並列オブジェクト指向言語における Synchronization Constraints と継承について *

2E−7

松岡 聡　米澤 明憲†

東京大学理学部情報科学科‡

1 Introduction

On developing large-scale programs with object-oriented concurrent programming (OOCP) languages, we generally acknowledge that *inheritance* is one of the most essential features. However, it has been previously pointed out that *inheritance* and *synchronization constraints* in concurrent object systems often conflict with each other[1, 2]. For this reason, some languages such as ABCL/1[13] do not employ inheritance. Although several solutions[3, 4, 7, 10, 12] have been proposed in the past, we argue that, unfortunately, most of the proposals render inheritance *totally useless*.

2 Synchronization Constraints, Specification, and DKSS

A concurrent object in a certain state can accept only a subset of its entire set of messages in order to maintain its internal integrity. We call such a restriction on acceptable messages the *synchronization constraint*. In most OOCP languages, the programmer gives either implicit or explicit program specification to control the set of acceptable messages. We call such specification the *synchronization specification*. The synchronization specification must always be *consistent with* the synchronization constraint of an object; otherwise the object might accept a message which it really should not accept, causing an error.

In some of the previous proposals, the programmer writes down explicit synchronization specification with what we call the *accept set*, i.e., the set of acceptable method keys[7]. Some other proposals provide indirect schemes for manipulating such sets[10, 12]. We categorize these schemes as the *Direct Keyset Specification Scheme (DKSS)*. Then, we can show for that, for proposals employing DKSS, the anomaly in inheritance occurs where *re-definitions of all relevant parent methods are necessary.*

3 An Example of Inheritance Anomaly

The example we give is a bounded buffer class, which also appears in[6, 7, 12].

Figure 1 shows a definition in the notation similar to Kafura's[7]. It is a first-in first-out buffer that can contain at most `size` items. The method `put()` stores one item in the buffer and `get()` removes the oldest one. (The code for accessing the local array storage for insertion and removal is omitted for brevity.) Two instance variables `in` and `out` count the total numbers of items inserted and removed, respectively, and act as indices into the buffer. Upon creation, the buffer is in the empty state, and the only message acceptable is `put()`; arriving `get()` messages are not accepted but kept in the message queue unprocessed. When a `put()` message is processed, the buffer is no longer empty and can accept both `put()` and `get()` messages, reaching a 'partial'

```
Class b-buf: Object { /* b-buf is a subclass of Object */
          int in, out;
 behavior: empty      = { put() };
          partial    = { put(), get() };
          full       = { get() };
   public: void b-buf() { in = out = 0;
                          become empty;
          }
          void put()   { in++;   /* insert an item */
                          if (in == out + size) become full;
                          else                  become partial;
          }
          void get()   { out++;  /* remove an item */
                          if (in == out) become empty;
                          else           become partial;
          }
}
```

Figure 1: The Bounded Buffer Class Example

(non-empty and non-full) state. When the buffer is full, it can only accept `get()`, and after processing the `get()` message, it becomes partial again.

In Figure 1, the **behavior** statements declare three sets of keys named `empty`, `partial`, and `full` assigned to {`put()`}, {`put()`,`get()`}, and {`get()`}, respectively. A synchronization specification is given using the **become** statements, each of which designating the set of method keys acceptable in the next state. We call such a set the *next accept set*. A method typically ends with conditional statements specifying the next accept set in order to maintain the consistency between the synchronization specification and the synchronization constraint. For example, in the definition of `get()`, when (`in == out`) (i.e., the buffer becomes empty), **become** `empty` is executed and the next accept set becomes {`put()`}, which does not contain `get()`; as a result, the `get()` messages become unacceptable.

Now, consider creating a class `x-buf`, a subclass of `b-buf`. `X-buf` has one additional method `get2()`, which removes the two oldest items from the buffer simultaneously. The corresponding synchronization constraint for `get2()` requires that at least two items remain in the buffer. As a consequence, the partial state must be partitioned into two — the state in which exactly one item exists, and the remaining states. In order to cope with the new constraint, we need another accept set `x-one` that represents the former state (Figure 2).

Then, notice that NONE of the methods (except the initializer) in `b-buf` can be inherited, and as a consequence, the programmer is forced to rewrite both `put()` and `get()` to maintain consistency with the new constraint! In fact, we can easily create a example where inheritance is almost totally useless for all languages classified as DKSS in a similar manner as above. Furthermore, we can show that the situation is even more serious — the addition of methods in the subclasses would *generally* exhibit this anomaly, except for special identifiable cases where previous proposals have claimed to work.

*On Synchronization Constraints and Inheritance in Concurrent Object-Oriented Lanugages

†Satoshi Matsuoka and Akinori Yonezawa

‡The University of Tokyo

```
Class x-buf: b-buf { /* x-buf is a subclass of b-buf */
behavior: x-empty   =                        renames empty;
          x-one     = {put(),get()};
          x-partial = {put(),get(),get2()} redefines partial;
          x-full    = {get(),get2()}       redefines full;
  public: void x-buf() { in = out = 0; become x-empty; }
          void get2()
             { out += 2;              /* addition of get2() */
               if (in == out)           become x-empty;
               else if (in == out + 1) become x-one;
               else                     become x-partial;
          }
/* below re-defines the corresponding methods in b-buf */
          void get() {
               out++;
               if (in == out)           become x-empty;
               else if (in == out + 1) become x-one;
               else                     become x-partial;
          }
          void put() {
               in++;
               if (in == out + size)    become x-full;
               else if (in == out + 1) become x-one;
               else                     become x-partial;
          }
}
```

Figure 2: The Extended Bounded Buffer Class Example

4 Why Does Anomaly Occur?

Now, what is the main cause of the anomaly? It is due to the properties of the class hierarchy with respect to accept sets. In the proposals, the accept sets are treated as first-class entities within the program description. Then, we can formally prove, based on Cook-Palsberg inheritance semantics[5], that the synchronization specifications of the parent classes must be modified on creation of a new subclass[11]. The only way to avoid this is to allow reference to the method keys of the child classes in the synchronization specifications of the parent classes. But this is not allowed, as *one-way references* of method keys from child classes to their parents is one of the general properties of inheritance.

5 Attaining the Efficiency of DKSS Using Program Transformation

We can further show that anomaly can be avoided by attaching a predicate to each method as a guard for synchronization specification. What then, was the motivation in the previous proposals for employing DKSS for synchronization specification in the first place? We speculate that *efficiency* is one of the prime motivations. Most naive implementations of guards would not be very efficient, as one must scan down and re-evaluate the guards in the queue for each message acceptance.

For the purpose of attaining the efficiency of the DKSS-with guards, we can use program transformation to convert the method definitions with guards to an equivalent one using DKSS[8]. The difference from the previous proposals is that this transformation is *totally invisible* to the programmer. Therefore, the full benefit of inheritance can be attained without sacrifices in efficiency. The correctness of the transformation up to arrival-order nondeterminism can be proven by showing that the concurrent objects before and after the transformation are bisimilar in the sense of Milner's CCS[9].

Acknowledgement

We thank Etsuya Shibayama, Takuo Watanabe, Makoto Takeyama, Ken Wakita and Ken-ichi Asai for valuable discussions and comments.

References

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

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

[3] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding concurrency to a statically type-safe object-oriented programming language. In *Proceedings of the 1989 Workshop on Object-Based Concurrent Programming*, volume 24(4), pages 18–21. ACM SIGPLAN, ACM, Apr. 1989.

[4] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *Proceedings of the 1989 Workshop on Object-Based Concurrent Programming*, volume 24(4), pages 102–104. ACM SIGPLAN, ACM, Apr. 1989.

[5] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA'89 Proceedings*, volume 24(10), pages 433–443. ACM SIGPLAN, ACM, Oct. 1989.

[6] D. Decouchant et al. A synchronization mechanism for typed objects in a distributed system. In *Proceedings of the 1989 Workshop on Object-Based Concurrent Programming*, volume 24(4), pages 105–107. ACM SIGPLAN, ACM, Apr. 1989.

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

[8] Satoshi Matsuoka Ken Wakita and Akinori Yonezawa. A safe inheritance scheme for concurrent object-oriented languages. In *To be presented at the 41st Annual Convention IPS Japan*, September 1990. (In Japanese).

[9] Robin Milner. *Communication and Concurrency*. Prentice Hall, Engle Cliffs, 1989.

[10] O. M. Nierstrasz. Active objects in Hybrid. In *OOPSLA'87 Proceedings*, volume 22(12), pages 243–253. ACM SIGPLAN, ACM, Oct. 1987.

[11] Ken Wakita Satoshi Matsuoka and Akinori Yonezawa. Synchronization constraints and inheritance: What is not possible — so what is? Technical Report 90-010, Department of Information Science, The University of Tokyo, 1990.

[12] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA'89 Proceedings*, volume 24(10), pages 103–112. ACM SIGPLAN, ACM, Oct. 1989.

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