Synchronization Constraints With Inheritance: What Is Not Possible — So What Is?*

Satoshi Matsuoka, Ken Wakita,
and Akinori Yonezawa

Department of Information Science†
The University of Tokyo

Keywords and Phrases
Concurrency, Synchronization Constraints
Inheritance, Encapsulation

Abstract

It has been pointed out that *inheritance* and *synchronization constraints* in concurrent object systems often conflict with each other. Several proposals have been made for resolving the conflicts. We show that, however, for many of the proposals, one cannot avoid anomaly in inheritance where re-definitions of all relevant parent methods are necessary. Our prime contribution is that we formally prove the occurrence of such an anomaly is not a minor problem in the language design, but is an inherent difficulty in the general scheme for specifying the behavior of objects with respect to synchronization. This is more serious than the violation of class encapsulation that has been pointed out by Snyder, for NONE of the parent methods can be inherited except for trivial cases. We then propose an alternative scheme in which the anomaly does not occur, while efficiency is retained using program transformation. Since this transformation is invisible to the programmer, the full benefit of inheritance can be attained in OOCP without sacrifices in efficiency.

*Technical Report 90-010, Department of Information Science, the University of Tokyo

†Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-3812-2111 (overseas +81-3-3812-2111) ex. 4108. E-mail: (Matsuoka) matsu@is.s.u-tokyo.ac.jp, (Wakita) kenny@is.s.u-tokyo.ac.jp, (Yonezawa) yonezawa@is.s.u-tokyo.ac.jp.

1 Introduction

In just a few years, massive parallel architectures will be available to professionals of numerous fields in the manner personal computers and workstations are today. *Professional computing* is the term we use for describing the computational activities of professionals requiring immense computational power. We claim that object-oriented concurrent programming (OOCP) serves as the basis for professional computing. Massive parallel architectures supporting OOCP languages are being designed and built atCaltech (Mosaic)[3], MIT (J-Machine)[8], among other places. Research on computational models for OOCP, such as the Actor model[1], design of languages based on those models, and their implementations are flourishing.

In developing large-scale programs for professional computing with OOCP languages, it is extremely important that *inheritance* is provided for high-level program abstraction/organization and re-usability. However, it has been previously pointed out that *inheritance* and *synchronization constraints* in concurrent object systems often conflict with each other[2, 4, 16]. Some have gone so far as not adopting inheritance in their languages[2, 21, 23, 24], or employed a flexible communication mechanism independent of the inheritance hierarchy[13]. Several proposals[5, 6, 9, 12, 15, 19] have been made in the past for controlling the anomaly arising from their simultaneous incorporation into OOCP languages. However, we can show for many of such proposals that the anomaly in inheritance occurs where re-definitions of all relevant parent methods are necessary.

The prime contribution of our present work is that, based on Cook's framework of inheritance semantics, we have formally proven that the anomaly is not a minor flaw in the language design that is easily circumvented, but is an inherent difficulty in the general scheme for specifying the behavior of objects with respect to synchronization. In fact, for a certain class of specification schemes in which many of the past proposals are categorized, it is ALWAYS necessary, except for insignificant situations which we formally identify, for the entire *synchronization specifications* made in the ancestor class definitions to be modified. This anomaly is more severe than the violation of class encapsulation that has been pointed out by Snyder[18], for in some of the schemes NONE of the parent methods can be inherited.

This anomaly can be avoided by attaching a predicate to each method as a guard for synchronization specification. Although the naive implementation of guards is not very efficient, we use program transformation to obtain code that is near-optimal in the sense that the time efficiency of the code would match that of the code written with the previous proposals. Since this transformation is invisible to the programmer, 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[14].

2 A Counter Example to Previous Proposals

Prior to our formal discussion on how the anomaly in inheritance occurs in the previous proposals, we present a simplified example aimed at attaining the reader's intuition. The example we give is a bounded buffer class, which also appears in[9, 12, 19].

When a concurrent object is in a certain state, it 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* of a concurrent object. 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 by specifying what we call the *accept set*, i.e., the set of acceptable method keys[12]. There are other proposals in which languages provides some indirect schemes for manipulating such sets[15, 19]. We show that such schemes cannot avoid causing serious anomaly in inheritance.

Figure 1 shows a definition of the bounded buffer class in the notation similar to Kafura's[12]. It is a first-in first-out buffer that can contain at most `size` items. It has two public methods `put()` and `get()`. 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 — the location of the next item to be put is indexed by (`in` *mod* `size`) and that of the oldest item in the buffer is indexed by (`out` *mod* `size`). 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' (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[1].

[1]Note that this cannot be done with successive messages sends of `get()`, as `get()` messages from

```
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 into a buffer */
                          if (in == out + size) become full;
                          else                  become partial;
           }
           void get()   { out++;  /* remove an item from a buffer */
                          if (in == out) become empty;
                          else           become partial;
           }
}
```

Figure 1: The Bounded Buffer Class Example

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, the methods `get()` and `put()` must be re-defined to maintain consistency with the new constraint.

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()`! This anomaly is more serious compared to the violation of encapsulation by inheritance[18], as the programmer defining the subclass must have complete access to/knowledge of the implementation of the ancestor classes. In fact, there could be cases where inheritance is almost totally useless, as shown above.

Then, is it ever possible to formulate a scheme for giving synchronization specification using method keys so that the anomaly in the manner above never occurs? We shall prove that this is impossible in general in the next section.

3 The Proof of Anomaly in Inheritance

In the previous section, a particular example demonstrated how the anomaly occurs with the previous proposals. To make our claim more general, we will *prove* that it is always necessary, except for some trivial situations, for the synchronization specifications in the parent classes to be modified in order to maintain consistency with the synchronization constraints. This is shown by employing a simple operational model of concurrent ob-

different objects may be interleaved.

```
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;
           }
/* the following 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

jects with inheritance and synchronization constraints. The outline of the proof is as follows[2]. First, we make a minor extension to the Cook-Palsberg inheritance semantics[7] to incorporate object states. Then, restricting our attention to state transitional behavior objects, we define functions that characterize the synchronization constraints of objects. Next, synchronization specifications are categorized into two schemes, one using predicates and the other using accept sets, and functions that characterize both schemes are given; consistency is then defined in terms of extensive equivalence of the functions. Finally, we prove the occurrence of the anomaly by constructing the characterization function for the scheme using accept sets, and showing that on creation of a subclass, the synchronization specifications in the superclasses must be modified in order to maintain consistency with the synchronization constraints.

3.1 Overview of Cook-Palsberg Inheritance Semantics and its Extensions

We first employ the semantics of inheritance defined by Cook and Palsberg[7]. We make minor extensions to incorporate object states, just simple enough for our purpose. Also,

[2]The proof, although not mathematically complex, is nevertheless somewhat long and tedious. We have omitted the minute details, and attempted to arouse reader intuition as much as possible. Subsection 3.1 can probably be safely skimmed on first reading.

Instances:	ρ, φ	$\in$	**Ins**
Classes:	κ	$\in$	**Cls**
Message Keys:	m	$\in$	**Key**
Primitive Functions:	f	$\in$	**Prim**
Method Expressions:	e	$\in$	**Exp**

Figure 3: Method System Domains

Undefined:			?		
Numbers:			**Num**		
Values:	α	$\in$	**Val**	$=$	**Beh** + **Num** + **Stat**
Behaviors:	σ, π	$\in$	**Beh**	$=$	**Key** $\rightarrow$ (**Fun** + ?)
Functions:	ϕ	$\in$	**Fun**	$=$	**Val** $\rightarrow$ **Val**
States:	ζ, ξ	$\in$	**Stat**		
Generators:			**Gen**	$=$	**Beh** $\rightarrow$ **Beh**

Figure 4: Semantic Domains

slight restrictions are made so that classes are *well-formed*. Other notations used here follow their paper.

3.1.1 Domains and Functions in the Method System

The definitions of method system domains are identical to Cook's (Figure 3). The definition of semantic domains is identical, except for the addition of **Stat**, the domain of states of instances (Figure 4). Intuitively, the state of an instance can be given with the values bound to its instance variables. (We could define it more precisely as is done for standard denotational semantics by introducing local locations for each instance, but for our purpose, the simple definition we give here suffices.) Introducing **Stat** also affects the domain of values, **Val**, as shown in Figure 4.

Two auxiliary functions are added to handle states. To specify the set of all possible states of instances of a class, we use the function *States* for each class κ. It has the property that for any class κ, $States(\kappa) \subset \mathbf{Stat}$. The auxiliary function *state* returns the current state of a given instance (Figure 5). We also extend Cook's definition of *par* (standing for 'parent') as follows:

$$par^n(\kappa) = \underbrace{par(par(\ldots par(\kappa)))}_{n} \quad (n \geq 1)$$

Henceforth, we adopt the notations $\kappa^\rho \stackrel{\text{def}}{=} class(\rho)$ and $\zeta^\rho \stackrel{\text{def}}{=} state(\rho)$ for brevity.

We also employ a portion of the functions in Cook's denotational system of inheritance, as shown in Figure 6. For convenience, we define $bec = \lambda\kappa.fix(gen(\kappa))$, which is the common behavior for all instances of a given class.

class	:	$\mathbf{Ins} \to \mathbf{Cls}$	the class of an instance
par	:	$\mathbf{Cls} \to (\mathbf{Cls} + ?)$	the parent class of a class
meth	:	$\mathbf{Cls} \to \mathbf{Key} \to (\mathbf{Exp} + ?)$	non-inherited methods for a class
root	:	$\mathbf{Cls} \to \mathbf{Bool}$	*true* if root class, *false* otherwise
States	:	$\mathbf{Cls} \to 2^{\mathbf{Stat}}$	the set of possible states for instances of a class
state	:	$\mathbf{Ins} \to \mathbf{Stat}$	the current state of an instance

(Abbreviations: $\kappa^\rho \stackrel{\text{def}}{=} class(\rho)$ and $\zeta^\rho \stackrel{\text{def}}{=} state(\rho)$)

Figure 5: Auxiliary Functions

beh	:	$\mathbf{Ins} \to \mathbf{Beh}$	the behavior of an instance (to be extended in Sect. 3.2)
gen	:	$\mathbf{Cls} \to \mathbf{Gen}$	the *generator* of a class
bec	:	$\mathbf{Cls} \to \mathbf{Beh}$	the behavior common to all instances of a class
bec	=	$\lambda\kappa.fix(gen(\kappa))$	(to be extended in Sect. 3.2)

Figure 6: Functions in the Denotational System of Inheritance

3.1.2 Well-Formed Classes (WFC)

Now that we have established the domains and functions in the method system, we need to define the notion of *well-formed classes (WFC)*. When a class is well-formed, all the methods available for the class are *well-defined*, that is, invocations of all methods that was defined at that class or at its superclasses do not result in `messageNotUnderstood` or infinite looping in the method lookup.

For our purpose, we assume that if a function is given a value in an illegal domain, it would return $\perp_?$. This allows us to define the following:

Definition 1 (Valid Domain Function) *Let f be a function. Then, $vd(f)$ is the subset of the domain of the function for which the mapped value is not $\perp_?$. To be more precise,*

$$vd(f) = \{x \mid x \in \text{domain of } f, f(x) \neq \perp_?\}.$$

On defining a method, we need a pair consisting of a method key ω and a method expression e_ω. We use a notation $\omega \mapsto e_\omega$ for such a pair. Note that both ω and e_ω are syntactic entities. We also need to specify exactly at which class in the inheritance tree a given method is defined.

Definition 2 *A method $\omega \mapsto e_\omega$ is* defined at *class κ iff $meth(\kappa)\omega \in \mathbf{Exp}$.*

We then restrict ourselves to *well-defined* methods, whose associated expression evaluate to a function in **Fun**:

Definition 3 (Well-Defined Method)
A method $\omega \mapsto e_\omega$ *is* well-defined at class κ *iff A method* $\omega \mapsto e_\omega$ *is defined at class* κ *and* $bec(\kappa)m \in$ **Fun**

We now define *well-formed class*, whose available methods are all well-defined.

Definition 4 (Well-Formed Class (WFC)) *A class* κ *is* well-formed *iff* κ *is a root class, or:*

1. $\forall m \in vd(meth(\kappa))$, m *is well-defined at* κ,
2. $par(\kappa)$ *is well-formed, and*
3. $\exists n$ *such that* $root(par^n(\kappa)) = true$ *(i.e., the message lookup does not 'loop').*

Some of the properties of WFCs, which we give without proof, are as follows:

- The well-formedness property is inherited; that is, when one creates a subclass of a WFC by adding only well-defined methods, then the created subclass is a WFC.
- A set of acceptable method keys of an instance, $(vd(beh(\rho)))$, is equivalent to the union of all keys of its class and of its ancestor classes.

3.2 Concurrent Objects with Synchronization Constraints

We will next define the operational behavior of concurrent objects with synchronization constraints. For our purpose, we use an extension of the model given by Shibayama[17], in which the state transition of an object is given with a function that takes three arguments: an object, the key of the message accepted, and the state of the object at the time of the message acceptance. The result of the function is the next state of the object. In our framework, we restrict the domain of primitive functions to **Stat** $\rightarrow$ **Stat**; for a pure function, i.e., a function that does not affect the state of the object, we define it as identity with respect to state transition. Then, *beh* becomes the state transition function, as beh : **Ins** $\rightarrow$ **Key** $\rightarrow$ **Stat** $\rightarrow$ **Stat**. Since our system has classes, we also require a function that gives the state transition behavior common to all instances of a class; this is achieved with *bec*, as bec : **Cls** $\rightarrow$ **Key** $\rightarrow$ **Stat** $\rightarrow$ **Stat**.

3.2.1 Accept Function

Our next step is to characterize the 'synchronization constraints'. For this, we introduce the *accept function*, which denotes the abstract condition of message acceptance. As we have indicated earlier, a message is acceptable by an object if the invocation of the associated method does not violate the assertion (concerning the synchronization constraints) that must hold for the current state of the object, in order for the internal consistency of the object to be maintained. This is a natural and powerful way to characterize the synchronization constraints, as object states can be partitioned by arbitrary predicates regarding its state.

The intuitive meaning of the accept function $\mathcal{A}$, which is defined below, is as follows: for an instance of a particular class, upon receipt of a message, when in a particular state, $\mathcal{A}$ decides whether it is safe to accept the message or not. For instance, the `get()` message in class `b-buf` should not be acceptable if the instance is in the empty state, i.e., informally $\mathcal{A}$(`b-buf`)(`get()`)(`in == out`) = *false* (Note: `in == out` stands for the empty state).

Definition 5 (Accept Function) Accept function $\mathcal{A}$ *is defined as follows:*

$$\mathcal{A} \;:\; \mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow \mathbf{Stat} \rightarrow \mathbf{Bool}$$

$$\mathcal{A}(\kappa)m\zeta \;=\; \begin{cases} \mathit{true} & \text{if } m \in vd(bec(\kappa)) \text{ and message } m \text{ is acceptable at state } \zeta \\ \mathit{false} & \text{if } m \in vd(bec(\kappa)) \text{ and message } m \text{ is not acceptable at state } \zeta \\ \perp_? & \text{otherwise} \end{cases}$$

3.2.2 Accept Sets

Alternatively, given an object and its state, we can enumerate the methods whose invocation do not violate the assertion that must hold for the current state of the object. By enumerating the keys of the methods, we obtain a finite set of method keys, which we call the *accept set*.

Definition 6 (Accept Set) *The* accept set b *is a finite set of method keys for an instance* ρ *of a WFC* κ^ρ*, for which the associated methods are acceptable for its current state* ζ^ρ*.*

Now, given $\mathcal{A}$, we can derive the *accept set function* $\mathcal{B}$ as a function returning, for a given state, the accept set of an instance of a class:

$$\begin{aligned} \mathcal{B} \;&:\; \mathbf{Cls} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}} \\ \mathcal{B} \;&=\; \lambda\kappa\zeta.\{m \,|\, m \in vd(bec(\kappa)), \mathcal{A}(\kappa)m\zeta = \mathit{true}\} \end{aligned}$$

Each accept set naturally satisfies the following condition: for any element key in the accept set, the method corresponding to the key is already defined at that class or its superclasses, *not those methods that are going to be defined at the subclasses.* This condition, which is a significant property of accept sets, is stated formally below; it is trivial to prove that it holds for any WFC κ.

$$\forall\zeta \in States(\kappa).(\mathcal{B}(\kappa)\zeta \subset vd(bec(\kappa)) \in 2^{\mathbf{Key}})$$

3.3 Schemes for Synchronization Specification: DPSS and DKSS

Before proceeding, we stress the point that $\mathcal{A}$ (and its derivative $\mathcal{B}$) merely characterize the object's synchronization constraints that must be *satisfied* to maintain its internal integrity, and does not characterize the *behavior* of an object with respect to message acceptance. The program description for controlling message acceptance, i.e., the *synchronization specification*, must be given within the textual definition of each class.

For example, in `b-buf` of Section 2, the synchronization constraint for an empty buffer is that only message `put()` can be accepted. Then, the synchronization specification is given in the definition of `b-buf` with **become** statements so that it would be *consistent* with the synchronization constraint i.e., the constraint is always satisfied. If we would mistakenly write **become** `full` for **become** `empty` in the method `get()`, then the synchronization specification would not be *consistent* with the synchronization constraint for `b-buf`, causing an error. In such a case, the synchronization constraint is said to be *broken*. Here, for a more precise treatment of the notion of consistency, we categorize schemes for synchronization specification into those using predicates and those using accept sets.

3.3.1 Direct Predicate Specification Scheme (DPSS)

The first category specifies a predicate per each method indicating the condition under which the message that invokes the method can be accepted. For example, one could attach a predicate as a guard within the text of the method expression, or give a separate description in the class definition[3]. We categorize such a scheme of synchronization specification as the *Direct Predicate Specification Scheme (DPSS)*. An example of DPSS will be given in Section 4.

The behavior of an object whose synchronization specification is given with DPSS is represented by a function $\mathcal{A}_P$: **Cls** $\rightarrow$ **Key** $\rightarrow$ **Stat** $\rightarrow$ **Bool**. In order for the synchronization specification of the object to be consistent with its synchronization constraint, $\mathcal{A}$ and $\mathcal{A}_P$ must be *extensively equivalent*, that is, for a given key, and state, they must evaluate to the same value; otherwise, the synchronization constraints would be broken. To present an informal example, $\mathcal{A}$(`b-buf`)(`get()`)(`in == out`) = $\mathcal{A}_P$(`b-buf`)(`get()`)(`in == out`) = *false* must hold. Otherwise, if $\mathcal{A}_P$(`b-buf`)(`get()`)(`in == out`) were *true*, an attempt would be made to `get()` from an empty buffer, which would result in an error.

Definition 7 *The accept function $\mathcal{A}$ and the DPSS function $\mathcal{A}_P$ are said to be* consistent *for class κ iff $\mathcal{A}(\kappa)$ and $\mathcal{A}_P(\kappa)$ are extensively equivalent for all $m \in vd(bec(\kappa))$ and $\zeta \in States(\kappa)$.*

3.3.2 Direct Keyset Specification Scheme (DKSS)

Alternatively, one comes up with an idea of directly specifying the accept sets as first-class entities within the program descriptions (which, as we shall see later, is the root of the anomaly). Again, these can be given within method definitions, or as separate descriptions within class definitions. We categorize this scheme as the *Direct Keyset Specification Scheme (DKSS)*.

The behavior of an object whose synchronization specification is given with DKSS is represented by a function $\mathcal{B}_K$: **Cls** $\rightarrow$ **Stat** $\rightarrow 2^{\mathbf{Key}}$. As was with $\mathcal{A}$ and $\mathcal{A}_P$, $\mathcal{B}$ and $\mathcal{B}_K$ must be extensively equivalent; otherwise synchronization constraints would be broken.

[3]We need not question whether the non-accepted message is either discarded or placed somewhere in the message queue, for it is irrelevant to the current argument.

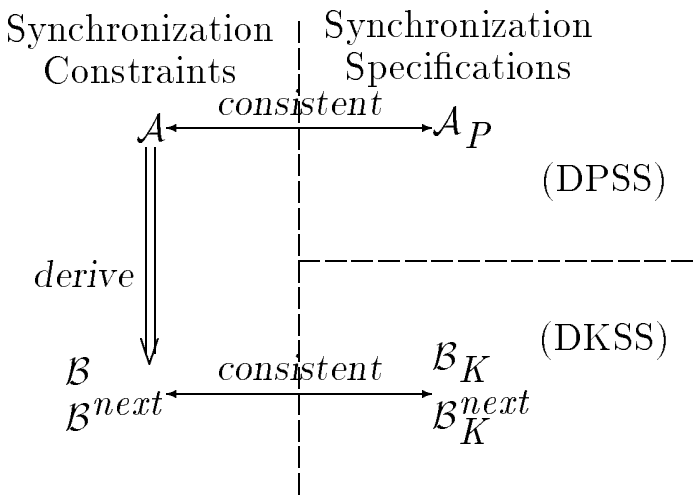

Figure 7: Relationships Between the Synchronization Constraints and the Synchronization Specifications

Another requirement derived from the equivalence is that the keys appearing in the DKSS program description must only be those of the methods that were defined at the class or at the ancestor classes — that is, methods defined at the subclasses cannot appear in the description. In more formal terms, it must satisfy for each class κ, the predicate $\forall \zeta \in States(\kappa).(\mathcal{B}_K(\kappa)\zeta \subset vd(bec(\kappa)))$. Here, one must be careful — this condition was naturally satisfied with $\mathcal{B}$, but in this case the keys given by a program could be arbitrary; for instance, one could refer to a method key of a subclass by mistake. Such cases must be detected by the interpreter/compiler and reported as an error.

Definition 8 *The accept set function $\mathcal{B}$ and the DKSS function $\mathcal{B}_K$ are said to be* consistent for class κ *iff $\mathcal{B}(\kappa)$ and $\mathcal{B}_K(\kappa)$ are extensively equivalent for all $\zeta \in States(\kappa)$.*

The relationship between $\mathcal{A}, \mathcal{A}_P, \mathcal{B}$, and $\mathcal{B}_K$ is illustrated in Figure 7. Notice that, although $\mathcal{B}$ is derived from $\mathcal{A}$, there is no direct connection between $\mathcal{A}_P$ and $\mathcal{B}_K$.

3.4 Previous Proposals and DKSS

DKSS is a generalization of the synchronization specification employed in many of the previous proposals. Although those proposals seem superficially different, they are in fact variations of DKSS.

- Some proposals represent our notion of accept sets as first-class identifiers[12]. In our framework, this is equivalent to systematically assigning a unique first-class identifier $\mathtt{a_i}$ to each accept set. Then, for the synchronization constraints, we can regard the function $\mathcal{B}(\kappa)$ as returning an identifier, i.e., $\mathcal{B}(\kappa)\zeta \longmapsto \mathtt{a_i}$. For synchronization specification, we employ $\mathtt{a_i}$ within the method expression e to denote an accept set; thus $\mathcal{B}_K(\kappa)$ also returns an identifier as a result, and the equivalence can be defined in terms of the identifier equivalence.

- Many proposals specify the *next accept set*, that is, the accept set for the next method invocation. In our framework, this corresponds to the accept set for the state after a state transition. The characterization of next accept sets induced by the synchronization constraints is given with the *next accept set function* $\mathcal{B}^{next}$, derived from $\mathcal{A}$ as follows:

$$\begin{array}{lcl} \mathcal{B}^{next} & : & \mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}} \\ \mathcal{B}^{next} & = & \lambda\kappa m\zeta.\{m' \mid m' \in vd(bec(\kappa)),\ \mathcal{A}(\kappa)m\zeta = true,\ \mathcal{A}(\kappa)m'(bec(\kappa)m\zeta) = true\} \end{array} \quad (1)$$

  The behavior of an object whose synchronization specification given with the scheme of this variation is represented by a function $\mathcal{B}_K^{next} : \mathbf{Cls} \rightarrow \mathbf{Key} \rightarrow \mathbf{Stat} \rightarrow 2^{\mathbf{Key}}$. We define the consistency requirement between $\mathcal{B}^{next}$ and $\mathcal{B}_K^{next}$ analogously to those of $\mathcal{B}$ and $\mathcal{B}_K$.

Now let us consider how some of the previous proposals can be regarded as DKSS:

- Kafura&Lee's proposal[12] of *behavior abstractions* assigns first-class identifiers called *behavior names* to the accept sets. A synchronization specification is transcribed within a method expression by specifying the next accept set with the *behavior name*. As we have shown, this can be regarded as a simple variation of DKSS.

- Tomlinson&Singh's proposal[19] of *enabled–sets* is similar to Kafura&Lee's proposal. The difference is that they give a first-class status to the accept sets to resolve the problems discussed in their paper. Inclusion of subclass method keys in one of parent's accept sets is prohibited by the condition given in Subsection 3.3, however. This, together with the fact that the synchronization specification is transcribed within the method expression, their proposal does not solve the anomaly as we will show.

- Nierstrasz's proposal of *delay queues* in Hybrid[15] is another variant of DKSS. A delay queue is associated with each method, which is either *opened* or *closed* as specified within the method expressions. The entire set of delay queues used in the object corresponds to the accept set of the object, and opening/closing queues can be regarded as element addition/deletion operations on the set, respectively.

3.5 Proof of the Anomaly in Inheritance with DKSS

We now prove that the anomaly in inheritance always occurs for DKSS. Our proof is done with $\mathcal{B}_K^{next}$ for convenience, but a similar proof can be formulated for $\mathcal{B}_K$. For simplicity, we only consider single inheritance, and that only a single method is defined at each class.

Let $\kappa_0, \kappa_1, \ldots, \kappa_{n-1}$ be well-formed classes, where $root(\kappa_0) = true$, and $par(\kappa_i) = \kappa_{i-1}$ for $1 \leq i \leq n-1$. Let $\omega_i \mapsto e_{\omega_i}$ be the well-defined method at class κ_i. We next refine the formulation of $\mathcal{B}_K^{next}$: with each definition of method $\omega_i \mapsto e_{\omega_i}$ at κ_i, we associate a function $\beta_i : \mathbf{Stat} \rightarrow 2^{\mathbf{Key}}$ satisfying the condition we described in Subsection 3.3, $\forall\zeta \in States(\kappa_i).(\beta_i(\zeta) \subset vd(bec(\kappa_i)))$. β_i is an abstraction of the program description of synchronization specification made with DKSS for the method $\omega_i \mapsto e_{\omega_i}$ at class κ_i.

$$
\begin{array}{lcl}
\mathit{slook} & : & \mathbf{Cls} \to \mathbf{Key} \to (\mathbf{Stat} \to 2^{\mathbf{Key}}) \\
\mathit{slook} & = & \lambda\kappa m.\ [\lambda e \in \mathbf{Exp}.\mathit{SyncSpec}(\kappa) \\
 & & \quad \lambda v \in ?.\ \mathbf{if}\quad \mathit{root}(\kappa) \\
 & & \qquad\qquad\quad \mathbf{then}\ \lambda\zeta \in \mathbf{Stat}.\emptyset \\
 & & \qquad\qquad\quad \mathbf{else}\ \mathit{slook}(\mathit{par}(\kappa))m \\
 & &](\mathit{meth}(\kappa)m)
\end{array}
$$

Figure 8: Auxiliary Function *slook* (in Cook's notation)

Such a description can either be given within the method expression e_{ω_i}, or separately within the class specification[4]. In both cases, β_i is said to be *defined at class* κ_i *for* ω_i. To present an informal example, if the class `x-buf` only were to add a single method `get2()`, then $\beta_{\texttt{x-buf}}(\texttt{in == out}) = \{\texttt{put()}\}$.

Next, we construct $\mathcal{B}_K^{next}$ from $\{\beta_i\}_{i=1\ldots n-1}$. Our construction becomes slightly subtle if methods are overridden; given κ_i and ω_i, we cannot simply use the corresponding β_i — rather, starting from κ_i, we must search up the class hierarchy until the definition of the method is found at some class; then the β_i defined at that class is the synchronization specification for ω_i. To be more precise, let $\mathit{SyncSpec}(\kappa_i)$ denote β_i that is defined at κ_i. We define an auxiliary function *slook*, which, given a class and a key, searches up the class hierarchy and returns β_i for the key defined at the class or the 'closest' ancestor class in the class hierarchy (Figure 8). By using *slook*, we can then construct $\mathcal{B}_K^{next}$ as:

$$\mathcal{B}_K^{next} = \lambda\kappa m\zeta.\mathit{slook}(\kappa)m\zeta \tag{2}$$

Function $\mathcal{B}_K^{next}$ constructed as above needs to be consistent with $\mathcal{B}^{next}$ for each class κ. When the consistency requirement is satisfied, we call each $\{\beta_i\}_{1,\ldots,n-1}$ the *class-specific accept set specification*:

Definition 9 (Class-Specific Accept Set Specification) *Each function* $\{\beta_i\}_{i=1\ldots n-1}$ *such that* $\beta_i : \mathbf{Stat} \to 2^{\mathbf{Key}}$ *is called the* class-specific accept set specification for class κ_i *if* $\mathcal{B}_K^{next}$ *constructed in the manner of equation (2) is consistent with* $\mathcal{B}^{next}$ *for each class* κ.

We also distinguish a special case of method addition for our proof: we say that *method (key)* ω_i *prohibits method (key)* ω_j if the next accept set for ω_i never contains ω_j:

Definition 10 (Prohibition of a Method) *For class* κ, *let* $\omega_i, \omega_j \in \mathit{vd}(\mathit{bec}(\kappa))$. *Then* ω_i prohibits ω_j *in class* κ *iff* $\forall\zeta \in \mathit{States}(\kappa).(\omega_j \notin \mathcal{B}^{next}(\kappa)\omega_i\zeta)$

On defining a subclass, the synchronization specification for a ancestor method need not be re-defined if the ancestor method prohibits the added method. In practice, however, it is rare for a method to prohibit another method — methods that do are very

[4]It does not matter whatever first-class status the accept set is given.

special in the sense that they are not affected by the state of the object prior to their invocation, and also tend to have an 'absolute' effect on the state of the object. In the bounded buffer example, we could define the method `clear()` which would clear the entire contents of the buffer; then `clear()` would prohibit `get()`, as the buffer would be in the empty state no matter what state it had been in prior to the invocation of `clear()`.

Now we are ready for our proof. First, let us examine the ideal case in which the anomaly does not occur. Consider the situation where $\mathcal{B}^{next}$ is consistent with $\mathcal{B}_K^{next}$ for all classes $\kappa_1, \ldots, \kappa_{n-1}$, and we are defining κ_n to be a subclass of κ_{n-1} by having a well-defined method $\omega_n \mapsto e_{\omega_n}$ at κ_n. By all means κ_n is now a WFC. Then, we would like to give a new program description of synchronization specification at κ_n with DKSS, so that none of the synchronization specification given in the superclasses need be modified. This means that every $\{\beta_i\}_{1\ldots n-1}$ remain unchanged, for any modification in the synchronization specification in the superclasses would reflect in the change of β_i's. At the same time, $\mathcal{B}_K^{next}$ constructed as in equation (2) must be consistent with $\mathcal{B}^{next}$ for all κ_i $(1 \leq i \leq n)$. Unfortunately, this is not possible, as we state and prove below.

Theorem 11 (Anomaly in Inheritance with DKSS) *Let $\kappa_0, \kappa_1, \ldots, \kappa_{n-1}$ be well-formed classes, where $root(\kappa_0) = true$, and $par(\kappa_i) = \kappa_{i-1}$ for $1 \leq i \leq n-1$. Let $\omega_i \mapsto e_{\omega_i}$ be the well-defined methods at class κ_i. Consider creating class κ_n, a subclass of κ_{n-1}, by defining a method $\omega_n \mapsto e_{\omega_n}$ which would be well-defined at class κ_n. Also, assume that:*

1. *$\{\beta_i\}_{i=1\ldots n-1}$ are class-specific accept set specifications, and*
2. *β_n is a characterization of synchronization specification given with next accept set variation of DKSS at class κ_n.*

Then, in order for $\mathcal{B}_K^{next}$ constructed as in equation (2) to maintain consistency with $\mathcal{B}^{next}$, every $\{\beta_i\}_{i=1\ldots n-1}$ must be modified if the corresponding ω_i (i) is not overridden, and (ii) does not prohibit ω_n in class κ_n.

The outline of the proof is as follows: in the `b-buf` example, the next accept set `partial` corresponds to all states such that `0 < out - in < size`. But when a new method `get2()` is added, set of states can be partitioned by the synchronization constraints for `get2()` in such a way that in one set, the new method is acceptable, and in the other, it is not. Correspondingly, the accept set must also be partitioned (`x-one` and `x-partial`). So, the program descriptions of synchronization specifications that referred to the accept sets in all parent methods need to be re-written (`get()`, `put()`). This is presented more formally below:

PROOF:
First, without the loss of generality[5] we can assume that $States(\kappa_i)$ are equivalent for all i. Now, we can partition $States(\kappa_n)$ into disjoint subsets $S_{\omega_n}, \overline{S_{\omega_n}}$ in such a way that

[5]The proof for monotonically increasing domain would be essentially the same, albeit a little more complicated.

$S_{\omega_n} \cup \overline{S_{\omega_n}} = States(\kappa_n)$, and ω_n is acceptable iff the state of the object is an element of S_{ω_n}, and vice-versa. More formally for all $\zeta \in States(\kappa_n)$,

$$\begin{cases} \zeta \in S_{\omega_n} & (\textbf{if } \mathcal{A}(\kappa_n)\omega_n\zeta = true) \\ \zeta \in \overline{S_{\omega_n}} & (\textbf{if } \mathcal{A}(\kappa_n)\omega_n\zeta = false) \end{cases}$$

For the parent class κ_{n-1}, consider ANY method $\omega_i \mapsto e_{\omega_i}$, where $1 \leq i \leq n-1$ and ω_i is not overridden, and ω_i does not prohibit ω_n in class κ_n. We can assume in general that for any state $\zeta \in States(\kappa_{n-1})$, the next accept set is not an empty set, i.e., $\exists b \subset vd(bec(\kappa_{n-1}))$ such that $b = \mathcal{B}^{next}(\kappa)\omega_i\zeta$ and $b \neq \emptyset$. This holds because if $b = \emptyset$, the object will no longer be able to accept any messages — and since state changes are only possible when an object accepts a message, such a state is a deadlock. (This is analogous to the agent bisimilar to **0** in CCS.)

Now, for ω_i, consider a state $\xi \in States(\kappa_n)$ where the next accept set contains ω_n; namely, $\omega_n \in \mathcal{B}^{next}(\kappa_n)\omega_i\xi$. We can easily show that such a state ξ is guaranteed to exist, as ω_n is not prohibited by ω_i in κ_n. Let b_Q denote the next accept set for ξ in the parent class κ_{n-1}, that is, $b_Q \stackrel{\text{def}}{=} \{\omega_{Q1}, \omega_{Q2}, \ldots, \omega_{Ql}\} = \mathcal{B}^{next}(\kappa_{n-1})\omega_i\xi = \mathcal{B}_K^{next}(\kappa_{n-1})\omega_i\xi$. There may be other states which also map to b_Q with $\mathcal{B}^{next}(\kappa_{n-1})\omega_i$. Let $Q \subset States(\kappa_{n-1})$ be a set of all such states; in other words, $Q = \{\zeta \mid \mathcal{B}^{next}(\kappa_{n-1})\omega_i\zeta = b_Q\}$. By the definition of $\mathcal{B}^{next}$ and the construction of $\mathcal{B}_K^{next}$, the corresponding $\beta_i(\zeta)$ defined for ω_i at κ_i must likewise be equal to b_Q for all $\zeta \in Q$ in order for $\mathcal{B}_K^{next}$ to maintain consistency with $\mathcal{B}^{next}$.

Next, partition Q into disjoint subsets $q, \overline{q}$ so that $q = Q \cap S_{\omega_n}$ and $\overline{q} = Q \cap \overline{S_{\omega_n}}$. Then, ω_n is acceptable only if $\zeta \in q$. So, by the definition of $\mathcal{B}^{next}$ in (1), we derive the following for each $\zeta \in Q$ (Figure 9):

$$\mathcal{B}^{next}(\kappa_n)\omega_i\zeta = \begin{cases} b_Q \cup \{\omega_n\} & (\zeta \in q) \\ b_Q & (\zeta \in \overline{q}) \end{cases} \tag{3}$$

Here, we can guarantee that q is non-empty by the construction of Q (since ξ was chosen so that the next accept set would contain ω_n in the first place) and thus $\mathcal{B}_K^{next}$ must be modified so that it is consistent with $\mathcal{B}^{next}$ for class κ_n. But we can easily show from the construction of $\mathcal{B}_K^{next}$ in equation (2) that this requires modifications to β_i, so that $\beta_i(\zeta) = b_Q \cup \{\omega_n\}$ if $\zeta \in q$, and $\beta_i(\zeta) = b_Q$ if $\zeta \in \overline{q}$. Since this applies to any ω_i such that $1 \leq i \leq n-1$ as the selection of ω_i was arbitrary, every β_i must be modified, if both (i) and (ii) hold for ω_i, in order to maintain consistency. ∎

As an additional note, careful readers might be concerned with the case $\overline{q} = \emptyset$; actually, this is the only case where indirect re-naming of accept sets by the previous proposals might work. Although we have deliberately omitted the treatment here for brevity, an analysis reveals that occurrence of such a case is just as rare as the prohibition of methods (every $\overline{q}$ must be $\emptyset$).

3.6 The Main Cause of Anomaly in Inheritance

What then, is the main cause of the anomaly? It is due to the properties of the class hierarchy with respect to accept sets. In DKSS, the accept sets are treated as first-

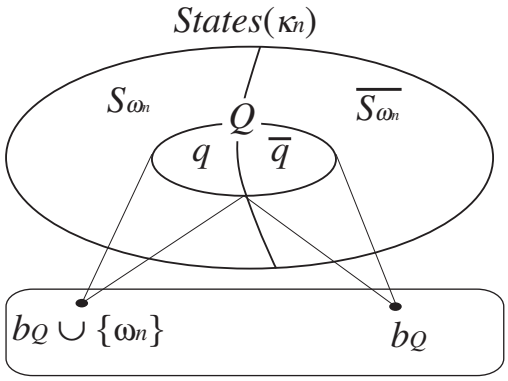

Figure 9: Proof of Anomaly in Inheritance

class entities within the program description. Then, as we have seen in the proof, the synchronization specifications of the parent classes must be modified on creation of a new subclass. The only way to avoid this in DKSS 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. If we were to allow inverse references from parents to their children to be predetermined, inheritance would be almost useless — once the class hierarchy is created, not only the hierarchy itself, but the methods contained therein cannot be modified.

It is easy to see that DPSS does not exhibit the anomaly of DKSS, since method keys are not (usually) treated as first-class entities within the predicates. As a result, the synchronization specification for a particular method is totally independent of the definitions of any other methods.

4 Program Transformation for DPSS

The idea of using predicates as guards as in DPSS is not new; notable examples are Hoare's CSP[11], concurrent logic programming languages[20], etc. In the context of OOCP, there were several, including [5, 6, 9, 24]. Then what was the motivation in the previous proposals for employing DKSS for synchronization specification instead of DPSS? We speculate that *efficiency* is one of the prime motivations. Naive implementations of DPSS would scan the message queue for a message whose associated guard evaluates to true; when such a message is found, it is removed from the queue and the corresponding method is invoked. However, this implementation would be inefficient because every method invocation requires the scanning of the message queue; if several messages remained unacceptable for a long duration, the associated guard could be evaluated repeatedly without success.

To alleviate the inefficiency, we use program transformation to obtain code that are near-optimal in the sense that the time efficiency of the code would match that of the

code written with languages that employ DKSS for synchronization specification. The main idea is to convert a class definition written with guards into the one using condition variables[6]. The transformation is invisible to the programmer — thus, the full benefit of inheritance can be enjoyed without making sacrifices in efficiency. The correctness of the transformation up to arrival-order nondeterminism is proven by using the *bisimilarity* relationship in CCS founded by R. Milner et. al.[14] Here, we present only an overview and some examples of the transformation. The details of the transformation rules and the proof of correctness are given elsewhere[22].

We illustrate how the bounded buffer class and extended buffer class are expressed and transformed (Figure 10). Here, we employ a syntax of extended C++ to express a method definition with a guard as:

`m(<formal arguments>)` **when**`(guard)` { `<body of method definition>` }

where `guard` is a boolean expression. Method `m` is invoked only when `guard` evaluates to **true**. For instance, in class `b-buf`, the guard (`in < out + size`) attached to `put()` assures that `put()` is not invoked when the buffer is full. The class `x-buf` is defined as a subclass of `b-buf` with three additional methods `get2()`, `tail()` and `empty?()`. Contrary to `get()`, which removes the oldest item from the buffer, `tail()` removes the most recent one. The method `empty?()` just reports whether the buffer is empty or not, and causes no side effect to the internal state of the buffer; thus, it can always be accepted (a keyword **always** is provided as an abbreviation for **when**(**true**)). As shown in Figure 10, all the methods defined at `b-buf` are inherited by `x-buf` without any changes to the methods or the attached guards.

```
Class b-buf: Object {       /* b-buf is a subclass of Object */
          int in, out;
  public: void b-buf()  { in = out = 0; }
          void put()    when (in < out + size) { in++; }
          void get()    when (in >= out + 1)   { out++; }
}
Class x-buf: public b-buf { /* x-buf is a subclass of b-buf */
  public: void x-buf()
          void get2()   when (in >= out + 2) { out += 2; }
          void tail()   when (in >= out + 1) { in--; }
          void empty?() always { return (in == out); }
}
```

Figure 10: Definition of Bounded Buffer and Extended Buffer in DPSS

Now, we transform the method definitions with the **when** clauses into the corresponding definition with condition variables in the following manner:

[6]Condition variables were first introduced in the monitor mechanism by C. A. R. Hoare[10]. The condition variables employed in our transformation, however, are slightly different, as no explicit signalling is done on the condition variables.

m(<formal arguments>) **when**(guard) { <body of method definition> }
$\Longrightarrow$ m(<formal arguments>) **on** c { <body of method definition> check(); }

Each method has its own message queue, and a disjoint set of queues is associated with each condition variable. Condition variables take truth values; when the object is ready to process the next message, the condition variables assigned **true** are checked. If element queues in the associated set of queues contain messages, one of the non-empty queues is chosen non-deterministically and the first message in the chosen queue is processed for method invocation. After the invocation, in order to reflect the internal state change of the object, a call to the private function `check()` is made to assign the results of evaluating the guard expressions to the corresponding condition variables. Figure 11 shows the result of transforming the `x-buf` class definition into the target language. Condition variables `c1`, `c2`, `c3`, and `c4` correspond to the guards for the methods `put()`, `get()`, `get2()`, and `tail()`, respectively. For the guard **always**, a special condition variable `T`, which is always **true**, is provided.

```
Class x-buf {
         int in, out;
         condition c1, c2, c3, c4;
 public: void b-buf()  { in = out = 0;   check(); }
         void put()    on c1 { in++;     check(); }
         void get()    on c2 { out++;    check(); }
         void get2()   on c3 { out += 2; check(); }
         void tail()   on c4 { in--;     check(); }
         int  empty?() on T  { check(); return (in == out); }
  local: void check() {
             c1 = (in < out + b-size) ? true : false; /* full? */
             c2 = (in >= out + 1)     ? true : false; /* more than one? */
             c3 = (in >= out + 2)     ? true : false; /* more than two? */
             c4 = (in >= out + 1)     ? true : false; /* more than one? */
         }
}
```

Figure 11: Transformed code.

One drawback with this transformation is that when the number of methods increases, the number of condition variables increases accordingly. Then, the overhead of `check()` becomes unnegligible since all the condition variables must be checked and updated. By performing optimizations on the transformed code, however, we can obtain a more efficient code as shown in Figure 12. Some of the optimizations employed here are as follows:

- *Merging Predicates:* In the body of `check()`, the same boolean expression (`in` $\geq$ `out` + 1) is evaluated twice for the condition variables `c2` and `c4`. We can remove this redundancy by merging `c4` with `c2`, and associating both `get()` and `tail()` with `c2`.

- *Removing Condition Variable Checking from Pure Functions:* A simple dependency analysis on `empty?()` reveals that `empty?()` is a pure function i.e., causes no side-effects to the internal state of the buffer; thus the values of condition variables should remain unchanged. As a result, a function call to `check()` in the method `empty?()` can be safely omitted.

- *Logically Deducing Assertable Conditions:* We can logically deduce that after invocation of `put()` the buffer would not be empty; thus we can safely assign **true** to `c1`. Similar analysis applies to other methods as well.

- *Re-Ordering of Conditional Variable Evaluation:* The condition for `c3 == true` implies the condition for `c2 == true`. By exchanging the order of the checking statements for `c2` and `c3`, we only need to check the conditions for `c2` if `c3 == false`.

These and other optimization techniques from traditional compilers would allow us to obtain codes that are near-optimal in the sense that the time efficiency of the code would match that of the code written with languages that employ DKSS for synchronization specification.

```
Class x-buf {
         int in, out;
         condition c1, c2, c3;
 public: void x-buf()  { in = out = 0;   check2(); }
         void put()    on c1 { in++;     check1(); }
         void get()    on c2 { out++;    check2(); }
         void tail()   on c2 { in--;     check2(); }
         void get2()   on c3 { out += 2; check2(); }
         int  empty?() on T  { return (in == out); }
  local: void check1() { c1 = (in < out + b-size); c2 = true;
                         c3 = (!c1 && (in >= out + 2)); }
         void check2() { c1 = true; c3 = (in >= out + 2);
                         c2 = (!c3 && (in >= out + 1)); }
}
```

Figure 12: Optimizations on Transformed Code

5 Acknowledgments

We would like to thank Etsuya Shibayama, Takuo Watanabe, and Makoto Takeyama for giving us numerous advises and suggestions during the course of our work. We would also like to thank Professor Takashi Masuda and other members of our department for many helpful comments and discussions.

References

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

[2] 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.

[3] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE computer*, 21(8):9–24, Aug. 1988.

[4] 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.

[5] 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.

[6] 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.

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

[8] W. Dally et al. Architecture of a message-driven processor. In *14th ACM/IEEE Symposium on Computer Architecture*, pages 189–196, Jun. 1987.

[9] 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.

[10] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 21(8):549–557, Oct. 1974.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[12] 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.

[13] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of the 1988 Conference on OOPSLA*, volume 23(11), pages 276–283. ACM SIGPLAN, ACM, Sep. 1988.

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

[15] O. M. Nierstrasz. Active objects in Hybrid. In *Proceedings of the 1987 Conference on OOPSLA*, volume 22(12), pages 243–253. ACM SIGPLAN, ACM, Oct. 1987.

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

[17] Etsuya Shibayama. How to invent distributed implementation schemes of an object-based concurrent language — a transformational approach —. In *Proceedings of the 1988 Conference on OOPSLA*, volume 23(11), pages 297–305. ACM SIGPLAN, ACM, Sep. 1988.

[18] Alan Snyder. Encapsulation and inheritance. In *Proceedings of the 1986 Conference on OOPSLA*, volume 21(11), pages 38–45. ACM SIGPLAN, Sep. 1986.

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

[20] Kazunori Ueda. Guarded Horn clause. In *Lecture Notes in Computer Science, Springer Verlag*, volume 221, pages 168–179. Springer Verlag, Berlin Heidelberg, 1986.

[21] Jan van den Bos and Chris Laffra. PROCOL a parallel object language with protocols. In *Proceedings of the 1989 Conference on OOPSLA*, volume 24(10), pages 95–102. ACM SIGPLAN, ACM, Oct. 1989.

[22] Ken Wakita. Implementation of synchronization constraints: Transformation and its correctness. Research report, Department of Information Science, the University of Tokyo, 1990.

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

[24] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the 1986 Conference on OOPSLA*, volume 21(11), pages 258–268. ACM SIGPLAN, Sep. 1986.