STUDIA UNIV. "BABEŞ-BOLYAI", INFORMATICA, Volume XLII, Number 2, 1997

# REUSE ANOMALY IN OBJECT-ORIENTED CONCURRENT PROGRAMMING

DAN MIRCEA SUCIU

**Abstract**. The integration of the concurrent mechanisms in object-oriented programming is without doubt, an attractive idea. Unfortunately, the design of some efficient programming languages that can achieve this thing is extremely difficult.

The primitives of communication and synchronization between concurrent activities interfere with features of the object-oriented programming causing the deterioration of some properties of some concepts that form the basis of those two programming techniques.

The inheritance anomalies represent the most intensively studied in the literature conflicts generated by such interference. A systematically analysis of these conflicts was not been done until now. Moreover, we want to prove that any of the classifications of these anomalies achieved until now are not complete.

We will show as well, that conflicts between concurrency and delegation or between concurrency and association are of the same type as the inheritance anomalies, and the key of these latter determines automatically the elimination of those other conflicts. In this way, we will suggest in the end the global treatment of all conflicts of this type (and not only of inheritance anomalies in particular) using the generic name of *reuse anomalies*.

## 1. Introduction

The integration of the concurrent mechanisms in object-oriented programming is without doubt, an attractive idea. Unfortunately, the design of some efficient programming languages that can achieve this thing is extremely difficult. The "efficient" attribute defines firstly the capacity of an object-oriented concurrent

Received by the editors: February 23, 1998.

1991 *Mathematics Subject Classification.* 68N15, 68Q10.

1991 *CR Categories and Descriptors.* D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages, methodologies*; D.3.2 [**Programming Languages**]: Language Classifications – *concurrent languages, object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *concurrent programming structures* .

programming language for owning the reunion of all advantages that are partially characteristic for every programming technique (concurrent, respectively object-oriented).

The problem of concurrency integration must be studied very seriously. A wrong approach of concurrent concepts can determine designing classes that satisfy only the necessities of a particular application. In this way, it will be difficult (even impossible) to reuse the respective classes within development of other applications. Combining concurrency with object-oriented mechanisms in object-oriented concurrent programming languages is not a simple process. In older researches (for example [YON87]) is highlighted the existence of some conflicts between inheritance and concurrency within the framework of such languages. These conflicts have determined many designers of object-oriented concurrent languages to exclude the inheritance from their languages or to implement mechanisms of coordination of flexible concurrent interactions independently of the hierarchy of inheritance.

The term of inheritance anomaly is used in the literature for the description of the conflict between inheritance and concurrency in the object-oriented programming languages, and it was used for the first time in [MAT90]. Though there were done many efforts for the analysis and elimination of these conflicts, the concrete solution to solve entirely this problem is still to wait for. The classifications of inheritance anomalies elaborate until now, are in our opinion unfinished, and the approach of these classifications is inadequate. Actually, the only papers that have had as subject exclusively the analysis and classifications of inheritance anomalies are [MAT93] and [ZEN97b]. The classifications of [MAT93][1] have been referred on many papers that have suggested new coordination mechanisms of the concurrent interactions integrated in an object-oriented environment.

Within the framework of this article, we will present the results obtained until now. In addition, we will analyze all constructions and mechanisms used to implement concurrency in concurrent object-oriented programming languages. Based on evaluation of these mechanisms, there will be determined the main causes which lead to the appearance of inheritance anomalies. A unified view concerning all these mechanisms as well can suggest ways of approach of some complete solutions, if these solutions exist.

Another goal of this article is that of getting a constructive criticism concerning previous classifications of the inheritance anomalies. We will also show that inheritance anomaly term is unsuitable. In this way we will prove that the mechanisms of initiation and coordination of the concurrent activities in object-oriented concurrent programming languages which exist at the moment, generate conflicts between concurrency and delegation, respectively between concurrency

[1] *An inheritance anomaly was defined in that paper as the phenomena through which the synchronizing code (in fact the coordinating code of the interactions between the concurrent activities) cannot be effectively inherited without nontrivial class re-definitions.*

and aggregation. The causes and the behavior of these conflicts are intimately related with those that determine inheritance anomalies. At the end of this paper, we will suggest a more general term for the description of these conflicts between concurrency and object-oriented primitives, namely the term of *reuse anomalies*.

In the second part, there are shortly presented the main concepts that are based on the object-oriented concurrent programming. In addition, a classification of the object-oriented concurrent programming languages through the viewpoint of the relationships between the concepts of concurrency and object is presented.

Moreover, an evaluation of the constructions used for the specification of interactions between the activities of the concurrent objects based on some criteria is achieved. Respecting these criteria is fundamental for the avoiding of the conflict appearance between concurrency and object-oriented programming concepts (the criteria are achieved based on considerations detailed in [PHI95], [MAT93], [FRO92] and [PAP89]).

Information is used in the third part, which is detached from the analysis of the concurrent constructions from the previous part, for establishing all cases in which inheritance anomalies arise. There are noticed all classifications of inheritance anomalies which exist until now, together with some critics concerning the way of approach of these classifications.

It is finally shown that the term of inheritance anomaly defines a particular conflict generated by the integration of the concurrency in an object-oriented context, and the term reuse anomaly is more suggestive.

## 2. Object Oriented Concurrent Programming

**2.1. General Aspects.** In the last two decades, the deșign of object models having concurrent features has represented a constant concern for many researchers. This was happening for mainly two reasons. On the one hand, as an effect of the obtained technological progress, many object-oriented programming languages having concurrent features have been designed during this time (over 100 such languages have been discussed and systemized in [PHI95]).

On the other hand, the fact is known that object-oriented programming has been developed having as a model our environment (seen as a set of objects among which several relationships exist and which communicate between them by message transmission). However, in the real world these objects are naturally concurrent, which leads to the normal trend of transposing this thing into programming.

It is interesting how two distinct criteria, the first one objective (determined by the rise of performances and complexities of the calculus systems), and the second one subjective (actually determined by "decency", which urges us to solve different abstract problems looking for similitude with the real world), have finally led to the development of some concepts, some programming techniques and implicitly of some efficient analysis and design methods for developing applications.

The concurrent programming has occurred before the object-oriented programming. It has been applied for the first time within the framework of procedural languages. Here the main problems studied have been concerned to the synchronization of the parallel execution of some instruction sequences and to the information transmission among many other concurrent activities.

Once with the appearance of object-oriented programming software development has met a qualitative and meaningful leap. In this way, the development of these programs (or applications) does not involve the decomposition of problems into algorithmic procedures, but independent objects that interacts among them. An evaluation of the coordinating primitives of these interactions will be achieved in a concurrent system.

2.2. **Evaluation of concurrency coordinating primitives.** A concurrent program (object-oriented or not) can put into execution parallel activities which do not interact among them. The activities are executed in this way, so that without knowing of each other and without a mutual influence. However, such programs are very rare. Very often, an interaction among activities is necessary for achieving some aims of the program. The interaction between two activities that are executed in parallel can arise in two cases:

- when the activities use some resource in common, as peripheral equipment, memory, buffer zones or variables, or
- when activities must cooperate in some way (for example, a certain activity uses results of some other activities).

Yet, the interactions among activities rise new problems that do not arise in the case of sequences programming or of independent concurrent activities. In this way, the simultaneous action of many activities concerning some shared resources for example, can lead mostly, either to unpredictable results, which come true by the softness of data, or even by unexpected endings of the concurrent programs.

The interaction between two activities became manifest under two ways: communication, respectively synchronization between activities.

Communication represents transmission of information among activities. It can be achieved by means of some shared variables (for the object-oriented concurrent programming this thing is achieved in the case of the intra-object concurrency) or through message transmission among activities (specifically for the inter-object concurrency).

Synchronization is achieved using restrictions on time evolution of an activity. Synchronization is used to avoid simultaneous access at a critical resource[2] (mutual exclusion) or for a postponed execution of an activity until the accomplishment of a certain condition (conditional synchronization).

[2] *critical resource* = *resource that cannot be used exclusively by a single activity*

The modeling of interactions among activities is done by means of some new primitives introduced in the language, because of "traditional" sequence programming are not adequate. These language primitives are generally represented in a programming language by means of diverse instruction or declaration modalities.

Primitives are generally common for both types of interaction, because they mutually involve themselves. In this way, communication between two activities involves the existence of a certain level of synchronization between them. That is because the operation of transmission of information by the first activity must precede the operation of receiving or taking over the information of the second activity. Mutually, synchronization of two activities involves the fact that the execution of an activity depends at a very moment on a certain information, or a specific action for the second activity. Information, respectively the achievement of the action can be known by the first activity, only in the case of communication to the second activity.

In the following, we will show the criteria that must be satisfied by the primitives of concurrent coordination in an object-oriented context. These criteria ensure that the specific qualities of concurrent programming or of object-oriented programming are not reduced or even cancelled. There has to be noticed from the very beginning that, unfortunately, none of the implemented primitives does completely comply with these criteria. The problem of inter-activities communication and mainly the problem of their synchronization are unsatisfactorily solved in the existent object-oriented concurrent programming.

The below enumerated criteria, have been described in [PHI95] and they have been determined on the basis of some previous studies included in diverse articles, among which: [MAT90], [MAT93], [FRO92], [PAP89] and [MEY93]:

**i)**: *Goal of Called-Oriented Coordination.* To guarantee an accurate programming, the complete specification of concurrent interactions must be implemented at the level of the called concurrent activity (or, more exactly, within the class which is concurrently accessed). If the principle isn't complied with, there is the risk of destroying the modularity of the classes, the activity (or object) which initiates the interaction has to know certain details of implementation, relatively to the called activity (object).

**ii)**: *Goal of Coordination Expressibility.* A primitive of interaction must allow the expression of certain types of condition, as follows:

- **a)**: *Intra-Object Concurrency.* The concurrent invocation of many methods of an object must be possible. The concurrent interaction primitives must offer the possibility of specification of methods that can be executed concurrent.
- **b)**: *State Proceed-Criteria.* The coordination primitives of concurrency must supply specification modalities of the calling opportunity of a method or the postponement of this, because of the non-observance of some conditions that depend on the intern status of the object.

**c):** *History Proceed-Criteria.* There must also exist a modality of specification whether a method can be called (the appealing being postponed) depending on methods of the object which have been called previously. This requirement of specification is included in b.), because there can be hold information concerning the history of the calling methods by simple addition of a member variable. However, because the conditions of synchronization dependant on the history of calling methods are frequently used in the object-oriented concurrent programming, the separate presentation of such a principle has been considered useful.

**iii):** *Goal of Isolated Coordination Code.* A concurrent interaction primitive must achieve the clear separation of the code that implements the functionality of a method to the adequate code of the concurrent interaction constraint. In the case of non-observance of this principle, the mechanism of inheritance can be affected, taking place the so called inheritance anomalies which will be described in detail in the next part.

**iv):** *Goal of Separable Coordination Code.* A mechanism of concurrent interaction must permit the separate inheritance of the interaction code.

In [SUC98] we have done an analysis of the characteristics of all primitives implemented in the object-oriented concurrent languages that still exist, related to the above stated criteria. Table 1 presents the synthesis of these results. The classification and description of all these primitives is not very important here. However, the main goals of the table presentation are those of suggesting the big amount of implemented primitives and for demonstrate that none of these completely follows the above-enumerated criteria. Details concerning the functioning of each mechanism of coordination treated in the table can be found in [SUC98].

## 3. Inheritance anomaly

One of the most important problems raised by the object-oriented concurrent programming is synchronization of concurrent objects. When a concurrent object is in a certain state, it can accept only one subset of the whole messages for maintenance of its internal integrity. This restrictions required on the object interfaces are named in the literature as *synchronization constraints* of the concurrent objects.

In the most object-oriented concurrent languages the responsibility of explicit code implementation within the framework of the methods regard to synchronization constraints, represent the developer's task.

For achieving this thing, the programmer must have at his disposal primitives for the implementation of synchronization and communication among the methods of objects. All types of primitives that were implemented in one of those over one hundred existing imperative object-oriented concurrent languages were presented

| Primitives | Calle-oriented Coordination | Coordination Expressibility | Isolated Coordination Code | Separable Coordination Code |
|---|---|---|---|---|
| Sync. by termination | No | a), b) | No | No |
| Semaphore, mutex, lock | No (sometimes yes) | a) | No | No |
| Conditional critical region | No (sometimes yes) | b) | No | No |
| Piggy-Backed sync. | No | b) | No | No |
| Monitor | Yes | - | Yes | Yes |
| Condition variables | Yes | b) | No | No |
| Conditional wait | Yes | b) | No | No |
| Delay Queue | Yes | b) | No | No |
| Include/exclude methods | Yes | b) | No | No |
| Behavior abstractions | Yes | b), c) | No | No |
| Actor model | Yes | a), b), c) | No | No |
| Method guard | Yes | a) (sometimes), b) | Almost yes | Yes |
| Enable set | Yes | b), c) | Almost yes | Yes |
| Path expression | Yes | a),c) | Yes | No |
| Life routine | Yes | a) (sometimes), b) | Almost yes | No |
| Generalized life routine | Yes | a), b) | Yes | Yes |
| (Un)Serialized Method | Yes | b) (restricted) | Yes | Yes |
| Reader/writer protocol | Yes | b) (restricted) | Yes | Yes |
| Reflective control | Yes | a), b) | Yes | No (poss. yes) |

TABLE 1

in the previous part. Likewise the previous part, we have suggested the fact that, in certain cases, the coordination code of the concurrent interactions cannot be inherited in fact without generating trivial redefinition of methods. This conflict has been identified and studied in many papers; in one of these papers, namely in [MAT90], this conflict was named as *inheritance anomaly*.

Three distinct cases have been here also identified, in which the utility of the inheritance concept is more diminished (in some particular cases even cancelled):

- the defining of a new subclass `K'` of the class `K` requires the redefining of its methods (the same thing available even for the descendants of the class `K'`)
- the modification of a new method m of the class `K` within the framework of the inheritance hierarchy involves the modification, apparently independent, of certain methods as well in class `K` as in its descendants.
- the defining of a method can force other methods (inclusively those that will be defined as subclasses in the future) to follow the specific protocol which wasn't necessary in the case in which the respective method wouldn't have to exist. The maintenance of the encapsulation property of classes will be therefore very difficult.

An important remark is that the communication and synchronization primitives of a specific language influence appearance of inheritance anomalies. It results from here that the problem of inheritance anomalies is generated by the semantic conflicts between the descriptions of synchronization and of the inheritance specific for a language, and not of the way in which the features of the language are implemented.

Two classifications of inheritance anomalies will be presented as follows, shown in [MAT93] and [ZEN97b]. We will prove further on, that these classifications are not complete. In addition, we will present another classification that won't take into consideration only the conflicts between concurrency and inheritance, but also the conflicts between concurrency and relationship of aggregation, respectively between concurrency and delegation mechanism. Based on this new classification, we will suggest a new naming which is more adequate for describing these two types of conflicts between concurrency and object-oriented programming concepts.

3.1. **Matsuoka-Yonezawa Classification.** This classification has been achieved and presented in [MAT93]. It was based on the demonstration of correctness of many object-oriented concurrent-programming languages ("correctness" interpreted through the viewpoint of the implementation of concurrent interaction and initiation mechanisms that was to hinder the appearance of inheritance anomalies). Unfortunately, although the Matsuoka-Yonezawa classification is praiseworthy to be the fruit of a laborious analysis of the conflicts between inheritance and concurrency, it has a lack generated by the modality in which this analysis has been approached.

In this way, the appearance of anomalies in diverse situations was demonstrated using examples; but the examples do not represent an adequate basis for classification. Therefore, it doesn't exist any guarantee that the classification is correct or complete. Particularly, such a classification cannot be used for demonstrating the absence of the inheritance anomalies within the framework of a programming language.

On the other hand, the example studied in [MAT93] hasn't been presented in a general context, but using particular mechanisms of specification of the concurrent interactions. In this way, the used mechanisms were: *explicit receiving of messages*, *path-expressions*, *life routines*, *behavior abstraction* [THO94], *enable-set* and *guarded methods* [FER95]. Nevertheless, as we have shown in the previous part, the multitude of the developed mechanisms is much greater.

To present clearly the categories of inheritance anomalies determined in [MAT-93] was used a classical example, namely the implementation of a bounded buffer (figure 1). In addition, Matsuoka and Yonezawa used the concept of abstract state of an object. For achieving a concise and suggestive presentation of both, we will resort to the modeling of the behavior of the object, using the statecharts, described by Harrel in [HAR86]. At these statecharts we have attached many formal annotations for the specification of the objects consistency (in figure 2 is presented as example the behavioral model of the `Buffer` class).

A first category of inheritance anomalies is that determined by the *partitioning of the abstract states*. In this way, let's suppose that the specialization of the **Buffer** class (figure 1) is wanted by addition of a new method, **Get2()**, which has to extract simultaneously from the buffer two elements (the first one).

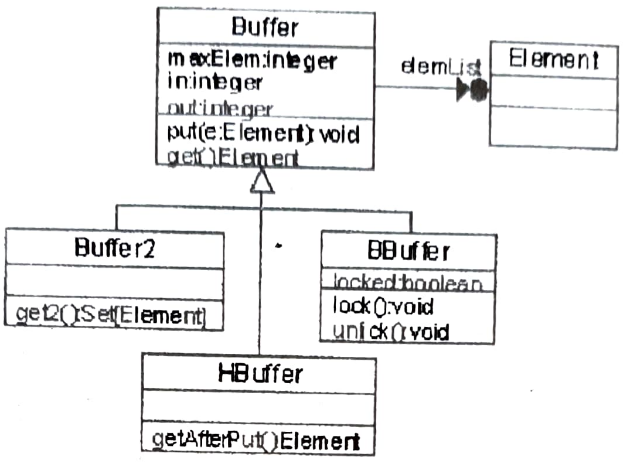

FIGURE 1. Hierarchy of classes used by Matsuoka-Yonezawa for classification of inheritance anomalies

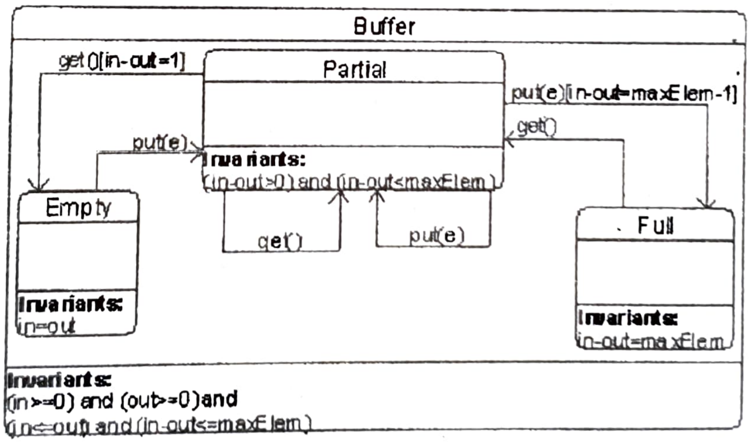

FIGURE 2. Behavior model of class **Buffer**

The behavior model of such a class, named **Buffer2**, is presented in figure 3. Here it's shown the **Partial** abstract state, characteristic for the **Buffer** class, which has in this case two sub-states, **Partial2** and One (actually the **Partial** state is replaced in the behavioral model by those other two states).
In examples presented in [MAT93] is shown that the addition of method **Get2()** in the **Buffer2** class using, for instance, enable-sets mechanism, implies the redefining of all methods of the **Buffer** class. That is because the partition of the **Partial** state must be treated by all these methods. On the other hand, it has been shown

that the mechanism with guarded methods doesn't lead to the appearance of this type of inheritance anomaly.

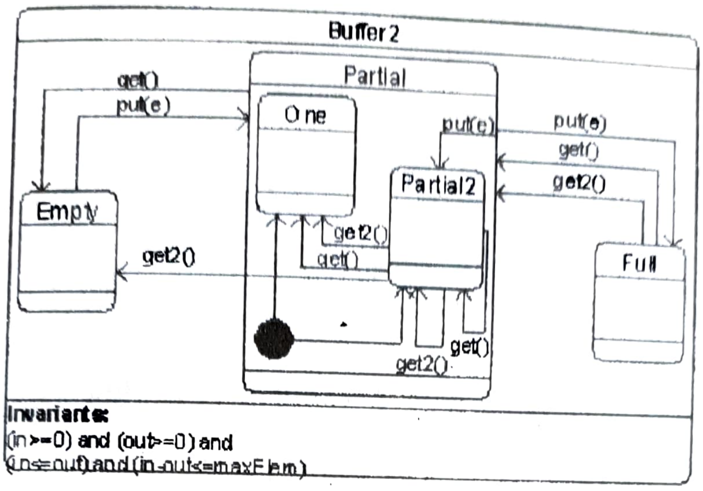

FIGURE 3. Behavior model of class **Buffer2**

A second category pointed out in [MAT93] is that of the inheritance anomalies determined by the history sensitiveness of states. The example that was studied by the authors of the paper, is that of a subclass, **HBuffer**, which contains a method **getAfterPut()** which can be executed only if the previous accepted method was **Put()**. So the method **getAfterPut()** won't be immediately executed, after the acceptance of a **Get()** method or **getAfterPut()**). This situation needs the addition of a new member variable (in the case of the mechanism of behavior abstractions or the enabled sets). The model of behavior presented in figure 4 is available for both cases.

For the enabled sets, re-definitions are necessary, because the new state must be taken into consideration by all methods. In the case of methods with guards, there are necessary considerable re-definitions because the suitable setting of the value of the new variable has to be achieved in all methods. Yet, in this last case, the cognition of the implementation of the redefined methods is not necessary, so there doesn't take place a violation of modularity.

The last category of inheritance anomalies is that determined by the modification of the abstract states. The shown example for defining this type of anomalies, consists of the possibility of obstruction the extraction and addition of the buffer elements. In this way the state of an object won't depend only on the numbers of buffer elements, but also on the value of the member variable locked (figures 1 and 5).

The blocking and release of the buffer is achieved by means of the **Lock**() methods, respectively **Unlock**() of the new created **BBuffer** class. This case was also implemented using the mentioned mechanisms of synchronization. In this way, it is necessary (in case of enabled sets) to add a new state. This addition

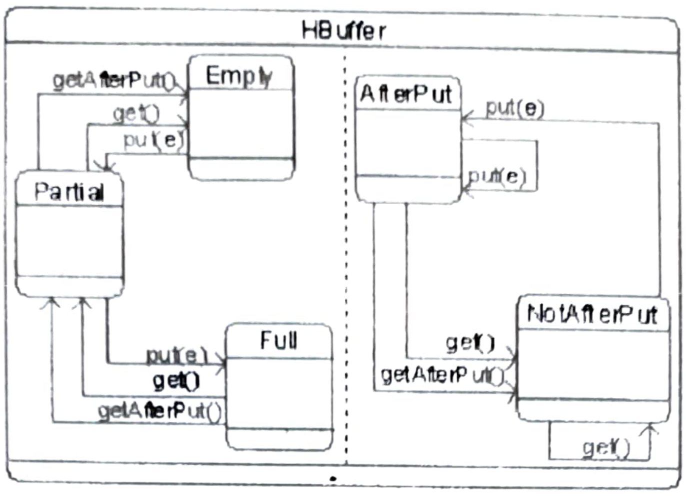

FIGURE 4. Behavior model of class **HBuffer**

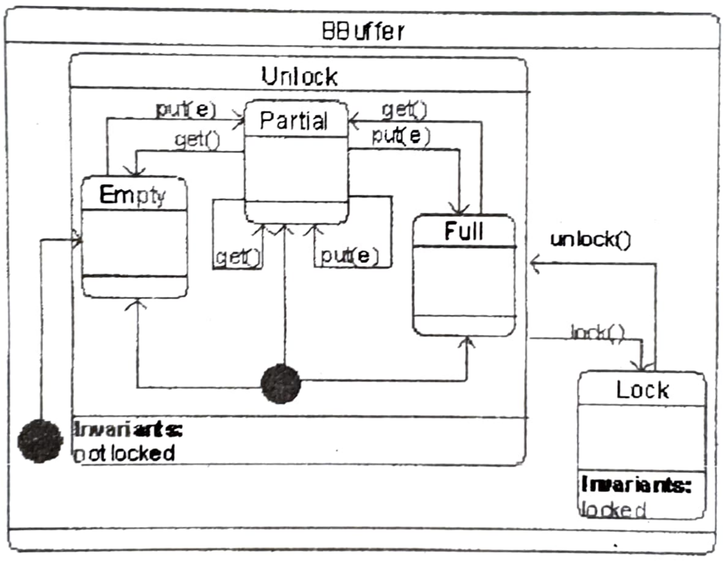

FIGURE 5. Behavior model of class **BBuffer**

involves the appearance of the inheritance anomalies, as you could see in the previous example. In the case of guarded methods, the modification of the guards' conditions is necessary for taking into account the new member variable locked. Therefore the redefining of the `Put()` and `Get()` methods is necessary without any altering of their functioning.

The classification of the inheritance anomalies achieved by Matsuoka and Yonezawa represents a first systematic determination attempt of the causes that lead to the appearance of inheritance anomalies. This classification has unfortunately no theoretical basis, basing strictly on the test of the cases where the multitude of abstract states of a class is altered in its descendants. Therefore, there doesn't exist any guarantee that the respective classification is complete or correct, its achievement depends strictly on the authors' "decency" and experience.

| Extensions | New method | Overriding method | Refers to parent method |
|---|---|---|---|
| E1 | No | Yes | No |
| E2 | No | Yes | Yes |
| E3 | Yes | No | No |
| E4 | Yes | No | Yes |
| E5 | Yes | Yes | No |
| E6 | Yes | Yes | Yes |

TABLE 2. Extension possibilities for a class

More than that, there doesn't considered the case which leads to an inheritance anomaly without modifying abstract states in a subclass. For above examples and using certain particular coordination mechanisms, the building of a new subclass of the Buffer class that contain a method which extracts the first element from buffer, leads to unjustified redefining of all methods of the superclass. This thing can be easily demonstrated using the mechanism of concurrent interaction with semaphores, for example.

3.2. **Zeng-Schach Classification.** The classification suggested by Zeng and Schach has been relatively recently achieved, in [ZEN97]. They suggest here an analysis of the way in which the inheritance functions, more precisely of the way in which the subclass can extend the superclass without taking into consideration either mechanism of particular concurrent interactions.

Syntactically, a subclass can extend the parent classes in three ways: by defining a new member variable, by defining a new method or by rewriting the inherited methods. Although the three modalities of extension are orthogonal, the addition of a new member variable involves logically the defining or re-defining of less than a method with which has to actuate. That's why Zeng and Schach considered that the appearance of inheritance anomalies is bound to the last two extension modalities of a superclass. This approach is totally different of that in [MAT93], because the inheritance anomalies are not watched, related to the abstract states sets of an object, but with the new defined methods of this.

More than that, the situations are taken into consideration where other methods of an object are mentioned within the framework of a new defined method. In table 2 there are presented the analyzed cases in [ZEN97] which have been considered as potential situations for generating non-trivial re-definitions of methods.

This fact represents one of the drawbacks of the approach. Namely the fact that has been taken into consideration only the problem of redefinition of methods and another aspect has been ignored, the characteristic aspect of inheritance anomalies, namely that of violation of class modularity.

Another drawback is that the analysis of those six cases presented in table 2, takes into consideration only the inter-object concurrency, considering that we deal with objects which own only one activity in execution at a very moment. Therefore, either in this case, it cannot be asserted that the results of the analysis lead to a complete classification.

The result of the analysis is not an astonishing one; it asserts the existence of two categories of anomalies. The first category, named the category of anomalies of forced overriding, is determined by the necessity of redefining of one or more methods in a subclass generated by definition of a method within the framework of this. The second determined category is that of the synchronization anomalies, which refers to the impossibility of calling a method because of its synchronization constraints.

The anomalies of the first category seem to correspond to the inheritance anomalies of the Matsuoka-Yonezawa classification. The here-achieved analysis doesn't permit the determination of those three subcategories, which was mentioned in the previous part.

On the other hand, the synchronization anomalies are not characteristic for the object-oriented concurrent programming. As the authors even asserted, these anomalies are present also in non object-oriented environments, and the problem is very similar with nested monitor call problem.

Therefore, the classification introduced here doesn't bring anything new. More than that, the source of anomaly appearance is not very clear spotlighted. This thing is due firstly to the approach of the problem that doesn't take into consideration particular implementations of mechanisms of concurrent interaction.

3.3. **Reuse Anomaly.** The concept of inheritance anomaly has been initially introduced for describing the conflicts that appear between concurrency and inheritance mechanism of the object-oriented programming. However, in the time, as can be seen in diverse articles with these topics in the last years, inheritance anomalies have been considered as representing the only main conflict proceeded from the fusion between concurrency and object-oriented concepts. This thing is totally wrong, because, as we will present further on, anomalies with a similar behavior take also place in the case of other mechanisms characteristic for the object-oriented programming. [ZEN97] represents one of the few papers where was taken in account other anomalies, not just those related with inheritance. Unfortunately, as we have seen, the achieved studies in this paper lead unsatisfactory conclusions. Although the forced overriding anomalies introduced in [ZEN97] seem to have a more general character, they replace the concept of inheritance anomaly only on the level of naming.

Between the classes that describe a specific problem may exist four types of relationships. A first type of relationship is that of inheritance which allows, as it is well known, the reuse of the structure and function of a class in the definition of another class.

There exist programming languages (as in the SINA example) which haven't implemented a mechanism for the rewriting of such a relationship. For simulating the inheritance in one of these languages, the mechanism of delegation is used. The relationship of delegation between two classes is also a reuse relationship and it supposes the potential redirection of received messages by an object to a so-called delegated object. The delegated object must be effectively included in the construction of the first object. The delegation is a stronger relationship than the inheritance, because it can stimulate this relationship, and more than that, it can model the dynamic evolution of systems (thing which the inheritance, as a static relationship between classes, cannot achieve).

The association relationship is the third possible relationship where two classes of a system can be present. This relationship is also named as reuse relationship, because at a very moment the object of a class is used by the services offered by an object of the associate class.

The forth relationship between the classes of a system is the aggregation relationship. It involves the existence of relationship of the type whole-part between the instances of classes being present in this relationship. This relation was considered by some authors as a particular type of association.

The four above described relationships also represent many ways of reuse of the defined classes in a certain library. We will show as follows that in a concurrent context, the existence of one of these relationships can determine in certain conditions non-trivial re-definitions of methods as serious encapsulation violation. Therefore, exactly the considered effects at the description of inheritance anomalies in [MAT93].

The relationships of delegation, aggregation and association involves at least, the existence of a communication between objects, communication which is based on the access or calling of public variables and methods of an object. The access at the public member variables of this one in a concurrent context and in the absence of an adequate synchronization mechanism can determine the bringing into a firmness state of the object. Therefore, methods or objects level which access the public structure of an object found in a relationship of aggregation, association or delegation, must be found communication and synchronization mechanisms for protection of the consistency of data. For the most of these mechanisms (as it has been demonstrated also in the case of inheritance anomalies) this thing supposes the knowledge of implementation of methods (public or not) from the object of which structure is accessed. Therefore the encapsulation of classes can be destroyed, and the reuse of classes which belongs to a library of classes, is once again burdened. More than that, the modification of classes within the framework of a library of classes involves the taking into consideration of the code of all classes of which it is in an above-mentioned relationship.

The fact is obvious that in the cases of associations and aggregations, the anomalies of this type can be eliminated by the consideration of all member variables of a class being private member variable.

However, the most of existent object-oriented programming languages allow the defining of many kinds of visibility for the member variables. More than that, in some languages (for example C++) the methods may return references of member variables. In this way the state of an object can be altered in the same manner as using public member variables (and involve the same anomalies).

Because of the obvious resemblance between this type of anomalies and that known in the literature as inheritance anomalies, we think the unitary treatment of them is more natural than the finding of a general naming of defining them. The most adequate naming seems to be that of reuse anomalies.

The criteria mentioned in the third part of this paper represents necessary conditions for avoiding the appearance of reuse anomalies and the untouched quality keeping of the introduced concepts as well the concurrency mechanisms as the object-oriented technique. Unfortunately, there hasn't been realized a communication and synchronization mechanism of the concurrent activities which are to satisfy all these principles.

## 4. Conclusions

The problems generated by the implementation of concurrency within the framework of the object-oriented programming haven't yet found a solution. The objective models developed until now do not satisfy all criteria which provide a correct and efficient programming.

We have shown in this paper that a rigorous analysis of the conflicts between the mechanisms of concurrent interaction specification and the concepts of object-oriented programming hasn't been achieved until now. The inheritance anomalies have been defined as a representation of conflicts between inheritance and concurrency. These anomalies are characterized by the re-definitions of methods and by the destruction of encapsulation.

We have also demonstrated that these are not the only types of conflicts specific for the inclusion of concurrency within an object-oriented framework and we have introduced the term of reuse anomaly. This term seems more adequate and offers the possibility of treatment of these conflicts on a higher level.

## References

[BRI93] Jean-Pierre Briot, *Object-Oriented Concurrent Programming: Introducing a New Programming Methodology*, Proceedings of the 7th International Meeting of Young Computer Scientists, 1993.

[ELE91] Petru Eles, Horia Ciocârlie, *Programarea concurenta în limbaje de nivel înalt*, Editura Stiintifica, Bucuresti, 1991.

[FER95] Szabolcs Ferenczi, *Guarded Methods vs. Inheritance Anomaly / Inheritance Anomaly Solved by Nested Guarded Method Calls*, SIGPLAN Notices 30(2): 49-58, 1995.

[FRO92] Svend Frolund, *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*, Proceeding of ECOOP'92, L. Madsen editor, Lecture Notes in Computer Science, vol. 615, pp. 185-196, Springer-Verlag, Utrecht, Netherlands, 1992.

[HAR86] David Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming, North-Holland, 1986.

[MAT90] Satoshi Matsuoka, Ken Wakita, Akinori Yonezawa, *Synchronization Constraints With Inheritance: What Is Not Possible - So What Is?*, Technical Report 10, Department of Information Science, University of Tokyo, 1990.

[MAT93] Satoshi Matsuoka, Akinori Yonezawa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, Research Directions in Concurrent Object-Oriented Programming, pp.107-150, MIT Press, Cambridge, 1993.

[MIT94] S. E. Mitchel, A. J. Wellings, *Synchronization, Concurrent Object-Oriented Programming and the Inheritance Anomaly*, Jun. 1994.

[PAP89] Michael Papathomas, *Concurrency Issues in Object-Oriented Programming Languages*, in D. Tsichritzis, editor, Object Oriented Development, pg. 207-245, University of Geneva, Switzerland, 1989.

[PAP97] M. Papathomas, Anders Andersen, *Concurrent Object-Oriented Programming in Python with ATOM*, Proceedings of the 6th International Python Conference, California, Oct. 1997.

[PHI95] Michael Phillipsen, *Imperative Concurrent Object-Oriented Languages*, Technical Report TR-95-049, International Computer Science Institute, Berkeley, Aug. 1995.

[SCU97] Marian Scuturici, Dan Mircea Suciu, Mihaela Scuturici, Iulian Ober, *Specification of active objects behavior using statecharts*, Studia Universitatis "Babes Bolyai", Informatica, Vol. II, Nr. 1, 1997.

[SUC97] Dan Mircea Suciu, *Limbaje de programare orientate obiect concurente*, PC REPORT, Aug. 1997.

[SUC98] Dan Mircea Suciu, *Anomalii de mostenire in programarea orientata-obiect concurenta*, referat de doctorat, mai 1998.

[THO94] Laurent Thomas, *Inheritance Anomaly in True Concurrent Object Oriented Languages: A Proposal*, Research Report, University of Tokyo, Department of Information Science, 1994.

[YON87] Akinori Yonezawa, Jean-Pierre Briot, *Inheritance and Synchronization in Concurrent Object Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Lecture Notes in Computer Science, no. 276, pp. 32-40, Springer - Verlag, 1987.

[ZEN97b] Nanshan Zeng, Stephen R. Schach, *A New Approach to the Inheritance Anomaly*, submitted for publication, 1997.

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

*E-mail address*: tzutzu@cs.ubbcluj.ro