22

Inheritance Anomaly - A Formal Treatment

Lobel Crnogorac[†], *Anand S. Rao*[‡] *and Kotagiri Ramamohanarao*[†]
[†]*Dept. of Computer Science, The University of Melbourne, Parkville Vic. 3052, Australia, E-mail: {lobel,rao}@cs.mu.oz.au*
[‡]*Australian Artificial Intelligence Institute, Level 6, 171 La Trobe Street, Melbourne Vic. 3000, Australia, E-mail: anand@aaii.oz.au*

Abstract

Inheritance is one of the key concepts in object-oriented programming (OOP). However, the usefulness of inheritance in concurrent OOP is greatly reduced by the inheritance anomalies. These anomalies have been subjected to intense research, but they are still only vaguely defined and often misunderstood. In this paper we show that concurrency is not the real cause of inheritance anomalies. We formally define the inheritance anomaly as a relationship between inheritance mechanisms and behavioural hierarchies. Our framework can be used to analyse the occurrence of inheritance anomalies in many different paradigms. A formal definition of the problem and a clear exposition of its causes are pre-requisites for a successful integration of inheritance and concurrency.

Keywords

inheritance, behavioural subtyping, concurrent object-oriented programming

1 INTRODUCTION

Inheritance is one of the key concepts in object-oriented programming (OOP). It is a widely used methodology for code re-use in sequential object-oriented programming. In recent years, the concepts from OOP have been applied in a concurrent setting, leading to the emergence of concurrent object-oriented programming (COOP) [7, 11, 12]. In its full generality COOP paradigm allows inter-object concurrency (multiple objects existing concurrently) and intra-object concurrency (multiple threads inside an object). It was found that most OOP concepts (*e.g.*, encapsulation) could be naturally integrated into COOP. However, the integration of inheritance and COOP has not been

smooth. One of the main problems with inheritance in COOP is the *inheritance anomaly* [8, 10, 11, 12, 13, 15]. Inheritance anomaly arises when additional methods of a subclass cause undesirable re-definitions of the methods in the superclass. Instead of being able to incrementally add code in a subclass the programmer may be required to re-define some inherited code, thus the benefits of inheritance are lost.

Inheritance anomalies have been researched extensively, but they are still only vaguely defined and often misunderstood. There is a wealth of language proposals in the literature trying to solve the problem of anomalies, but almost no formal work has been done. The main practical progress has been made in the area of languages that do not allow intra-object concurrency. For comparison purposes the proposals were usually evaluated on a set of standard benchmark examples introduced by Matsuoka and Yonezawa [11]. The aim was to successfully avoid the anomaly in the benchmark examples.

Inheritance anomaly has never been formally defined in its full generality, although a particular type of anomaly, *state-partitioning*, has been formally investigated by Matsuoka *et. al.* [10]. We feel that a general formal treatment of the problem is needed in order to precisely define the inheritance anomaly and to formally compare the different proposals. Without a formal treatment we cannot be certain that the set of benchmark examples is exhaustive (there could be other undiscovered types of anomalies). The formal treatment should also analyse the causes of the inheritance anomaly. It is widely believed that interference between concurrency and inheritance is the cause of inheritance anomalies. However, the appearance of inheritance anomalies in other paradigms based on object-oriented concepts points to the need for a much more thorough examination of their causes. For example, inheritance anomaly in real-time specification languages [1] could not be caused by an interference between inheritance and concurrency because these languages are purely sequential. The introduction of inheritance into agent-oriented programming (AOP) [14] also leads to the appearance of inheritance anomalies [5]. The requirements we put on a framework for inheritance anomaly are:

- The framework must be general enough to allow a uniform analysis of inheritance anomaly for any existing inheritance mechanism in any existing object-oriented paradigm. For example, inheritance anomaly in truly concurrent languages (allowing intra-object concurrency [3]) with active objects seems to be a much harder problem than the more constrained case (only inter-object concurrency). The framework must be able to capture the general problem of inheritance anomaly in COOP.
- The framework must formally pinpoint the cause of the anomaly. These results should be used to predict appearances of anomalies in new paradigms, and explain why the anomaly doesn't arise in sequential OOP.
- Are the known types of anomalies the only possible anomalies? Are they caused by different reasons or are they just different manifestations of a sin-

gle conflict? The framework must be able to derive the complete taxonomy of the types of inheritance anomalies across different paradigms.
- By using the framework it should be possible to explore whether an ideal solution actually exists. Also, the framework must be able to formally explain and justify the choices made in the development of better inheritance mechanisms (*e.g.*, the need to separate inheritance of synchronisation code from inheritance of functionality code should be formally justified [12]).
- The framework must present a formal definition of inheritance anomaly.

We share the views in [2, 13] that inheritance should be an unconstrained, flexible tool for code re-use. Inheritance should maximise the amount of code that can be re-used when defining a new specification from an existing specification. *Subtyping* is concerned with the use of objects, and is usually based on method signatures. Whenever we require an object, a subtype of that object would do equally well. Inheritance and subtyping are distinct concepts [2], since inheritance is concerned with the internal structure of objects (code sharing), while subtyping is concerned with the external behaviour of objects (the way objects are used). The work on *behavioural subtyping* [2, 9] extends the concept of subtyping to more general notions of behaviour.

The key insight of this paper is the connection between the notions of inheritance and subtyping: the inheritance mechanism should be powerful enough to incrementally mimic the behavioural (subtype) hierarchy. Hence, inheritance should be able to reach any "sub-behaviour" of a given specification without any need for re-definitions. Inheritance anomaly is defined as the failure of the inheritance mechanism to incrementally mimic the behavioural hierarchy. The aim of this paper is to provide a formal definition of inheritance anomaly and to address most of the previously stated requirements.

After a brief overview of the problem in COOP (Section 2), we introduce the domain of syntactic specifications of objects (Section 3). An inheritance mechanism is defined as a transition relation on the set of syntactic specifications. Defining inheritance in terms of a transition relation means we can avoid giving a particular semantics to inheritance. Thus, we avoid unnecessarily constraining our framework. In Section 4 we introduce the concept of behavioural hierarchy, which can be viewed as a generalised notion of subtyping. The relationship between syntactic specifications and behaviours is given by an abstraction function which maps specifications of objects into actual behaviours. The formal definition of inheritance anomaly is based on the relationship between the behavioural hierarchy and the inheritance mechanism. An inheritance mechanism is anomaly-free with respect to a given behavioural hierarchy if it can mimic that hierarchy in an incremental fashion. Inheritance anomaly is highly language dependent. This dependence is encoded in the transition relation. The framework is then applied in the context of sequential and concurrent OO languages (Section 5). We show that our formal definition matches informal examples of anomalies given in literature. We give

fundamental results stating theoretical limitations of inheritance mechanisms in COOP. In particular, we show that an ideal solution does not exist.

The primary contribution of this paper is the use of the correspondence between an inheritance mechanism and a behavioural hierarchy to motivate a formal definition of the inheritance anomaly. Our framework can be used to analyse the occurrence of inheritance anomalies in many different paradigms. The analysis based on the formal definition provides a clearer understanding of the causes of the anomaly. We provide results that explain the recent directions of research into better inheritance mechanisms [3, 12] and show that an ideal solution does not exist. A formal definition of the problem and a clear exposition of its causes are the pre-requisites for a successful integration of inheritance into the existing object-oriented paradigms.

2 OVERVIEW OF INHERITANCE ANOMALY IN COOP

This section gives a brief overview of the problems caused by the inheritance anomaly in COOP. The examples used are due to Matsuoka and Yonezawa [11]. The anomaly results in the inability of COOP languages to inherit synchronisation code without re-definitions. Concurrent object-oriented programming languages have to provide facilities for expressing synchronisation constraints of objects. For example, the programmer needs to be able to express that adding an element into a full buffer or removing an element from an empty buffer is not allowed. Inheritance anomaly (in the context of COOP) is the conflict between concurrency and inheritance where extensive re-definitions of inherited methods are necessary in order to maintain the synchronisation constraints of concurrent objects. Matsuoka and Yonezawa [11] have distinguished three kinds of inheritance anomalies in COOP languages:

- *state-partitioning:*

Execution of a concurrent object can be thought of as a sequence of transitions between states. Each state is determined by the current values of all the state variables and the methods that are acceptable. The state-partitioning anomaly occurs when the addition of a new method further partitions the state-space. A new state is added to the set of states a class can be in. All the other states remain. The code changes in the superclass are caused by the difference in the number of states possible in the superclass and the subclass. The extra states possible in the subclass have to be accounted for in all the methods in the superclass. The classical example involves a language based on *accept sets* [7, 15]. The synchronisation scheme of these proposals uses explicit sets to determine which methods are acceptable at any time. The methods that are not currently acceptable are either rejected, or suspended and placed into a message queue. The keyword *become* denotes the explicit switching between the states. Consider the situation in Figure 1. Class *Buffer* implements a bounded buffer (storing at most MAX elements). It provides methods *put* and *get* which add and remove a single element from the buffer.

The synchronisation code, expressed by explicit accept sets, needs to describe the following constraints: "Method *put* is acceptable unless *Buffer* is full. Method *get* is acceptable unless *Buffer* is empty. Method *numOfElements* is always acceptable." We define a subclass of *Buffer*, *NewBuffer* with an additional method *get2* that removes two elements. Method *get2* can be used only if *NewBuffer* contains more than one element. This is an extra state that the object could be in. The anomaly appears when the *become* statements of *put* and *get* (as well as the most of the *behaviour* block) need to be re-defined to accommodate for the extra state.

```
class  Buffer{
int in=0, out=0;
behaviour:empty = {put(x).numOfElements}
          partial = {put(x).get.numOfElements}
          full = {get.numOfElements}
method   numOfElements
    code for numOfElements
method   put(x)
    code for put
    if (numOfElements==MAX) become full;
    else become partial;
method   get
    code for get
    if (numOfElements==0) become empty;
    else become partial;
}

class  NewBuffer: Buffer{
behaviour:/* the set empty is inherited cleanly */
          partial = {put(x).get.get2.numOfElements}
          full = {get.get2.numOfElements}
          one = {put(x).get.numOfElements}
method   get2
    code for get2
    if (numOfElements==0) become empty;
    else if (numOfElements==1) become one;
    else become partial;
method   put(x)
    code for put
    if (numOfElements==MAX) become full;
    else if (numOfElements==1) become one;
    else become partial;
method   get
    code for get
    if (numOfElements==0) become empty;
    else if (numOfElements==1) become one;
    else become partial;
}
```

Figure 1 State-partitioning

State-partitioning has been circumvented by proposals that employ method guards instead of accept sets [11]. A method is accepted if its guard evaluates to *true*, otherwise it is suspended (placed into a message queue) or rejected. The syntax is "**method** *method_signature when guard*" (Figure 2). Here, *NewBuffer* is constructed from *Buffer* by adding the statement "**method** *get2 when numOfElements > 1*" along with the code for *get2*. Unlike the situation in Figure 1, re-definitions of *put* and *get* are not necessary with method guards. The guards solve state-partitioning anomaly. However, method guards and accept sets can't prevent the next two types of anomalies.

- *state-modification:*

Additions of new methods to a class can introduce a finer-grained distinction for the set of states under which the methods can be invoked. Code re-definitions are caused by the need to account for this finer-grained distinction of states. The standard example involves adding a locking capability to the *Buffer* class. Methods *put* and *get* can be accepted only when the object is unlocked. The method *numOfElements* is not affected by the new locking operations. Method guards (Figure 2) are used to specify the conditions under which methods are accepted. State-modification arises from the need to add new variables to dis-

```
class  Buffer{
int in=0, out=0;
method   numOfElements /* always accepted */
    code for numOfElements
method   put(x) when (numOfElements < MAX)
    code for put
method   get when (numOfElements > 0)
    code for get
}
class  LockableBuffer: Buffer{
Bool locked = false;
method   lock{ locked=true;}     /* always accepted */
method   unlock{ locked=false;} /* always accepted */
method   put(x) when ( !locked &&
         (numOfElements < MAX))
    code for put
method   get when (!locked && (numOfElements > 0))
    code for get
}
```

Figure 2 State-modification

tinguish between states, *e.g.*, the variable *locked* in Figure 2. Methods *put* and *get* have to be re-defined to take *locked* into account.

- *history-only-sensitiveness*:

In COOP we often encounter situations that depend on history of an object. The need to re-define code arises because the methods in the superclass need to leave some trace of their execution for the future (usually, the methods are re-defined to update some new variables). The standard example involves extending the *Buffer* class with a new method *gget* which behaves exactly like *get*, except that it cannot be invoked immediately after an execution of *put*. To define this new class, *HistoryBuffer*, methods *put* and *get* are re-defined to use a new variable *after_put*. Thus, the benefits of inheritance are lost.

3 MODELLING INHERITANCE AS A TRANSITION RELATION

An inheritance mechanism defines the way a new class specification can be obtained by re-using code from an existing class specification. New services may be added, the inherited services re-defined or omitted. An inheritance mechanism of a language is usually given by defining the semantics of its inheritance operator [4]. We take a different approach. Here we formalise the inheritance mechanism of an arbitrary language as a transition relation on the set of syntactic specifications. A pair of syntactic specifications forms a transition if the second specification can be obtained from the first by using the inheritance rules. Expressing all inheritance mechanisms in terms of transition relations provides a uniform view of different languages. It avoids giving a particular semantics to inheritance, thus it does not constrain the formal framework to one paradigm. This is important since there is no clear agreement about the inheritance semantics in COOP or AOP (unlike in sequential OOP).

We capture a language with inheritance as a set of syntactic specifications that do not use inheritance, and a transition relation between them. Let *Spec* be the set of all possible syntactic specifications (without using inheritance) of classes in some language. In an OOP language, *Spec* is the countably infinite set of all classes that can be written without using the inheritance operator of the language. For example, the definition of class *Buffer* (Figure 2) is an element of the set *Spec* of the illustrated language. However, class *LockableBuffer* is not an element of *Spec* since it is defined by using the inheritance operator ":". We assume that an element $p \in Spec$ is a function $p : Keys \rightarrow Exp_\perp$ where *Keys* is a countably infinite set of names. The set $Exp_\perp$ is the set of expressions that can be written in the language. It is a partial order (actually, a flat cpo) under the $\preceq$ ordering. In OOP $p \in Spec$ maps method signatures and variables (the keys) to their respective bodies (the expressions).

Definition 1 A *preordering* on a set D is a binary relation that is reflexive and transitive. A *partial ordering* is an antisymmetric preordering. Let $e, f \in Exp_\perp$. If $e \preceq f$ then either $e = \perp$ or $e = f$. We extend $\preceq$ to functions. ■

Definition 2 An *inheritance mechanism* is a pair $(Spec, \dashrightarrow)$ where $\dashrightarrow \subseteq Spec \times \Delta \times Spec$. An element of $\dashrightarrow$, (p, δ, q) is called a *transition* where $p, q \in Spec$ and $\delta \in \Delta$. Δ is the set of syntactic entities specifying the differences between p and q. We write $p \overset{\delta}{\dashrightarrow} q$ for $(p, \delta, q) \in \dashrightarrow$. Furthermore, $p \overset{\delta \in \Delta^*}{\dashrightarrow} q$ is used to denote the reflexive and transitive closure of $\dashrightarrow$ *i.e.*, a sequence of individual transitions. Overloading of the notion $p \overset{\delta}{\dashrightarrow} q$ is harmless. ■

The transition relation $\dashrightarrow$ is a set of triples (p, δ, q). Transitions specify how inheritance can be used to move from a syntactic specification p to a new specification q by specifying the differences (*e.g.*, new methods) in δ^*. Transitions may simulate re-definitions, deletions or omissions of components of syntactic specifications. Hence, very general inheritance mechanisms can be defined. The sets *Spec* and Δ are determined by the language that is being analysed. Since $q \in Spec$, Definition 2 assumes that everything that can be defined by means of inheritance can also be defined without it.

Example 1 Figure 2 shows a single transition of the inheritance mechanism employing method guards. Here, p corresponds to *Buffer* and δ corresponds to *LockableBuffer*. The syntactic specification q is not shown however. It would correspond to a *fully expanded* version of *LockableBuffer* with the method *numOfElements* and the variables *in* and *out* explicitly defined. Thus, in practice, a language specifies transitions by giving the modification δ from p. ■

Inheritance anomaly arises when instead of being able to incrementally add code in a subclass, the programmer is required to re-define some inherited code. In order to capture this property we need to formally make a distinction between "incrementally adding code" and "re-defining code".

Definition 3 Transition $p \overset{\delta}{\dashrightarrow} q$ is *incremental* if* $p \preceq q$. The subset of all incremental transitions is denoted $\dashrightarrow_I \subseteq \dashrightarrow$. ■

An incremental modification means that new functionality is added to a syntactic specification without re-defining any of the existing services. Hence, if $p \overset{\delta}{\dashrightarrow}_I q$ then whenever p maps a key to a defined expression, q maps the same key to the same expression also. Furthermore, if p maps a key to $\perp$, then q maps the same key to $\perp$ or to a defined expression. The relation $\dashrightarrow_I$ corresponds to the elegant, incremental use of inheritance.

---

*The definition of transition relation can be extended to model multiple inheritance. We focus on single inheritance in this paper.

*In most contexts this is actually a double implication. However, in order to model inheritance in languages that use *self* and *super* [4] the definition of incremental transitions needs to be modified to include some additional transitions. Basically, re-instantiation of *self* and *super* is modelled by fully expanding all references to them and by including the transition in $\dashrightarrow_I$. Details are beyond the scope of this paper.

4 THE DEFINITION OF INHERITANCE ANOMALY

This section presents the formal definition of inheritance anomaly. As discussed earlier in Section 1 an anomaly-free inheritance mechanism needs to be powerful enough to simulate the behavioural hierarchy in an incremental fashion. If a subclass **preserves and extends the behaviour** of its parent then we would like to re-use the parent's specification as a whole. The subclass would be defined incrementally from the original specification without any need for re-definitions of the parent's methods. Alternatively, if a subclass **modifies the behaviour** of its parent (behaviour of the parent is not preserved) all we can expect to re-use are parts of the parent's specification (some methods are inherited cleanly while other methods are re-implemented).

We now introduce the notion of *behavioural hierarchy.* Behavioural hierarchy is a partially ordered set $(Beh, \leq)$. The set *Beh* is the set of all possible behaviours, ordered by a partial ordering $\leq$ which defines the meaning of behaviour preservation/extension. That is, if $\theta, \phi \in Beh$ and $\theta \leq \phi$ then ϕ somehow preserves and extends θ. Consider the relationship between the sets *Spec* and *Beh.* The behaviour of an object specification is determined by the language semantics. The semantics of the chosen language defines an *abstraction* function α, which maps the set *Spec* into the set *Beh.* The abstraction function maps a syntactic specification into its behaviour. Hence, $\alpha : Spec \rightarrow Beh$ is a function mapping a single element of *Spec* into a single element of *Beh.* Syntactic specification p *implements* the behaviour θ if $\theta \equiv \alpha(p)$ (where $\equiv$ is induced by the partial order $\leq$). In general α is not injective since different syntactic specifications may be mapped to the same behaviour. Intuitively, we can implement a required behaviour in infinitely many ways, *e.g.*, by changing the names of variables. In general, α is not surjective ($\alpha(Spec) \subseteq Beh$, where $\alpha(Spec)$ is the image of α over *Spec*) since *Spec* is countable, while *Beh* may be uncountable. The relationship between inheritance mechanism and behavioural hierarchy is the basis for the following definition.

Definition 4 Let $(Spec, \dashrightarrow)$ be an inheritance mechanism, with a behavioural hierarchy $(Beh, \leq)$ and an abstraction function α. Let $p, q \in Spec$ and $\theta \in Beh$. Define $G_p = \{q : p \overset{\delta \in \Delta^*}{\dashrightarrow} q\}$. Let $I_p = \{q : p \overset{\delta \in \Delta^*}{\dashrightarrow}_I q\}$, $B_p = \{q : \alpha(p) \leq \alpha(q)\}$. Finally, let $S_p = \{\theta : \alpha(p) \leq \theta\}$. ■

Consider Figure 3. For each syntactic specification p we define sets G_p, I_p, B_p and S_p. The set G_p is the set of all syntactic specifications that can be obtained from p by repeated applications of the inheritance mechanism. Note that $G_p \subseteq Spec$, but commonly, $G_p = Spec$ since in many inheritance mechanisms we can obtain any given specification from p by repeated re-definitions, deletions and additions. The set I_p is a subset of G_p which allows only incremental transitions from p. The set of all *syntactic specifications* which preserve and extend the behaviour of p is denoted by B_p. In

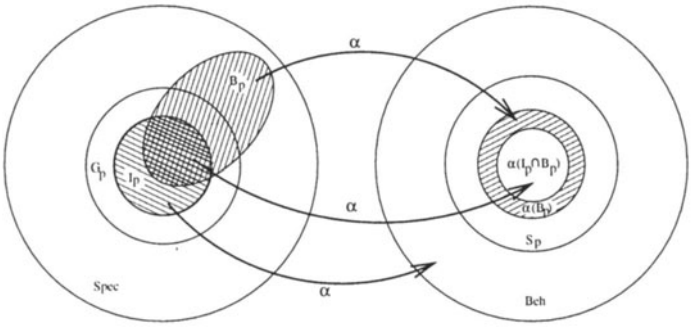

Figure 3 The definition of inheritance anomaly

general, I_p is not a subset of B_p because some incremental modifications can modify behaviour. This depends on the particular context, determined by α and $(Beh, \leq)$. For example, in AOP under a "natural" definition of agent behaviour it is possible to modify the behaviour of an agent by simple, incremental additions of new plans. The intersection $(I_p \cap B_p)$ is the set of all specifications which preserve and extend the behaviour of p and which can be incrementally (without re-definitions) obtained from p. The set S_p is the set of all possible *behaviours* that preserve and extend the behaviour of p. As shown in Figure 3, $S_p \subseteq Beh$. In general α may not be surjective, therefore the set B_p maps to $\alpha(B_p)$ which is a subset of S_p. In general $\alpha(I_p \cap B_p) \subseteq \alpha(B_p) \subseteq S_p$.

Definition 5 An inheritance mechanism $(Spec, \dashrightarrow)$ is *anomaly-free* with respect to $(Beh, \leq)$ iff $\forall p \in Spec$, $\alpha(I_p \cap B_p) = \alpha(B_p)$. ■

Intuitively, consider the following scenario: a) The programmer has defined a specification p which implements the behaviour $\alpha(p)$; b) The programmer envisions a specialisation of $\alpha(p)$ which preserves and extends this behaviour; c) If this specialisation can be implemented in the language by some specification r then the programmer must be able to incrementally obtain a specification q from p, such that q also implements the required specialisation. Proposition 1 shows that this scenario is a consequence of Definition 5.

Proposition 1 *Let $p, r \in Spec$. If the inheritance mechanism $(Spec, \dashrightarrow)$ is anomaly-free with respect to $(Beh, \leq)$ and if $\alpha(p) \leq \alpha(r)$ then $\exists \delta \in \Delta^*, q \in Spec$ such that $p \overset{\delta}{\dashrightarrow}_I q$ and $\alpha(r) \equiv \alpha(q)$, where $\equiv$ is induced by $(Beh, \leq)$.*
Proof: *Since $\alpha(p) \leq \alpha(r)$ we have $r \in B_p$. Since $\alpha(I_p \cap B_p) = \alpha(B_p)$ by assumption, we can find $q \in I_p \cap B_p$ such that $\alpha(q) \equiv \alpha(r)$.* ■

The concept of inheritance anomaly is defined with respect to a given behavioural hierarchy. An inheritance mechanism may be anomaly-free with

respect to one hierarchy, while it may have anomalies with respect to another hierarchy. This observation leads to an explanation of the inconsistency in the current literature. Until now, the definition of behavioural hierarchy was only given informally by researchers, through examples. In the case of *HistoryBuffer* (Section 2) some researchers have claimed that it illustrates an anomaly [11, 12], while others have claimed that it actually *modifies* the behaviour of *Buffer* [13] and therefore should not be considered an anomaly. This inconsistency arises from different assumptions about the behavioural hierarchy. A formal definition of behaviour hierarchy is needed to unambiguously define the problem. The dependence of the inheritance anomaly on the chosen behavioural hierarchy gives rise to a new view of the most essential cause of the anomaly. The inheritance anomaly arises because the inheritance mechanisms suitable for one behavioural hierarchy may not be suitable for a different hierarchy. Of course, in the context of COOP it is still correct to view the anomaly as being caused by an interference between inheritance and concurrency. However, this view does not generalise well. For example, inheritance anomalies have been discovered in sequential real-time specification languages [1] (behaviour hierarchy would be based on temporal information).

5 EXAMPLE ANALYSIS

This section presents an example analysis of inheritance mechanisms in sequential and concurrent OOP. We examine two different behavioural hierarchies, which are based on the externally observable behaviour of objects. The object model that we adopt views objects as encapsulated message acceptors. The only way to communicate with such an object is by sending it messages. Thus, the externally observable behaviour of an object is defined by the way it reacts to messages. This behaviour is independent of the implementational details. The objects are assumed to be executing at most one service at a time. Thus, we do not deal with intra-object concurrency here. The formal definition of inheritance anomaly is illustrated by applying it to the examples of Section 2. Finally, we prove a general result which states the theoretical limitations in the search for anomaly-free inheritance mechanisms.

In the context of our chosen object model we define behaviour of an object to be the set of all possible sequences of messages that the object can accept. We use the concept of *traces* [6]. Suppose that an external observer notes down the message acceptances. A *trace* is a finite sequence of message acceptances by an object. The set of all such finite sequences of message acceptances of an object is called the *traces* of the object and is denoted by *traces*(*object*). The set $Messg \subseteq Keys$ is the set of all keys which can be sent in messages and which are used to identify services. Thus, *Messg* is the set of all symbols that can appear in traces (*i.e.*, the *alphabet*), and the set of all possible traces is denoted $Messg^*$. An element of *Keys* is simply a method name or private variable name. Objects enforce encapsulation by using keys in $Keys - Messg$

for their private services. The set of services (with their keys belonging to *Messg*) corresponds to the public interface of an object. A message can be accepted only if it matches a key of one of the services offered by the object. However, a message that matches one of the services may sometimes be not accepted. This case arises in COOP where messages are accepted according to the current state of the object. For example, a *get* message sent to a *Buffer* object may be accepted (if the object is non-empty) or not accepted (if the object is empty, in which case the message may be rejected or suspended).

Definition 6 Behaviour of an object O is $traces(O)$. The set $\mathcal{P}(Messg^*)$ is the set of all subsets of $Messg^*$, *i.e.*, the set of all possible behaviours (hence, $Beh = \mathcal{P}(Messg^*)$ in this context). Behaviour $\theta \in \mathcal{P}(Messg^*)$ can also be viewed as a language over the alphabet *Messg*. Let $Reg \subset \mathcal{P}(Messg^*)$ be the set of all regular languages over *Messg*, that is, all languages (behaviours) that can be accepted by finite state machines. We use some standard operations on traces. Let $\theta \in \mathcal{P}(Messg^*)$ and let t be a trace in θ. The *restriction*, $t \restriction D$ denotes the trace t when restricted to symbols in the set D. The *length* of trace t is denoted $\#t$. *Catenation* constructs a trace from a pair of traces s and t by putting them together in that order. The result is denoted $s\hat{}t$. *Head* of a trace t gives the first symbol in t and is denoted t_0. The symbol m *occurs* in t iff $\#(t \restriction \{m\}) > 0$. Similarly, m *occurs* in θ iff $\exists t \in \theta \mid m$ *occurs* in t. ■

Example 2 $\langle put, put, get\rangle$ is a trace of the behaviour of bounded *Buffer* from Figure 1 (assuming MAX is large), whereas $\langle put, get, get\rangle$ is not. $\alpha(Buffer) = \{t \in Messg^* \mid \forall s, v \in Messg^*, t = s\hat{}v \implies 0 \leq (\#(s \restriction \{put\}) - \#(s \restriction \{get\})) \leq MAX\}$. In other words, a trace of *Buffer* must have at least as many occurrences of *put* as it has of *get*, but the difference must be at most MAX. Additionally, every initial segment of this trace (*i.e.*, every prefix) must have the same property (*e.g.*, to disallow $\langle put, get, get, put, put\rangle$). It follows that every non-empty trace of *Buffer* starts with *put* (consider the initial segment of length 1) since *Buffer* starts off empty. The empty trace, $\langle\rangle$, is a trace of *Buffer*. To illustrate the operations on traces, we have: $\langle put, put, get\rangle\hat{}\langle put, get\rangle = \langle put, put, get, put, get\rangle$. $\#\langle put, put, get\rangle = 3$ and $\langle put, get, get\rangle_0 = put$. ■

Behaviours are ordered by defining the relation $\leq$. Behaviour ϕ *preserves/extends the behaviour* θ if $\theta \leq \phi$. Intuitively, ϕ can behave like θ (can preserve θ), but it can also exhibit some additional behaviour (ϕ extends the behaviour θ). The first behavioural ordering (denoted $\leq_1$) is given by:

Definition 7 Let $\theta, \phi \in \mathcal{P}(Messg^*)$. Then, $\theta \leq_1 \phi$ iff $\forall m \in Messg$, m *occurs* in $\theta \implies m$ *occurs* in ϕ. Also, $\theta \equiv_1 \phi$ iff $\theta \leq_1 \phi$ and $\phi \leq_1 \theta$. ■

Example 3 $\{\langle put, get\rangle, \langle put, put\rangle\} \equiv_1 \{\langle put\rangle, \langle get\rangle\}$ because the symbols *put* and *get* appear on both sides. Also, "the behaviour of *Buffer*" $\leq_1$ "the behaviour of *NewBuffer*" since *NewBuffer* may accept an extra message *get2*. ■

It can be checked that $\leq_1$ is a partial order, and that $\equiv_1$ is an equivalence relation on $\mathcal{P}(Messg^*)$. Definition 7 states that $\theta \leq_1 \phi$ if the traces in ϕ contain more distinct symbols than the traces in θ. Hence, ϕ offers a larger set of services than θ. Behavioural equivalence $\equiv_1$ states that two behaviours are equivalent if they offer the same set of services (*i.e.* the same public interface). Definition 7 is equivalent to simple subtyping without covariant/contravariant rules [2] since messages contain only service names, and parameters are ignored (this can be extended). Sometimes, we need to distinguish between behaviours based on the actual sequencing of accepted messages. A stricter notion of behaviour preservation/extension, denoted by $\leq_2$, is given by:

Definition 8 Let $\theta, \phi \in \mathcal{P}(Messg^*)$ and consider $s, t, v \in Messg^*$. Then, $\theta \leq_2 \phi$ iff $\theta \subseteq \phi$ and $\forall s \in \phi, s = t\hat{}v$ for some $t \in \theta$ and for some v (possibly empty) such that the symbol v_0 (if it exists) never occurs in a trace of θ. The relation $\equiv_2$ is defined as in Definition 7. It follows that $\theta \equiv_2 \phi$ iff $\theta = \phi$. ■

Again, we can check that $\leq_2$ and $\equiv_2$ define a partial order and an equivalence relation on $\mathcal{P}(Messg^*)$. Definition 8 states that two objects display equivalent behaviour if they can engage in exactly the same sequences of message acceptances (their traces must be identical). Furthermore, ϕ preserves/extends the behaviour θ if ϕ can engage in all the sequences that θ can engage in, and it may also engage in some additional sequences. However, such an additional trace of ϕ must start with a trace from θ until a new message (that never occurs in θ) is accepted by ϕ. Thus, ϕ and θ are identical until ϕ accepts a new message, after which ϕ produces some additional functionality.

Example 4 The behaviour of *LockableBuffer* (Figure 2) preserves/extends the behaviour of *Buffer* because it contains the same traces as *Buffer* (if the observer is not sending *lock* messages), but it also contains additional traces, all of which start with some trace from *Buffer*. For instance, the trace $\langle put, put, lock, numOfElements, unlock, get\rangle$ is such an additional trace which starts with the trace $\langle put, put\rangle$ from *Buffer*. Similarly, the trace $\langle lock, unlock, put\rangle$ is a trace of *LockableBuffer* which starts with the empty trace from *Buffer*. Hence, "the behaviour of *Buffer*" $\leq_2$ "the behaviour of *LockableBuffer*". Under Definition 8, *HistoryBuffer* (Section 2) also preserves/extends the behaviour of *Buffer*. For instance, it introduces new traces $\langle put, put, get, gget, put\rangle, \langle put, put, put, get, gget\rangle, \langle put, put, put, get, gget, gget\rangle$. Again, "the behaviour of *Buffer*" $\leq_2$ "the behaviour of *HistoryBuffer*". The behaviours of *HistoryBuffer* and *LockableBuffer* are incomparable. ■

Definitions 7 and 8 give two different versions of behaviour preservation/extension. By using these definitions we can explain the set of standard informal examples that have been used as the "definition of inheritance anomaly" in literature. Different definitions of behaviour and behavioural hierarchy would be used to analyse inheritance mechanisms in other paradigms (*e.g.*, AOP, real-time specification, actor-based, COOP with internal concurrency). Note that we have not distinguished between the behaviour of classes and the behaviour of instances of classes in our definition of behaviour. This distinction is not necessary for the simple examples of Section 2, but it should be incorporated into a more thorough analysis.

Example 5 Consider Figure 1. We have, "the behaviour of *Buffer*" $\leq_2$ "the behaviour of *NewBuffer*". However, there is no incremental transition from the given specification of *Buffer* to any specification of *NewBuffer*. Hence, this inheritance mechanism is not anomaly-free with respect to $(\mathcal{P}(Messg^*), \leq_2)$. Similar arguments can be formulated for other examples in Section 2. Thus, the formal definition matches the informal examples. ■

We present the results of our example analysis and discuss their implications.

Theorem 1 *Consider a typical sequential object-oriented language with a simple inheritance mechanism* $(Spec, \dashrightarrow)$ *where* $p \dashrightarrow q$ *iff* q *has additional methods, or* q *has re-defined some methods from* p. $(Spec, \dashrightarrow)$ *is anomaly-free with respect to both* $(\mathcal{P}(Messg^*), \leq_1)$ *and* $(\mathcal{P}(Messg^*), \leq_2)$.
Proof: *Firstly note that* $\forall p \in Spec, \alpha(p) = \{m \in Messg \mid p(m) \neq \perp\}^*$. *If* $\alpha(p) \leq_1 \theta$ *and if* θ *is of the above form then* $\exists q \in Spec$ *such that* $p \dashrightarrow_I q$ *and* $\alpha(q) \equiv_1 \theta$ *(*q *simply defines all* $m \in Messg$ *which occur in* θ, *and for which* $p(m) = \perp$*). The case for* $\leq_2$ *is identical.* ■

Consider a sequential class *Buffer* (*e.g.*, Figure 2 with the method guards removed). The behaviour of this class is the set of all possible sequences of its services, *i.e.*, $\{put, get, numOfElements\}^*$. In other words, a sequential *Buffer* cannot refuse a message if the message key matches one of its services (of course, the message may return an error or fail, but the observer only observes message acceptances). Contrast this with the behaviour of bounded *Buffer* (in COOP), which is an element of *Reg*. Most COOP languages (that employ synchronisation code) can implement any regular language (at least) *i.e.*, $Reg \subseteq \alpha(Spec)$ where $\alpha(Spec)$ denotes the image of α over *Spec*. Inheritance mechanisms suitable for $\leq_2$ in sequential OOP may not be suitable for $\leq_2$ in COOP. Note that there is no anomaly under $\leq_1$ in COOP.

Definition 9 An inheritance mechanism is *behaviour preserving* under $(Beh, \leq)$ iff $p \overset{\delta}{\dashrightarrow}_I q \implies \alpha(p) \leq \alpha(q)$. ■

We introduce the notion of *behaviour preserving inheritance mechanisms* thus classifying inheritance mechanisms into two types. Behaviour preserving inheritance mechanisms are based on the principles used in sequential OOP. An ideal inheritance mechanism in COOP would be behaviour preserving. Such a mechanism produces only extensions of behaviour, if used incrementally. A simple inheritance mechanism, as used in sequential OOP is behaviour preserving . It can be shown that the proposals based on method guards and accept sets, informally described in Section 2 are also behaviour preserving.

```
class Buffer{
int in=0, out=0;
method numOfElements{...}
pre:    numOfElements < MAX
method put(x){...}
pre:    numOfElements > 0
method get{...}
}
class LockableBuffer: Buffer{
Bool locked=false;
method lock{...}
method unlock{...}
pre:    (get) ∧ !locked
pre:    (put) ∧ !locked
}
```

Figure 4

Example 6 Figure 4 illustrates a non-behaviour-preserving inheritance mechanism that avoids the anomaly from Figure 2. Pre-conditions (denoted by "**pre**") act as guards, but they can also be incrementally composed in the subclasses. The semantics of a set of pre-conditions is given by the conjunction of their conditions. Note that a syntactic specification would use two distinct keys for the two pre-conditions of the method *put* (hence, the transition is incremental). It can be seen that the incremental addition of pre-conditions is essentially "modifying" the synchronisation constraints of the object in an incremental manner. For example, if the pre-condition of *put* in *LockableBuffer* is changed to *locked* (instead of *!locked*) then the behaviour of *Buffer* is not preserved. A behaviour preserving mechanism is clearly preferable. ■

Theorem 2 *Given an inheritance mechanism* $(Spec, \dashrightarrow)$ *and* α *such that* $Reg \subseteq \alpha(Spec)$, *if* $(Spec, \dashrightarrow)$ *is behaviour preserving under* $(\mathcal{P}(Messg^*), \leq_2)$ *then it is not anomaly-free.*

Proof: *Assume the mechanism is anomaly-free. We construct* $p \in Spec$ *such that* $\alpha(p) = \{m_1, m_2\}^*$ *for some* $m_1, m_2 \in Messg$. *Let* $s \in \alpha(p)$ *and* $m \in Messg$ *such that* m *doesn't occur in* $\alpha(p)$. *Then,* $t = s\hat{}\langle m\rangle \notin \alpha(p)$. *Consider the behaviour* $\theta = \alpha(p) \cup \{t, t\hat{}\langle m_1\rangle\}$. $\theta \in Reg$ *(by closure) and* $\alpha(p) \leq_2 \theta$, *hence* $\exists r \in Spec$ *such that* $\alpha(r) \equiv_2 \theta$. *By Proposition 1,* $\exists q \in Spec$ *such that* $p \dashrightarrow_I q$ *and* $\alpha(q) \equiv_2 \theta$. *At any instant, the state of* q *is determined by the values of its instance variables (*q *has a superset of the set of instance variables of* p *since* $p \dashrightarrow_I q$*). Suppose that after accepting* t, q *is in some state* S. *Construct* q' *such that* S *is the initial state of* q' *(by changing the initial values of variables). We have* $\alpha(q') = \{\langle\rangle, \langle m_1\rangle\}$. *Construct* p' *such that* S *(restricted to the variables of* p*) is the initial state of* p'. *We have* $\alpha(p') = \{m_1, m_2\}^*$ *since all states of* p *and* p' *can accept* m_1 *and* m_2. *Hence, we have* $p' \dashrightarrow_I q'$ *(since* $p \dashrightarrow_I q$ *and the same simple mapping of initial values was used to obtain* p' *from* p *and* q' *from* q*). However,* $\alpha(p') \not\leq_2 \alpha(q')$ *since* $\{m_1, m_2\}^* \not\leq_2 \{\langle\rangle, \langle m_1\rangle\}$. *Hence, the inheritance mechanism is not behaviour preserving.* ■

Theorem 2 states that there is no behaviour preserving, anomaly-free inheritance mechanism in COOP. Hence, proposals employing method guards and accept sets must give an anomaly. Theorem 2 does not apply to sequential OOP (α does not map to *Reg*). Further research into this area should lead to more formal classifications of the different types of anomalies. The immediate implication of Theorem 2 is that there is no ideal solution to the problem of inheritance anomalies in COOP. Hence, it supports the recent direction of research [3, 12] which separates the actual inheritance mechanisms of the synchronisation code and the functionality code, leading to non-behaviour-preserving inheritance mechanisms. The trade-off expressed by Theorem 2 is that an inheritance mechanism is either not powerful enough (not anomaly-free), or it is too powerful (not behaviour preserving). In other words, the use of pre-conditions leads to the possibility of making mistakes (incrementally adding new code may not preserve the original behaviour).

6 CONCLUDING REMARKS

This paper investigated the problem of the inheritance anomaly. We claim that a formal approach is needed to provide a better understanding of the problem, which would lead towards the design of better inheritance mechanisms. The main contribution of the paper is the use of the correspondence between an inheritance mechanism and a behavioural hierarchy to motivate a formal definition of the inheritance anomaly in a general setting. The formal definition shows that the interference between inheritance and concurrency is not the most basic cause of the inheritance anomaly. Rather, the anomaly arises because the inheritance mechanisms suitable for one behavioural hierarchy may not be suitable for a different hierarchy.

We presented some theoretical limitations of inheritance mechanisms, thus justifying the recent trends in the search for a cleaner integration of inheritance and concurrency. In particular, we proved that an ideal solution is not possible in COOP. The given framework can be the basis for a formal comparison of the different inheritance mechanisms. It could also be used to formally construct the complete taxonomy of the types of inheritance anomalies. Further work in this area should concentrate on applying the formal treatment to languages that allow intra-object concurrency.

REFERENCES

[1] M. Aksit, J. Bosch, W. van der Sterren, and L. Bergmans. Real-time specification inheritance anomalies and real-time filters. In *ECOOP'94*, LNCS 821, pages 386–407. Springer-Verlag, 1994.

[2] P. America. Designing an object-oriented programming language with behavioural subtyping. LNCS 489, pages 60–90. Springer-Verlag, 1990.

[3] M.Y. Ben-Gershon and S.J. Goldsack. Using inheritance to build extendable synchronisation policies for concurrent and distributed systems. In *TOOLs Pacific '95*, pages 109–121, 1995.
[4] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA'89*, pages 433–443, 1989.
[5] L. Crnogorac and A. S. Rao. Inheritance by extensions and restrictions in agent systems. In *ACSC'97*, Sydney, Australia, February 1997.
[6] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.
[7] D. G. Kafura and K. H. Lee. Inheritance in Actor based concurrent object-oriented languages. In *ECOOP'89*, pages 131–145, UK, 1989.
[8] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. How to overcome the inheritance anomaly. In *ECOOP'96*, LNCS 1098. Springer-Verlag.
[9] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994.
[10] S. Matsuoka, K. Wakita, and A. Yonezawa. Synchronization constraints with inheritance: What is not possible - so what is? Technical Report 10, Dept. of Information Science, the University of Tokyo, 1990.
[11] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in COOP*, chapter 1, pages 107–150. MIT Press, 1993.
[12] C. McHale. *Synchronisation in COO Languages: Expressive Power, Genericity and Inheritance.* PhD dissertation, Trinity College, 1994.
[13] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *ECOOP'93*, LNCS 707, pages 220–246.
[14] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
[15] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA '89*, pages 103–112. ACM Press, 1989.

7 BIOGRAPHY

Dr Anand Rao is the Chief Research Scientist at the Australian AI Institute. He obtained his PhD from the University of Sydney in 1988, and spent a year at IBM's T.J. Watson Research Center. He has published a number of papers in reactive planning and recognition; families of Belief-Desire-Intention (BDI) logics and their properties; and agent-oriented languages and methodologies. **Prof. Kotagiri Ramamohanarao** received his PhD from Monash University in 1980. He is well known for his contributions in the areas of dynamic hash indexing, partial match retrieval and deductive database systems with over 100 refereed papers. He has been a program committee member for several prestigious international conferences including VLDB, ICDE, ICLP, EUROPAR and ISLP. **Lobel Crnogorac** is a PhD student at The University of Melbourne working on incorporation of inheritance into AOP.