and modifications of terms and rewriting rules to the base class.

In this approach, the programmer directly specifies the objects desirable behavior and need not specify synchronization at all; thus inheritance anomalies are avoided altogether. The strong points of Meseguer's approach are as follows: (1) Concurrent rewriting logic provides accurate execution semantics of the language. (2) No explicit description of parallelism and synchronization actions are required.

On the other hand, it has following weak points: (1) To construct a real programming language, one has to use a proper subset of rewriting logic (actually Mesegeur has done this for Maude langugage). (2) Even on this restricted subset, the execution requires dynamic pattern matching as a primitive operation. Thus, execution speed will not be very good.

Generally speaking, differences between the Mesegeur's approach and others (including our approach) are similar to differences between specification-based languages (Obj, Larch, IOTA, ...) and traditional programming languages. The former is based on logical specification and is more accurate, but tends to be complex or slower. The latter is more or less an abstraction of current machine hardware, thus the gap between its execution model and logical specification tends to be larger, but efficient implementation is possible. We are aimed at the latter direction.

Now, we compare p6i against works other than Meseguer's. In the above categories, explicit message acceptance, accept/enabled set, and concurrency annotation do not provide satisfactory solution to inheritance anomaly problems. Guard-based scheme is unsatisfactory because of problems listed in section 2.3.

Matsuoka's proposal does contain guards, but as it have many intersting aspects, we compare his approach with p6i more throughly. Compared with this, the weak point of p6i is as follows: (1) Synchronization descriptions are closely tied to the method code and cannot be reused separately.

However, the synchronization conditions in the method headers and state setting code in the method bodies can be inherited separately through interface inheritance and implementation inheritance. Moreover, we can gather state setting code as private methods and share them from many places; the same also holds for after daemons.

The strong points of p6i are as follows: (1) The operation on instance variables, or concrete states, and abstract states can be described at a single place (method bodies). (2) A smaller number of language constructs and compact notation leads to more learnable, comprehensive language. (3) State matching

is actually a simple bit-mask operation; it is simple, efficient and side-effect free.

Another difference is that the abstract state information is accessible from other objects. Although this is not a prerequisite for state abstraction-based synchronization, this is important, as the avoidance of substitution anomaly (section 3.3) is crucial to parallel object-oriented languages.

7 Conclusion

In this paper, we have presented state abstraction-based synchronization for parallel object-oriented programming languages and its inheritance-enabled extension. Our scheme solves inheritance anomaly problems described by Matsuoka and Yonezawa[MAT 93] in a straightforward manner. Compared to the previous works, we could contribute the following: (1) clean and comprehensive language design which is (2) efficiently implementable, (3) avoiding typical inheritance anomaly problems, and (4) address the problem of substitution anomaly.

On the other hand, we need more to implement research in the following directions: (1) Investigate the relation of our model with theoretical works such as process calculus[MIL 89] or regular types[NIE 93]. (2) Test our ideas on larger systems and accumulate programming experiences. (3) Transport existing implementation to other (presumably distributed) platforms.

Acknowledgment

We would like to thank Prof. Satoshi Matsuoka of the Tokyo Institute of Technology for suggesting relevant readings. We are also thankful to the OOPSLA'97 and POPL'98 referees for their helpful and insightful suggestions.

References

[AME 90] AMERICA, P.: A Parallel Object-Oriented Language with Inheritance and Subtyping, Proc. OOPSLA/ECOOP'90, pp. 161-168, 1990.

[AND 92] ANDERSEN, B.: Ellie: A General, Fine-Grained, First-Class, Object-Based Language, JOOP, vol. 5, no. 2 and 3, 1992.

[BAQ 95] BAQUERO, C. et al.: Integration of Concurrency Control in a Language with Subtyping and Subclassing, Proceedings of USENIX COOTS'95, 1995.

[CAR 93] CAROMEL, D.: Toward a Method of Object-Oriented Concurrent Programming, CACM, vol. 36, no. 9, pp. 90-102, 1993.

[FRO 96] FRØLUND, S.: Coordinating Distributed Objects, MIT Press, 1996.

[ISH 92] ISHIKAWA, Y.: Communication Mechanism on Autonomous Objects, Proc. OOPSLA'92, pp. 303-313, 1992.

[KAU 89] KAFURA, D. G., LEE, K. H.: Inheritance in Actor based concurrent object-oriented languages, Proc. ECOOP'89, pp. 131-145, 1989.

[KUN 91] KUNO, Y.: 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.

[KUN 96] KUNO, Y., OHKI, A., UBAYASHI, N.: "Symmetric" Message Passing and Its Implementation (in Japanese), IPSJ 96-PRO-8-12, 1996.

[KUN 97] KUNO, Y., OHKI, A.: p6: A State Abstraction-Based Parallel Object-Oriented Language (In Japanese), Trans. IPSJ, vol. 38, no. 3, pp. 563-573, 1997.

[LIS 94] LISKOV, B., WING, J.: A Behavioral Notion of Subtyping, TOPLAS, vol. 16, no. 6, pp. 1811-1841, 1994.

[LOH 93] LÖHR, K-P.: Concurrency Annotations for Reusable Software, CACM, vol. 36, no. 9, pp. 81-89, 1990.

[LEC 96] LECHNER, K. et al.: (Objects + Concurrency) & Reusability — A Proposal to Circumvent the Inheritance Anomaly, Proceedings of ECOOP'96 (Springer LNCS 1098), pp. 232-247, 1996.

[MAT 93] MATSUOKA, S., YONEZAWA, A.: Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in G. Agha, A. Yonezawa, P. Wegner, eds., Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[MCH 94] MCHALE, C.: Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance, Ph. D Thesis, University of Dublin, Trinity College, 1994.
`ftp://ftp.dsg.cs.tcd.ie/pub/doc/gsg-86b.ps.gz`

[MES 93] MESEGUER, J.: Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming, Proc. ECOOP'93 (Springer LNCS-707), pp. 247-267, 1993.

[MEY 93] MEYER, B.: Systematic Concurrent Object-Oriented Programming, CACM, vol. 36, no. 9, pp. 56-80, 1993.

[MIL 89] MILNER, R.: Communication and Concurrency, Prentice Hall, 260p, 1989.

[NEU 91] NEUSIUS, C.: Synchronizing Actions, Proc. ECOOP'91 (Springer LNCS 512), pp. 118-132, 1991.

[NIE 93] NIERSTRASZ, O.: Regular Types for Active Objects, Proc. OOPSLA'93, pp. 1-15, 1993.

[TAU 94] TAURA K. et al.: StackThreads: An Abstract Machine for Scheduling Fine-Grained Threads on Stock CPUs, in Proc. Workshop on Theory and Practice of Parallel Programming (Springer LNCS 907), pp. 121-136, 1994.

[TOM 89] TOMLINSON, C., SINGH, V.: Inheritance and Synchronization with Enabled-Sets, Proc. OOPSLA'89, pp. 103-112, 1989.

Appendix: Complete List of Bounded Buffer Example in p6

```
aint = array[int]                                                    %1
bbuf = class { full, empty, mid }                                    %2
  slot arr:aint, size, cnt, ipt, opt:int                             %3
  new = method(n:int) replies(bbuf{empty})                           %4
    return(bbuf$[arr:aint!new(n),size:n,cnt:0,ipt:0,opt:0]!{empty})  %5
  end nw
  put = method({empty,mid}b:bbuf{mid,full}, val:int)
    b.ipt := (b.ipt+1)//b.size; b.cnt := b.cnt+1; b.arr[b.ipt] := val %6
    if b.cnt = b.size then b!{full} else b!{mid} end
  end put
  get = method({full,mid}b:bbuf{mid,empty}) replies(int)
    b.opt := (b.opt+1)//b.size; b.cnt := b.cnt-1; v:int := b.arr[b.opt]
    if b.c = 0 then b!{empty} else b!{mid} end
    return(v)
  end get
end bbuf
```

1. `array[`T`]` means array of type T. The array size is not part of the type.
2. Start of class definition, followed by the list of abstract states that this class object can have.
3. Declare instance variables.
4. Start of method definition. In p6, there are no distinction between class methods and instance methods. All methods can be invoked through the expression: *typename*`!`*methodname*`(`*arg*1`, `*arg*2`, ...)` However, when the type of the first argument is the class type itself, the following expression can also be used: *arg*1`!`*methodname*`(`*arg*2`, ...)` This can be regarded as an instance method.
5. `bbuf$[...]` allocates the object's implicit record and initializes its fields.

   expression`!{`*statename*`}` set abstract state for an object, whose type must be same as the srrounding class type.
6. `//` is a modulo operator.