Classifying Inheritance Mechanisms in Concurrent Object-Oriented Programming

Lobel Crnogorac[1], Anand S. Rao[2], and Kotagiri Ramamohanarao[1]

[1] Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia,
{lobel,rao}@cs.mu.oz.au
[2] Mitchell Madison Group, Level 49, 120 Collins Street, Melbourne, Victoria 3000, Australia,
anand_rao@mmg.net.au

Abstract. Inheritance is one of the key concepts in object-oriented programming. However, the usefulness of inheritance in concurrent object-oriented programming is greatly reduced by the problem of inheritance anomaly. Inheritance anomaly is manifested by undesirable re-definitions of inherited code. The problem is aggravated by the lack of a formal analysis, with a multitude of differing proposals and conflicting opinions causing the current state of research, and further directions, to be unclear. In this paper we present a formal analysis of inheritance anomaly in concurrent object-oriented programming. Starting from a formal definition of the problem we develop a taxonomy of the anomaly, and use it to classify the various proposals. As a result, the major ideas, trends and limitations of the various proposals are clearly exposed. Formal analysis of the anomaly and a thorough exposition of its causes and implications are the pre-requisites for a successful integration of inheritance and concurrency.

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) [2]. In its full generality the 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* [22,23,24,25]. 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

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 571–601, 1998.

in the literature trying to solve the problem, but almost no formal work has been done. For comparison purposes the proposals were usually evaluated on a set of standard examples introduced by Matsuoka and Yonezawa [23]. Such informal approach cannot guarantee that some examples of anomaly are not omitted from consideration. Many proposals claim to have solved the anomaly, however there is a serious lack of agreement due to a large variety of informal definitions used by different researchers, and to the lack of a formal framework. Consequently, there is no satisfactory method of comparing the various language proposals.

Inheritance anomaly is most commonly believed to be caused by the interference between inheritance and concurrency. In this paper we adopt a much more precise view of the cause of the problem, building upon our previous work [11] which presents the formal definition of the inheritance anomaly. As a result it becomes clear that concurrency is not the real cause of inheritance anomaly, implying that the problem may also arise in other paradigms.

Starting from an intuitive explanation of the issues involved (Section 2) we present our formal framework (Section 3). Inheritance anomaly is defined in this framework as a relationship between (behavioural) subtyping and inheritance. This general definition is then specialised to a particular version of the anomaly which has been the focus of most research in COOP literature. The main contribution of this paper is the development of a formal taxonomy for this particular version of inheritance anomaly (Section 4), thus enabling us to classify many of the languages proposed in literature (Section 6). Inheritance anomaly is highly language dependent. However, by using our framework we are able to avoid most of the language dependent issues in the different proposals. As a result, our comparison yields a clear, uniform view of the current state of research showing that the main progress has been made in the area of languages that do not allow intra-object concurrency. The version of inheritance anomaly analysed in this paper has motivated a large number of proposed solutions. However, we prove that an ideal solution does not exist. We also present some examples of the anomalies which have not been considered in literature before, and discuss the major ideas used to minimise the effects of the anomaly (Section 5).

Finally, our framework is general and can be easily used to investigate other versions of the anomaly in COOP, as well as to investigate inheritance anomaly in paradigms other than COOP [3,10].

2 Inheritance anomaly

In this section we present an intuitive explanation of the inheritance anomaly problem. We do not consider any particular language, relying only on the most basic notions of object-oriented programming such as methods, classes, class-based inheritance and subtyping.

2.1 Incremental inheritance hierarchies in OOP

Inheritance is one of the major components of the OOP paradigm. It is a syntactic mechanism for code re-use (or code sharing). The main benefits of inheritance

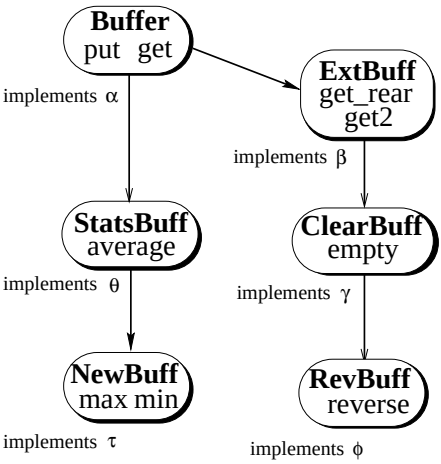

Fig. 1. An incremental inheritance hierarchy. The arrows represent inheritance and point from the superclass to the subclass.

are the reduction of the implementational effort, since new classes can be developed by starting from already written and tested classes, as well as the improved flexibility of the software system, since changes to one class are automatically reflected in its subclasses.

Consider the inheritance hierarchy in Figure 1. Class *Buffer* is a *superclass* of class *ExtBuff*. Class *ExtBuff* is a *subclass* of class *Buffer*, and a superclass of class *ClearBuff*. The interesting property of this inheritance hierarchy is that all subclasses are derived from their superclasses only by adding new methods. None of the subclasses re-define any of their inherited methods. For example, class *ExtBuff* defines new methods *get_rear* and *get2*, and inherits the methods *put* and *get*. In our terminology we refer to this type of inheritance hierarchy as an *incremental inheritance hierarchy*. Incremental inheritance hierarchies exhibit maximum code re-use. A superclass in such a hierarchy is completely re-used by its subclass, since none of its methods are re-defined in the subclass.

The inheritance relationship between classes also suggests another relationship. If class B inherits from class A, it will necessarily have at least all the variables and methods that class A has. Therefore, whenever we require an instance of class A, an instance of class B would do equally well. Note that in this case we are concerned with the relationship between the external behaviour of instances, rather than with the internal structure of a class (as in the case of inheritance). A *type* is a set of instances that have some "*externally observable behaviour*" in common [4]. Hence, any instance belonging to the type can produce the common behaviour. A class implements a type if each instance of the

class belongs to the type. Type τ is a *subtype* of type θ if $\tau \subseteq \theta$. An instance belonging to τ also belongs to θ, hence a class that implements τ also implements θ. In other words, any instance in a subtype can produce the behaviour common to its supertype, thus it can be used in place of an instance belonging to the supertype. Unless otherwise specified the statement "*A implements* θ" will be used to mean that θ is the smallest type (the most specialised type) that A implements. The full definition of a type depends on the precise meaning of "*externally observable behaviour*". Let us concentrate for the moment on a very simple definition of a type, commonly used in OOP:

> A type is characterised by a set of methods $\{m_1, \ldots, m_n\}$. Every instance belonging to the type has the common property that at least the methods $m_1, \ldots, m_n$ are defined for it.

According to this definition of types, the type β in Figure 1, characterised by the methods $\{put, get, get\_rear, get2\}$ is a subtype of type α since each instance in β (*e.g.*, an instance of *ExtBuff*), is also an instance in α. That is, *ExtBuff* implements β and α, but β is the smallest type that it implements. In order to keep the definition of a type simple we ignore the issue of method parameters - this can be easily extended.

Definition 1. A *chain* is a finite sequence of types $\alpha \supseteq \beta \supseteq \gamma \ldots$ □

A chain is a set of types in which there are no unrelated types, all types are related by the subtyping relation. The set $\{\beta, \theta\}$ in Figure 1 is not a chain since each type requires a method that the other type does not have. When a system is designed as a chain of subtypes this generally means that re-usable abstractions have been discovered, by factoring out the common properties into re-usable components. Missing abstractions, overly specialised components, or deficient object modelling may prevent the system from being designed as a chain of subtypes, thus seriously impairing the re-usability of that collection of classes. Also, subtype chains are usually much easier to understand than more complicated relationships that exist in other subtype hierarchies. A chain of subtypes can be understood through a single conceptual relationship - *specialisation*. Every subtype corresponds to a specialisation of its supertype.

Observation 1 *Every path in an incremental inheritance hierarchy implements a chain of subtypes.*

In Figure 1 *ExtBuff* implements β which is a subtype of α, *ClearBuff* implements γ which is a subtype of β etc. Whenever a subclass adds new methods, then the type implemented by the subclass is necessarily a subtype of the type implemented by its superclass.[1]

Observation 2 *Every chain of subtypes can be implemented by an incremental inheritance hierarchy.*

[1] Actually, for this definition of a type, every path in any inheritance hierarchy (as long as the deletion of methods is disallowed) defines a chain of subtypes.

The second observation is the converse of the first one. Given a chain of subtypes $\alpha \supseteq \beta \supseteq \gamma \ldots$, it is always possible to construct an incremental inheritance hierarchy with the root superclass implementing the type α, its subclass implementing the type β etc.

Observations 1 and 2 together imply that the notions of incremental inheritance and subtyping are interchangeable. The two observations trivially hold for our simple definition of a type. However, with slightly different definitions of a type the two observations do not necessarily hold. With a more restrictive definition (in which the subclass needs to satisfy more conditions in order to implement a subtype) Observation 1 is less likely to hold. For example, Observation 1 was found not to hold in the context of recursive types [8] (with contravariant, *i.e.*, argument occurrences of *self*), which uses a more restrictive definition. Adopting a less restrictive definition of a type (*e.g.*, a definition that includes contravariance/covariance rules) may invalidate Observation 2, since there are many more subtype chains, and some of them may not be implementable by incremental inheritance hierarchies. The work on behavioural subtyping [4,19] examines the different notions of a type in OOP.

2.2 Inheritance anomaly in COOP

Concurrent object-oriented programming (COOP) paradigm introduces the concept of an *active* object. Most commonly an active object has its own thread of control, unlike objects in OOP which are *passive*. The model of an active object in COOP is most commonly based on the *actor* model [1], in which each object maintains a *mail queue* for receiving messages. Concurrency implies the need for synchronisation, without which the state of an active object may become inconsistent. Therefore, COOP introduces the notion of *interface control*, an entity (possibly a separate thread of execution, or a locking mechanism) that controls which methods are allowed to proceed and execute, and which methods are suspended. Messages initially enter the mail queue of an object, and are suspended there until the interface control allows them to proceed. Interface control is said to enforce concurrency (or synchronisation) constraints.

Example 1. Consider the class *Buffer* from Figure 1 in the context of COOP. Method *put* stores an element, while *get* removes an element. In order to maintain consistency the interface control of *Buffer* needs to constrain *put* to be acceptable only when the object is not full. Similarly, *get* is constrained to be acceptable only when the object is not empty. The notation $\langle m_1, \ldots, m_n \rangle$ is used to denote a sequence of accepted messages $m_1, \ldots, m_n$. Assuming an instance of *Buffer* starts off being empty, the sequence $\langle put, put, get \rangle$ is a valid sequence of the object while $\langle put, get, get, put, get \rangle$ is not. □

Suppose that we characterise a type by a set of message sequences $\{u_1, \ldots, u_k\}$. Every instance belonging to this type has the property that at least each u_i is a valid sequence of the instance. If $\tau \subseteq \theta$ then each instance in τ is also an element of θ, and therefore is capable of accepting the set of message

sequences common to θ. Hence, any instance from τ can be used in place of an instance from θ, without any observable change in the resulting behaviour.

Clearly, the two observations from Section 2.1 hold without change in COOP, if a type is simply characterised by a set of methods. However, in the context of COOP Observation 1 holds for a different, more restrictive notion of a type, in which a type is characterised by a set of message sequences. Consider the class *ExtBuff* (Figure 1) in the context of COOP. *ExtBuff* incrementally adds new methods *get_rear* and *get2* to class *Buffer*. Immediately we observe that any valid sequence of an instance of *Buffer* is also a valid sequence of an instance of *ExtBuff*. This holds because if we only send messages *put* and *get* to an instance of *ExtBuff* then this instance is indistinguishable from an instance of *Buffer*. Therefore, *ExtBuff* implements a subtype of the type implemented by *Buffer*. Naturally, an instance of *ExtBuff* may also have many more additional valid sequences involving all four of its methods, thus extending the behaviour defined by class *Buffer*. Therefore, in COOP a subclass in an incremental inheritance hierarchy is related to its superclass by an even stronger relationship (based on the sequences of message acceptances) than the simple relationship based on sets of methods.

The problem of *inheritance anomaly* arises in COOP because for many COOP languages Observation 2 does not hold with the stronger notion of a type. We proceed with an example.

2.3 An example of inheritance anomaly

Consider Figure 2, which illustrates an implementation of the concurrent *Buffer* class. This implementation uses the concept of *method guards*, which are Boolean conditions of the form "`method` *method_name* `when` *guard*" [12,23]. A message is acceptable only if the guard of the corresponding method evaluates to *true*. The *Buffer* class shown in Figure 2 implements the synchronisation constraints discussed in Example 1. Suppose that we wish to implement a new type λ characterised by the following property:

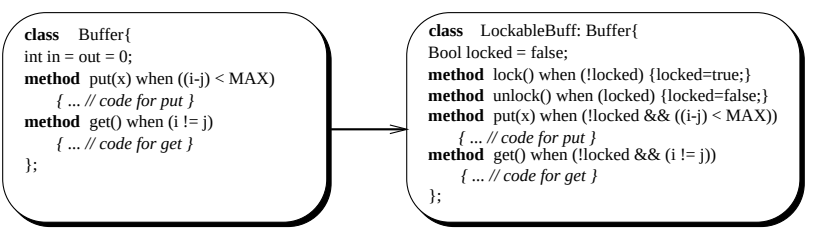

Fig. 2. An example of inheritance anomaly

> Any instance belonging to λ is capable of accepting any message sequence in the set that characterises α (the type of *Buffer*). Furthermore, at any moment an instance belonging to λ is capable of accepting a message *lock*, which must be followed by an acceptance of *unlock*, unless it is the final message of the sequence.

Informally, an instance belonging to λ behaves like an instance of *Buffer*, but it can also be locked, thus becoming incapable of accepting any message except *unlock*. After being unlocked the instance again behaves like an instance of *Buffer*. We have $\lambda \subseteq \alpha$ since any instance in λ satisfies the characteristic property of α. Ideally, we would like to implement the chain $\alpha \supseteq \lambda$ as an incremental inheritance hierarchy, by constructing *LockableBuff* as another subclass of *Buffer* which inherits the complete specification of *Buffer* without any re-definitions. Simply adding the methods *lock* and *unlock* does not implement the desired subtype (*e.g.*, *put* could still be accepted after the message *lock*), leading to a complete re-write of the inherited methods. This is an example of inheritance anomaly since the two-class hierarchy shown in Figure 2 is not an incremental hierarchy. The benefits of inheritance are lost since all the inherited methods must be re-written. This example (introduced originally in [23]) shows the difficulties encountered when a simple, natural approach of method guards is chosen. The problems arising from inheritance anomalies are numerous.

Firstly, the total amount of code required to implement the desired subtype is much larger than necessary.

Secondly, if the methods of *Buffer* need to be re-implemented (*e.g.*, in order to improve efficiency), then this change is not reflected in *LockableBuff* unless the methods *put* and *get* are re-implemented in *LockableBuff* as well. Hence, re-implementation is not localised. Changes that should be local thus become spread all through the hierarchy in presence of an anomaly.[2]

Thirdly, inheritance anomaly increases the effort of making functionality changes to a system. It is well known that user needs are rarely stable. Additional functionality has to be constantly integrated into existing applications, and the existing functionality needs to be constantly improved. Suppose that we wish to add a new method *numElems* (returns the current number of elements) to each class of the hierarchy in Figure 1. This can be achieved by simply adding *numElems* to the definition of *Buffer*. This change is reflected in the whole hierarchy. However, suppose that we wish to add the *lock*/*unlock* capabilities to each class in Figure 1. This cannot be achieved by simply re-writing *Buffer* into *LockableBuff*. Rather, it is necessary to re-write every class of the hierarchy. Figure 3a shows one such re-implementation. Again, instead of just making local changes we are forced to spread the changes all through the hierarchy.

Fourthly, inheritance anomaly reduces the re-usability of classes. In principle, instead of re-writing the class *Buffer* into *LockableBuff* it is possible to achieve the same effect by letting the class *Buffer* inherit from a new root class *Lock*

[2] Languages that do not enforce encapsulation with respect to subclasses (including the simple illustrative language in Figure 2) have even more problems since subclass methods may depend on the implementational details of the superclass [28].

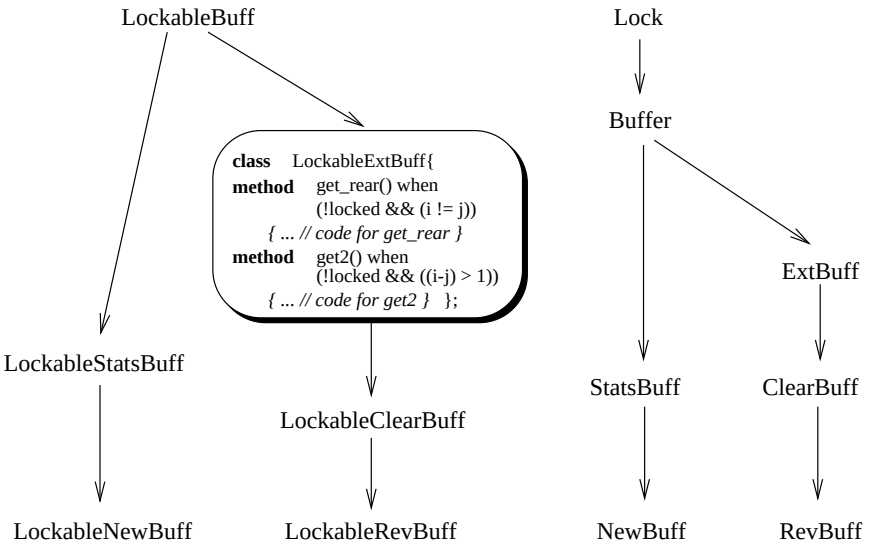

Fig. 3. The effects of the anomaly on inheritance hierarchies

(Figure 3b). The class *Lock* is a re-usable component which can be used in exactly the same way in numerous other hierarchies.[3] However, due to the occurrence of inheritance anomaly, in many languages we are forced to re-write every class of the hierarchy after adding *Lock* as the new root class. Thus, the smooth evolution of the hierarchy is prevented. The burden on the programmer increases, since the programmer is forced to check whether re-definitions are required, even when the anomaly does not occur. The problems become even more serious in the context of compiled libraries, where the programmer may have no access to the code, and is therefore unable to implement the re-definitions.

Inheritance anomaly is a serious problem which has a large impact on the implementation and evolution of a software system. It is crucial that a thorough, formal analysis of the problem is undertaken, before attempting to propose a solution. In the next section we formalise the notions of incremental inheritance, behavioural subtyping and inheritance anomaly.

3 Formal framework

In this section we present a formal framework of inheritance and behavioural subtyping, developed initially in [11]. Inheritance anomaly is defined as a relation-

[3] The problems may arise if the inheritance hierarchy already contains methods *lock* or *unlock*. Thus, the orthogonality of functionalities is required.

ship (informally expressed by Observation 2) between incremental inheritance hierarchies and behavioural subtyping.

3.1 Modelling inheritance as a transition relation

An inheritance mechanism defines the way a new class can be obtained by reusing code from an existing class. A general inheritance mechanism allows new methods to be added, the inherited methods to be re-defined or omitted. An inheritance mechanism of a language is usually given by defining the semantics of its inheritance operator [9]. We take a different approach. We formalise the inheritance mechanism of an arbitrary language as a transition relation on the set of classes. A pair of classes forms a transition if the second class can be derived from the first by employing the inheritance mechanism. Expressing all inheritance mechanisms in terms of transition relations allows us to separate the issues of inheritance from other, language specific issues. Thus, we obtain a general, uniform view of inheritance in different COOP languages.

Method System Domains			Method System Operations		
Instances	p, q, r	$\in$ **Instance**	*class*	:	**Instance** $\rightarrow$ **Class**
Classes	P, Q, R	$\in$ **Class**	*instances*	:	**Class** $\rightarrow \mathcal{P}$(**Instance**)
Message Keys	m	$\in$ **Key**	*methods*	:	**Class** $\rightarrow$ **Key** $\rightarrow$ **Exp**$_\perp$
Method Expressions	e, f	$\in$ **Exp**	$\mathcal{M}$	:	**Class** $\rightarrow \mathcal{P}$(**Key**)
			$\mathcal{M}(P)$		$= \{m : methods(P)m \neq \perp\}$

Fig. 4. Method System domains and operations

We employ a variation of the method system formalism of Cook and Palsberg [9], shown in Figure 4. Method systems are a simple formalisation of object-oriented programming encompassing instances, classes, and method descriptions, which are mappings from message keys to method expressions. The operation *class* gives the class of an instance, while *instances* returns the set of all possible instances of a class. For a given class the operation *methods* maps message keys to either the corresponding expression (the method code), or to the undefined element $\perp$. The operation $\mathcal{M}$ returns the set of all defined methods for a class. The set $Exp_\perp$ is a partial order under the "*less defined than or equal to*" ordering defined as follows.

Definition 2. A *preordering* on a set is a binary relation that is reflexive and transitive. A *partial ordering* is an antisymmetric preordering. Let $e, f \in Exp_\perp$. Then, e is "*less defined than or equal to*" f, written $e \preceq f$, if either $e = \perp$ or $e = f$. We extend $\preceq$ to functions. □

The main difference from the Cook and Palsberg's framework is that we do not allow a class to inherit from another class. That is, inheritance is not a part

of our method system formalism. Rather, we capture a language with inheritance as a set of classes that can be defined in the language without using inheritance, and a transition relation between them.

Definition 3. An *inheritance mechanism* is a pair $(Class, \dashrightarrow)$ where $\dashrightarrow \subseteq Class \times \Delta \times Class$. An element of $\dashrightarrow$, (P, δ, Q) is called a *transition* where $P, Q \in Class$ 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. Remark that 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 derive a new class Q from class P by specifying the differences (*e.g.*, new methods) in δ.[4] Transitions may simulate re-definitions, additions or deletions of components of classes. Hence, very general inheritance mechanisms can be modelled, including the use of *self* and *super* [9]. The sets *Class* and Δ are determined by the language being analysed. Since $Q \in Class$ (Q is defined without inheritance) Definition 3 assumes that everything that can be defined by means of inheritance can also be defined without it.

Example 2. The hierarchy in Figure 1 illustrates several transitions of some inheritance mechanism. We have *Buffer* $\in$ *Class*. Since *ExtBuff* is defined by inheriting from *Buffer*, in our framework *ExtBuff* $\notin$ *Class*. However, it is simple to construct *ExtBuff*$'$, a class which explicitly defines *put,get,get2* and *get_rear*. *ExtBuff*$'$ is an element of *Class*, and it corresponds to the *fully expanded* version of *ExtBuff*. The transition is *Buffer* $\overset{get2,get\_rear}{\dashrightarrow}$ *ExtBuff*$'$. In practice, a class specifies the transition by giving the modification δ from the superclass. □

We now formalise the notion of an incremental inheritance hierarchy. Intuitively, every transition used in such a hierarchy must be an *incremental* transition, that is, it must only define new methods without re-defining any of the inherited methods.

Definition 4. Transition $P \overset{\delta}{\dashrightarrow} Q$ is *incremental* iff $methods(P) \preceq methods(Q)$. The subset of all incremental transitions is denoted $\dashrightarrow_I \subseteq \dashrightarrow$. □

If $P \overset{\delta}{\dashrightarrow}_I Q$ then whenever $methods(P)$ maps a message key to the corresponding method expression, $methods(Q)$ maps the same key to the same expression also (Definition 2). Alternatively, if $methods(P)$ maps a key to $\perp$, then $methods(Q)$ maps the same key to $\perp$ or to a defined expression (thereby adding new methods).

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

Example 3. All the transitions in Figure 1 are incremental. The transition from *Buffer* to *LockableBuff′* (the class constructed by expanding *LockableBuff* in Figure 2) is not incremental since the method expressions of *put* and *get* differ in the two classes (the guards are modified). Note that the definition of *LockableBuff′* is almost identical to the *LockableBuff* in Figure 2, except that *LockableBuff′* does not inherit from *Buffer*. □

3.2 The definition of inheritance anomaly

This section presents the formal definition of inheritance anomaly. We firstly define the notion of $imp(P)$, a function which returns the most specialised type that P implements. This function is firstly used to formalise Observation 1 by defining the concept of *behaviour preservation*. The definition of inheritance anomaly is then obtained by formalising Observation 2.

Definition 5. Let $(X, \leq)$ be a partially ordered set and let Y be a subset of X. An element $x \in X$ is a *lower bound* for Y iff $x \leq y$ for all $y \in Y$. A lower bound x for Y is the *greatest lower bound* for Y iff, for every lower bound x' for Y, $x' \leq x$. When it exists, we denote the greatest lower bound for Y by $\sqcap Y$. We use the following two well known results [21] in the paper: If x is a lower bound for Y and $x \in Y$, then $\sqcap Y = x$. If $\sqcap Y$ exists then it is unique. □

Suppose that the notion of a type has been defined, and that *Types* is the set of all possible types. We now consider the smallest type implemented by a class.

Definition 6. Consider $\theta \in Types$. Class P implements θ iff $\forall p \in instances(P)$, $p \in \theta$. Furthermore, $imp(P) = \sqcap\{\theta : P \text{ implements } \theta\}$. An inheritance mechanism $(Class, \dashrightarrow)$ is *behaviour preserving* with respect to *Types* iff $P \overset{\delta}{\dashrightarrow}_I Q \implies imp(Q) \subseteq imp(P)$. □

For any notion of types, *Types*, we require the existence of $imp(P)$ for any class P. $imp(P)$ denotes the smallest type that P implements, and it is a function since the greatest lower bound is unique. The function imp is not injective in general, since there are usually many different implementations of the same type. An inheritance mechanism is behaviour preserving if along any path in an incremental hierarchy the subclass is related to its superclass by the subtype relation.

Definition 7. Consider an inheritance mechanism $(Class, \dashrightarrow)$ and $P, Q \in Class$. Define $G_P = \{Q : P \overset{\delta \in \Delta^*}{\dashrightarrow} Q\}$. Let $I_P = \{Q : P \overset{\delta \in \Delta^*}{\dashrightarrow}_I Q\}$, $B_P = \{Q : imp(Q) \subseteq imp(P)\}$. Finally, let $S_P = \{\tau \in Types : \tau \subseteq imp(P)\}$. □

Consider Figure 5. For each class P we define sets G_P, I_P, B_P and S_P. The set G_P is the set of all classes that can be obtained from P by repeated applications of the inheritance mechanism. Note that $G_P \subseteq Class$, but commonly, $G_P = Class$ since in many inheritance mechanisms we can obtain any given class from P by repeated re-definitions, deletions and additions. The set I_P is a subset of

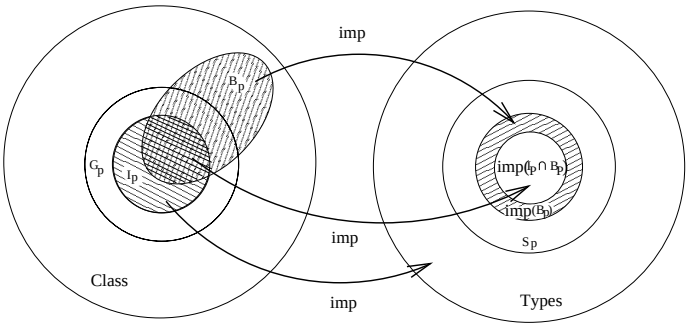

Fig. 5. The definition of inheritance anomaly

G_P which allows only incremental transitions from P. Let $imp(P) = \theta$. The set of all classes which implement subtypes of θ is denoted by B_P. In general, I_P is not a subset of B_P unless the inheritance mechanism is behaviour preserving. The intersection $(I_P \cap B_P)$ is the set of all classes which implement the subtypes of θ, and which can be incrementally obtained from P. The image of B_P under imp is denoted $imp(B_P)$ and it is a subset of S_P, the set of all possible subtypes of θ. In most cases we have $S_P = imp(B_P)$.

Definition 8. An inheritance mechanism $(Class, \dashrightarrow)$ is *anomaly-free* with respect to *Types* iff $\forall P \in Class, imp(I_P \cap B_P) = imp(B_P)$. □

Consider the scenario from Figure 2, Section 2.3: The programmer has defined class *Buffer* which implements type α. The programmer envisions a (non-empty) subtype of α (type λ), containing instances that preserve and extend the behaviour of instances of *Buffer*. The programmer should be able to incrementally obtain *LockableBuff* (which implements the subtype λ) from *Buffer*. Proposition 1 shows that this scenario is a consequence of Definition 8. Of course, if the programmer does not require a subtype of α (*i.e.*, a modification of the behaviour is required) then some inherited methods may need to be re-defined.

Proposition 1. Let $P \in Class$. Inheritance mechanism $(Class, \dashrightarrow)$ is anomaly-free with respect to the given definition of *Types* iff $\forall Q \in Class$, if $imp(Q) \subseteq imp(P)$ then $\exists \delta \in \Delta^*, R \in Class$ such that $P \overset{\delta}{\dashrightarrow}_I R$ and $imp(R) = imp(Q)$.
Proof: Assume the inheritance mechanism is anomaly-free. Clearly, $Q \in B_P$. By Definition 8 $\exists R \in (I_P \cap B_P)$ such that $imp(R) = imp(Q)$. Hence, $P \overset{\delta}{\dashrightarrow}_I R$ for some $\delta \in \Delta^*$. The converse statement can be proven similarly. □

Definition 8 is the general definition of inheritance anomaly in COOP. A particular version of this definition is obtained by providing a particular notion

of *Types*. An inheritance mechanism may be anomaly-free with respect to one such version of the anomaly, while it may have anomalies with respect to another. This observation leads to an explanation of the inconsistency in the current literature. Until now, the notion of a type that causes inheritance anomaly was only given informally by researchers, through examples. Hence, the difference in these notions (*e.g.*, between [25] and [23]) leads to the different conclusions about the occurrences and the implications of inheritance anomalies.

Inheritance anomaly is most commonly believed to be caused by the 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 [3]. Our formal framework allows the analysis of inheritance anomaly in different paradigms (*e.g.*, [10] presents an application in the area of agent-oriented programming).

4 Taxonomy

This section presents a formal taxonomy of the inheritance anomaly which arises when the COOP notion of a type (as discussed in Section 2.2) is used in Definition 8. The taxonomy is then developed by considering two additional notions of a type, which are shown to define subsets of the set of all anomalies. We prove that even for a subset of all anomalies, an ideal solution does not exist in COOP, thus strengthening our previous result [11].

The main goal in this section is to ensure that all inheritance anomalies that have been considered in literature are encompassed by the taxonomy. The formal approach leads to examples of anomalies that have not been discussed in literature before, thus illustrating the disadvantages of the informal approach. Note that it is possible to develop an arbitrarily fine-grained taxonomy of the anomaly. However, this would unnecessarily complicate our effort to formally present the major trends amongst the various language proposals. We also relate our taxonomy to the three different types of anomalies identified in [23].

Trace Semantics	Derived State Semantics
$beh : \mathbf{Instance} \to \mathcal{P}(\mathbf{Key}^*)$	$state : \mathcal{P}(\mathbf{Key}^*) \to \mathbf{Key}^* \to \mathcal{P}(\mathbf{Key}^*)$
$bec : \mathbf{Class} \to \mathcal{P}(\mathbf{Key}^*)$	$state(\xi)u = \begin{cases} \{z : u\hat{}z \in \xi\} & \text{if } u \in \xi \\ \bot & \text{otherwise} \end{cases}$
$bec(P) = \sqcap\{beh(p) : p \in instances(P)\}$	$States : \mathcal{P}(\mathbf{Key}^*) \to \mathcal{P}(\mathcal{P}(\mathbf{Key}^*))$
	$States(\xi) = \{state(\xi)u : u \in \xi\}$

Fig. 6. Semantic domains and operations

For each of the three different notions of a type we require a precise statement of the behavioural property common to all instances belonging to a type. The semantic domains and operations (Figure 6) are used to formalise the behaviour of an instance. The notion of behaviour used in this paper considers an object

to be a "*black box*" which either accepts, or does not accept a given message, while an external observer notes down all the message acceptances. Hence, the behaviour of an instance is a set of message sequences. Consider the trace semantics in Figure 6. Function *beh* gives the set of all possible message sequences that an instance can accept[5], while *bec* returns the behaviour of a class. The behaviour of a class is defined as the common behaviour of all of its instances. If θ is a set of instances then the notation $\sqcap_{beh}\theta$ denotes the greatest lower bound of the behaviours of instances in θ, *i.e.*, $\sqcap\{beh(p) \ : \ p \in \theta\}$. We use the formalism of *traces* [16] to manipulate message sequences. A trace $u \in Key^*$ is a finite sequence of messages. *Concatenation* constructs a trace from a pair of traces u and v by putting them together in that order. The result is denoted $u\hat{}v$. *Head* of a trace u is the first symbol in u, and is denoted u_0.

Definition 9. Let $\xi, \zeta \in \mathcal{P}(Key^*)$. Then, $\xi \sqsubseteq_T \zeta$ iff $\xi \subseteq \zeta$ and $\forall u \in \zeta, u = v\hat{}z$ for some $v \in \xi$ and for some z (possibly empty) such that the symbol z_0 (if it exists) never occurs in ξ. Non-empty set of instances θ is an element of $Types_T$ iff $\forall p \in \theta, \sqcap_{beh}\theta \sqsubseteq_T beh(p)$. Furthermore, if $q \in Instance$ such that $\sqcap_{beh}\theta \sqsubseteq_T beh(q)$ then $q \in \theta$. □

Definition 9 states that a set of instances θ is a type, characterised by $\sqcap_{beh}\theta$, if and only if every instance in θ can accept at least all the message sequences in $\sqcap_{beh}\theta$. An instance in θ may also accept some additional sequences. However, such an additional trace of the instance must start with a trace v from $\sqcap_{beh}\theta$ until a new message z_0 (that never occurs in $\sqcap_{beh}\theta$) is accepted by the instance. Thus, an instance belonging to θ behaves identically to $\sqcap_{beh}\theta$ until it accepts a new message, after which it produces some additional functionality. Furthermore, θ is the set of *all* instances satisfying the characteristic property of θ.

Example 4. $bec(Buffer) \sqsubseteq_T bec(LockableBuff')$ since $bec(LockableBuff')$ contains the same traces as $bec(Buffer)$ (if the observer is not sending *lock*/*unlock* messages), but it also contains additional traces involving *lock*/*unlock*, all of which start with some trace from $bec(Buffer)$. For instance, the trace $\langle put, put, lock, unlock, get\rangle$ is such an additional trace which starts with the trace $\langle put, put\rangle$ from $bec(Buffer)$. Similarly, the trace $\langle lock, unlock, put\rangle$, which starts with the empty trace from $bec(Buffer)$, is a trace in $bec(LockableBuff')$. □

One limitation of the $Types_T$ definition of a type is that it does not satisfactorily capture the case when instances of the same class have different behaviour. For example, consider a class $Buffer(n)$ which given an argument n, can be used to instantiate an instance that can store n elements. Under $Types_T$, the only type that this class implements is the type characterised by $\{\langle\rangle\}$ - the type which contains all instances of all classes (consider the case $n = 0$, *bec* must return $\{\langle\rangle\}$). As a consequence, the information about the common behaviour of instances of $Buffer(n)$ is lost. This limitation can be avoided by refining the definition of

[5] It is assumed that this set is prefix-closed. That is, if $\langle put, put, get\rangle$ is a valid sequence, then $\langle\rangle, \langle put\rangle, \langle put, put\rangle$ are also valid sequences.

$Types_T$ (outlined in Section 7). In this section we use the simpler definition in order to focus on the taxonomy issues. Therefore, we assume that all instances of a class have the same behaviour, that is, $\forall p \in instances(P)$, $beh(p) = bec(P)$.

Proposition 2. Suppose some relation $\sqsubseteq$ captures the common property of all instances that belong to a certain type. If $\sqsubseteq$ is a preordering then $\sqcap_{beh}\theta \sqsubseteq \sqcap_{beh}\tau$ iff $\tau \subseteq \theta$. Furthermore, for any class P, $imp(P) = \theta$ exists and $\sqcap_{beh}\theta = bec(P)$.
Proof: Suppose $\sqcap_{beh}\theta \sqsubseteq \sqcap_{beh}\tau$ and consider $p \in \tau$. We have $\sqcap_{beh}\tau \sqsubseteq beh(p)$. By transitivity of $\sqsubseteq$ we obtain $\sqcap_{beh}\theta \sqsubseteq beh(p)$, and therefore $p \in \theta$. Now suppose $\tau \subseteq \theta$. Consider an instance p such that $beh(p) = \sqcap_{beh}\tau$. By reflexivity of $\sqsubseteq$ we have $p \in \tau$. Hence, $p \in \theta$ also, and $\sqcap_{beh}\theta \sqsubseteq beh(p)$. Since $beh(p) = \sqcap_{beh}\tau$ we obtain $\sqcap_{beh}\theta \sqsubseteq \sqcap_{beh}\tau$. Consider a class P and a type θ such that $\sqcap_{beh}\theta = bec(P)$. Let p be an instance of P. By reflexivity of $\sqsubseteq$ and by the assumption $bec(P) = beh(p)$, we have $\sqcap_{beh}\theta \sqsubseteq beh(p)$. Hence for any instance of P, $p \in \theta$ and therefore P implements θ. Suppose P also implements τ. Hence, $\sqcap_{beh}\tau \sqsubseteq bec(P)$ and since $\sqcap_{beh}\theta = bec(P)$ we obtain $\sqcap_{beh}\tau \sqsubseteq \sqcap_{beh}\theta$. By the previous result $\theta \subseteq \tau$. Hence, θ is a lower bound of the set $\{\tau : P \text{ implements } \tau\}$, and since it is also an element of this set, $\theta = imp(P)$. □

Proposition 3. $\sqsubseteq_T$ is a preordering.
Proof: Suppose $\xi \sqsubseteq_T \zeta$ and $\zeta \sqsubseteq_T \omega$, where $\xi, \zeta, \omega \in \mathcal{P}(Key^*)$. Then, $\xi \subseteq \zeta$ and $\forall u \in \zeta, u = v\hat{}z$ for some $v \in \xi$ and for some z (possibly empty) such that the symbol z_0 (if it exists) never occurs in ξ. Clearly, $\xi \subseteq \omega$. Consider $u \in \omega$. $u = v\hat{}z$ for some $v \in \zeta$ and since $\xi \sqsubseteq_T \zeta$ we have $v = v_1\hat{}v_2$ for some $v_1 \in \xi$. Hence, $u = v_1\hat{}(v_2\hat{}z)$ and by construction the first symbol of $v_2\hat{}z$ never occurs in ξ. Hence, $\xi \sqsubseteq_T \omega$. Also, by the definition of $\sqsubseteq_T$ it is clearly reflexive. □

Propositions 2 and 3 imply that for every class there exists a unique smallest type that it implements. Such a type, denoted $imp_T(P)$ is characterised by the behaviour of class P. We can now consider practical examples of inheritance anomaly with respect to $Types_T$. Consider two classes P and Q. If the most specific type implemented by Q ($imp_T(Q) \in Types_T$) is a subtype of the most specific type implemented by P then by Proposition 1 there should exist a class R that can be incrementally derived from P and which implements $imp_T(Q)$. Otherwise, an inheritance anomaly arises. It is important to examine whether our definition corresponds to the informal examples given in literature.

Example 5. In Example 4 it was noted that $bec(Buffer) \sqsubseteq_T bec(LockableBuff')$. By Proposition 2 we obtain $imp_T(LockableBuff') \subseteq imp_T(Buffer)$. Hence, in an anomaly-free inheritance mechanism there must exist a class, incrementally derivable from *Buffer*, which implements $imp_T(LockableBuff')$. However, in the language shown in Figure 2 such incremental transition cannot be written. Therefore, this is an example of inheritance anomaly. The other examples of anomalies given in literature also satisfy our definition [11]. □

Inheritance anomaly induced by $Types_T$ is only one version of the general problem of inheritance anomaly in COOP. There are many other useful notions

of a type that could be analysed, for example, a subtype may be defined to restrict the possible message sequences of the supertype. The reason inheritance anomaly induced by $Types_T$ has been the main focus of research to date is that Observation 1 holds for this notion. It was then often incorrectly assumed that Observation 2 should also hold for $Types_T$.

We now consider two subsets of the set of anomalies induced by $Types_T$. In order to formally define these two subsets we use a stronger notion of a type. We consider an object to be a "*state machine*" which evolves from its current state to a new state by accepting a message and executing the corresponding method expression. The first subset of the anomalies is obtained by requiring all instances in the same type to evolve through the same number of strongly related core states (Definition 10). Thus, all instances in the type can be considered to be implemented by very similar state machines. The second subset of the anomalies is obtained by allowing instances in the same type to evolve to additional states, but only if those states are restricted versions of core states (Definition 11).

Intuitively, the state of an object is determined by the values of its instance variables. However, this is not externally observable. Therefore, we consider a state to be the set of currently acceptable message sequences. Consider an instance p of *Buffer*. Initially, p can accept any sequence in $beh(p)$. Hence, the initial state of p is simply $beh(p)$. After accepting a message *put*, the set of acceptable message sequences changes, *e.g.*, the sequence $\langle get, put\rangle$ which was not acceptable from the initial state is now acceptable.[6] We say that p has evolved from the initial state to a new state. Note that accepting different message sequences may lead to the same state. For example, the message sequences $\langle put\rangle$, $\langle put, put, get\rangle$ and $\langle put, get, put\rangle$ all lead to the same state of p. Consider the derived state semantics in Figure 6. Function *state* takes the initial state of an instance and returns the state of the instance after it has accepted the message sequence u. The function *States*, given the initial state of an instance, returns the (possibly infinite) set of all states of the instance.

Proposition 4. Suppose $\xi, \zeta \in \mathcal{P}(Key^*)$ and $\xi \sqsubseteq_T \zeta$. Then, $\forall u \in \xi, state(\xi)u \sqsubseteq_T state(\zeta)u$.

Proof: Consider $u \in \xi$. Since $\xi \subseteq \zeta, u \in \zeta$ and $state(\zeta)u \neq \perp$. Let $t \in state(\xi)u$. Then $u\hat{}t \in \xi$ and $u\hat{}t \in \zeta$. Hence, $t \in state(\zeta)u$ and we have $state(\xi)u \subseteq state(\zeta)u$. Let $z \in state(\zeta)u$. It follows that $u\hat{}z \in \zeta$ and since $\xi \sqsubseteq_T \zeta$, $u\hat{}z = (u\hat{}z_1)\hat{}z_2$ for some $u\hat{}z_1 \in \xi$ and the first symbol of z_2 never appears in ξ. Hence, $z_1 \in state(\xi)u$. □

Example 6. Consider an instance p of *Buffer*. After accepting a message sequence u, p is in some state ξ. An instance q of *ExtBuff*, after accepting u is in some state ζ. Intuitively, we expect a strong relationship between the states ξ and ζ. Proposition 4 states that any message sequence acceptable from ξ is also acceptable from ζ. This must hold since *ExtBuff* implements a subtype of the type implemented by *Buffer*, and therefore must be capable of behaving like

[6] Note that the sequence $\langle put, get, put\rangle$ is the corresponding trace which is acceptable from the initial state.

Buffer. However, ζ may allow some additional message sequences involving the new messages *get2* and *get_rear*. In general, after accepting such an additional sequence, q may evolve to a completely new state not related to any state of p. In this example however, q never evolves to a state unrelated to p. An instance of *LockableBuff* does evolve to a new, unrelated state by accepting a message *lock*. In the resulting state messages *put* and *get* are both rejected, which is never the case for an instance of *Buffer*. □

The first subset of the set of inheritance anomalies induced by $Types_T$ is defined by considering the restricted notion of a type in which an instance belonging to a subtype never evolves to a new, unrelated state.

Definition 10. Let $\xi, \zeta \in \mathcal{P}(Key^*)$. Then, $\xi \sqsubseteq_f \zeta$ iff $\xi \sqsubseteq_T \zeta$ and $\exists f : States(\xi) \rightarrow States(\zeta)$ such that f is a bijection. Non-empty set of instances θ is an element of $Types_f$ iff $\forall p \in \theta, \sqcap_{beh}\theta \sqsubseteq_f beh(p)$. Furthermore, if $q \in Instance$ and $\sqcap_{beh}\theta \sqsubseteq_f beh(q)$ then $q \in \theta$. □

Definition 10 states that a set of instances θ is a type, characterised by $\sqcap_{beh}\theta$, if and only if θ is a type under Definition 9, and each instance in θ has the same number of states. By Proposition 4, every state of $\sqcap_{beh}\theta$ is related to a state of an instance in θ. Since the number of states is the same, it follows that every state of an instance is related to a state of $\sqcap_{beh}\theta$ as well. Hence, all instances belonging to θ behave as very similar state machines.

Proposition 5. $\sqsubseteq_f$ is a preordering. The proof is straightforward. □

Proposition 6. The set of anomalies with respect to $Types_f$ is a subset of the set of anomalies with respect to $Types_T$.
Proof: Consider an occurrence of inheritance anomaly for classes P and Q, with respect to $Types_f$. Hence, $imp_f(Q) \subseteq imp_f(P)$ and $\nexists \delta \in \Delta^*, R \in Class$ such that $P \overset{\delta}{\dashrightarrow}_I R$ and $imp_f(R) = imp_f(Q)$. By Proposition 2 $bec(P) \sqsubseteq_f bec(Q)$. Hence, $bec(P) \sqsubseteq_T bec(Q)$ and $imp_T(Q) \subseteq imp_T(P)$. Suppose $\exists \delta' \in \Delta^*, R' \in Class$ such that $P \overset{\delta'}{\dashrightarrow}_I R'$ and $imp_T(R') = imp_T(Q)$. By Proposition 2 we have $bec(R') \sqsubseteq_T bec(Q)$ and $bec(Q) \sqsubseteq_T bec(R')$. From Definition 9 it follows that $bec(R') = bec(Q)$, and therefore $imp_f(R') = imp_f(Q)$. Hence, R' cannot exist since it would solve the anomaly with respect to $Types_f$. □

The notion of types $Types_f$ induces a subset of the problem. It is the smallest subset that we consider in this paper, hence it is the easiest to solve. Examples of COOP languages that are not anomaly-free with respect to $Types_f$ are the languages with *centralised* interface control, and languages that use *accept sets*. Languages employing centralised interface control (*e.g.*, Eiffel// [7],POOL-I [4], [14] etc.) adopt bodies that explicitly regulate the acceptance of messages. These languages are not anomaly-free with respect to Definition 10 because centralised interface control groups all synchronisation constraints into a single body. An

addition of a new method in a subclass always requires the re-definition of the body, otherwise the new method could never be executed.

Some languages that employ decentralised interface control are also not anomaly-free with respect to $Types_f$. Most of these languages have synchronisation control based on the concept of accept sets, that is, at each moment the synchronisation code specifies the set of currently acceptable methods. The set of acceptable methods changes as the object evolves, and this is typically achieved by each method explicitly specifying the next accept set.

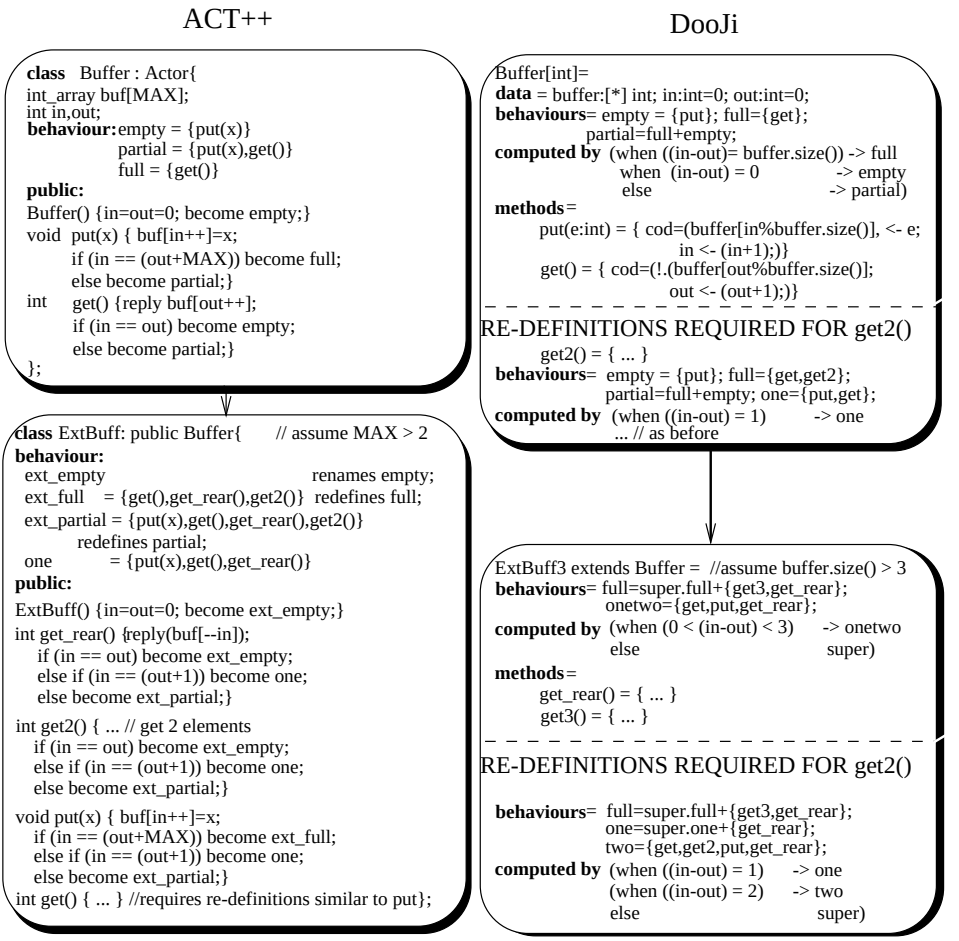

Fig. 7. Inheritance anomalies induced by $Types_f$

Example 7. Figure 7 illustrates an example of inheritance anomaly with respect to $Types_f$ in ACT++ [17]. *ExtBuff* inherits from *Buffer* and adds new methods *get_rear* (acceptable unless the object is empty) and *get2* (which removes two elements and hence is acceptable if there are at least two elements). Each method specifies the next accept set by a `become` statement. In order to define *ExtBuff* in ACT++ extensive re-definitions of the inherited methods *put* and *get* are required. Note that, even if *ExtBuff* only defined the new method *get_rear*, this would still be an occurrence of inheritance anomaly. The inherited methods would not require re-definitions. However, the `behaviour` block would still need to be re-written, thus reducing the flexibility of the hierarchy (*e.g.*, a new method could not be added to *Buffer* without re-defining *ExtBuff*). The problems are reduced in DooJi [29] (*e.g.*, *ExtBuff* can be solved) which instead of explicit `become` statements has a `computed by` block which computes the current accept set. Suppose that instead of implementing *ExtBuff*, we define class *ExtBuff3* with a new method *get3* which removes three elements. DooJi succeeds in localising re-implementation and localising some changes to the hierarchy (*e.g.*, a method *numElems* could be easily added). This is achieved by the use of the "+" operator to re-define the `behaviours` block, thus preserving flexibility. However, the definition of *ExtBuff3* is not re-usable. Furthermore, suppose that we add the method *get2* to *Buffer*. Naturally, some re-definitions are required in class *Buffer*, but inheritance anomaly also causes undesirable re-definitions in *ExtBuff3*. □

The ACT++ example shown in Figure 7 was initially introduced by Matsuoka and Yonezawa [23], where this type of anomaly was termed the *state-partitioning* anomaly. It can be checked that every state-partitioning anomaly satisfies Definition 10. Hence, state-partitioning anomalies are examples of anomalies with respect to $Types_f$. It was formally proven [22] that the languages employing accept sets always suffer from the state-partitioning anomaly. It follows that all such languages are not anomaly-free with respect to $Types_f$. The anomaly induced by $Types_f$ is avoided in proposals which use *method guards*.

Proposition 7. A COOP language that employs method guards is anomaly-free with respect to $Types_f$. The proof is given in the Appendix. □

We now define the second subset of inheritance anomalies.

Definition 11. Let $\xi, \zeta \in \mathcal{P}(Key^*)$. Then, $\xi \sqsubseteq_R \zeta$ iff $\xi \sqsubseteq_T \zeta$ and $\forall s \in States(\zeta)\ \exists s' \in States(\xi)$ such that, $\forall u \in s$ either u contains a symbol m which never appears in ξ, or $u \in s'$. Non-empty set of instances θ is an element of $Types_R$ iff $\forall p \in \theta, \sqcap_{beh}\theta \sqsubseteq_R beh(p)$. Furthermore, if $q \in Instance$ such that $\sqcap_{beh}\theta \sqsubseteq_R beh(q)$ then $q \in \theta$. □

Definition 11 states that a set of instances θ is a type, characterised by $\sqcap_{beh}\theta$ if and only if θ is a type under Definition 9 and every state of an instance in θ is a restriction of some state of $\sqcap_{beh}\theta$. That is, an instance from θ may have more states than $\sqcap_{beh}\theta$ (reached when the newly defined messages are accepted). Each such additional state must contain traces from some state of $\sqcap_{beh}\theta$ or traces that involve the new methods.

Example 8. The example of *LockableBuff* satisfies Definition 11. The state obtained after accepting *lock* is a restriction of the corresponding state of an instance of *Buffer*, since it contains the trace $\langle\rangle$ (any state of an instance of *Buffer* contains this trace), and all the other traces involve the new message *unlock*. Another example of inheritance anomaly induced by $Types_R$ arises when a class *HistoryBuff* is derived from *Buffer*. *HistoryBuff* adds a new method *gget*, which behaves exactly like *get*, except that it cannot be executed immediately after an execution of *put*. The sequences $\langle put, put, get\rangle$ and $\langle put, get, put\rangle$ now lead to two different states, unlike in the case of *Buffer*. The state obtained by accepting the first trace is a restriction of the corresponding state of an instance of *Buffer* (it contains additional traces that involve *gget*), while the state obtained by accepting the second trace is also a restriction of the original state. Many proposals succeed in localising re-implementation for this example, but none succeed in localising changes to the hierarchy and preserving re-usability. Furthermore, many proposals deteriorate considerably when handling more complicated cases involving constraints that depend on the history of invocations (*e.g.*, [27]). The example of *HistoryBuff* was introduced in [23] as a separate type of anomaly - *the history sensitive* anomaly, while *LockableBuff* was named *state-modification* anomaly. In this paper both, *LockableBuff* and *HistoryBuff* are captured by Definition 11. Intuitively, these two examples can be seen as being related, since no methods are acceptable *after* an execution of *lock*, while *gget* is not acceptable *after* an execution of *put*. It is possible to refine our taxonomy in order to separate these two examples. □

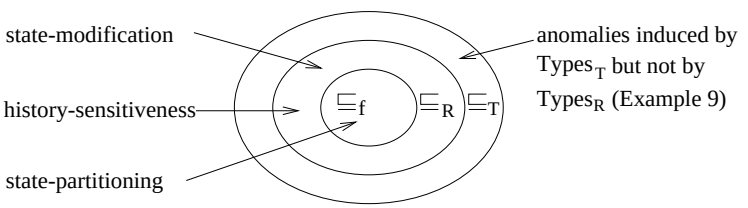

Fig. 8. The anomalies induced by the three definitions of a type

Proposition 8. $\sqsubseteq_R$ is a preordering. The proof is straightforward. □

Similarly to Proposition 6 it can be shown that the set of anomalies induced by $Types_R$ is a subset of the set of anomalies induced by $Types_T$, leading to the hierarchy in Figure 8.

Theorem 1. Consider an inheritance mechanism $(Class, \dashrightarrow)$. If $(Class, \dashrightarrow)$ is behaviour preserving with respect to $Types_T$ then it is not anomaly-free with respect to $Types_R$. The proof is given in the Appendix. □

Theorem 1 states that even for a subset of the anomalies (induced by $Types_R$), an anomaly-free, behaviour preserving COOP language cannot be designed. Most COOP languages are naturally behaviour preserving with respect to $Types_T$. The problems with non-behaviour preserving languages are discussed in Section 5. It follows that an ideal solution to the version of inheritance anomaly induced by $Types_R$ (and $Types_T$) does not exist.

The third subset of the inheritance anomalies considered in this paper contains the anomalies that do not satisfy the conditions of Definition 10 or Definition 11. This type of anomaly has not been discussed in literature, but it naturally arises from our taxonomy. We present an example for illustration.

```
class ResourceManager{
int granted_A = granted_B = 0;
guards:
    guard gA    ( granted_A < max_A )
    guard gB    ( granted_B < max_B )
    guard fA    ( granted_A > 0 )
    guard fB    ( granted_B > 0 )
synchronizers:
    gA  enables access_A;
    gB  enables access_B;
    fA  enables free_A;
    fB  enables free_B;
methods:
  int access_A(request) { ... granted_A++; ... }
  int access_B(request) { ... granted_B++; ... }
  int free_A(request) { ... granted_A--; ... }
  int free_B(request) { ... granted_B--; ... }};
```

```
class PriorityAccess{
int priority = 0;
method_sets:  mset P_OFF  #{priority_off}
              mset P_ON   #{priority_on}
synchronizers:
  initially          enables all-except(P_OFF);
  (priority==1)      enables all-except(P_ON);
  (priority==0)      enables all-except(P_OFF);
methods:  void priority_on() { ... priority=1; ... }
          void priority_off() { ... priority=0; ... } };

class ResourceManager: PriorityAccess{
guards: guard gA priority || (granted_A < max_A)
        guard gB priority || (granted_B < max_B) };
  // note that if method sets were used in
  // the original definition of
  // ResourceManager, then even more
  // re-definitions would be required
```

Fig. 9. An example of the third type of anomaly

Example 9. Consider a hierarchy of resource managers which is derived from the class *ResourceManager* shown in Figure 9. This class, implemented in ABCL [23], is used to control access to a pool of two types of resources, A and B. At most max_A resources of type A, and at most max_B resources of type B can be granted at any time. Suppose that a new class *PriorityAccess* is written. This class is intended to be a re-usable class which adds two new methods: *priority_on* and *priority_off*. If we let *PriorityAccess* class be the superclass of all classes in the *ResourceManager* hierarchy then we expect the following behaviour: A client of an instance of the new *ResourceManager* class is allowed to use *priority_on*, after which all *access_A* requests disregard the max_A limit. The method *priority_off* resets the object to use the limit again. None of the current proposals are capable of avoiding re-definitions of the hierarchy. Figure 9 shows the re-definitions required for *ResourceManager* in ABCL. □

Finally, we look at the problem of inheritance anomalies in the presence of internal concurrency (intra-object concurrency). Many languages allow multiple

threads within an object. However, we only consider the languages that allow the programmer to specify the constraints under which methods are allowed to run concurrently. Internal concurrency is introduced by refining the definition of *Key*, the set of message keys. For each key $m \in Key$ the refined set Key_{intra} contains keys m_s (denoting the start of an invocation), and m_e (denoting the end of an invocation). Function $beh_{intra} : Instance \rightarrow \mathcal{P}(Key^*_{intra})$ returns the set of all acceptable sequences of message starts and completions.[7]

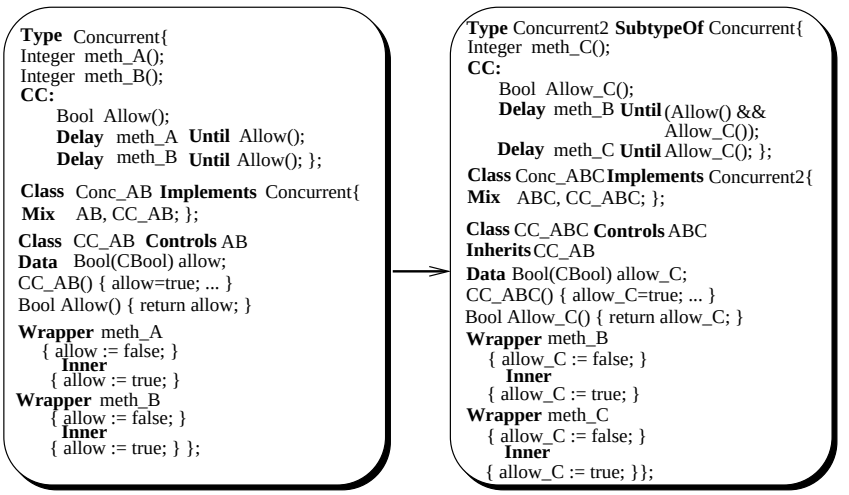

Fig. 10. Inheritance anomaly in presence of internal concurrency - BALLOON [5]

Example 10. Consider a class *Conc_AB* which defines two mutually exclusive methods, *meth_A* and *meth_B*. Furthermore, at most one invocation of either method can be active at a time. Hence, $beh_{intra}(Conc\_AB)$ contains $\langle meth\_A_s, meth\_A_e, meth\_B_s, meth\_B_e\rangle$, but it does not contain traces $\langle meth\_A_s, meth\_B_s, meth\_B_e, meth\_A_e\rangle$ or $\langle meth\_A_s, meth\_A_s, meth\_A_e, meth\_A_e\rangle$. Consider a class *Conc_ABC* which defines a new method *meth_C*. This method can be executed concurrently with *meth_A*, but not with *meth_B*. As before, at most one invocation of *meth_C* can be active at a time. We have, $beh_{intra}(Conc\_AB) \sqsubseteq_T beh_{intra}(Conc\_ABC)$. Some additional traces are $\langle meth\_A_s, meth\_C_s, meth\_A_e, meth\_A_s, meth\_C_e, meth\_A_e\rangle$ and $\langle meth\_C_s, meth\_C_e, meth\_B_s, meth\_B_e\rangle$. □

The definition of inheritance anomaly and the taxonomy of the anomaly developed in this section are easily generalised to include internal concurrency

[7] Some assumptions are relevant. For example, each completion must occur after the corresponding start.

by simply using the new semantic operations $beh_{intra}, bec_{intra}, state_{intra}$ and $States_{intra}$. Clearly, every anomaly described until now is a special case of this generalised notion of inheritance anomaly. Hence, internal concurrency possibly causes even more inheritance anomalies.

Example 11. Figure 10 illustrates an implementation of classes *Conc_AB* and *Conc_ABC* in BALLOON [5]. BALLOON supports a complete separation of subtyping from inheritance. Hence, types *Concurrent* and *Concurrent2* specify the concurrency control (CC block), classes *AB* and *ABC* (which are not shown) implement the sequential methods *meth_A, meth_B* and *meth_C*. Classes *CC_AB* and *CC_ABC* implement the concurrency control of the methods using synchronisation variables *allow* and *allow_C*. Finally, the classes *Conc_AB* and *Conc_ABC* are obtained by mixing the sequential methods and their concurrency control. Since $beh_{intra}(Conc\_AB) \sqsubseteq_T beh_{intra}(Conc\_ABC)$ an anomaly-free language should allow an incremental inheritance transition from *Conc_AB* to *Conc_ABC*. However, re-definitions (*Concurrent2* and *CC_ABC* cannot be derived incrementally) are required in BALLOON. Consider Figure 11, which illustrates an approach proposed in [6]. Synchronisation class *Sync_DE* allows concurrent execution of methods *D* and *E*. At most one invocation of each method can be active at a time, but the two methods may be active concurrently. Consider a subtype of the type implemented by *Sync_DE*. Instances belonging to this subtype (implemented by *Flexible_DE*) behave like instances of *Sync_DE*, but they also allow disabling and enabling of internal concurrency. This subtype cannot be implemented by incremental inheritance from *Sync_DE*, because the `allow_start` condition of all methods needs to be re-written. □

The example of inheritance anomaly in BALLOON (Figure 10) is induced by the internal concurrency version of the notion $Types_f$. It occurs in most languages that support internal concurrency. This shows that the research into the inheritance anomaly in the context of internal concurrency is still in the initial stages, mainly due to the additional complexity which makes an informal analysis impractical.

5 Minimising the anomaly

This section examines three major ideas that can be used to minimise the effects of inheritance anomaly. The three ideas that we consider are *separation of concerns*, *non-behaviour preserving inheritance*, and *generic policies*. One of the major aims of COOP is to separate the concerns of concurrency from the functionality concerns. Then, the concurrency part of a class can be re-used separately from the functionality part of the class. Since the separation of concerns in COOP increases the possibility of re-use, it needs to be investigated whether this may lead to a reduction of the effects of the inheritance anomaly.

Example 12. Recall the implementation of *LockableBuff* in Figure 2. The code for *put* and *get* had to be re-written even though only the guards of the two methods

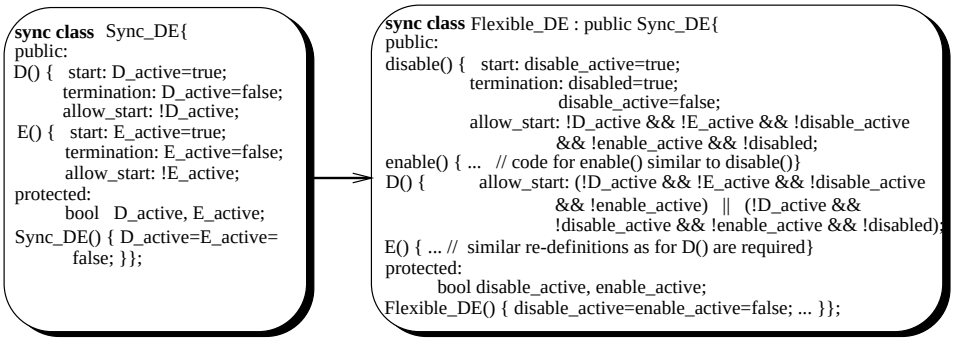

Fig. 11. The anomaly in presence of internal concurrency - synchronisation classes [6]

required modifications. If each method is separated into a *guard* part, and a *functionality* part, then the amount of re-definition can be reduced. For instance, in the case of *put* the statement "`guard` *put(x)* `when` (*!locked ...*)" would be the only re-definition required. In the method system framework this separation of concerns is modelled by using multiple message keys for each method. Hence, $\mathcal{M}(Buffer) = \{put^c, put^f, get^c, get^f\}$ where $methods(Buffer)put^c$ maps to the guard expression, while $methods(Buffer)put^f$ maps to the functionality code.□

Example 12 illustrates how separation of concurrency from functionality can reduce the effects of the anomaly. Note however that nothing prevents the programmer from mixing the functionality code with the concurrency code. Since guards have access to instance variables they in general depend on the implementational details of methods. Hence, re-implementing a method may lead to re-implementation of the concurrency code as well. Similarly, if concurrency issues are mixed in with the functionality code then an occurrence of inheritance anomaly may lead to unnecessary changes in the functionality code, as well as to the changes in the concurrency code. Clearly, the mixing of concurrency and functionality reduces the possibility of re-use, and it should be prevented by enforcing a complete separation of the two concerns.

Definition 12. Consider a COOP language that employs the separation of concurrency and functionality by multiple message keys. Suppose $\mathcal{M}(P) = \mathcal{M}(Q) = \{{m_1}^f, {m_1}^c, {m_2}^f, {m_2}^c, \dots\}$, and $\forall {m_i}^c \ methods(P){m_i}^c = methods(Q){m_i}^c$. The language is *implementation independent* iff for any such P and Q we have $imp(P) = imp(Q)$. □

In an implementation independent COOP language the type implemented by a class is independent from the functionality part of the class definition. Essentially, the functionality code of the class is encapsulated from the synchronisation code of the class. Examples of implementation independent languages

are [6,14,24,26]. Some of these proposals encapsulate the concurrency issues into a separate class in order to facilitate re-use of the concurrency part. Naturally, a language may be implementation independent with respect to one notion of types, while not being implementation independent with respect to a different notion. We now show that if a language is implementation independent with respect to $Types_T$ then re-definition of the functionality code is never required.

Proposition 9. Consider a COOP language that is implementation independent with respect to $Types_T$. An inheritance anomaly in this language never requires the re-definition of the functionality part of any method.
Proof: Consider classes P, Q and suppose $imp_T(Q) \subseteq imp_T(P)$. Since we have an occurrence of inheritance anomaly it follows that $\nexists \delta \in \Delta^*, R \in Class$ such that $P \overset{\delta}{\dashrightarrow}_I R$ and $imp_T(R) = imp_T(Q)$. Let $\mathcal{M}(P) = \{{m_1}^f, {m_1}^c, \ldots, {m_n}^f, {m_n}^c\}$ and $\mathcal{M}(Q) = \{{m_1}^f, {m_1}^c, \ldots, {m_n}^f, {m_n}^c, \ldots\}$. Construct R which is identical to Q except that $methods(R){m_i}^f = methods(P){m_i}^f$ for $1 \leq i \leq n$. By Definition 12, $imp_T(R) = imp_T(Q)$, and by the construction of R it does not re-define any functionality code of P. □

Hence, any language which enforces a complete separation of concurrency from functionality always avoids any re-definitions of the functionality code in presence of inheritance anomalies. This reduces the total amount of code required, and maintains the flexibility of the hierarchy under re-implementation (it does not help with localising functionality changes and with re-usability). The effects of the anomaly are reduced, but the anomaly still occurs since re-definitions of the concurrency code are still required.[8] Many proposals that claim to have solved the inheritance anomaly actually only succeed in re-using the functionality code. Solving the inheritance anomaly in our framework means re-using the whole specification of the superclass - including the synchronisation code.

The second approach we consider is a consequence of Theorem 1 which states that if an anomaly-free inheritance mechanism exists then it must be non-behaviour preserving. In such a mechanism addition of new methods does not necessarily lead to a subtype of the original type. In practice, this means that the mechanism has access to, and is capable of changing meta-level information.

Example 13. Recall the example of class *HistoryBuff* from Example 8. Suppose that in some language it is possible to write a re-usable class *History*. Any subclass of *History* will have all the invocations of all of its methods logged. Then, *HistoryBuff* can be incrementally defined from *Buffer* (which now is a subclass of *History*) by defining the new method *gget*. The guard of *gget* would use some constructs introduced by *History*, *e.g.*, `int` *gget()* `when` (*last_invocation()* `!=` *put*). Such a language would solve this particular inheritance anomaly, but the implementation of *History* would most likely add a non-practical overhead. □

[8] It is still possible to mix in the functionality code into the concurrency code, but this can be prevented by restricting the expressiveness of the concurrency component, or by using different languages for the different components [14].

Non-behaviour preserving inheritance mechanisms are more expressive than behaviour preserving inheritance mechanisms, and therefore they are likely to handle more inheritance anomalies. However, there are very few proposals which employ non-behaviour preserving concepts. This is a consequence of very serious problems that arise from such mechanisms. Firstly, non-behaviour preserving mechanisms do not guarantee the subtype relationship. Hence, for each case it must be manually checked (undecidable in general) whether the subclass implements a subtype. Secondly, the efficiency issues are likely to make such proposals prohibitively expensive. However, more research into static analysis techniques and into optimisation techniques is needed before a definite answer can be given concerning non-behaviour preserving inheritance mechanisms.

The third approach towards reducing the effects of the anomaly is to build libraries of generic synchronisation/concurrency policies [20,24]. Then, instead of attempting to inherit the concurrency part of its superclass, a subclass may simply instantiate a different policy. Generic policies are especially useful in the context of internal concurrency, where there is a small number of frequently used policies (*e.g.*, ReadersWriter, ReadersPriority etc.).

6 Classification

In this section we present the results of classifying the various proposals from literature. Firstly, the languages are compared with respect to their inheritance mechanism (centralised, decentralised with accept sets, or decentralised with method guards[9]), whether they are behaviour preserving, and whether they are implementation independent. We use "-" to denote that some feature is not supported. We use `KEYS` to denote that a language employs the separation of keys, but is not implementation independent. Secondly, the inheritance mechanisms are compared with respect to inheritance anomalies. `LOC-R` signifies that a language achieves the localisation of re-implementation.

Our comparison shows that none of the proposals solve the anomaly. Most proposals only succeed in localising re-implementation of some subset of the anomalies. Furthermore, none of the three types of anomalies has been solved in the context of internal concurrency - all of the proposals suffer from anomalies with respect to the internal concurrency definition of $Types_f$. Three proposals [14,15,23] implement limited forms of non-behaviour preservation. This enables a successful solution to the *LockableBuff* example. Because of the limited nature of their non-behaviour preserving constructs these proposals avoid most of the problems associated with non-behaviour preservation, in particular they do not sacrifice efficiency. There are other issues, not considered in this paper, that are important in comparing the different proposals. For instance, on the conceptual level it is interesting to examine which proposals succeed in presenting a single concept of inheritance that is used uniformly to inherit the

[9] Two proposals employ pattern matching with some similarity to the mechanism of method guards.

concurrency and the functionality code. The proposals should also be compared with respect to efficiency.

language	inheritance mechanism	behaviour preserving	implementation independent	inheritance anomaly			
				$Types_f$	$Types_R$	$Types_T$	internal
CUBL [27]	GUARDS	Yes	KEYS	Yes	No	No	-
ACT++ [17]	ACCEPT SETS	Yes	No	No	No	No	$Types_f$
ROSETTE [30]	ACCEPT SETS	Yes	No	No	No	No	$Types_f$
DEMETER [20]	GUARDS	Yes	KEYS	Yes	LOC-R	LOC-R	$Types_f$
MAUDE [25]	PATTERNS	Yes	No	Yes	No	No	$Types_f$
MAUDE [18]	PATTERNS	Yes	No	Yes	LOC-R	LOC-R	$Types_f$
[26]	GUARDS/SETS	Yes	Yes	Yes	LOC-R	LOC-R	$Types_f$
[6]	GUARDS	Yes	Yes	Yes	LOC-R	LOC-R	$Types_f$
BALLOON [5]	GUARDS	Yes	KEYS	Yes	LOC-R	LOC-R	$Types_f$
GUIDE [12]	GUARDS	Yes	KEYS	Yes	No	No	$Types_f$
[13]	GUARDS	Yes	No	Yes	LOC-R	No	-
DOOJI [29]	GUARDS/SETS	Yes	KEYS	LOC-R	No	No	$Types_f$
[15]	GUARDS	No	No	Yes	No	No	-
[14]	CENTRALISED	No	Yes	LOC-R	LOC-R	LOC-R	$Types_f$
DESP [24]	GUARDS	Yes	Yes	Yes	LOC-R	LOC-R	$Types_f$
ABCL [23]	GUARDS/SETS	No	No	Yes	LOC-R	LOC-R	-

7 Conclusion and further work

This paper investigated the problem of the inheritance anomaly. Starting from a formal definition of the anomaly we developed a taxonomy used to compare the various inheritance mechanisms proposed in literature. We presented a theoretical limitation of inheritance mechanisms, showing that an ideal solution for the version of the anomaly that has been investigated in the COOP literature does not exist. As a result it becomes clear that the problem of inheritance anomaly has not been solved, with most proposals merely reducing its harmful effects.

There are many other notions of types in COOP that could be examined. For example, the code that specifies internal concurrency is commonly added to a class as an attempt to increase efficiency. Therefore, the natural notion of a type would encompass all versions of the same class with different degrees of concurrency ($Types_T$ separates different degrees of concurrency into different types). Also, it may be desirable to separate inter-object and internal concurrency in order to be able to re-use them separately [29].

Inheritance anomalies may arise in many other paradigms, *e.g.*, agent-oriented programming [10], coordination languages, real-time specification languages [3] etc. It is important to note that discovering an anomaly in a paradigm does not necessarily imply that the anomaly will cause problems in practice. For example, inheritance anomaly exists in the context of sequential OOP. As noted in Section 2.1, if the notion of a type includes contravariance/covariance rules then Observation 2 does not hold, and inheritance anomaly arises. The reason this anomaly does not create problems in OOP is that most OOP inheritance

mechanisms never incrementally produce subclasses with parameter types that are different from the parameter types of methods in the superclass.[10] Therefore, this notion of a type is not interesting with respect to inheritance anomaly.

Another possible extension of our framework is to refine $Types_T$ in order to capture the case when the behaviour of a class is different from the behaviour of its instances (Section 4). This can be achieved by characterising each type by a set of behaviours, rather than by a single behaviour as in Definition 9. Such generalisation of $Types_T$ is required in the analysis of more expressive mechanisms, which make use of message arguments and various other control information (*e.g.*, a message may be accepted based on the identity of the sender, the sender's location, the time it was sent etc.).

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986. 575
2. G. Agha, P. Wegner, and A. Yonezawa. Proceedings of the ACM SIGPLAN workshop on object-based concurrent programming. *SIGPLAN Notices*, 24(4), 1989. 571
3. M. Aksit, J. Bosch, W. van der Sterren, and L. Bergmans. Real-time specification inheritance anomalies and real-time filters. In *Proceedings of ECOOP'94*, LNCS 821, pages 386–407, Bologna,Italy, July 1994. Springer-Verlag. 572, 583, 597
4. P. America. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages*, LNCS 489, pages 60–90, Noordwijkerhout, The Netherlands, June 1990. Springer-Verlag. 573, 575, 587
5. C. Baquero, R. Oliveira, and F. Moura. Integration of concurrency control in a language with subtyping and subclassing. In *USENIX Conference on Object-Oriented Technologies (COOTS'95)*, Monterey, California, USA, June 1995. 592, 593, 597
6. 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, Melbourne,Australia, November 1995. Prentice-Hall. 593, 594, 595, 597
7. D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–101, September 1993. 587
8. W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of POPL'90*, pages 125–135, San Francisco, California, January 1990. 575
9. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA'89*, pages 433–443, New Orleans, 1989. ACM Press. 579, 579, 580
10. L. Crnogorac, A. Rao, and K. Ramamohanarao. Analysis of inheritance mechanisms in agent-oriented programming. In *Proceedings of IJCAI'97*, pages 647–652, Nagoya, Japan, August 1997. Morgan Kaufmann Publishers. 572, 583, 597
11. L. Crnogorac, A. Rao, and K. Ramamohanarao. Inheritance anomaly - a formal treatment. In *FMOODS'97*, pages 319–334, England, July 1997. Chapman & Hall. 572, 578, 583, 585

[10] There are some exceptions. For instance, the use of *self* does change the parameter types in the subclass. Such a mechanism actually turns out to be anomaly-free.

12. D. Decouchant et. al. A synchronization mechanism for typed objects in a distributed system. *SIGPLAN Notices*, 24(4):105–107, April 1989. 576, 597
13. S. Ferenczi. Guarded methods vs. inheritance anomaly - inheritance anomaly solved by nested guarded method calls. *SIGPLAN Notices*, 30(2):49–58, February 1995. 597
14. G. Florijn. Object protocols as functional parsers. In *Proceedings of ECOOP'95*, LNCS 952, pages 351–373, Aarhus, Denmark, August 1995. Springer-Verlag. 587, 595, 595, 596, 597
15. S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP'92*, LNCS 615, pages 185–196, Utrecht, The Netherlands, June 1992. Springer-Verlag. 596, 597
16. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985. 584
17. D. G. Kafura and K. H. Lee. Inheritance in Actor based concurrent object-oriented languages. In *ECOOP'89*, pages 131–145, UK, 1989. Cambridge University Press. 589, 597
18. U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. How to overcome the inheritance anomaly. In *ECOOP'96*, LNCS 1098, Linz, Austria, 1996. Springer-Verlag. 597
19. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994. 575
20. C. Lopes and K. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *Proceedings of ECOOP'94*, LNCS 821, pages 81–99, Bologna, Italy, July 1994. Springer-Verlag. 596, 597
21. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974. 581
22. S. Matsuoka, K. Wakita, and A. Yonezawa. Synchronization constraints with inheritance: What is not possible - so what is? Technical Report 10, University of Tokyo, 1990. 571, 589
23. 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. 571, 572, 576, 577, 583, 583, 589, 590, 591, 596, 597
24. C. McHale. *Synchronisation in COO Languages: Expressive Power, Genericity and Inheritance*. PhD dissertation, Trinity College, 1994. 571, 595, 596, 597
25. J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings of ECOOP'93*, LNCS 707, pages 220–246, Kaiserlautern, Germany, July 1993. Springer-Verlag. 571, 583, 597
26. C. Neusius. Synchronising actions. In *Proceedings of ECOOP'91*, LNCS 512, pages 118–132, Geneva, Switzerland, July 1991. Springer-Verlag. 595, 597
27. A. Poggi. Interface methods: a means for the integration of inheritance in a concurrent OOP language. *Informatica*, 20:125–134, 1996. 590, 597
28. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA'86*, pages 38–45. ACM Press, September 1986. 577
29. L. Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *IEEE TENCON'94*, pages 541–545, August 1994. 589, 597, 597
30. C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of OOPSLA '89*, pages 103–112, New Orleans, 1989. ACM Press. 597

Appendix

Proof of Proposition 7 (sketch): The complete proof requires a formalisation of the semantics of guards. Consider Proposition 1 and classes P, Q such that $imp_f(Q) \subseteq imp_f(P)$. Suppose that $\mathcal{M}(P) = \{m_1, \ldots, m_n\}$ and $\mathcal{M}(Q) = \{m_1, \ldots, m_k\}$ where $n \leq k$. Without loss of generality we assume that $k = n + 1$. Suppose that both P and Q use method guards. We incrementally construct R from P such that $imp_f(R) = imp_f(Q)$. Since $bec(P) \sqsubseteq_f bec(Q)$ it is sufficient to construct a guard for m_{n+1} in R which accepts m_{n+1} iff Q's guard accepts it. We would like to use Q's guard in the construction of R, but this cannot be done directly since Q's guard makes use of instance variables of Q which are different from R. Therefore, we construct the guard for m_{n+1} as follows: The guard has a local pool of instances of P and Q. Upon reception of m_{n+1} the guard needs to determine which sequence of messages led to the current state of the object. This is achieved by trying all possible message sequences of length $1, 2, ...$ Clearly, this process will eventually find either the message sequence which led to the current state, or another sequence which also leads to the current state. The first such sequence is then applied to an instance of Q, followed by the message m_{n+1}. The guard of R then accepts the original message m_{n+1} iff the guard of Q accepts m_{n+1}. Note that the guard of R is side-effect free with respect to instance variables of R. Hence, R satisfies Proposition 1. □

Proof of Theorem 1: Firstly note that we will only consider the languages that are at least as expressive as finite state machines (otherwise, some useful synchronisation constraints cannot be defined). That is, let $Reg \subset \mathcal{P}(Key^*)$ be the set of all regular languages over Key. We assume that $Reg \subseteq bec(Class)$. Hence, every regular language can be accepted by (is the behaviour of) instances of some class. Assume the mechanism is anomaly-free with respect to $Types_R$, and behaviour preserving with respect to $Types_T$. Consider $P \in Class$ such that $bec(P) = \{m_1, m_2\}^*$ for some $m_1, m_2 \in Key$. Let $u \in bec(P)$ and $m \in Key$ such that m does not occur in $bec(P)$. Then, $v = u^\frown\langle m\rangle \notin bec(P)$. Consider the type $\theta \in Types_R$, where $\sqcap_{beh}\theta = bec(P) \cup \{v, v^\frown\langle m_1\rangle\}$. We have $bec(P) \sqsubseteq_R \sqcap_{beh}\theta$, hence $\theta \subseteq imp_R(P)$. To check this note that v starts with $u \in bec(P)$. Also, every state of $\sqcap_{beh}\theta$ contains traces that involve either m, m_1 or m_2, and every trace that involves m_1 and/or m_2 is a trace in the (only) state of $bec(P)$. By closure $\sqcap_{beh}\theta \in Reg$. Hence, $\exists Q \in Class$ such that $bec(Q) = \sqcap_{beh}\theta$, hence $imp_R(Q) \subseteq imp_R(P)$. **A:** By Proposition 1, $\exists \delta \in \Delta^*, R \in Class$ such that $P \xrightarrow{\delta}_I R$ and $imp_R(R) = imp_R(Q) = \theta$.

The second assumption is that whenever $P \dashrightarrow_I Q$ and $z \in bec(Q)$ then it is possible to construct P', Q' such that $P' \dashrightarrow_I Q'$, $bec(Q') = state(bec(Q))z$ and $bec(P') = state(bec(P))w$ for some $w \in bec(P)$.[11] Q' and P' differ from Q and P only in their initial states. The state of an instance is determined by the values of its instance variables. Hence, Q' is obtained from Q by changing the

[11] This assumption is neeeded because at the moment the inheritance mechanism is unrestricted allowing quite "*unnatural*" mechanisms.

initial values of some variables. Depending on the language this may invalidate the incremental transition between P to Q'. Hence, we require that it is always possible to change the initial values of variables in P (obtaining P') so that $P' \dashrightarrow_I Q'$. This assumption holds for all COOP languages we are aware of.[12]

Returning to point **A**, consider $v = u\hat{}\langle m\rangle \in bec(R)$ and construct R' such that $bec(R') = state(bec(R))v$. By assumption we have $P' \overset{\delta'}{\dashrightarrow}_I R'$ and $bec(P') = state(bec(P))w$ for some $w \in bec(P)$. However, $bec(R') = \{\langle\rangle, \langle m_1\rangle\}$, while $bec(P') = \{m_1, m_2\}^*$ (P has only one state). Hence, $bec(P') \not\sqsubseteq_T bec(R')$ and $imp_T(R') \not\subseteq imp_T(P')$. Therefore, the inheritance mechanism is not behaviour preserving. □

[12] If P and Q use constructors to specify the initial values of variables, then P' and Q' are simply constructed by modifying the two constructors without affecting the incremental relationship between them. If the initial state is specified by the initial values of variables (as in Figure 2), then any variable changed to obtain Q' is changed to the same initial value in P' (if the variable exists in P).