Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages

Satoshi Matsuoka Kenjiro Taura Akinori Yonezawa
Department of Information Science, The University of Tokyo
{matsu,tau,yonezawa}@is.s.u-tokyo.ac.jp

Abstract

Re-use of synchronization code in concurrent OO-languages has been considered difficult due to *inheritance anomaly*, which we minimize with our new proposal. Designed with high practicality in mind, we propose language primitives (plus their implementation) with the following characteristics: (1) it allows multiple *synchronization schemes*—the language schemes for *programming* synchronization—to coexist and be integrated, (2) re-use of synchronization code is done similarly to sequential OO-languages for user familiarity, (3) it offers high degree of encapsulation—even synchronization schemes could be encapsulated in superclasses in many cases, and (4) it can be efficiently implemented on conventional MPPs. We demonstrate the effectiveness of our proposal with solutions to the example inheritance anomaly cases from [16]. We also give an overview of the implementation architecture, along with preliminary benchmarks. The proposed language primitives are being incorporated into our ABCL/onAP1000 running on Fujitsu's 512-node MPP, AP1000.

1 Introduction

High-performance parallel computing on Massively Parallel Processors (MPPs) is one of the most important topics in computer science today. Although most MPP programming still uses traditional languages such as FORTRAN, several research projects are focused on implementing Concurrent Object-Oriented (OO) languages on MPPs. Concurrent-OO languages provide high computational and modeling power through (1) concurrency of objects, plus (2) familiar OO-software engineering concepts such as *encapsulation*, *code re-use*, and *application frameworks*[32], and can serve as a powerful platform for parallel applications on MPPs[1]. Much work on establishing useful concurrent-OO language models have been quite fruitful: they include Actors[1], POOL[3, 4], Hybrid[20, 21], ABCL[34], and Maude[17].

Despite such background, plus promising results in impressive pioneering research systems such as POOL/DOOM[6], we do acknowledge that concurrent-OO languages have not yet seen widespread use in practice. We believe that the two major impediments are as follows:

'Myth' that 'Real'(Parallel) Message Passing is Slow:
There is still a belief amongst the OO-community that message passing in a literal sense—parallel over processor interconnections—is a few orders of magnitudes slower compared to sequential procedure invocation. Since most programmers use sequential-OO languages that implement methods with plain procedures, the term 'message passing' has become synonymous to 'procedure-calling' in OO-programming. In parallel programming, performance is the key issue, and such unwarranted 'myth' has had negative effects.

Inheritance Anomaly—Difficulty in Re-using Synchronization Code:
Several works have pointed out the conflicts between inheritance and concurrency in OO-languages[3, 12, 23, 30, 8], where attempts to inherit and re-use the code of concurrent objects results in extensive breakage of encapsulation. In [16], we have coined such a phenomenon as *inheritance anomaly* (See Section 2). One resulting critical drawback is that it becomes very difficult to construct a clean application framework—the most effective OO-software engineering discipline[32]—

[1] To quote Robin Milner at ECOOP'91 Object-Based Concurrency Workshop panel, "I can't understand why objects are not concurrent in the first place (in OO-languages such as Smalltalk)."

with concurrent-OO languages.

Very recently, several research groups have so far been successful in demonstrating that the slow-performance 'myth' can be overcome. In particular, our work on ABCL/onEM-4[33] and ABCL/onAP1000[28] have achieves near sequential-OO message passing performance, as we briefly overview in Section 5. The purpose of this paper is to present our latest results in solving the latter problem of providing efficient concurrent-OO language design that minimizes inheritance anomaly, promote code re-use, and aid the construction of parallel application frameworks. Although some number of proposals have been made in this regard, early attempts([12, 30]) were shown to be restrictive in [15]. The problem has sparked interest of several concurrent-OO researchers who have come out with various 'solutions' [26, 10, 13, 11, 18], especially after circulations of analysis papers such as [23, 15] and the early draft of [16]. However, we feel that they were not completely satisfactory with regards to practical applications.

Our current proposal is designed with high practicality in mind. It extends and extensively refines the ideas in the past proposals to (1) separate and localize the synchronization schemes from the main bodies of methods, allowing fine-grained inheritance/overriding, and to (2) allow dynamic operations on the methods themselves, in order to control which messages are acceptable by an object. Furthermore, it has the following novel and favorable characteristics:

- Our proposal allows multiple *synchronization schemes*—the basic language features such as guards for *programming* the synchronization of objects—to coexist and be integrated, so that the best scheme can be chosen to program given synchronization constraints.
- The manner we re-use the synchronization code is syntactically similar to superclass method references in sequential OO-languages (e.g.,`super`). Thus, users with experience in OO-programming can readily adapt to our proposal. Inheritance rules are made to depend on each synchronization scheme, however, because the most 'natural' way of inheritance differs among the schemes.
- We offer a high degree of encapsulation and re-use for synchronization code. Furthermore, even synchronization schemes could be encapsulated in superclasses in many cases by proper exporting of class information by the user.
- Expressiveness is not our sole concern—we have also devised speed- and space-efficient incorporation into the software architecture of the aforementioned ABCL/onAP1000. In particular, all space/time-consuming data structure construction for object synchronization can be done at compile-time.

Throughout the paper, we demonstrate the effectiveness of our proposal by resolving the example inheritance anomaly cases taken from [16]. We also give an overview of the implementation architecture, and some preliminary benchmarks that support our claim of efficiency. The proposed language primitives are being incorporated into the MPP version of our concurrent-OO language ABCL, currently being implemented on AP1000 and CM-5. We are also porting practical applications we have written into the new language (such as N-body simulation and Genome RNA secondary structure prediction).

2 Background: What is Inheritance Anomaly?

We first briefly outline the inheritance anomaly problem. For a more thorough analysis, readers are referred to [16]. In order to control the synchronization between objects, the set of messages a concurrent object can receive is made to depend on its state. We call such a restriction on acceptable messages the *synchronization constraint* of a concurrent object. For example, consider a bounded buffer with methods `put()` and `get()`, where `put()` stores an item in the buffer and `get()` removes the oldest one; then, the synchronization constraint is that one should not `get()` from a buffer whose state is *empty*, etc. The satisfaction of constraints is not achieved automatically; the user must somehow program the methods to implement the object behavior that satisfy the synchronization constraint. *Synchronization code* is the portion of the method code where such synchronization is controlled. Synchronization code of an object must always be *consistent with* its synchronization constraint; otherwise, semantical error could result.

In order to program synchronization code, a concurrent-OO language provides some primitives and/or general schemes for object-wise synchronization, such as guarded methods. We refer to

the scheme for achieving object-wise synchronization using those primitives in the language as the *synchronization scheme* of the language. Unfortunately, it has been pointed out that *synchronization code cannot be effectively inherited without non-trivial class re-definitions*. This conflict, which we have coined as *inheritance anomaly*[15], has been recently analyzed and categorized[16]. Inheritance anomaly breaks the encapsulation of classes more severely compared to sequential OO-languages, because it is possible to create a general example where NONE of the parent methods can be inherited. One notable fact is that the occurrence of inheritance anomaly *depends on* the synchronization scheme of the language; in other words, there are cases where re-definitions are required for one synchronization scheme while unnecessary in another.

There were several early attempts to provide robust synchronization code re-use in concurrent-OO languages (e.g., [2, 9, 12, 30, 19]). They can be largely classified into those based on *guarded methods*, and those based on *accept set specification*: The former assigns a boolean guard predicate to each method, and only messages that satisfy the guard can be accepted. The latter has an object specify the next set of messages to be accepted within its method code. In [16], we have analyzed three major categories of inheritance anomaly in those proposals, which are: (1) *Partitioning of Acceptable States (State Partitioning Anomaly)*, (2) *History-only Sensitiveness of Acceptable States*, and (3) *Modification of Acceptable States (State Modification Anomaly)*. To aid reader intuition, we describe each one with a canonical example of re-using the code of a bounded-buffer class.

2.1 Partitioning of Acceptable States (State Partitioning)

State partitioning anomaly occurs in the accept-set based schemes. Figure 1 is a definition of class `b-buf` (bounded buffer) using a synchronization scheme called *behavior abstraction*[12], a variant of the accept-set based scheme: the user specifies, within its method code, the next set of methods that can be accepted by the object with the `become` statement followed by the symbolic name of the set. Addition of new methods in a subclass is handled by re-defining the set to contain the name of the new method appropriately.

Unfortunately, naive accept-set based schemes cannot cope with 'partitioning' of acceptable

```
Class b-buf: ACTOR {// b-buf is an Actor
   int in, out, buf[SIZE];
behavior:
   empty = {put}; partial = {put, get};
   full = {get};
public:
   void b-buf() {
      in = out = 0; become empty;
   }
   void put(int item) {
      in++; //store an item
      if (in == out + size) become full
      else become partial;
   }
   int get() {
      out++; //remove an item
      if (in == out) become empty
      else become partial;
}}
```

Figure 1: `B-buf` and `x-buf` with Behavior Abstractions

state[15]: consider creating a class `x-buf2`, a subclass of `b-buf` (Figure 2). `X-buf2` has an additional method `get2()`, which removes the two oldest items from the buffer simultaneously. The corresponding synchronization constraint for `get2()` requires at least two items exist—thus, the `partial` state must be partitioned into (1) `x-one` (only one item exists), and (2) `x-partial` (the remaining states). Then, all the non-initializer methods in `b-buf` (i.e., `get` and `put`) *must be re-defined*. This is due to the following reason: the set of possible 'states' an object can be partitioned into disjoint subsets according to the synchronization constraint of the object. When a new method is added in the subclass, this partitioning may require further partitioning to account for the synchronization constraint for the new method (e.g., `get2()` above, illustrated with Figure 3(A)). When this state partitioning is determined within the method code using explicit conditional statements as in the example, method re-definitions are *required*, because the new partitioning must be accounted for within all the methods. Note that, this is not entirely resolved by making accept sets first-class values as is with *enable sets*[30], because this partitioning cannot be affected by the operations upon the accept-set data.

```
Class x-buf2: b-buf { // x-buf2 is a subclass of b-buf
behavior:
   x_empty   = rename empty;
   x_one     = {put,get};
   x_partial = {put,get,get2()} redef partial;
   x_full    = {get,get2()}     redef full;
public:
   void x-buf2() { in = out = 0;
      become x-empty;
   }
   intpair get2() { out += 2; //definition of get2
      if (in == out)        become x_empty;
      else if (in == out+1) become x_one;
      else                  become x_partial;
   }
   //The following re-defines the methods in b-buf.
   void put(int item) {
      in++; //store an item
      if (in == out)        become x_empty;
      else if (in == out+1) become x_one;
      else                  become x_partial;
   }
   int get() { // requires a similar re-definition.
      :
}
```

Figure 2: State Partitioning Anomaly with Accept-Sets

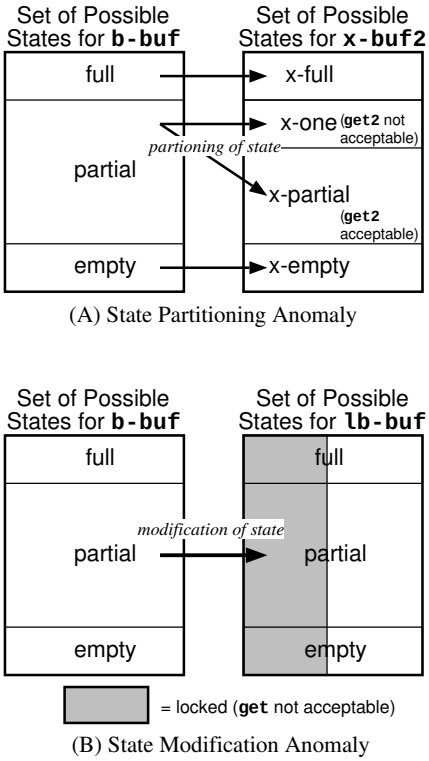

Figure 3: Conceptual Illustrations of Inheritance Anomalies

2.2 History-only Sensitiveness of Acceptable States

State partitioning anomaly can be avoided with guarded methods, because they are able to directly judge whether the message is acceptable or not under a given state. Thus, even if the new methods were added, the guards would not need be re-defined: for example, the guard `(in >= out + 2)` would precisely satisfy the synchronization constraint for `get2()`. However, other types of anomalies occur, as we illustrate in Figure 4: Consider a subclass of `b-buf`, `gb-buf` that adds a single method, `gget()`. The behavior of `gget()` is almost identical to that of `get()`, with the exception is that it cannot be accepted immediately after the invocation of `put()`. As a consequence, both `get()` and `put()` must be re-defined as in Figure 4. The reason for the anomaly occurrence is that we cannot judge the state for accepting the `gget()` message with the guard declarations in `b-buf`, requiring addition of a flag variable `after-put` and the associated re-definition of guards. To be more specific, `gget()` is a *history-only sensitive* method. Since the proper valuation of this variable must be done in all the methods, the requirement of re-definitions of all parent methods arose.

2.3 Modification of Acceptable States

We next consider the `Lock` class, an abstract *mix-in* class whose purpose is to be 'mixed-into' other classes in order to add the capability of locking an object. Upon accepting a message `lock()`, the object suspends the acceptance of further messages until it accepts `unlock()`. When `Lock` is mixed-into `b-buf` to create `lb-buf`, it should not affect the method codes of `b-buf` since the state of the object with respect to `lock()` and `unlock()` is totally orthogonal to the effect of other messages. However, this is not the case — we must add an instance variable `locked` which indicates whether the object is currently 'locked' or 'unlocked'. Then, the inherited methods such as `put` or `get()` must be overridden in order to account for `locked` (Figure 5). What anomaly has occurred here? The execution of the methods in `Lock` modifies the set of states under which the methods inherited from

```
Class b-buf: ACTOR {
  int in, out, buf[SIZE];
public:
  void b-buf() { in = out = 0; }
  void put() when (in < out+size) { in++; }
  int get() when (in >= out+1)   { out++; }
}
// gb-buf is a subclass of b-buf with gget()
Class gb-buf: b-buf {
   bool after-put;
public:
   void gb-buf() { after-put = False};
   // Definition of gget()
   int gget() when (!after-put && (in >= out+1))
     { out++; after-put = False; }
   // The following must be re-defined
   void put(int item) when (in < out+size)
     { in++; after-put = True; }
   int get() when (in >= out+1)
     { out++; after-put = False; }
}
```

Figure 4: History-only Sensitive Anomaly with Guards

```
Class Lock: ACTOR {
   bool locked;
public:
   void Lock() {locked = 0};
   void lock() when (!locked) {lock = 1};
   void unlock() when (locked) {lock = 0};
}
// lb-buf is a subclass of b-buf
//   with Lock mix-in
Class lb-buf: b-buf,Lock {
public:
   void lb-buf();
   // The following methods must be re-defined
   void put(int item)
         when (!locked && (in < out+size))
     { in++; }
   int get() when (!locked && (in >= out+1))
     { out++; }
}
```

Figure 5: State Modification Anomaly with Guards

the parent could be invoked (Figure 3(B)). Thus, the mixing-in of `lock` to `lb-buf` introduces finer-grained distinction for the set of states under which `get()` (or `put()`) can be invoked. This would require the modification of the method guards to account for the new synchronization constraint, resulting in *state modification anomaly.*

3 Our Proposed Solution to Inheritance Anomaly

Recently, several researchers have proposed to minimize the effect of inheritance anomaly and promote code re-use in concurrent-OO languages [26, 10, 13, 24, 11, 18]. Altogether, they have identified to some degree that (1) localization of the changes in the synchronization code, and (2) first-classing of synchronization schemes (possibly with reflection) are significant in providing the necessary flexibility to minimize code re-definitions. However, there are still problems:

(1) Limited Expressiveness/Encapsulation:
We've seen that no single synchronization scheme would be a panacea for inheriting synchronization code, i.e., accept-set and guard each has its advantages and drawbacks. Instead, our premise is that the user should be able to *choose* the synchronization scheme deemed appropriate for a given synchronization constraint. This can be achieved by maintaining the orthogonality/encapsulation property not only for the synchronization code, but also for the *synchronization schemes* employed in the superclasses, i.e., the employed schemes are implementation details that *should not be exposed to the subclasses.* The previous proposals restrict the user to a single synchronization scheme, thus limiting expressiveness, and/or have not addressed the encapsulation issue at all. As a result, re-use that could lead to inheritance anomaly, which could otherwise be avoided using different synchronization schemes, become difficult.

(2) Little or No Implementation/Performance Analysis:
For all proposals, efficiency of the implementation of their schemes have been given little consideration: in particular, none consider whether their implementation could be done with a comparable efficiency to sequential OO-languages, say, to Smalltalk, SELF, or C++. In fact, the overhead for some of them could easily be hundreds of instructions per each method invocation, which would prohibit their usage in practical situations.

(3) Awkward Inheritance of Synchronization Code:
Some proposals allow inheritance of synchronization code separately from main method bodies, but do so in a syntactically and conceptually different way compared to standard method inheritance. Thus, users are faced with two different

inheritance systems, possibly with no obvious conceptual model of how they interact.

Our solution derives from and retains the favorable characteristics of the previous proposals, but takes further steps to resolve the above problems for practical use. More specifically, it (1) allows the programmer to achieve encapsulation of the synchronization code and scheme in the parent classes—the synchronization scheme employed in a class definition can be encapsulated, allowing the subclasses to employ *entirely different synchronization schemes from their superclasses* in many cases, if the superclass exports proper information to its subclasses. At the same time (2) the overhead of synchronization at run-time is minimized by smooth integration of the underlying implementation into the efficient runtime architecture of ABCL/onAP1000[28]. Furthermore, (3) inheritance of synchronization code is in principle specified in a similar manner to standard method inheritance for user familiarity, but with customized inheritance rules appropriate for each synchronization scheme.

3.1 Overview of Execution Model of Concurrent Objects

The execution model of our concurrent object is an extension of ABCM[34], the computational model for the language ABCL: an object sends messages asynchronously, either *past type* ("asynchronously send and no-wait", syntactically denoted by *Receiver* `<-` *Msg*) or *now type* ("asynchronously send and wait for a reply", syntactically denoted by *Receiver* `<--` *Msg*). The transmission order of messages between two objects are preserved (*transmission order preservation law*). The now-type is similar to procedure call, except that (1) the receiver can continue its execution even after it has returned a reply message and (2) the *reply destination* (reply message box) of a now-type message is a first-class object, which can be passed around as message arguments.

Upon message reception, an object executes messages in a mutually exclusive manner. An object is *dormant* if it is not processing a message, and *active* if it is. All the messages received during active mode are placed in its message queue. When in dormant mode, the object scans its message queue from its head, and accepts the first message that satisfies its synchronization constraint.

3.2 How do we Inherit Synchronization Code without Unnecessary Redefinitions?

In order to eliminate or 'minimize' inheritance anomaly, we propose the combined use of *method sets*, *synchronizers*(extended form of guarded methods), and *transition specifications* (state transition directives of *accept-sets*), that features (1) appropriate separation from the actual method bodies, (2) specialized inheritance rules, and (3) orthogonality between the synchronization schemes. Below, we outline the general principles that guided the design of our language features.

One of the primary principles of object-orientation is the encapsulation of *methods* by abstracting their operations and hiding their underlying implementation. The only interface open to the outside world is the set $M(C)$ of methods (or, method names) that an object can accept for class C. Concurrent objects further impose restriction to this set depending on its encapsulated 'abstract state', in order to maintain consistency with its synchronization constraint. All the abstract states must be identifiable within the synchronization code of classes, with which users control the accept set depending on each state. Such code can be characterized by a mapping from the domain $S(C)$ of abstract state for the class to the powerset of methods $2^{M(C)}$ (i.e., the set of accept sets).

We note that the mapping need not have explicit representation as a single function within user method code; rather, it could be 'spread out' in code definitions. For example, with method guards the mapping can be constructed trivially as follows: For all $s \in S(C)$,

$$< s, \{m \mid m \in M(C), G(m)(s) = true\} >$$

where $G(m)$ is the guard function associated with method m in class C.

The internal state of an object can usually be grouped into equivalence classes depending on possible accept sets. Conversely, it is usually possible to discern the internal state of the object by its current accept set[2]. In other words, we can define a modulo $S(C)/2^{M(C)}$ in a standard way, and let each element of $2^{M(C)}$ represent the corresponding partitioned state. Based on this observation, we provide explicit *method sets* within our language

[2]One could construct cases where this does not hold. However, such cases would not be in good accordance with the concept of object encapsulation.

as user-defined accept sets, in order to allow the programmer to encapsulate the abstract state of a concurrent object with respect to its synchronization constraints.

When an object executes a method, it may cause a change in its internal state; the state change could manifest as change in the values of its instance variables, or could be some 'historical state' that does not. Whatever the case, the transitional behavior between the abstract states is realized by the underlying implementation of the synchronization code. By properly defining a class so that method sets precisely describe the abstract states, general subclassing/code re-use can be done in the following way: (1) synchronization code is separated from the main body of methods, so that they can be inherited and re-defined separately, avoiding unnecessary re-definitions of method bodies; (2) if no new states/transitional behavior need to be identified in the subclasses, then re-definitions can be confined to static set-operations on the method sets; (3) if the re-definitions affect the abstract states in a non-trivial way as exemplified in Section 2 (such as state partitioning), then the user augments, via inheritance, the method sets and the transitional behavior dictated by the synchronization code.

Even for case (3), if the abstract state is *internally identifiable*, that is, one can always identify the abstract state of an object solely from the internal values of its instance variables, then the programmer is free to use whatever synchronization scheme that he deems appropriate in his synchronization code of the subclasses, provided that the different schemes can be combined orthogonally, and a new mapping can be constructed transparently for the methods in the superclasses. In other words, encapsulation not only of the synchronization code, but also *of the synchronization scheme* becomes possible in such a case.

In our proposal, the execution model of concurrent objects is extended with the following synchronization scheme: First, the object evaluates its set of *synchronizers* to determine, within its current accept set, which messages are acceptable according to its current state. The object scans the message queue for an acceptable message, and executes the corresponding method. After its execution is completed, the object evaluates the *transition specification* associated with the method, and alters its accept set with the method sets specified in the transition. The programmer can choose and combine the synchronization schemes in the synchronization code to best express his synchronization constraints (i.e., with various forms of synchronizers and transition *types*). Both synchronization schemes are defined separately from the main body of methods, and employ (user-defined) method set in operations upon the accept set. The separated definition is referred to as the *synchronization specification* of a given class.

In defining subclasses, The constituent primitives of synchronization specification—method sets, synchronizers, and transition specifications—can be individually inherited and re-used in a 'fine-grain' manner, using a similar syntax to standard method inheritance but with separate inheritance rules for each primitive. Furthermore, since the different synchronization schemes—synchronizers and transition specifications—are designed to be orthogonal, we achieve encapsulation of synchronization schemes when the object state is internally identifiable.

3.3 Method Sets

A *method set* is a set of methods (identifiers) bound with the corresponding method bodies. It is worthy to note in the outset that method sets are designed NOT to be full first class objects, in that they cannot be assigned to variables and such, and are only subject to restricted run-time extensions. This allows the *Virtual Function Table (VFT)* (the table that holds pointers to the compiled methods code according to their method identifiers) to be determined at compile time for efficient execution and low storage space; the details are discussed in Section 5.

Primitive method set constructor has the form `#{`*method name*`,...}`. There are also some primitive set operations, such as '`|`'(union). Other than within the method definitions, the method sets in the program must be assigned to explicit identifiers to be referenced, disallowing arbitrary run-time set operations. The basic definition form is:

`mset` *name* `#{`*method name*`, ...};`

Example 1 Consider the `b-buf` class with the two methods, `put` and `get`:

```
class b-buf {
method_sets:
   // only 'put' to empty buffer
   mset EMPTY   #{put};
   // only 'get' from full buffer
   mset FULL    #{get};
   // both possible otherwise
   mset PARTIAL EMPTY | FULL;
```

```
        :
}
```

A *method qualifier* is a limited form of qualifier expression to denote a constructor for a set of methods. The restriction on the qualifier is that the resulting set must be computable at compile time. Currently, the supported form is as follows:

- `all`—all the methods of the class including the inherited methods,
- `defined`—the defined methods of the class, excluding the inherited ones, and
- `all_except(`*method set*`,...)`—all the methods of the class except the ones of the specified method sets.

Example 2 The following constructs a pair of mutually exclusive method set: `LOCKED` is a singleton set containing `unlock`, while `UNLOCKED` contains all the methods defined at the class, except `unlock`.

```
class Lock {
method_sets:
  mset LOCKED    #{unlock};
  mset UNLOCKED  all-except(LOCKED);
       :
}
```

Method set definitions can be re-defined in subclasses. If the overridden definitions of the superclass needs to be referenced, the subclasses may refer to them using the `super` operator followed by the method set identifier[3] (see Example 3). All the method sets are recomputed in the subclass to account for re-definitions, so that any changes are propagated to the derived method sets.

One special case is as follows: when the name of a method is not syntactically manifest in any of the method set definitions, then the method is called a *synchronization free* method. Such a method is added implicitly and uniformly to all the method sets of the class. This is to allow non-constrained methods to be freely invokable by default. When the method is later explicitly used in a construction of a method set in a subclass, this implicit addition is nullified for that subclass and its siblings.

In addition, the method qualifiers are also (re)computed when one defines a new method in the subclass: for example, if a new method `foo` is added in a subclass of `Lock`, it automatically becomes a member of the `UNLOCKED` method set for the class, because `UNLOCKED` is defined as `all-except(LOCKED)`.

Example 3 Consider defining a class `x-buf` as a subclass of `b-buf` of Example 1 by adding the following two methods: (1) `last`, which removes the last element that was `put`, and (2) `empty?` which checks whether the buffer is empty or not. Since the synchronization constraint for `last` is identical to that of `get`, it is added to the method set where `get` was a member, namely, `FULL`. Method `empty?` is a synchronization-free method, and is automatically added to all the method sets. Furthermore, the method sets that were derived from `FULL`, i.e., `PARTIAL`, are also automatically updated in `x-buf` as well.

```
method_sets:
  mset FULL  super FULL | #{last};
    // PARTIAL  is automatically redefined.
```

Altogether, method set `PARTIAL` in `x-buf` is `#{put, get, last, empty?}`.

A method set can be statically bound to a guard expression (see below), so that a method set in the superclass can be partitioned into multiple method sets, depending on runtime object states. Such *guarded method set* is introduced to cope with the state partitioning anomaly, allowing transitions to have (full) flexibility of guards. Below is the syntax:

```
mset name #{method name, ...}
  when guard expression;
```

Note, however, that we do not allow arbitrary runtime set operations on the method sets; the purpose of the guarded method set definitions is to maintain the encapsulation property without method sets being full first-class entities for efficiency reasons.

Example 4 The following definition of `PARTIAL` allows partitioning of the state in the subclass:

```
method_sets:
  mset PARTIAL  super PARTIAL
    when (size == 1);
  mset PARTIAL  super PARTIAL | #{get2}
    when (size > 1);
```

3.4 Synchronizers

A *synchronizer* is a combination of *guard expressions*, *enabling specifier*, and a list of method sets. In essence, it is similar to a guarded method, but is more flexible in that a single guard can be assigned to multiple methods in method sets.

[3] In practice, matters are more complicated due to multiple inheritance.

A guard expression is basically a side-effect free boolean expression involving (1) instance variables and (2) method arguments (in particular, Local variables of individual methods are not allowed in the guards to maintain encapsulation). The named method argument must exist in all the methods associated with the guard (both for synchronizers and transitions) or compile-time error will result. A guard expression can also contain *acceptance inquiry* function `enabled()` and `disabled()`, which takes either a method name or a method set as an argument. The guard expressions can be assigned symbolic names with `guard` definitions, so that they can be re-used and possibly redefined in subclasses.

`guard` *name guard expression*`;`

The synchronizer thus becomes:

```
synchronizers:
  guard enables method set name, ...;
```

A special keyword `initially` can be specified in place of a guard in order to indicate which method set is enabled initially upon object creation. Also, a synchronizer can be chosen NOT to be inherited for methods in a given method set with `override` *method set*.

Example 5 Synchronization specification for `b-buf` can be made with synchronizers in the following way:

```
guards:
  guard empty_g   (size == 0);
  guard full_g    (size == MAX_SIZE);
  guard partial_g (!e_guard && !full_g);
synchronizers:
  partial_g  enables PARTIAL;
  empty_g    enables EMPTY;
  full_g     enables FULL;
```

3.5 Transition Specifications

Transition specifications can be used as an alternative synchronization scheme to synchronizers. A *transition* corresponding to a method is executed immediately after the completion of the method body. Its purpose is to specify the transitional behavior of an object's accept set, that reflects the synchronization constraint dictated by the internal state of the object.

The transitions are specified on a method-by-method basis, via *transition specifications* for the method. A method name or a method set is given immediately after the keyword `transition` to specify the transition specification for the method (or, correspondingly, the member methods of the method set). The transition specification is composed of multiple lines of *transitions*. Each transition is associated with a *transition type*, followed by a method set and an optional guard; combined, they designate the effect of the method set upon the current accept set, such as replacing the current accept set with the designated method set, etc. (described below). The optional guard governs the condition under which the transition is executed. Altogether, the syntax of transition specification for a given method is as follows:

```
transitions:
  transition {method-name-or-set1}() {
    transition-type-1  method-set
        {when guard-expression 1};
                :
    transition-type-n  method-set
        {when guard-expression n};
  }
  transition {method-name-or-set2}() {
                :
```

Each guard expression of the transition is evaluated sequentially from transitions 1 through n, and the transition of the first guard to evaluate to `true` is executed exclusively. This not only automates the disambiguation of multiple possible transitions, but also allows for finer control of inherited transitions when we consider inheritance.

The currently available *transition types* are as follows: `become`, `push`, `enable`, `disable`, `restore`, `wait_once`, `enable_once`, `disable_once`, and `is`. Below is their brief description:

1. `Become:` Replaces the accept set entirely with the specified method set.

2. `Push:` Not only replaces the current accept set with the specified method set, but also 'pushes' it so that it can be restored with a subsequent `restore`.

3. `Enable:` Enables the methods in the method set in addition to the ones in the current accept set. (Effectively, the new accept set is the set union of the old accept set and the specified method set).

4. `Disable:` Being complement to `enable`, disables the methods that are elements of the argument method set and the current accept set (effectively, the set difference is taken).

5. `Restore:` Restores the method set to the one prior to performing `push`, `enable`, or `disable`.

6. `Wait_once:` The current accept set is 'pushed' and replaced as is with `push`, but is subsequently restored with a implicit `restore` transition in the next accepted message, just prior to execution of the real transition. This effectively allows handshaking-type protocol to be programmed easily, as is with the `wait-for` construct of ABCL/1.

7. `Enable_once`, `disable_once:` A combination of `enable` or `disable` with automatic `restore`, as is with `wait_once`.

8. `Is:` A special-purpose keyword to inherit the transitions of the parent class.

The transition specifications of the parent class are inherited, but overriding them can be done in a more sophisticated manner compared to overriding of method sets and synchronizers:

- Each class can have its *default transition specification*, indicated by a special keyword `default` instead of a method name. If the method does not have any transition specification in its class or its superclasses, the default transition specification is used *if one is defined*. The default transition specification can also be overridden along the inheritance chain.

- The transition `is self` *method-name* refers to the entire transition specification of that method, allowing sharing among the methods. The expression `is super` *method-name* is the similar, except that the search for the corresponding reference starts from the immediate superclass as is with Smalltalk-80 (with disambiguation rules for multiple inheritance).

- There is an automatic inheritance rule to relieve the programmer from explicit declaration of inheritance of transition specifications: when one defines a transition for a method, if there are no lines with the `is` specification, an implicit `super` *method-name* is assumed to exist as the *last* transition in the specification, effectively inheriting the entire transition specification of the superclass for the method. (In order to prohibit this automatic inheritance, the last line of the transition specification can made into a special form `override`.)

- The guard expression can be substituted for `otherwise` to be the *otherwise transition*. This transition is selected when there are no guards that become true. When there are multiple otherwise transitions, the one in the most specific subclass supersedes the others in the superclasses.

Note that, combined with the ordered evaluation rule of the transitions, the above inheritance mechanism of transition specifications allows the programmer the freedom to define transitions that either precede or succeed the superclass transitions with arbitrary placement of the `super`. Default transitions can be referenced by `self` and `super` as well, so that individual methods can customized upon the default transitions of its superclass.

To clarify the combined effect of the above inheritance rules, suppose that we have classes $C_1 \ldots C_m$, C_i inheriting from C_{i-1}, all C_is defining transition specification for method `M` as follows:

```
class C_i : C_{i-1} {
            :
transition M(args) {
  tr-type_i^1  mset_i^1  when guard_i^1;
            :
  tr-type_i^{N_i}  mset_i^{N_i}  when guard_i^{N_i};
  tr-type^{o_i}  mset^{o_i}  otherwise;
            :
}}
```

then, the resulting transition specification for class C_m would effectively be as follows, after inheritance has been 'folded out':

```
transition M(args) {
  tr-type_m^1  mset_m^1  when guard_m^1;
            :
  tr-type_m^{N_m}  mset_m^{N_m}  when guard_m^{N_m};
  tr-type_{m-1}^1  mset_{m-1}^1  when guard_{m-1}^1;
            :
  tr-type_{m-1}^{N_{m-1}}  mset_{m-1}^{N_{m-1}}  when guard_{m-1}^{N_{m-1}};
            :
  tr-type_1^1  mset_1^1  when guard_1^1;
            :
  tr-type_1^{N_1}  mset_1^{N_1}  when guard_1^{N_1};
  tr-type^{o_m}  mset^{o_m}  otherwise;
    // otherwisetransition of C_m takes precedence.
}
```

Example 6 Here is an alternative definition of `b-buf` using transitions.

```
transitions:
transition get() {
```

```
    become EMPTY when (size == 0);
    become FULL  when (size == BUFSIZE);
    become PARTIAL otherwise;
  transition put() {
    // transition of put the same as get
    is self get();
  }
```

More comprehensive examples of transition specifications are presented in Section 4.

3.6 Inheriting Synchronization Code

When the programmer creates a new subclass, he (re)defines the new methods, and also (re)defines the method sets, synchronizers, and transitions to satisfy the new synchronization constraint, re-using much of the synchronization code of the superclasses. As we have seen, inheritance rules are customized for each synchronization scheme according to their characteristics. The required updating of synchronization code is encapsulated within the synchronization specification of the class, i.e., the method sets, synchronizers and transition specifications. The main bodies of the methods in the superclasses, by contrast, are unaffected in the subclasses, avoiding the inheritance anomaly and achieving encapsulation. Synchronizers and transition specifications operate in an orthogonal way, achieving encapsulation of synchronization schemes when object state is internally identifiable.

In general, inheritance anomaly is avoided as follows: state partitioning anomaly does not occur when using synchronizers. For transition specifications, there are two ways of expressing the state partitioning in the subclasses: first is to augment the sets of transitions with the additional partitioning required. By appropriate placement of `super`, only the states that are partitioned for the new methods in the subclass need to be modified, and majority of the parent transitions are re-used automatically by the implicit inheritance rule of transition specifications. Second is to employ method sets bound with (dynamically computed) guards. By describing the new partition of the method sets with the guards, the transitions in the superclasses can be refined with re-definitions only for the relevant method sets. For state modifications such as `lock`, synchronizers can be re-defined, or alternatively, transitions can also be used effectively to 'switch' the method sets according to the state. We do not employ expensive higher-order term structure encodings and pattern matchings as in Maude[18], but provide comparable descriptive power.

4 Examples of Avoiding Inheritance Anomaly

The inheritance anomaly examples in Section 2 are now programmed using our proposal. We show that (1) the synchronization code is encapsulated in the synchronization specifications and does not manifest in the main body of the methods, and (2) separate inheritance (rules) for method sets, synchronizers, and transitions allow fine-grain reuse of superclass synchronization code, keeping redefinitions very small. Furthermore (3) synchronization schemes are also encapsulated; separate solutions for the same problem are programmed using either synchronizers or transitions (except `gget`). We also emphasize that synchronization scheme of the superclasses are also encapsulated, i.e., the solutions would work irrespective of the choice of the synchronization scheme in the original `b-buf` first presented below:

Bounded Buffer with Synchronizers:

```
Class b-buf {
  int size = in = out = 0;
  int item[MAX_SIZE];
method_sets:
  // only 'put' to empty buffer
  mset EMPTY    #{put};
  // only 'get' from full buffer
  mset FULL     #{get};
  // both possible otherwise
  mset PARTIAL  EMPTY | FULL;
synchronizers:
  (0 < size && size < MAX_SIZE) enables PARTIAL;
  (size == 0) enables EMPTY;
  (size == MAX_SIZE) enables FULL;
methods:
  void put(int item) {
    size--; out = (out+1) % max_size;
    return item[out]; }
  int get() {
    size++; in = (in + 1) % max_size;
    item[in] = x; }
}
```

Bounded Buffer with Transition Specifications:

```
Class b-buf {
  int size = in = out = 0;
  int item[MAX_SIZE];
method_sets:
  mset EMPTY    #{put};
  mset FULL     #{get};
  mset PARTIAL  EMPTY | FULL;
methods:
```

```
  void put(int item) {
    size--; out = (out + 1) % max_size;
    return item[out]; }
  int get() { size++; in = (in + 1) % max_size;
    item[in] = x; }
transitions:
  // the default transition specification
  transition default {
    become EMPTY when (size == 0);
    become FULL  when (size == BUFSIZE);
    become PARTIAL otherwise;
  }
}
```

4.1 State Partitioning Anomaly—Method get2:

The method obtains two elements atomically from the buffer. State partitioning in the subclass is trivially satisfied with guards. With accept sets, there are two possible solutions: One is to augment the transition specifications of each method to realize the partition. The other is to use method sets bound with guards—method sets are 'refined' with guards to dynamically add `get2` as an element depending on the internal state of the object.

Solution with Synchronizers:

```
Class x-buf2: b-buf { //x-buf2 subclass of b-buf
// Optional re-definitions of method sets,
//   necessary if the future subclasses
//   are to use transitions.
method_sets:
  mset  FULL    super FULL | #{get2};
  mset  PARTIAL super PARTIAL | #{get2};
  mset  ONE     super PARTIAL;
synchronizers:
  (size > 1) enables get2
methods:
  int get2() {
    // code to return two elements; }
}
```

Solution with Transition Specifications (1):

```
Class x-buf2: b-buf {
method_sets:
  mset  FULL    super FULL | #{get2};
  mset  PARTIAL super PARTIAL | #{get2};
  mset  ONE     super PARTIAL;
methods:
  int get2() {
    // code to return two elements; }
transitions:
  // account for new state partitioning
  transition default {
    become ONE when (size == 1);
    // implicit 'is super default()'
  }
}
```

Solution with Transition Specifications (2):

```
Class x-buf2: b-buf {
method_sets:
  mset FULL  super FULL
    when (size == 1)
  mset FULL  super FULL | #{get2}
    when (size > 1)
  mset PARTIAL super PARTIAL
    when (size == 1)
  mset PARTIAL super PARTIAL | #{get2}
    when (size > 1)
  mset ONE   super PARTIAL
methods:
  int get2(){
    // code to return two elements; }

// Transitions need not be re-defined
}
```

4.2 History-Only Sensitiveness—Method gget in class gb-buf:

Although not all history sensitiveness can be resolved with our scheme, cases where only the previous message affects the accept set can be handled gracefully with the family of `_once` transitions. Here, we present a solution to the `gget` method in `gb-buf`; notice that the solution works irrespective of whether `b-buf` is specified with synchronizers or transitions. A more elaborate inter-object protocol can be designed by the combined use of both synchronizers and transitions.

Solution with Transition Specifications:

```
class gb-buf: b-buf {
method_sets:
  mset AFTER-PUT  #{gget};
  mset FULL  super FULL | AFTER-PUT;
methods:
  // gget identical to get except
  //   for synchronization constraint
  int gget() { return super get(); }
transitions:
  transition put() {
    // Only once immediately after put
    disable_once AFTER-PUT;
  }
  // gget automatically handled by the
  //   default: transition, if b-buf
  //     were specified with transitions.
}
```

As another example of history sensitiveness, we can simulate the ALTERNATION behavioral class given by Rheghizzi et. al.[24] as a mixin. We define two 'abstract' method sets `FOPS` and `SOPS` in the mixin to be instantiated by the client subclass; the accept sets alternate between the two sets on each method invocation:

Alternation mixin:

```
class alternate {
```

```
method_sets:
  mset FOPS #{}; // abstract; redefined in subclasses
  mset SOPS #{}; // abstract; redefined in subclasses
transitions:
  // Alternate between FOPS and SOPS
  transition FOPS() {
    become SOPS otherwise;
  }
  transition SOPS() {
    become FOPS otherwise;
  }
}
```

4.3 State Modification Anomaly—Method Lock/Write-Lock/Unlock:

The `write-lock` class defines a two-level lock where the method `lock` locks the object exclusively so that no other methods can access it until it receives a corresponding `unlock` message, whereas the `write-lock` message allows side-effect free methods to be invoked. `Write-lock` is to be used as a mixin, so that the invokable methods under write-locked state can be extended in the subclasses. It is also straightforward to refine the locks by constructing a hierarchy of locks. Although it is possible to define `write-lock` using either synchronizers or transition specifications, we only present the definition with the latter for brevity. (Interested readers are referred to [14] for full details.)

Solution with Transition Specifications:

```
class write-lock {
method_sets:
  mset LOCKED        #{unlock}
  // redefined in subclass
  mset WRITE-LOCKED  self LOCKED;
  mset UNLOCKED      all-except(LOCKED);
methods:
  // Locking can be handled entirely
  //  with transitions.
transitions
  transition lock() { push LOCKED; }
  transition write-lock() { push WRITE-LOCKED; }
  transition unlock() { restore; }
}
```

We give two examples of the use of `lock`: One is to add a method to the `lock` class itself which inquires the status of the lock, which is not possible with Frølund's proposal[13]. Notice how the inheritance of method sets allows easy re-use of existing lock code:

```
class lock2: lock {
method_sets:
  //  Always invokable.
  mset ALWAYS    #{inquire-lock};
  mset LOCKED    super LOCKED | ALWAYS;
  // WRITE-LOCK is automatically updated
  mset UNLOCKED  super UNLOCKED | ALWAYS;
methods:
  lock_state inquire-lock() { return lock_var; }
}
```

As an example of use of `write-lock` as a mix-in, we define a class `lb-buf`, which is a bounded buffer that allows locking; in particular, if the buffer is write-locked, then only the `empty?` method can be invoked. Further extensions to `lb-buf` is possible by augmenting the method set `READ-ONLY`. We note that this type of encapsulated extensibility is much more cumbersome with other proposed schemes:

```
class lb-buf: b-buf,write-lock {
method_sets:
  // Can be extended in subclasses
  mset READ-ONLY     #{empty?};
  mset WRITE-LOCKED
     super WRITE-LOCKED | READ-ONLY;
//  No other definitions are necessary
//   Note that empty? is in both
//   UNLOCKED and WRITE-LOCKED
}
```

5 Overview of Implementing our Proposal Efficiently

To aim for practical use, our proposal is carefully crafted not to sacrifice run-time execution efficiency, despite its expressive power. Efficient implementation of our proposal is possible as an extension to the run-time architecture of our ABCL/onAP1000[28], whose key techniques allowed orders of magnitude improvement in execution efficiency:

(1) Integration of Stack-Based and Queue-Based Scheduling:
The scheduling mechanism of ABCL/onAP1000 avoids unnecessary queue manipulation by employing efficient stack-based scheduling as much as possible, significantly reducing the cost of intra-node processing.

(2) Multiple Virtual Function Tables (Multiple VFTs):
Objects synchronize on message reception by processing them under mutual exclusion, enqueueing any messages received during the processing. We eliminate this cost by integrating the synchronization/mutual exclusion check into *VFT lookup*, which is a standard scheme for implementing generic method dispatch in C++. By such inte-

gration into what is already a necessary cost, local asynchronous message passing can be optimized to 8-25 SPARC instructions/2.3μseconds. (c.f., C++ virtual function takes 8-10 instructions.)

(3) Active Message-based Inter-node Message Sending:
We reduce the overhead of remote message passing by employing the *Active Messages*[31] concept, which by only specifying the address of the message handler address customized for each message, eliminates the tags and the associated overhead caused by their interpretation. As a result, the cost of inter-object remote asynchronous message passing can be low as 8.9μseconds on AP1000.

Other techniques include low latency remote object creation via object-address prefetching/fault function tables, integrated long-range load-balancing and distributed garbage collection, etc.[28, 29]. The N-queens benchmark on a 512-node AP1000 has so far exhibited 440 times speedup, over purely sequential C++ code using essentially the same search algorithm (extra cost includes distributed termination detection using parallel tree merge).

We have managed to integrate both the synchronizers and the transition specifications into the run-time architecture of ABCL/onAP1000. This is achieved by (A) creating a set of MVFTs statically customized with special-purpose procedures for each method set, (B) augmenting the scheduling of active objects in ABCL/onAP1000, and (C) incorporating a new run-time structure called a *transition stack*. Due to space limitations, we cannot present the technical details—interested readers are referred to [28, 14].

In order to demonstrate the involved cost, we give the results of the preliminary benchmarks on intra-node asynchronous message passings to the `b-buf` objects described in Section 4 on AP1000 in Table 1, and also compare the result with that of the original ABCL/onAP1000. A local `put/get` message pair was sent repeatedly to a dormant object, and averaged out. For the original ABCL/onAP1000, a message with null arguments was sent repeatedly to invoke an empty method. The cost of unoptimized message passing (including guard evaluation for synchronizers and transitions) is approximately twice in comparison to a null message passing on ABCL/onAP1000. The compiler can optimize down to ABCL/onAP1000 message passing speed in ideal cases, however, when the method is a small, leaf method in the message chain.

The overhead incurred by our synchronization scheme is significantly smaller compared to other cost of concurrent execution in our implementation architecture. Integration of new features merely tacks on the small difference in the overhead for intra-node messages given in Table 1. Although larger-scale benchmarks are still undergoing, we believe the above figures, plus the observed speedups in larger-scale ABCL/onAP1000 benchmarks, support our claim of efficiency.

6 Discussions and Conclusion

6.1 Why Method Sets are not Full First-class 'Sets'?

Our proposal is carefully crafted to allow the structure of all the VFTs to be determined statically at compile time. This is to avoid the necessity of constructing or operating upon method sets at run-time, because such operations are costly: the number of methods for user-defined classes in large application frameworks typically exceed one hundred in real-life settings[7], meaning that each virtual table could well be over 400 bytes in size. Thus, the cost of dynamic creation/copy/updating of virtual tables would be prohibitive. Another concern is the storage space; if each concurrent object were to have a modifiable VFT, it would quickly overwhelm the available memory space—for example, our 13-queens benchmark created over 4 million objects, most of which are simultaneously alive; In this case VFT alone would consume approximately 1.6 Gigabytes in the system, if we assigned a customized VFT per each object. One could devise some clever sharing techniques, but they would involve considerable overhead of indirections as well as run-time management of sharing.

By contrast, our scheme avoids the necessity of dynamic construction and direct memory operations on the VFTs. Even method sets bound with guards can be case-analyzed at compile time. Since the maximum number of VFTs generated is bound by a constant factor proportional to the number of classes multiplied by the size of the synchronization specifications (method sets + transition lines), the number remains manageable in practice.

The use of multiple virtual table has been independently proposed by [25] for supporting fine-grained, object-wise protection capabilities. Unfortunately, the proposal involves run-time creation of

Activity	*SPARC Instructions*				
	`b-buf` Synch. (Unoptim.)	`b-buf` Synchrnzr. (Optimized)	`b-buf` Trans. (Unoptim.)	`b-buf` Trans.Opt. (Optimized)	ABCL/ onAP1000 (Null Mess.)
Check Locality	4	4	4	4	3
Lookup and Call	6 (w/arg)	6 (w/arg)	6 (w/arg)	6 (w/arg)	5 (wo/arg)
Guard Evaluation	15	7	0	0	0
Switch VFTP to Active Mode	3	0	3	0	3
Execution of Method Body	8	8	7	7	– (empty)
Check Message Queue	3	0	3	0	3
Switch VFTP to Dormant Mode	3	0	3	0	3
Polling of Remote Message	5	0	5	0	5
Transition Execution	11	3	22	14	0
Stack Pointer Adj. and Return	3	3	3	3	3
Total Instrs. (`put()`)	61	31	56	34	25
Message Instrs. (incl. Guards)	52	22	48	26	25
Total Time (μsec, AP1000@25Mhz)	5.7	3.4	5.5	3.4	2.3

Table 1: Benchmark of Intra-node Message to Dormant `b-buf` Object on AP1000.

customized virtual table per each object, exactly what we avoid.

6.2 Inheriting Synchronization Code with Encapsulation—Good Design is Still Important

Method sets as 'open' interface in inheritance of synchronization code assimilates the benefits of method inheritance by giving the programmer opportunities to augment the synchronization specifications of the superclasses with very minor modifications in a syntactically familiar way. In particular, our design allows automatic re-use and encapsulation in many common situations, promoting definitions of abstract classes that serve as basis of application frameworks for parallel programs. Still, language mechanisms alone are not omnipotent, and we do not claim that our proposal alone is the panacea; rather, we stress the importance of both the detailed user knowledge of the synchronization constraints, and good OO-design disciplines in reuse. When inheriting methods, it is impossible to re-use or refine them in subclasses, unless the user has a good understanding of their behavior. Analogously, the user has to have a good understanding of the abstract state that each method set represents plus their transitional behavior when re-using synchronization code to maintain proper encapsulation.

6.3 Which Synchronization Scheme do we Use?

The user has the liberty of using the 'appropriate' synchronization scheme depending on the synchronization constraint he has to satisfy, and in many cases either scheme can be used, just as there are many ways to implement the functionality of a method. There are, however, situations where one is more preferable, depending on (1) semantical and/or (2) efficiency issues. For example, cases involving simple state partitioning are better programmed using synchronizers, whereas cases where a certain set of methods needs to be enabled/disabled in protocols are better programmed using transitions. As for efficiency, the use of transitions adds an extra overhead of indirection and transition stack management, but there are cases where it is better, e.g., when the message queue of an object becomes long, synchronizers have to evaluate the guards for each message in the queue, whereas transitions need not[27]. Establishing a concrete guideline is a subject of our future work.

6.4 Previous Work

Early idea of separate inheritance of guards/method bodies/state transitions was given by Shibayama[26], where methods are categorized into *primary*, *constraint*, and *transition* methods, and each can be separately defined/inherited/overridden. The proposal was not as developed nor as practical compared to our scheme, however, for reasons includ-

ing: (1) there was no way to operate upon *sets* of methods as abstract synchronization states of objects, resulting in lack of encapsulation as well as other drawbacks such as cumbersome re-definitions when multiple methods are affected, (2) almost identical inheritance rules were applied to each category, which made code re-use awkward, and (3) no implementation schemes nor benchmarks were given—in particular, Shibayama's proposal requires a number of successive method delegations to describe state transitions, whose execution could be a considerable overhead.

Frølund has proposed a synchronization scheme that allows flexible re-use purely based on guarded methods[13]. The difference is that methods are essentially *disabled* with guards instead of being enabled as in our proposal. As for semantical issue of conformance, differences are difficult to judge—there are cases where 'disabling' is better, whereas one can also consider situations where 'enabling' is better. Both are possible with our scheme, the former implicitly and the latter via the `super` reference to synchronizers of the superclasses. Other categories of anomalies are not well-handled in [13]: in the `lock` example, it is only possible to specify a single exemption method from disabling with the `all-except(`*method-name*`)` construct, e.g.,

```
(lock_var == 1) disables all-except(lock);
```

Here, it is impossible to add the method `inquire-lock` as we did in Section 4, because `all-except(lock)` prohibits any further extensions on the constraint on `lock_var`, which in turn disallows addition of methods which *can* be invoked under the same synchronization constraint as `lock`. No implementation issues were discussed, either.

Reghizzi et. al. takes a very conservative approach[24]: (1) synchronization code is totally separated within the definitions of *behavior classes (b-class)*, which are mixed into *free classes (f-class)* to create classes with appropriate concurrent behavior (called *b-inheritance* in their terminology), and (2) no further subclassing is permitted once such mixin occurs (the resulting class is called *regulated classes or r-class*). B-classes themselves are not inheritable either. Mapping between names in a b-class and actual method names in f-classes is specified at the time of mixin. Because synchronization code is *never* inherited, anomaly obviously does not occur. However, the approach seems too restrictive: aside from the inability to specialize on concurrent object classes (r-classes), it would be very difficult to cope with synchronization constraints that depend on (non-generic) internal states of objects of specific classes. One interesting point in their work is the comprehensive set of operators available in their synchronization scheme for supporting intra-object concurrency. Although some of their examples can be simulated with ours as exemplified in Section 4, it remains to be seen whether their scheme could be subsumed if we extended our approach to intra-object concurrency.

Ishikawa proposed a communication mechanism between concurrent objects that introduced the notion of method set[11], similar but different from ours. The mechanism is intended for the re-use of communication protocols more complex than the standard client-server protocol. It is difficult to compare the proposal to ours, because the original intent of the mechanism was not on solving inheritance anomaly. In the paper, Ishikawa does present a 'solution' to the `lock` example, but the solution is limited, because as far as we understand the solution cannot be extended to abstract mix-in class examples such as `write-lock`. Also, since state transitions are directly programmed in the methods, state partitioning anomaly easily occurs. Furthermore, no implementation schemes nor performance results were given; in fact we believe it is difficult to implement the proposal efficiently, because the basic operations require first-class operations on the *visible set*—which dictates the accept-set of an object—per each method invocation.

Meseguer recently proposed a solution to the inheritance anomaly problem with the Maude language[18], which is base on concurrent rewriting logic[17]. Inheritance anomaly is avoided in Maude with side conditions placed on the rewrite rules of the logic serving as guards, avoiding the state-partitioning anomaly. In addition, rewrite rules in Maude can operate on the term structures themselves as first class values, providing implicit reflective capabilities. This allows history information to be encoded within the term structure of the class definition. For example, the `lock` example can be solved by encoding the class identifier as a first class value, and using pattern matching on the identifier value to distinguish locked and unlocked state. We have not yet completely analyzed the difference in the descriptive power of our proposal and Maude, but there is a major difference when practice is considered: Although [18] does give some initial ideas of implementation, it is still not obvious how Maude can be practically implemented on a conventional MPP. The main problem

is that, because the solutions are based on a powerful run-time pattern-matching on first-class term values, a single message send could take at the least several hundreds of instructions, unless some novel implementation scheme is devised.

6.5 Conclusion and Current Work

The prime objective of OOCP languages is to provide maximum computational and modeling power through concurrency of objects plus OO-software engineering disciplines. Unfortunately, synchronization code is not trivially inheritable due to inheritance anomaly. We have investigated various categories of inheritance anomaly, and presented language constructs that facilitate multiple synchronization scheme constructs to be integrated, allowing proper encapsulation as well as efficient implementation. Currently, we are working on the followings:

- Completing the implementation of the compiler and the language system. There are many other implementation issues that we have investigated, such as concurrent OO-specific optimizations, distributed garbage collection and long-term load-balancing[29], etc. In particular we are planning to devise more extensive compiler optimizations to 'compile-away' unnecessary guards and other synchronization code when possible.

- Throughout the paper, inheritance was used as a means of code re-use. Relationship with another aspect of inheritance, subtype classifications, might be required. Recent work by America et. al. to separate the subtyping hierarchy from the inheritance hierarchy in the POOL family of concurrent-OO languages such as POOL/I and POOL/S[5, 4] could bring the two aspects together: in their formalism, subtyping relationship is determined solely by the observed external behavior and not the internal structure. Using such formalism, it could be possible to 'type' the behavior of a concurrent object whose synchronization was specified using our scheme. Recent works towards active type systems e.g., by Nierstraz[22], could also be beneficial in this regard.

- Construction of practical parallel applications on top of those systems. Applications for N-body simulation and Genome RNA secondary structure prediction are being ported.

Acknowledgements

Many thanks to Gul Agha, Pierre America, Jens Palsberg, José Meseguer, Dennis Kafura, Etsuya Shibayama, Yutaka Ishikawa, Makoto Takeyama, Ken Wakita, the anonymous referees, and the members of the ABCL project group for helpful comments and discussions.

References

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

[2] Mehmet Aksit and Anand Tripathi. Data abstraction and mechanism in Sina/ST. In *Proceedings of OOPSLA'88*, volume 23, pages 267–275. SIGPLAN Notices, ACM Press, September 1988.

[3] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[4] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA'90*, volume 25, pages 161–168. SIGPLAN Notices, ACM Press, October 1990.

[5] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May,1990*, number 489 in Lecture Notes in Computer Science, pages 60–90. Springer-Verlag, February 1991.

[6] J. K. Annot and P. A. M. Haan. POOL and DOOM: The object oriented approach. In P. C. Treleaven, editor, *Parallel Computers: Object-Oriented, Functional, Logic*, chapter 3, pages 47–79. John Wiley & Sons, Ltd., 1990.

[7] Brian M. Barry. Prototyping a real-time embedded system in smalltalk. In *Proceedings of OOPSLA'89*, volume 24, pages 255–265. SIGPLAN Notices, ACM Press, October 1989.

[8] Jean-Piere Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *Proceedings of ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 33–40. Springer-Verlag, 1987.

[9] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 102–104. SIGPLAN Notices, ACM Press, April 1989.

[10] Denis Caromel. Programming abstractions for concurrent programming — a solution to the explicit/implicit control dilemma. In B. Meyer, J. Potter, M. Tokoro, and J. Bezivin, editors, *Proceedings of TOOLS 3, Sydney*, pages 245–253, November 1990.

[11] Yutaka Ishikawa. Communication mechanisms on autonomous objects. In *Proceedings of OOPSLA'92*, volume 27, pages 303–314. SIGPLAN Notices, ACM Press, October 1992.

[12] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor based concurrent object-oriented languages. In *Proceedings of ECOOP'89*, pages 131–145. Cambridge University Press, 1989.

[13] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP'92*, 1992.

[14] Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming.* PhD thesis, the University of Tokyo, Tokyo, Japan, June 1993.

[15] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Synchronization constraints with inheritance: What is not possible — so what is? Technical Report 10, Department of Information Science, the University of Tokyo, 1990.

[16] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, 1993.

[17] José Meseguer. A logical theory of concurrent objects. In *Proceedings of OOPSLA'90*, volume 25, pages 101–115. SIGPLAN Notices, ACM Press, October 1990.

[18] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings of ECOOP'93*, 1993. (to appear).

[19] Christian Neusius. Synchronizing actions. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 1991.

[20] Oscar Nierstrasz. Active objects in Hybrid. In *Proceedings of OOPSLA'87*, volume 22, pages 243–253. SIGPLAN Notices, ACM Press, October 1987.

[21] Oscar Nierstrasz and Michael Papathomas. Viewing objects as patterns of communicating agents. In *Proceedings of OOPSLA'90*, volume 25, pages 38–43. SIGPLAN Notices, ACM Press, October 1990.

[22] Oscar Nierstraz. Regular types for active objects. In *Proceedings of OOPSLA'93*, September 1993.

[23] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, chapter 12, pages 207–245. Université de Geneve, 1989.

[24] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 148–166. Springer-Verlag, 1991.

[25] Joel Richardson, Peter Schwartz, and Luis-Felipe Cabrera. Cacl: Efficient fine-grained protection for objects. In *Proceedings of OOPSLA'92*, volume 27, pages 263–275. SIGPLAN Notices, ACM Press, October 1992.

[26] Etsuya Shibayama. Reuse of concurrent object descriptions. In B. Meyer, J. Potter, M. Tokoro, and J. Bezivin, editors, *Proceedings of TOOLS 3, Sydney*, pages 254–266, November 1990.

[27] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Efficient implementation of fine-grained object-wise synchronization schemes. (To be submitted), 1993.

[28] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of 4th ACM Symposium on Principles and Practices of Parallel Programming (PPoPP'93), San Diego*, pages 218–228, May 1993.

[29] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Incorporating locality management and gc in massively-parallel object-oriented languages. In *Proceedings of Joint Symposium on Parallel Processing*, pages 277–282. IPSJ, May 1993.

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

[31] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, volume 20, pages 256–266, Gold Coast, Australia, May 1992.

[32] Rebecca Wirfs-Brock and Ralph Johnson. Surveying current research issues in object-oriented design. *Commun. ACM*, 33(9):105–124, September 1990.

[33] Masahiro Yasugi, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proceedings of 6th ACM International Conference on Supercomputing.* ACM, 1992.

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