Why Inheritance Anomaly Is Not Worth Solving

Vincent Gramoli
NICTA and The University of Sydney
vincent.gramoli@sydney.edu.au

Andrew E. Santosa
Oracle Labs Australia
andrew.santosa@oracle.com

Abstract

Modern computers improve their predecessors with additional parallelism but require concurrent software to exploit it. Object-orientation is instrumental in simplifying sequential programming, however, in concurrent setting, programmers adding new methods in a subclass typically have to modify the code of the superclass which inhibits reuse, a problem known as the *inheritance anomaly*. There have been much efforts by researchers in the last two decades to solve the problem by deriving anomaly-free languages. Yet, these proposals have not ended up as practical solutions, thus one may ask why.

In this article, we investigate from theoretical perspective if a solution introduced extra code complexity. We model object behavior as regular language, and we show that freedom from inheritance anomaly necessitates a language where ensuring Liskov-Wing substitutability becomes a language containment problem, which in our modeling is PSPACE hard. This indicates that it is not humanly feasible to ensure that subtyping holds in an anomaly-free language. Anomaly freedom thus predictably leads to software bugs and we doubt the value of providing it.

From the practical perspective, the problem is already solved. Inheritance anomaly is part of the general *fragile base class problem* of object-oriented programming, that arises due to code coupling in *implementation* inheritance. In modern software practice, the fragile base class problem is circumvented by interface abstraction to avoid implementation inheritance, and opting for composition as means for reuse. We discuss concurrent programming issues with composition for reuse.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, concurrent programming structures, inheritance

General Terms Design, Reliability, Theory, Verification

Keywords object-oriented programming, concurrent programming, inheritance anomaly

1. Introduction

For software to be faster on nowadays energy-efficient CPU cores, it has become crucial to leverage the internal parallelism provided by most of the computational devices, including cell-phones and tablets. Object-oriented programming appears as the paradigm of choice to simplify concurrent programming [4, 12, 14, 24, 36], however, *concurrent object-oriented programming* (*COOP*) remains a difficult task. Many COOP languages are prone to what is called *inheritance anomaly* [25]. Inheritance anomaly generalizes several issues of COOP languages outlined in the literature, including the reusability of a class with synchronization code [34] and the interference of concurrency-control with inheritance [20, 42]. It refers to the inability the *incrementally* extending a class without modifying or referring to its implementation details, even though the extension is incremental in nature. Precisely because the execution of the added subclass methods may have unintended effect to the superclass methods and vice versa, the parent methods have to be rewritten. Such redefinitions annihilate the encapsulation and reusability appeal of object-oriented programs. Although there have been much efforts by researchers in the last two decades to solve the problem by deriving anomaly-free languages, these proposals have not ended up as practical solutions, therefore one may ask why.

Other than reuse, COOP programmers have another important concern for inheritance, that is to maintain an intuitive relationship between the superclass and its subclass. This intuitive relationship is often referred to as the Liskov-Wing substitution principle [22] where it is desirable that a subclass implements a subtype of its superclass's, such that a subclass object may substitute a superclass object. Maintaining this principle is crucial as a way to extend previously developed code in a way that minimizes errors and localizes the software maintenance. There are even verification and

testing approaches to ensure substitutability [22, 35]. The subtyping guarantee in an incremental inheritance is termed *behavior preservation*. Crnogorac et al. presented a contradiction between anomaly freedom and behavior preservation [6], however, its proof is based on a property that was satisfied by all COOP languages known to them at the time[1] It therefore does not answer whether it is possible, violating this property, for a COOP language to satisfy both behavior preservation and anomaly freedom. We answer this question by defining the execution behavior of an object as a regular language accepted by *nondeterministic finite-state automata* (*NFA*), and demonstrate that in our formalism, checking behavior preservation in a language that is anomaly free is PSPACE hard. It is therefore not humanly feasible to ensure subtyping in an anomaly-free language. This predictably leads to software bugs, as a subclass object may not be a substitute for a superclass object. Our result also justifies the use of verification or testing techniques by an anomaly-free language compiler to ensure substitutability. However, as these checks may be expensive, we doubt the value of providing anomaly freedom.

Interestingly, inheritance anomaly does not appear to be a major concern for practitioners. In the cases where inheritance is needed, practitioners generally bypass inheritance anomaly without even knowing them [38]. From the practical perspective, the problem is already solved. Inheritance anomaly is part of the general *fragile base class problem* of object-oriented programming [18], that arises due to code coupling in *implementation* inheritance. Implementation inheritance is precisely the situation under which inheritance anomaly arises. In modern software practice, the fragile base class problem is circumvented by interface abstraction to avoid implementation inheritance, and opting for composition as means for reuse [11]. Composing concurrent code, however, does not come for free. The issues of atomicity, lock freedom, deadlock freedom, and thread safety remain.

Section 2 explains the issues of inheritance anomaly and behavior preservation. In Section 3, we formalize them using NFA to represent object behavior. Section 4 shows the inherent difficulty to maintain behavior preservation when a language is free from inheritance anomaly. Section 5 introduces concurrent programming issues that remain when avoiding implementation inheritance altogether. Section 6 presents the related work and Section 7 concludes.

2. Implementation Inheritance Problems

2.1 Inheritance Anomaly

Inheritance anomaly originally refers to the necessity of redefining some inherited methods to maintain the integrity of concurrent objects caused by three identified reasons [25]. It was originally illustrated using a concurrent bounded buffer class `BBuf` with methods `put` and `get` that respectively adds an element if the buffer is not full or removes an element if the buffer is not empty. We show its Java version in Figure 1. `BBuf` uses a variable `state` which can have the value of either `EMPTY`, `PARTIAL`, or `FULL` to represent the possible states the buffer is in. Although this is rather contrived, it illustrates the anomalies better.

```
public class BBuf {
    protected Object[] buf;
    protected int MAX;
    protected int current = 0;
    protected int state;
    protected static final int EMPTY = 0;
    protected static final int PARTIAL = 1;
    protected static final int FULL = 3;

    public BBuf(int max) {
        MAX = max;
        buf = new Object[MAX];
        state = EMPTY;
    }
    public synchronized void put(Object v)
        throws Exception {
        while (state==FULL) { wait(); }
        buf[current] = v;
        current++;
        state = (current>=MAX? FULL : PARTIAL);
        notifyAll();
    }
    public synchronized Object get()
        throws Exception {
        while (state==EMPTY) { wait(); }
        current--;
        Object ret = buf[current];
        state = (current<=0? EMPTY : PARTIAL);
        notifyAll();
        return ret;
    }
}
```

Figure 1. Java Concurrent Bounded Buffer `BBuf`

Example 1 (Partitioning of states). The *partitioning of states* occurs when a class is extended with a method that is enabled in states that cannot be expressed by the current fields of the superclass. To extend `BBuf` with a `get2` method that removes two elements of the buffer in a row, a new state, say `ONE`, must indicate whether a single element remains in the buffer to prevent `get2` from executing; and the original `put` and `get` must be accordingly redefined to deal with this new state. We show the subclass `XBuf2` in Figure 2.

Example 2 (History-only sensitivity of states). The *history-only sensitivity of states* occurs when the states in which a method can be executed depends on the concurrent history

[1] Informally, when the class Q incrementally extends a class P, where Q has an execution state s, then there are classes P' and Q' such that Q' incrementally extends P' with the inital state of Q' is set to s, and P' has an initial state that is some execution state of P. We provide the formal description in the re-explanation of the proof in Appendix B.

```
public class XBuf2 extends BBuf {
    protected static final int ONE = 4;

    public XBuf2(int max) { super(max); }
    public synchronized void put(Object v)
        throws Exception {
        while (state==FULL) { wait(); }
        buf[current] = v;
        current++;
        state = (current==1? ONE :
                 (current>=MAX? FULL : PARTIAL));
        notifyAll();
    }
    public synchronized Object get()
        throws Exception {
        while (state==EMPTY) { wait(); }
        current--;
        Object ret = buf[current];
        state = (current==1? ONE :
                 (current<=0? EMPTY : PARTIAL));
        notifyAll();
        return ret;
    }
    public synchronized Object[] get2()
        throws Exception {
        while (state==EMPTY || state==ONE)
            { wait(); }
        current -= 2;
        Object[] ret = new Object[2];
        ret[0] = buf[current+1];
        ret[1] = buf[current];
        state = (current<=0? EMPTY : PARTIAL);
        notifyAll();
        return ret;
    }
}
```

Figure 2. Java Class `XBuf2`

of preceding method calls. This issue typically arises when extending the buffer `BBuf` with a `gget` method that cannot be executed immediately after a `put`. As this history information cannot be represented by the current state of the `BBuf` buffer and the guards, it requires a new field to indicate at anytime what is the last executed method – existing methods must be redefined to update it.

Example 3 (Modification of states). The *modification of states* occurs when an identifiable behavior is *mixed-in* to a subclass. The usual example is the behavior representable by a `Lock` class with `lock` and `unlock` methods being mixed-in to the buffer `BBuf` to be able to disable/enable method executions in its `LBBuf` subclass. Inheritance anomaly stems from the fact that the changes necessary to be able to lock the objects of `LBBuf` are not localized, but rather applies to all the `BBuf`'s methods.

```
public class BBuf {
  sync {
    put : (state != FULL);
    get : (state != EMPTY);
  }
  ...
  public void put(Object v)
    throws Exception {
    buf[current] = v;
    current++;
    state = (current>=MAX? FULL : PARTIAL);
  }
  ...
}

public class NewBBuf extends BBuf {
   sync {
      put: (super.putConstr) &&
               (Previous event==get);
      get: (super.getConstr) &&
               (Previous event==put);
   }
}
```

Figure 3. Non-Preservation of Behavior in Jeeg

2.2 Behavior Preservation

We first provide an example to motivate the section. Independent guard-based languages such as Jeeg [30] are anomaly free with respect to the three cases. However, Jeeg does not preserve superclass behavior via incremental inheritance. In Figure 3, `BBuf` is rewritten in Jeeg syntax using `sync` block to separate method guards from their code. We only display the `put` method to save space. Note that its code is less cluttered by concurrency concerns. We extend `BBuf` to its subclass `NewBBuf`, which has a `sync` block that adds constraints to the guards of `put` and `get`. For instance, `put`'s guard has the same guard as in the superclass referenced by `super.putConstr`, while having a new constraint `Previous event==get` requiring the last executed method to be `get`. The guard of `get` is redefined similarly. In this way, the invocation of `put` or `get` requires that the other method had previously been executed. The subclass `NewBBuf` therefore does not preserve the behavior of `BBuf` as none of the methods in `NewBBuf` is enabled initially.

The language Eiffel [27] has a strong support for behavior preservation via *design by contract*, where a programmer can provide class invariants, and pre- and post-conditions of methods. These conditions clarify the integrity contraint of a class, as well as the client's and the supplier's obligations in an interaction between objects. A method's redefinition satisfies *assertion redeclaration rule*, where its precondition can only be weakened, and its postcondition strengthened, respectively using `require else` and `ensure then` pre- and post-condition specifications. Their satisfaction by the method's implementation ensures that an object of a subclass

Method System Domains	
Classes	$P, Q \in \mathit{Class}$
Instances	$p, q, r \in \mathit{Instance}$

Method System Operation		
$\mathit{instances}$	:	$\mathit{Class} \to \mathcal{P}(\mathit{Instances})$

Figure 4. Simple Method System Domains and Operation

can substitute an object of the superclass, as the subclass method does not require more guarantee from the caller, nor does it guarantee less to the caller. In a concurrent setting, a precondition in particular becomes a guard. We hence have an explanation for the deadlock in Figure 3, namely a violation of the assertion redeclaration rule due to the strengthening of the preconditions of the methods. However, the question remains on the expressiveness of the syntactically-encouraged substitutability. It turns out that Eiffel is not free from inheritance anomaly. In the modification of states example, we require the method guards to be strengthened, as their executions can only commence when the object is unlocked. This fact raises the question on whether anomaly freedom and behavior preservation can coexist.

3. A Subtyping Based on NFA

3.1 Object-Orientation

Following [6], we base the domains and the operations of our object-orientation model on a simplified *method system* [5], summarized in Figure 4. We define a concurrent object-oriented programming language using an infinite set *Class* of classes, a set *Instance* of instances, and an operation $\mathit{instances} : \mathit{Class} \to \mathcal{P}(\mathit{Instance})$ with the obvious meaning. We define an *inheritance mechanism* as the pair $(\mathit{Class}, \dashrightarrow)$ with $\dashrightarrow$ a binary relation on the classes, such that for all $P, Q \in \mathit{Class}$, $P \dashrightarrow Q$ denotes that Q is a *subclass* of P or that P is a *superclass* of Q. The model subsumes both single and multiple inheritance. An *incremental* inheritance mechanism is the means by which P is incrementally extended (via addition of fields/methods) to obtain Q, for any P and Q of *Class*. $P \dashrightarrow_I Q$ means that Q incrementally extends P.

3.2 A Typing Domain

We consider a concurrent system of objects that are accessed through methods that modify their state. Our formal treatment shall be based on indivisible atomic operations, called *steps*, on the fields of an object, regardless of granularity, that is, they can be as large-grained as methods, or as fine-grained as atomic machine instruction. To express inheritance anomaly examples of [25], however, we assume that each such operation corresponds to an atomic execution of a method. The behavior of an object, which we shall formalize later, is some set of sequences of steps that can be applied to the object. Our notion of type shall be based on behavior.

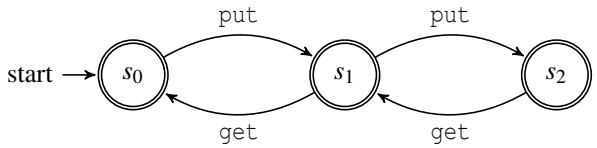


Figure 5. A Two-Element Buffer Represented as a Finite State Machine $A_{\texttt{BBuf}}$

A type belongs to a set $\mathit{Types} \subseteq \mathcal{P}(\mathit{Instance})$. For $\tau, \sigma \in \mathit{Types}$, σ is a *subtype* of τ iff $\sigma \subseteq \tau$. Moreover, an object $p \in \mathit{Instance}$ *has a type* τ iff $p \in \tau$. To properly define the typing domain *Types*, we need to first define the notion of behavior. To this end we employ a set Σ of *keys*, which are method names denoting its execution. We abstract away the arguments of the execution, keeping in mind that method calls with different arguments can be modeled using different keys. Σ^* is the set of *finite* sequences of the elements of Σ called *traces*. Assuming every method terminates, each trace intuitively represents a sequence of steps that finishes in a bounded amount of time. We write the empty trace as $\langle\rangle \in \Sigma^*$. Given, two traces $u, v \in \Sigma^*$, we refer to their concatenation as $u.v \in \Sigma^*$. Here we define a *nondeterministic finite automaton* (*NFA*) A as a quadruple (S, s_0, δ, F) with S as the finite set of *states*, $s_0 \in S$ as the *initial* state, $\delta : S \times \Sigma \to S$ as the *state transition* function, and $F \subseteq S$ as the *final* or *accepting* state[2]. The language accepted by NFA $A = (S, s_0, \delta, F)$, denoted $\mathcal{L}(A) \subseteq \Sigma^*$ is the set of traces containing the empty trace $\langle\rangle$ and $\langle m_1, \ldots, m_k\rangle$ such that there is a sequence of states $s_0, \ldots, s_k$ where $s_i \in S$ for any $0 \leq i \leq k$, for any m_{i+1} with $0 \leq i < k$, $s_{i+1} = \delta(s_i, m_{i+1})$, and $s_k \in F$. We also call a trace $w \in \mathcal{L}(A)$ a *word*. The set of all regular languages of alphabet Σ is denoted $\mathit{Reg}(\Sigma^*)$, that is, $\mathit{Reg}(\Sigma^*) = \{T \mid T = \mathcal{L}(A) \text{ for some } A, T \subseteq \Sigma^*\}$. The *behavior* of an object is defined by a function $\mathit{beh} : \mathit{Instance} \to \mathit{Reg}(\Sigma^*)$. Therefore, $\mathit{beh}(p)$ for any $p \in \mathit{Instance}$ is given by $\mathcal{L}(A)$ for some NFA A [3].

Example 4 (Bounded buffer). Consider $\texttt{put}, \texttt{get} \in \Sigma$ that we use to represent the behavior of a bounded buffer class `BBuf` with two methods: `put` and `get` that respectively inserts an element into the buffer and retrieves an element from the buffer. Figure 5 is a graphical representation of a NFA $A_{\texttt{BBuf}} = (S, s_0, \delta, S)$ that exactly captures the behavior of a bounded buffer of size two that is initially empty. $S = \{s_0, s_1, s_2\}$ is the set of states, with s_0, s_1, and s_2 respectively represent the state when the buffer is empty, has a single item, or has two items in store and where s_0 is the initial state. All states are final. An initial state is represented by an arrow without a source. The other arrows collectively

[2] For brevity we deviate from the usual convention by having the alphabet Σ, that is usually clear from the context, not included in the tuple.

[3] One may consider the use of context-free, context-sensitive, or recursive languages, however, regular languages have better decidability. Our result to be presented later depends on language containment, a problem that is undecidable even in context-free setting [19].

represent the transition function δ, where $\delta(s_i, m) = s_j$ refers to an arrow from s_i to s_j labeled m.

We define operations on behavior, consisting of the *interface extension* and the tensor *product* of NFA. The interface extension captures the notion of adding interfaces as the result of an incremental inheritance, while the product captures a restriction on the resulting behavior. The extension of $A = (S, s_0, \delta, F)$ with a set of keys $\Delta \subseteq \Sigma$, denoted $\Delta \triangleright A$ is the NFA $(S, s_0, \delta \cup \{(s, m, s') \mid s, s' \in S, m \in \Delta\}, F)$, which is defined when $(s, m, s') \notin \delta$ for any $m \in \Delta$ and $s, s' \in S$. The product $A = (S, s_0, \delta, F)$ of $A_1 = (S_1, s_1, \delta_1, F_1)$ and $A_2 = (S_2, s_2, \delta_2, F_2)$, denoted $A_1 \times A_2$ is defined when $S_1 \cap S_2 = \emptyset$ as follows.

$$\begin{array}{rcl} S & = & \{\{s,t\} \mid s \in S_1, t \in S_2\} \\ s_0 & = & \{s_1, s_2\} \\ \delta & = & \{(\{s,t\}, m, \{s',t'\}) \mid (s,m,s') \in \delta_1, (t,m,t') \in \delta_2\} \\ F & = & \{\{s,t\} \mid s \in F_1, t \in F_2\} \end{array}$$

It is easy to see that $\times$ is commutative.

We define our typing domain based on the aforementioned operations. We first define the preorder $\sqsubseteq$ that relates behavior in $Reg(\Sigma^*)$. We shall also use the notation $\sqcap_{beh}\tau$ to mean $\cap_{p \in \tau} beh(p)$.

Definition 1 (*Types*). *Let $T, U \in Reg(\Sigma^*)$. Then, $T \sqsubseteq U$ iff*

1. *$T \subseteq U$, and*
2. *for any NFA A such that $T = \mathcal{L}(A)$, there is $\Delta \subseteq \Sigma$ and NFA A' such that $U = \mathcal{L}((\Delta \triangleright A) \times A')$*[4].

Non-empty set of instances $\tau \in Types$ iff for any $p \in Instances$, whenever $\sqcap_{beh}\tau \sqsubseteq beh(p)$ then $p \in \tau$, and vice versa.

Intuitively, all objects that belong to a type of the typing domain *Types* has a common behavior. The behavior of each object subsumes the common behavior (Condition 1 in Definition 1) and is an extension of the common behavior by addition of new interfaces, and the restriction of the new behavior (Condition 2 of Definition 1).

We now state an intuitive relation between a type $\tau \in$ *Types* and an NFA that specifies its behavior, with proof in Appendix A.2.

Proposition 1. *If $\tau \in Types$, then $\sqcap_{beh}\tau = \mathcal{L}(A)$ for some NFA A.*

The following lemma formalizes the notion of subtyping based on *Types*. We provide its proof in Appendix A.3.

Lemma 2. *When $\tau, \sigma \in Types$ then σ is a subtype of τ iff $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$.*

Further in our discussion, we use the function *imp* : *Class* $\rightarrow$ *Types* that return the type $\tau \in$ *Types* that is implemented by a class. For $\tau \in$ *Types*, class *P implements* τ iff $instances(P) \subseteq \tau$. Now, $imp(P) = \cap_{instances(P) \subseteq \tau} \tau$.

[4] This condition is a prerequisite for the *safety preservation* in Section 3.3.

3.3 Subtyping Preserves Reachability and Safety

Our notion of subtyping is reasonable as it covers important cases of substitutability. According to Liskov-Wing substitution principle [22], when σ is a subtype of τ :

> Let $\varphi(p)$ be a property provable about objects p of τ.
> Then $\varphi(q)$ should be provable for objects q of σ.

In our setting when σ is a subtype of τ then $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$ (Lemma 2). We show that this notion of subtyping ensures at least two kinds of important property preservations in a subtyping: *reachability preservation* and *safety preservation*. We state reachability preservation as the following proposition. It is easy to see that it holds without a proof.

Proposition 3 (Reachability preservation)**.** *When $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$ holds and a trace $w \in \sqcap_{beh}\tau$ then all objects p of type τ (that is, $p \in \tau$) can execute w. Similarly, all objects q of type σ (that is, $q \in \sigma$) can also execute w, as $\sqcap_{beh}\sigma \sqsubseteq beh(q)$.*

Hence, our notion of subtyping captures reachability properties. In addition, it also preserves *safety* properties, which we can define to be some subset Φ of Σ^*, where any trace $u \notin \Phi$ is regarded an "error" trace. A type τ satisfies a safety property Φ iff $\sqcap_{beh}\tau \subseteq \Phi$ and Φ only includes keys from $\sqcap_{beh}\tau$ (as Φ, a "specification" of τ, should only concern the type τ). When σ is a subtype of τ, we want that σ also satisfies Φ wrt. traces containing only keys in $\sqcap_{beh}\tau$.

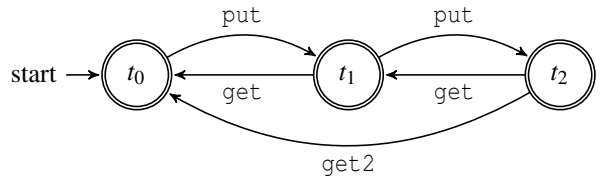
Proposition 4 (Safety preservation)**.** *When $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$ holds and a trace, consisting only of keys in $\sqcap_{beh}\tau$ does not belong to $\sqcap_{beh}\tau$, then $\sqcap_{beh}\sigma$ does not include it either.*

Proof. As assuming that $\sqcap_{beh}\tau = \mathcal{L}(A_\tau)$, then for any $p \in \tau$, $beh(p) = \mathcal{L}((\Delta \triangleright A_\tau) \times A_1)$ for some $\Delta \subseteq \Sigma$ and NFA A_1. When $A_\tau = (S_\tau, q_\tau, \delta_\tau, F_\tau)$, then $\Delta \triangleright A_\tau$ is only defined when for any $m \in \Delta$, $(s, m, s') \notin \delta_\tau$ for any $s, s' \in S_\tau$. Therefore, $\Delta \triangleright A_\tau$ does not add new traces consisting only of keys in $\sqcap_{beh}\tau$. From Proposition 9, we know that $\mathcal{L}((\Delta \triangleright A_\tau) \times A_1) = \mathcal{L}(\Delta \triangleright A_\tau) \cap \mathcal{L}(A_1) \subseteq \mathcal{L}(\Delta \triangleright A_\tau)$ and therefore the product does not add new traces either. □

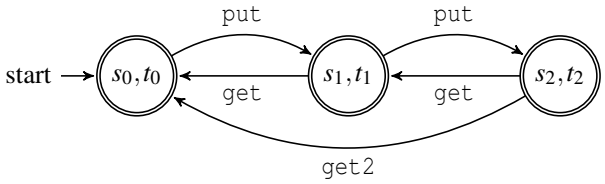
3.4 Subtyping in Cases of Inheritance Anomaly

As demonstrated in [25], inheritance anomaly arises due to inability of languages to incrementally implement three cases of behavioral relationships between superclass and subclass. We show that these relationships are also captured by our notion of subtyping.

Example 5 (Partitioning of states). We consider the partitioning of states example in Section 2.1. We assume that $\tau = imp(\texttt{BBuf})$, and therefore for any $p \in \tau$, $\sqcap_{beh}\tau \sqsubseteq beh(p)$. By Definition 1, all instances belonging to τ exhibit the execution behavior shown in Figure 5, that is, $\sqcap_{beh}\tau = \mathcal{L}(A_{\texttt{BBuf}})$. We also assume $\sigma = imp(\texttt{XBuf2})$ according to our typing domain *Types*, that is, $\sigma \in$ *Types*, and therefore for any $p \in \sigma$, $\sqcap_{beh}\sigma \sqsubseteq beh(p)$. When σ is a subtype of τ, then it should be that $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$. We show that this is the case.



(a) $A_{\texttt{get2}}$



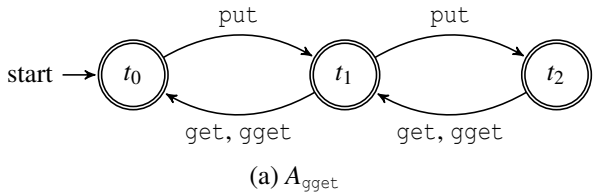
(b) $A_{\texttt{XBuf2}}$

Figure 6. Subtyping in Partitioning of States

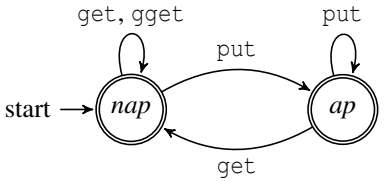
Figure 6 shows two NFA $A_{\texttt{get2}}$ and $A_{\texttt{XBuf2}}$. $A_{\texttt{XBuf2}}$ intuitively represents the behavior exhibited by all instances in σ, that is, $\sqcap_{beh}\sigma = \mathcal{L}(A_{\texttt{XBuf2}})$. The NFA $A_{\texttt{get2}}$ of Figure 6 (a) captures the behavior of `get2` wrt. the bounded buffer, which is an operation to remove two consecutive elements from the buffer. (In defining $A_{\texttt{get2}}$ we do not need to have a knowledge on $A_{\texttt{BBuf}}$'s states.) Now, $A_{\texttt{XBuf2}} = (\{\texttt{get2}\} \rhd A_{\texttt{BBuf}}) \times A_{\texttt{get2}}$ and therefore $\mathcal{L}(A_{\texttt{XBuf2}}) = \mathcal{L}((\{\texttt{get2}\} \rhd A_{\texttt{BBuf2}}) \times A_{\texttt{get2}})$. Obviously, $\mathcal{L}(A_{\texttt{BBuf}}) \subseteq \mathcal{L}(A_{\texttt{XBuf2}})$, and therefore $\mathcal{L}(A_{\texttt{BBuf}}) \sqsubseteq \mathcal{L}(A_{\texttt{XBuf2}})$ and $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$.

Example 6 (History sensitivity). In the the example for history-only sensitivity of states in Section 2.1, the method `gget` can be executed when the last execution was not `put`. We assume that the subclass of the bounded buffer is named `GBBuf`. Here we assume that $\tau = imp(\texttt{BBuf})$ and $\sigma = imp(\texttt{GBBuf})$ with respect to *Types*, and therefore $\tau, \sigma \in$ *Types*. Similar to Example 5, we assume that $\sqcap_{beh}\tau = \mathcal{L}(A_{\texttt{BBuf}})$. Figure 7 shows three NFA: $A_{\texttt{gget}}$, A_{ap}, and $A_{\texttt{GBBuf}}$. The NFA $A_{\texttt{gget}}$ intuitively captures the behavior of `gget` wrt. the buffer size, A_{ap} the same wrt. how it is affected by the execution of `put`, and $A_{\texttt{GBBuf}}$ captures the behavior of an instance of the class `GBBuf`, that is, $\sqcap_{beh}\sigma = \mathcal{L}(A_{\texttt{GBBuf}})$. Observe that $A_{\texttt{GBBuf}} = (\{\texttt{gget}\} \rhd A_{\texttt{BBuf}}) \times (A_{\texttt{gget}} \times A_{ap})$, and hence $\mathcal{L}(A_{\texttt{GBBuf}}) = \mathcal{L}((\{\texttt{gget}\} \rhd A_{\texttt{BBuf}}) \times (A_{\texttt{gget}} \times A_{ap}))$. It is easy to see that $\mathcal{L}(A_{\texttt{BBuf}}) \subseteq \mathcal{L}(A_{\texttt{GBBuf}})$ as both $A_{\texttt{gget}}$ and A_{ap} imposes no restriction on the traces in $\mathcal{L}(A_{\texttt{BBuf}})$. Therefore, $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$.

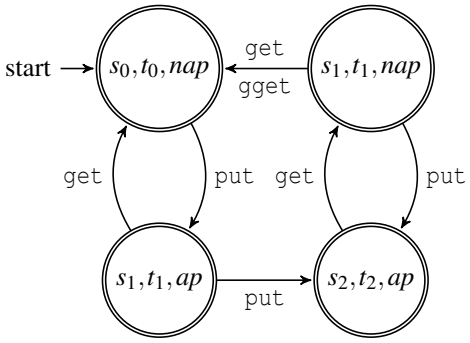
Example 7 (Modification of states). Let us re-visit the example for modification of states in Section 2.1. Recall that the class which is a version of the bounded buffer with additional `lock` and `unlock` methods is called `LBBuf`. Here we assume that $\tau = imp(\texttt{BBuf})$ and $\sigma = imp(\texttt{LBBuf})$ with respect to *Types*, and therefore $\tau, \sigma \in$ *Types*, and also that $\sqcap_{beh}\tau = \mathcal{L}(A_{\texttt{BBuf}})$. Figure 8 shows the NFA A_{lo} and $A_{\texttt{LBBuf}}$. The behavior of the instances in σ is intuitively captured



(a) $A_{\texttt{gget}}$



(b) A_{ap}



(c) $A_{\texttt{GBBuf}}$

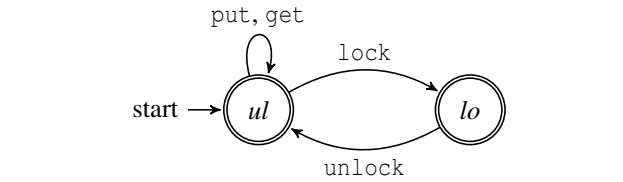
Figure 7. Subtyping in History-Only Sensitivity

by $\mathcal{L}(A_{\texttt{LBBuf}})$, that is, $\sqcap_{beh}\sigma = \mathcal{L}(A_{\texttt{LBBuf}})$. The equation $A_{\texttt{LBBuf}} = (\{\texttt{lock}, \texttt{unlock}\} \rhd A_{\texttt{BBuf}}) \times A_{lo}$ holds, and therefore $\mathcal{L}(A_{\texttt{LBBuf}}) = \mathcal{L}((\{\texttt{lock}, \texttt{unlock}\} \rhd A_{\texttt{BBuf}}) \times A_{lo})$. It is important to note that the NFA A_{lo} does not impose any restriction on the execution of `put` and `get` of `BBuf`. Therefore, the execution traces of $p \in \sigma$ includes executions of `lock` and `unlock`, but the executions of `put` and `get` follows the same behavior as that of τ. That is, $\mathcal{L}(A_{\texttt{BBuf}}) \subseteq \mathcal{L}(A_{\texttt{LBBuf}})$, and therefore $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$.

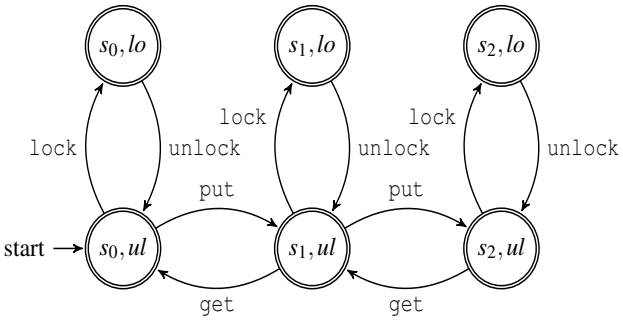
4. Intractability of Behavior Preservation under Anomaly Freedom

In the Jeeg example of Figure 3, an instance of `NewBBuf` can execute neither `put` nor `get`. With respect to *Types* (Definition 1), $imp(\texttt{NewBBuf}) \not\sqsubseteq imp(\texttt{BBuf})$ as $\sqcap_{beh} imp(\texttt{BBuf}) \not\sqsubseteq \sqcap_{beh}\ imp(\texttt{NewBBuf})$ (Lemma 2) because $\sqcap_{beh} imp(\texttt{BBuf}) \not\subseteq \sqcap_{beh}\ imp(\texttt{NewBBuf})$ (Definition 1). For this particular example, it is computationally easy to show that behavior preservation does not exist, as a compiler can simply examine that none of the guards are enabled in the initial state. The question is whether all cases of behavior preservation under anomaly-free incremental inheritance are easy to ensure. We show that this is not the case.

We first restate the definition of behavior preservation [6].



(a) A_{lo}



(b) $A_{\texttt{LBBuf}}$

Figure 8. Subtyping in Modification of Acceptable States

Definition 2 (Behavior Preservation)**.** *An inheritance mechanism* $(Class, \dashrightarrow)$ *is* behavior preserving *iff for any* $P, Q \in Class$, $P \dashrightarrow_I Q \Rightarrow imp(Q) \subseteq imp(P)$.

Informally, a language is behavior preserving if any incremental inheritance implements a subtyping. We contrast this to anomaly freedom, which holds in a language whenever any subtyping requirement can be satisfied by an incremental inheritance. Following is its formal definition [6].

Definition 3 (Anomaly Freedom)**.** *An inheritance mechanism* $(Class, \dashrightarrow)$ *is* anomaly free *iff for any* $P \in Class$ *the following holds: if there is a* $Q \in Class$ *s.t.* $imp(Q) \subseteq imp(P)$ *then there is* $R \in Class$ *s.t.* $P \dashrightarrow_I R$ *and* $imp(R) = imp(Q)$.

We now start building our proof of intractability of behavior preservation in the presence of anomaly freedom. In Eiffel [27], behavior preservation is encouraged syntactically, such as by disallowing method guards to be strengthened in a subclass. Strictly speaking, Eiffel does not syntactically ensure behavior preservation, as programmers still have the freedom to implement a method that does not satisfy its contract, however, here we consider languages where substitutability is derivable from program syntax. We identify such languages as having the property of *sufficient* behavior preservation, as incremental inheritance guarantees a stronger relation than simply subtyping.

Definition 4 (Sufficient Behavior Preservation)**.** *A mechanism* $(Class, \dashrightarrow)$ *satisfies a* sufficient *behavior preservation property iff it is behavior preserving, and there is a relation* $\rho : Types \times Types$ *such that for any* $P, Q \in Class$:

1. $P \dashrightarrow_I Q \Rightarrow \rho(imp(P), imp(Q))$, *and*
2. $imp(Q) \subseteq imp(P) \not\Rightarrow \rho(imp(P), imp(Q))$.

Intuitively, a mechanism $(Class, \dashrightarrow)$ is sufficiently behavior preserving iff there is an additional restriction on the subtyping relationships between the subclass and superclass. Such a strong behavior preservation, however, conflicts with anomaly freedom.

Lemma 5. *Sufficient behavior preservation and anomaly freedom do not coexist in any inheritance mechanism.*

Proof. Anomaly freedom holds in a mechanism $(Class, \dashrightarrow)$ only if for any $P, Q \in Class$, if $imp(Q) \subseteq imp(P)$ then there is $R \in Class$ s.t. $P \dashrightarrow_I R$ while $imp(R) = imp(Q)$. Therefore, when $(Class, \dashrightarrow)$ is sufficiently behavior preserving, by substituting the term $P \dashrightarrow_I R$ with $\rho(imp(P), imp(Q))$ and simplifying the resulting formula, we get an entailment of $\rho(imp(P), imp(Q))$ by $imp(Q) \subseteq imp(P)$, which contradicts the definition of sufficient behavior preservation. □

Recall the discussion of `LBBuf` from Section 2.2. Even though `LBBuf` implements a subtype of that implemented by `BBuf` (Example 7), such subtyping cannot be implemented in Eiffel via inheritance.

The following lemma will be used in the proof of our main theorem. We provide its own proof in Appendix A.4.

Lemma 6. *For any* $\Delta \subseteq \Sigma$ *and NFA* A_1 *and* A_2 *with* $(\Delta \rhd A_1) \times A_2$ *defined, then* $\mathcal{L}(A_1) \subseteq \mathcal{L}((\Delta \rhd A_1) \times A_2)$ *iff* $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$.

We now state our main result.

Theorem 7. *Consider an inheritance mechanism* $(Class, \dashrightarrow)$. *Ensuring behavior preservation when* $(Class, \dashrightarrow)$ *is anomaly free is PSPACE hard.*

Proof. Assume $(Class, \dashrightarrow)$ is both anomaly free and behavior preserving. We consider how we implement such $(Class, \dashrightarrow)$. From Lemma 5, it is impossible to make use of a sufficient behavior preservation, as this violates anomaly freedom. Therefore, we have to ensure that the behavior preservation of Definition 2 holds, that is, $P \dashrightarrow_I Q \Rightarrow imp(Q) \subseteq imp(P)$. From Lemma 2, in order to establish $imp(Q) \subseteq imp(P)$ in an incremental inheritance $(P \dashrightarrow_I Q)$ we can establish instead $\sqcap_{beh} imp(P) \sqsubseteq \sqcap_{beh} imp(Q)$. From Proposition 1, $\sqcap_{beh} imp(P) = \mathcal{L}(A_P)$ and $\sqcap_{beh} imp(Q) = \mathcal{L}(A_Q)$ for some NFA A_P and A_Q. From Definition 1 we therefore need to establish two:

1. $\mathcal{L}(A_P) \subseteq \mathcal{L}(A_Q)$, and
2. there is $\Delta \subseteq \Sigma$ and A_1 s.t. $\mathcal{L}(A_Q) = \mathcal{L}((\Delta \rhd A_P) \times A_1)$.

The first condition is a regular language containment. When the second condition is also satisfied for some Δ and A_1, we have that $\mathcal{L}(A_P) \subseteq \mathcal{L}((\Delta \rhd A_P) \times A_1)$. From Lemma 6, this is equivalent to $\mathcal{L}(A_P) \subseteq \mathcal{L}(A_1)$. Since even the satisfaction of the second condition does not eliminate language containment condition, ensuring behavior preservation in anomaly-free language is therefore polynomially reducible from regular language containment, a PSPACE hard problem. □

We have shown the intractability of ensuring behavior preservation in an anomaly-free language. It is therefore not humanly feasible to ensure subtyping in an anomaly-free language, hence to ensure subtyping, the use of potentially expensive verification or testing techniques by an anomaly-free language compiler is required.

5. Avoiding Implementation Inheritance

Inheritance anomaly is an instance of a more general fragile base-class problem, which is known to plague implementation inheritance. While some constraints were listed as a designer's check list to bypass it [29], more recent observations favored composition over inheritance as a simpler alternative [11, 21]. With composition for reuse, we focus more on the problems that are specific to concurrency, listed here.

Atomicity. Preserving the atomicity of concurrent operations during composition is important. For example, the method `contentEquals` of the `java.lang.String` class of JDK 1.4 abnormally raises an `ArrayIndexOutOfBoundsException` because of this issue [9]. Guaranteeing that the composite method is atomic is not trivial, be the original methods atomic or not, and many research efforts were devoted to detect it, e.g., [9]. The same problem remains a topical issue as 56 new atomicity composition issues were identified recently in existing applications [38].

Lock freedom. The `java.util.concurrent` library of JDK7 aims at being reusable, however, most of the optional `size` methods implemented by its `Collection` (e.g., `Concurrent SkipListMap`) are not atomic, which makes them "not very useful" as the documentation indicates. This problem stems from performance considerations that influenced the design of the corresponding data structures whose elements are not locked independently. The preferred lock-free technique relies generally on adding an implementation of compare-and-swap or load-link/store-conditional.

A related problem is to keep the load of a lock-free hash table low. It is highly desirable to keep this load, which indicates the number of nodes per buckets, constant yet correctly resizing the number of buckets may be impossible because of the way concurrent methods (like `insert`) execute. This limitation led to the adoption of an elaborate linked list that can be easily extended as an alternative to traditional lock-free hash tables [39].

Deadlock freedom. Deadlock may occur in a composition. For example, let P export a `put`(x) and a `get`(x) that both acquire a lock on the object x they aim at adding or removing, and let another class P' referring to an object `o` of P as a component, exports `replace`(x,y) that invokes `o.put`(x) and then `o.get`(y). Deadlock may occur when `replace`(a,b) is invoked concurrently with `replace`(b,a). Deadlock is still known today as a major source of the complexity in COOP languages [32].

Thread safety. Thread safety ensures that no inconsistent state can lead to an exception or an error. It provides a weaker guarantee than atomicity but is often sufficient. Enforcing thread safety in code reuse by inheritance may result in inheritance anomaly [41]. However, in composition-based reuse, programmers already know intuitively how to avoid problems. Here, if a similar thread-safe class is provided, it is generally more natural to try reuse this class rather than a non-thread-safe one to derive another concurrent one [13], as synchronizing a non-thread-safe class requires additional efforts. If the thread-safe code to be copied and optimized is unavailable, skilled concurrent programmers would typically know that a universal construction is preferable to transform the sequential object into an atomic one [16] as atomicity precisely represents the preservation of the behavior of the sequential object, making the obtained semantics easy to reason about. Adding atomicity requires no redefinitions of knowledge of the components's implementation details. In Java, very minimal modifications are necessary to convert `Collections` into concurrent ones: simply wrapping them into `Collections.synchronized` or `copyOnWriteArray` makes their methods thread-safe. Here there is no reuse anomaly as it requires no redefinitions.

6. Related Work

6.1 Studies on Inheritance Anomaly

The term inheritance anomaly was coined in by Matsuoka and Yonezawa [25]. Our technical framework is based on a simplified presentation of the formal framework of Crnogorac et al. [6], with major differences. First, [6] provides a result for non-coexistence of anomaly freedom and behavior preservation, however, the proof of [6] is rather weak: it depends on a particular property of the COOP languages the authors were aware of at the time. (We re-explain the proof in Appendix B using our formalization.) The proof therefore does not remove the possibility of co-existence of anomaly freedom and behavior preservation in a language without such property. Although our intractability result of Theorem 7 of Section 4 is weaker than non-coexistence, it is purely based on our formal definitions. Our result therefore corrects that of [6] and clarifies the cost of an anomaly-free language in terms of code complexity. Second, compared to [6], our NFA-based typing domain is more intuitive, where addition of new interfaces ($\triangleright$) and restriction of new behavior ($\times$) can be represented graphically. In this, our work is also related to *typestates* [40] and AND decomposition of *Statecharts* [15]. We also provide, in addition, a demonstration on how subtyping in our framework ensures Liskov-Wing substitutability wrt. reachability and safety properties.

Although the design of our typing domain is aimed at covering the cases of inheritance anomaly in [25], there are several other cases of inheritance anomalies identified in the literature. *Real-time inheritance anomaly* is identified in [2] where a real-time interrupt handling results in modifica-

tion of methods which would not be necessary otherwise. Regardless of the real-time context, the anomaly is similar to another case of anomaly mentioned in [6] that results from the necessity to weaken synchronization. *Composition anomaly* [3] is a generalization and formalization of inheritance anomaly as the composition of the superclass and the uniquely subclass features, similar to our notion to the interface extension ($\triangleright$), however, our $\sqsubseteq$ relation of Definition 1 provides a more precise notion of extension of behavior by such composition. A particular kind of inheritance anomaly also occurs in languages that support internal object concurrency, when the execution of the new method must be made mutually exclusive to the superclass's methods [41].

6.2 Solutions to Inheritance Anomaly

Inheritance anomaly has driven the design of many COOP languages, most of which aims at better clarity and programming flexibility by separating concurrency from functionality issues. We classify the approaches based on the increasing degree of separation.

Guarded methods. Some COOP languages abstracts concurrency into *method guards*, which are per-method specifications of wait conditions, as a means to separate concurrency code from functionality. [8], for instance, views inheritance anomaly as *nested conditional critical region* (*nested CCR*) problem, where a method is considered a conditional (guarded) critical region. Code reuse is achieved through `super` keyword to invoke superclass method. This evaluates the superclass method's guard. In CJava [7], a call to `super` from the functional part of a method executes only the functional part of the superclass method, and a call to `super` within a guard only the evaluates the superclass method's guard. Similar approach is found in TAO [31] and in Java concurrent programming [17]. Guards-based language introduced in [33] employs *synchronizing actions*, whose core component is a definition of *method sets* used in guards.

Synchronization sections. Matsuoka and Yonezawa remarked that having methods guards that can be independently defined from the methods themselves seems to be a promising approach [25]. A second class of languages compartmentalize concurrency concerns within a section of the class definition. The solution of Matsuoka and Yonezawa employs *method sets* section and *synchronizers* section. Programmers specify sets of methods in the method sets section and writes guards that governs the execution of the methods of the method sets in the synchronizer section. A related approach is found in [41] with *behavior names* and *behavior change expressions* that correspond to method sets and synchronizers. The design of [10] also adopts method sets and synchronizers. [26] also supports the separation of concurrency concerns into a section of a class where guards that refer to state variables are defined. Similar decoupling of concurrency from functionality is also found in the use of `sync` section of Jeeg [30] that is shown to be free from the three anomalies of [25].

Concurrency classes and aspect orientation. A third class of COOP languages clearly separates concurrency into class-like constructs. In the language DRAGOON [1], for instance, concurrency concerns are encapsulated in a *behavioral class,* which is imported by the class that implements the functionality. The *composition filters* approach [2] allow the definition of concurrency control in an interface class separated from the implementation. A composition filter relates user-defined named conditions on object states to sets of methods, such that the conditions act as guards. [37] proposes *behavior definitions* where guards on execution history are specified using predefined primitives. These approaches are related to that of *aspect-oriented programming* (*AOP*), the difference being AOP originally proposes an automatic *weaving* of *aspects*, including functionality, concurrency, and distribution aspects [23], whereas the aforementioned approaches require manual integration.

In the above approaches, it is rather easy to write conflicting wait conditions of subclass and its superclass such that some execution traces of the superclass are no longer executable by the subclass, violating behavior preservation. Evidently, abstraction and separation do not directly contribute to substitutability, confirming our theoretical result.

7. Conclusion

Many years of research on solving inheritance anomalies have not been effective in convincing language developers to use anomaly-free languages, usable for two decades now [25]. We demonstrate theoretically that any COOP language design that achieves complete anomaly freedom necessarily makes Liskov-Wing substitutability hard. Further, inheritance anomaly can be considered as an instance of the fragile base class problem, which is currently solved in practice by using composition instead of inheritance for implementation reuse [11]. We list concurrent programming problems pertaining to composition.

Acknowledgments

Andrew E. Santosa thanks Yasuro Kawata and Mamoru Maekawa for their supervision on the topic of this article.

References

[1] C. Atkinson. *Object-Oriented Reuse, Concurrency and Distribution: An Ada-Based Approach.* Addison-Wesley, 1991.

[2] L. Bergmans and M. Aksit. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Computing*, 36(1):32–52, July 1996.

[3] L. Bergmans, B. Tekinerdogan, M. Glandrup, and M. Aksit. Composing software from multiple concerns: A model and composition anomalies. In *Proc. ICSE 2000 (2nd) Workshop on Multidimensional Separation of Concerns*, 2000.

[4] D. Caromel, L. Mateu, G. Pothier, and E. Tanter. Parallel object monitors. *Concurr. Comput. : Pract. Exper.*, 20(12): 1387–1417, 2008.

[5] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In Meyrowitz [28], pages 433–443.

[6] L. Crnogorac, A. S. Rao, and K. Ramamohanarao. Classifying inheritance mechanisms in concurrent object oriented programming. In *12th ECOOP*, pages 571–600. Springer, 1998.

[7] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in Java. In *4th OOIS*, pages 504–514. Springer, 1997.

[8] S. Ferenczi. Guarded methods vs. inheritance anomaly: Inheritance anomaly solved by nested guarded method calls. *SIGPLAN Notices*, 30(2):49–58, Feb. 1995.

[9] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03*, pages 338–349. ACM, 2003.

[10] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, 1994.

[12] N. Giacaman and O. Sinnen. Parallel iterator for parallelizing object-oriented applications. *International Journal of Parallel Programming*, 39:232–269, 2011.

[13] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley, 2005.

[14] P. Grogono and B. Shearing. Concurrent software engineering: preparing for paradigm shift. In *C3S2E '08*, pages 99–108, 2008.

[15] D. Harel. On visual formalisms. *Comm. ACM*, 31(5):514–530, May 1988.

[16] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Sys.*, 15(5):745–770, 1993.

[17] D. Holmes. Java: Concurrency, synchronization and inheritance. `http://www.mri.mq.edu.au/~dholmes/java-concurrency.html`, accessed 1998, 1998.

[18] A. Holub. Why extends is evil. *JavaWorld*, Aug. 2003. `http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html`.

[19] J. E. Hopcroft. On the equivalence and containment problems for context-free languages. *Mathematical Systems Theory*, 3 (2):119–124, 1969.

[20] D. G. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages. *Comput. J.*, 32(4):297–304, 1989.

[21] S. D. Kent. *A programming language for software components*. PhD thesis, Queensland University of Technology, 2010.

[22] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Sys.*, 16(6):1811–1841, Nov. 1994.

[23] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox PARC, Feb. 1997.

[24] Y. Lu, J. Potter, and J. Xue. Ownership types for object synchronisation. In *10th APLAS*, pages 18–33, 2012.

[25] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press, 1993.

[26] C. McHale, B. Walsh, S. Baker, and A. Donnelly. Scheduling predicates. In *5th ECOOP Workshop*, volume 612 of *LNCS*, pages 177–193. Springer, 1992.

[27] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1996.

[28] N. Meyrowitz, editor. *Proceedings of OOPSLA '89, October 1–6, 1989, New Orleans, Louisiana, USA*, Oct. 1989. ACM.

[29] L. Mikhajlov and E. Sekerinski. The fragile base class problem and its solution. Technical Report TUCS-TR-117, 1997.

[30] G. Milicia and V. Sassone. Jeeg: temporal constraints for the synchronization of concurrent objects. *Concurrency - Practice and Experience*, 17(5–6):539–572, 2005.

[31] S. E. Mitchell and A. J. Wellings. Synchronisation, concurrent object-oriented programming and the inheritance anomaly. *Computer Languages*, 22(1):15–26, Apr. 1996.

[32] B. Morandi, S. Nanz, and B. Meyer. Record-replay debugging for concurrent scoop programs. Technical Report arXiv:1111.1170v1, ETHZ, Nov. 2011.

[33] C. Neusius. Synchronizing actions. In *5th ECOOP*, volume 512 of *LNCS*, pages 118–132. Springer, 1991.

[34] M. Papathomas. *Concurrency in Object-Oriented Programming Languages*, pages 31–68. Prentice Hall, 1995.

[35] M. Pradel and T. R. Gross. Automatic testing of sequential and concurrent substitutability. In *35th ICSE*, pages 282–291. IEEE / ACM, 2013.

[36] A. Ricci, M. Viroli, and G. Piancastelli. simpA: an agent-oriented approach for programming concurrent applications on top of Java. *Sci. Comput. Prog.*, 76(1):37–62, 2011.

[37] A. E. Santosa, Y. Kawata, and M. Maekawa. A solution to inheritance anomaly based on reusable behavior definitions. In *10th PDCS*, pages 586–589. IASTED/ACTA Press, 1998.

[38] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA '11*, pages 51–64, 2011.

[39] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

[40] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[41] L. Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *TENCON '94*, volume 2, pages 541–545. IEEE, 1995.

[42] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In Meyrowitz [28], pages 103–112.

A. Proofs

A.1 Auxiliary Propositions

Here we state two useful properties on the NFA operations.

Proposition 8. *For any NFA A and $\Delta \subseteq \Sigma$ such that $\Delta \rhd A$ is defined, $\mathcal{L}(A) \subseteq \mathcal{L}(\Delta \rhd A)$.*

Proof. The empty sequence $\langle\rangle \in \mathcal{L}(A)$ for any NFA A, and therefore $\langle\rangle \in \mathcal{L}(A)$ and $\langle\rangle \in \mathcal{L}(\Delta \rhd A)$. When $A = (S, s_0, \delta, F)$ and $w = \langle m_1, \ldots, m_k\rangle \in \mathcal{L}(A)$, there is a sequence of states $s_0, \ldots, s_k$ where $s_i \in S$ for $0 \leq i \leq k$, $s_k \in F$, and $(s_i, m_{i+1}, s_{i+1}) \in \delta$ for $0 \leq i < k$. When $\Delta \rhd A = (S, s_0, \delta', F)$, then for any $(s, m, s') \in \delta$, it also holds that $(s, m, s') \in \delta'$. Therefore, $w \in \mathcal{L}(\Delta \rhd A)$ if $w \in \mathcal{L}(A)$, and hence $\mathcal{L}(A) \subseteq \mathcal{L}(\Delta \rhd A)$. □

Proposition 9. *When $A_1 \times A_2$ is defined, then $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.*

Proof. In this proof we assume that $A_1 = (S_1, t_0, \delta_1, F_1), A_2 = (S_2, u_0, \delta_2, F_2)$, and $A_1 \times A_2 = (S, s_0, \delta, F)$.

First we prove that $\mathcal{L}(A_1 \times A_2) \subseteq \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. For this, we show that for any w such that $w \in \mathcal{L}(A_1 \times A_2)$, it is also true that $w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. In case $w = \langle\rangle$, the property is trivially true as $w \in \mathcal{L}(A_1)$, $w \in \mathcal{L}(A_2)$, and $w \in \mathcal{L}(A_1 \times A_2)$. Next we consider the case when $w \neq \langle\rangle$. When $w \in \mathcal{L}(A_1 \times A_2)$, then $w = \langle m_1, \ldots, m_k\rangle$ with $(s_i, m_{i+1}, s_{i+1}) \in \delta$ for any $0 \leq i < k$ and for some $s_0, \ldots, s_k \in S$ with $s_k \in F$. By the definition of tensor product, there are t_i, $t_{i+1} \in S_1$ and u_i, $u_{i+1} \in S_2$ such that $(t_i, m_{i+1}, t_{i+1}) \in \delta_1$ and $(u_i, m_{i+1}, u_{i+1}) \in \mathit{delta}_2$ for any $0 \leq i < k$, and with $t_k \in F_1$ and $u_k \in F_2$. Therefore necessarily $w \in \mathcal{L}(A_1)$ and $w \in \mathcal{L}(A_2)$.

Next we prove that $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \subseteq \mathcal{L}(A_1 \times A_2)$. The case when $w = \langle\rangle$ is trivial as by definition w belongs to $\mathcal{L}(A)$ for any NFA A. Now let us assume that $w = \langle m_1, \ldots, m_k\rangle \neq \langle\rangle$. When $w \in \mathcal{L}(A_1)$ and $w \in \mathcal{L}(A_2)$ then there are states $t_0, \ldots, t_k \in S_1$ and $u_0, \ldots, u_k \in S_2$ with $(t_i, m_{i+1}, t_{i+1}) \in \delta_1$ and $(u_i, m_{i+1}, u_{i+1}) \in \delta_2$ for $0 \leq i < k$, and with $t_k \in F_1$ and $u_k \in F_2$. Therefore, there are states $s_0, \ldots, s_k \in S$ such that $(s_i, m_{i+1}, s_{i+1}) \in \delta$ and $s_k \in F$ when each s_i is $\{t_i, u_i\}$ for $1 \leq i \leq k$. Hence, $w \in \mathcal{L}(A_1 \times A_2)$. □

A.2 Proof of Proposition 1

We show that when $\tau \in \mathit{Types}$, then $\sqcap_{beh}\sigma = \cap_{p \in \sigma} \mathit{beh}(p) = \mathcal{L}(A)$ for any $\sigma \subseteq \tau$ and for some NFA A. We induct on $|\sigma|$, the size of σ. In the base case when $|\sigma| = 0$, $\cap_{p \in \sigma} \mathit{beh}(p) = \Sigma^*$. In this case, $A = (\{s_0\}, s_0, \{(s_0, m, s_0) \mid m \in \Sigma\}, \{s_0\})$.

For the inductive case, when $|\sigma| = k$, we inductively assume that there is a NFA $A = (S, s_0, \delta, F)$ such that $\sqcap_{beh}\sigma = \mathcal{L}(A)$. Now we extend σ into $\sigma' = \sigma \cup \{p\}$ using some object $p \in \mathit{Instance}$ such that $|\sigma'| = k + 1$. Here we show a construction of a NFA A' such that $\sqcap_{beh}\sigma' = \mathcal{L}(A')$. By our assumption on Page 4, $\mathit{beh}(p) = \mathcal{L}(A_p)$ for some NFA A_p. Therefore, $\sqcap_{beh}\sigma' = \mathcal{L}(A) \cap \mathcal{L}(A_p)$. From Proposition 9 we know that if there is a NFA A'' such that 1) $\mathcal{L}(A_p) = \mathcal{L}(A'')$ and 2) $A \times A''$ is defined, then the NFA A' that satisfies the above requirement is $A \times A''$. We now construct such NFA $A'' = (S'', s_0'', \delta'', F'')$ from $A_p = (S_p, s_p, \delta_p, A_p)$ using a bijection $\eta : S_p \to S''$ such that $S_p \cap S'' = \emptyset$, where $S'' = \{\eta(s) \mid s \in S_p\}$, $s_0'' = \eta(s_p)$, $(\eta(s), m, \eta(s')) \in \delta''$ iff $(s, m, s') \in \delta_p$, and $F'' = \{\eta(s) \mid s \in F_p\}$. Notice that in this way, 1) $\mathcal{L}(A_p) = \mathcal{L}(A'')$ and 2) $A \times A''$ is defined.

A.3 Proof of Lemma 2

(If case.) We assume that $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$ holds. When $p \in \sigma$, since $\sqcap_{beh}\sigma \sqsubseteq \mathit{beh}(p)$, then $\sqcap_{beh}\tau \sqsubseteq \mathit{beh}(p)$, and therefore $p \in \tau$. Hence, σ is a subtype of τ.

(Only if case.) From Proposition 1, there are NFA $A_\tau = (S_\tau, s_\tau, \delta_\tau, F_\tau)$ and $A_\sigma = (S_\sigma, s_\sigma, \delta_\sigma, F_\sigma)$ such that $\sqcap_{beh}\tau = \mathcal{L}(A_\tau)$ and $\sqcap_{beh}\sigma = \mathcal{L}(A_\sigma)$.

For $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\sigma$ to hold, according to Definition 1 we need to establish two:

1. $\mathcal{L}(A_\tau) \subseteq \mathcal{L}(A_\sigma)$, and
2. there is $\Delta \subseteq \Sigma$ and NFA A' s.t. $\mathcal{L}(A_\sigma) = \mathcal{L}((\Delta \rhd A_\tau) \times A')$

We note that by definition σ is a subtype of τ iff $\sigma \subseteq \tau$, and therefore $\sqcap_{beh}\tau \subseteq \sqcap_{beh}\sigma$ and therefore first condition is trivially satisfied.

For the second condition, by the premise that σ is a subtype of τ, we can say that $\sigma \subseteq \tau$ from our definition of subtyping. Since $\tau, \sigma \in \mathit{Types}$, for any $p \in \sigma$, by Definition 1, $\sqcap_{beh}\sigma \sqsubseteq \mathit{beh}(p)$. Since $\sigma \subseteq \tau$, it is also the case that $p \in \tau$, and therefore by the same Definition 1, $\sqcap_{beh}\tau \sqsubseteq \mathit{beh}(p)$.

Again by Definition 1, further there are $\Delta_2, \Delta_4 \subseteq \Sigma$ and NFA $A_2 = (S_2, s_2, \delta_2, F_2)$ and $A_4 = (S_4, s_4, \delta_4, F_4)$ such that $\mathit{beh}(p) = \mathcal{L}((\Delta_2 \rhd A_\tau) \times A_2) = \mathcal{L}((\Delta_4 \rhd A_\sigma) \times A_4)$. We assume that $A_2 = A_4$ and that $\Delta_2 \rhd A_\tau = \Delta_4 \rhd A_\sigma$ with both the lhs $\Delta_2 \rhd A_\tau$ and the rhs $\Delta_4 \rhd A_\sigma$ defined. We further define $\Delta_\tau = \{m \mid (s, m, s') \in \delta_\tau\}$ and $\Delta_\sigma = \{m \mid (s, m, s') \in \delta_\sigma\}$, and since both the $\Delta_2 \rhd A_\tau$ and $\Delta_2 \rhd A_\tau$ interface extensions are defined, $\Delta_\tau \cup \Delta_2 = \Delta_\sigma \cup \Delta_4$ with $\Delta_\tau \cap \Delta_2 = \Delta_\sigma \cap \Delta_4 = \emptyset$. Since $\mathcal{L}(A_\tau) \subseteq \mathcal{L}(A_\sigma)$, necessarily $\Delta_\tau \subseteq \Delta_\sigma$, and therefore there is Δ_5 such that $\Delta_4 \cup \Delta_5 = \Delta_2$ and $\Delta_4 \cap \Delta_5 = \emptyset$. Hence, $A_\sigma = \Delta_5 \rhd A_\tau$. We define the NFA A_{any} to be $(\{s_{any}\}, s_{any}, \delta_{any}, \{s_{any}\})$ such that $s_{any} \notin S_\tau$ and for any $m \in \Sigma$, $(s_{any}, m, s_{any}) \in \delta_{any}$. Therefore, $\mathcal{L}(A_{any}) = \Sigma^*$. Moreover, $(\Delta_5 \rhd A_\tau) \times A_{any}$ is defined and $\mathcal{L}(\Delta_5 \rhd A_\tau) = \mathcal{L}((\Delta_5 \rhd A_\tau) \times A_{any})$. Hence, since $\mathcal{L}(\Delta_5 \rhd A_\tau) = \mathcal{L}(A_\sigma)$, therefore $\mathcal{L}((\Delta_5 \rhd A_\tau) \times A_{any}) = \mathcal{L}(A_\sigma)$, and we satisfy the second condition.

A.4 Proof of Lemma 6

From Proposition 9, $\mathcal{L}((\Delta \rhd A_1) \times A_2) = \mathcal{L}(\Delta \rhd A_1) \cap \mathcal{L}(A_2)$. Therefore, $\mathcal{L}(A_1) \subseteq \mathcal{L}((\Delta \rhd A_1) \times A_2)$ iff $\mathcal{L}(A_1) \subseteq \mathcal{L}(\Delta \rhd A_1)$ and $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$. From Proposition 8, $\mathcal{L}(A_1) \subseteq \mathcal{L}(\Delta \rhd A_1)$. Therefore, $\mathcal{L}(A_1) \subseteq \mathcal{L}((\Delta \rhd A_1) \times A_2)$ iff $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$.

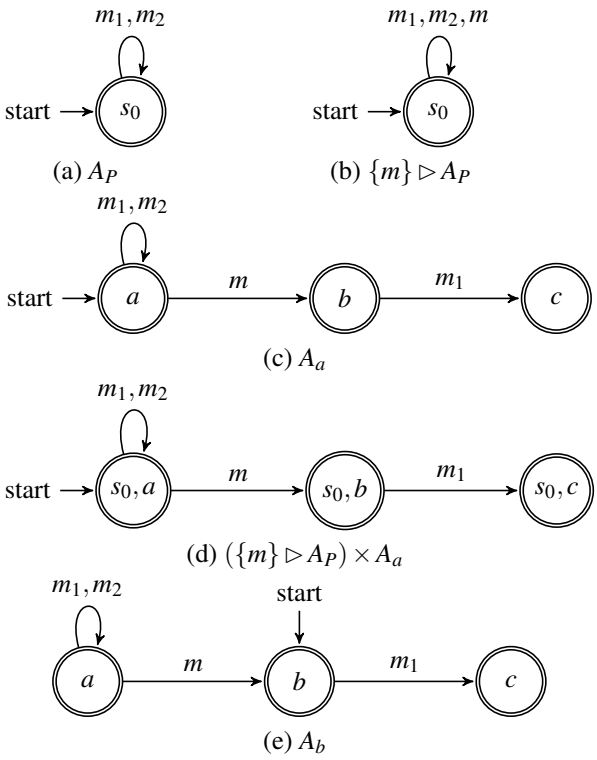


Figure 9. NFA Explaining Crnogorac et al.'s Result

B. Crnogorac et al.'s Proof Re-Explanation

Crnogorac et al.'s formal system in [6] employs a variety of typing domains. Here we re-explain their results uniformly using our domain *Types* of Definition 1. The following corresponds to Theorem 1 in [6]:

Theorem 10. *Consider an inheritance mechanism* $(Class, \dashrightarrow)$. *If* $(Class, \dashrightarrow)$ *is behavior-preserving with respect to Types then it is anomaly-free with respect to Types.*

Proof. The proof proceeds by contraposition: assuming an anomaly-free $(Class, \dashrightarrow)$, we show that such mechanism cannot be behavior preserving. We first assume a class P where $\sqcap_{beh} imp(P) = \{m_1, m_2\}^*$. This is the language accepted by the automaton A_P in Figure 9 (a). Now we assume a class Q such that $\sqcap_{beh} imp(Q)$ is the language accepted by the automaton in Figure 9 (d). This automaton is obtainable by constructing $(\{m\} \rhd A_P) \times A_a$ with $\{m\} \rhd A_P$ and A_a of Figures 9 (b) and (c), respectively. Here $\sqcap_{beh} imp(P) \sqsubseteq \sqcap_{beh} imp(Q)$, therefore by Lemma 2, $imp(Q) \subseteq imp(P)$. **A:** From the definition of anomaly freedom (Definition 3), there is a class R such that $P \dashrightarrow_I R$ with $imp(R) = imp(Q)$.

At this point the proof employs a *critical property:*

> Whenever $P \dashrightarrow_I Q$ and $z \in \sqcap_{beh} imp(Q)$ then it is possible to construct P', Q' such that $P' \dashrightarrow_I Q', \sqcap_{beh} imp(Q') = \{v \mid z.v \in \sqcap_{beh} imp(Q)\}$, and for some $w \in \sqcap_{beh} imp(P)$, $\sqcap_{beh} imp(P') = \{v \mid w.v \in \sqcap_{beh} imp(P)\}$.

The property entails that if by incremental inheritance $P \dashrightarrow_I Q$, and if Q' is obtainable from Q by setting the initial state differently to a reachable state (the state reached after the execution of a trace, which in the statement of the property is z) of Q, then there is a reachable state of P (after executing w) such that P' is obtainable from P by setting the initial state differently to that reachable state, such that $P' \dashrightarrow_I Q'$. Crnogorac et al. claimed that, "This assumption holds for all COOP languages we are aware of" [6].

Returning to point **A**, here we have that $P \dashrightarrow_I R$. Now by the critical property we can construct classes P' and R' such that $P' \dashrightarrow_I R'$ with $\sqcap_{beh} imp(R')$ is the language accepted by the automaton A_b of Figure 9 (e) which is $\{\langle\rangle, \langle m_1 \rangle\}$, as the initial state b of Figure 9 (e) is the result of executing $\langle m \rangle$ on the automaton of Figure 9 (d), which represents the behavior of R, i.e., $\sqcap_{beh} imp(R)$. Here, $\sqcap_{beh} imp(R') = \{v \mid \langle m \rangle . v \in \sqcap_{beh} imp(R)\}$. Again by the critical property the initial state of P' would be the result of executing some trace on A_P, which always ends in the same state, hence, $\sqcap_{beh} imp(P') = \sqcap_{beh} imp(P) = \mathcal{L}(A_P)$. Therefore, although $P' \dashrightarrow_I R'$ holds, $\sqcap_{beh} imp(P') \not\sqsubseteq \sqcap_{beh} imp(R')$, and therefore by Lemma 2, $imp(R') \not\subseteq imp(P')$ and behavior preservation (Definition 2) is violated. □

From a language designer's point of view, the above proof is incomplete due to the use of the critical property. A question remains: Violating the critical property, is it possible to design a COOP language where both anomaly freedom and behavior preservation hold? We demonstrate that although this is logically possible, it is intractable to ensure behavior preservation in the presence of anomaly freedom.