Special Issues in Object-Oriented Programming

Workshop Reader of the 10th European Conference on Object-Oriented Programming
ECOOP '96
Linz
# Table of Contents

I: Adaptability in Object-Oriented Software Development 5

M. Aksit, B. Tekinerdogan, L. Bergmans, K. Mens, P. Steyaert, C. Lucas, K. Lieberherr

Adaptability in Object-Oriented Software Development: Workshop Report 7

B. Tekinerdogan, M. Aksit

Achieving adaptability through separation and composition of concerns 12

M. Aksit, B. Tekinerdogan, L. Bergmans

Right Abstractions on Adequate Frameworks for Building Adaptable Distributed Applications 24

B. Garbinato, P. Felber, R. Guerraoui

Adaptability in Object-Oriented Embedded and Distributed Software 29

T. Ihme, E. Niemelä

Reuse Contracts: Managing Evolution in Adaptable Systems 37

K. Mens, P. Steyaert, C. Lucas

Adaptation through End-user Tailoring 43

A. March

II: Composability Issues in Object Orientation (CIOO '96) 53

L. Bergmans, P. Cointe

Composability Issues in Object Orientation: CIOO '96 Workshop Report 55

L. Bergmans, P. Cointe

Aspect Oriented Programming 63


An Introduction to Composability Issues 75

L. Bergmans

Research Topics in Composability 81

C. Lucas, P. Steyaert, K. Mens

From Inheritance to Feature Interaction 87

C. Prehofer

Composition through Superimposition 94

J. Bosch

Issues in Composability of Synchronization Constraints in Concurrent Object-Oriented Languages... 102

F. Sánchez, J. Hernández, M. Barrena, J. M. Murillo, A. Polo
Issues in Composability of Synchronization Constraints in Concurrent Object-Oriented Languages

Fernando Sánchez, Juan Hernández, Manuel Barrena, J. M. Murillo & Antonio Polo

Computer Science Department, University of Extremadura, Spain
e-mail: {fernando, juanher, barrena, juannamu, polo}@unex.es

Abstract

In the last few years, the so-called inheritance anomaly has been one of the issues that has attracted a great deal of research in concurrent object-oriented languages (COOLs). However, although some apparently promising approaches have been proposed, most of them do not give a complete solution to this problem. Furthermore, these approaches do not favour the extensibility and reusability of synchronization constraints (SC), because of the different aspects involved, namely behaviour, data and synchronization, are not cleanly/clearly separated. For this reason, composability in COOLs is gaining more interest. In this paper we analyze the problems arising in composing and reusing SC through inheritance. Moreover, we propose a solution that avoids the classical inheritance anomalies and that allows the programmer to specify the SC in a composable way.

1 Introduction

One of the main reasons of the success of the object-oriented model is the high degree of reusability for building and maintaining large and complex systems. To make feasible this reusability, techniques such as part-of relations, delegation and inheritance are used. Inheritance is by far the best-known and most widely used technique for structuring O-O software. The reasons for this are twofold. First, inheritance is easily applied by programmers with experience in sequential O-O languages. Second, it allows to reuse composable aspects of objects such as behavior and data structures.

Nevertheless, a new composable aspect appears in COOLs: synchronization constraints, but it is widely known the interference between inheritance and concurrency in COOLs [3, 5, 8, 12]. Consequently, SC become an essential composable aspect in

1. This research has been done with the support of the Comisión Interministerial de Ciencia y Tecnología, Spain, under contract TIC96-0551.
these languages. Precisely, the analysis of various issues in composability of SC through inheritance in the area is the main goal of this paper.

The rest of the paper is organized as follows. In section 2 we describe required features in the design of COOLs related to composition of SC. Section 3 shows the main reasons because previous proposals fail in their attempt to integrate these features. In section 4, our proposal will be outlined. Finally, we present our conclusions in section 5.

2 Required features when composing SC

In this section we describe some features that are not only desirables but also actually required in the composition of SC in COOLs in order to obtain a higher degree of code re-use and software adaptability [3, 6, 12]. We claim that previous proposals do not pay enough attention to an effective integration of the features that we specifically describe below:

**Reusability and Extensibility**: Does the language provide support for reusing the SC already defined in the definition of new ones within the same hierarchy? This is an important property because most of the SC in a superclass are applicable in its subclasses. Moreover, SC surely will need to be extended in order to cope with the new requirements imposed in subclasses. As a consequence, it is necessary to be able to re-use all kinds of codes, that is, not only methods, but also SC.

**Polymorphism**: Does the language provide support for reusing the synchronization constraints defined for a class in the definition of new ones in different hierarchies? Polymorphism of SC [12] is useful when a synchronization policy may be applied to different classes. For instance, the so-called concurrent read-exclusive write is a common synchronization policy widely used in many application domains, and the policy itself is independent of the specific domain.

**Substitutability**: Do the changes in the implementation of a class induce changes in the synchronization constraints? Sometimes we want to introduce changes in the implementation of a class, e.g. for performance reasons. These changes should not implicate changes in SC. In other words, we are looking for the adaptability of software in the area of COOLs.

Whatever solution integrating successfully the mentioned features must also take into consideration the two issues we describe below.

- When designing a class library we cannot make assumptions about the possible application domain where such classes are going to be reused. For example, the behavior of a bounded buffer is the same in sequential and concurrent environments. Composition of SC must provide support for using the behavior of the

---

1. An extended version of this work may be found in [6]
bounded buffer in either sequential or concurrent environments. A clean separation between SC and basic object behavior allows this issue to be fulfilled.

• Another issue to be addressed in the composition of objects is the potential re-use of different parts of the object. Sometimes programmers are not interested in reusing the entire object but only one aspect from it, e.g. synchronization constraints. Flexibility must be provided for reusing independent composable aspects. In other words, we are looking for mechanisms favouring the composability of software in COOLs.

The main contribution of this paper is not only identifying why prior proposals fail in the integration of all the features described above, but also establishing the requirements for its effective integration. Furthermore, we propose a solution that avoids the classical inheritance anomalies and easily integrates the mentioned features.

3 Problems in other proposals

Most of the efforts to compose SC through inheritance have been mainly oriented towards the re-use and extension of SC within the same class hierarchy by ignoring polymorphism and substitutability. The aim of this section is to identify the problems because other proposals do not integrate these features.

• **Reusability and Extensibility**: Since the SC are members of the concurrency protocol of a class, then a derived class must be able to reuse and extend the SC of its superclass. Proposals allowing reusability and extensibility in different ways are [3, 4, 8, 13]. The common factor in them is the separation of synchronization code from the method bodies. Otherwise, inheritance anomalies would appear [7, 14].

• **Polymorphism**: Matsuoka et al. [8] offer a solution to solve inheritance anomalies with a high degree of encapsulation and re-use for the synchronization code in subclasses. However, their proposal disallows polymorphism of SC because the definition of SC is highly attached to the methods of the class to be synchronized. Another inconvenience to achieve polymorphism that also appears in this proposal is the use of instance variables in the specification of SC.

DRAGOON [2] is a language providing polymorphism of SC by using generic synchronization classes. However, it disallows extensibility of SC because inheritance between classes specifying synchronization policies is forbidden.

Another attempt to integrate polymorphism is made by McHale in [9] through the use of Generic Synchronization Policies (GSPs). However, this proposal also presents deficiencies when SC needs to be expressed in terms of instance variables. Since references to instance variables are not allowed in this proposal, then they must be duplicated in synchronization code which leads inevitably to inconsistencies in presence of intra-object concurrency [6].
L. Bergmans extends the composition filter model for composing concurrent objects in a new version of Sina [3]. Although in [3] Bergmans states that his model supports polymorphism, he does not exemplify it.

We state that polymorphism may be achieved by providing a clean separation between methods and SC. In other words, the separation of the involved concerns: behaviour and synchronization [1].

- **Substitutability**: although this is one of the main goals of the object-oriented paradigm, it has been surprisingly ignored in most of the proposals in COOLs. Any solution where the specification of SC involves instance variables presents a great lack of substitutability and, consequently, software adaptability is lost. Matsuoka [8] and Frølund’s [4] are two significant proposals disallowing substitutability.

In consequence, the use of instance variables in the synchronization code leads to anomalies in both polymorphism and substitutability. If instance variables are needed in SC, then a suitable mechanism must be provided.

**Specification of Synchronization Constraints**

Methods and Synchronization Constraints are specified in the same class definition. Separate specification of Synchronization Constraints and Methods

**Problems:**
- It is not possible to reuse SC among different hierarchies (polymorphism of SC)
- Reuse of SC implies forced inheritance of methods.

**Advantages:**
- Promotes reusability, extensibility, substitutability and polymorphism of SC.

*Exemplars: {2,3,6,9,11,13}*

**Ways to access the state of the object**

**Instance Variables in SC?**

Yes

**Advantages:**
- Promotes reusability, extensibility and substitutability.

*Exemplars: {6,9}*

No

**Problems:**
- Classical inheritance anomalies.

*Exemplars: {7,10,14}*

**Functional Access**

*Exemplars: {3,6,11,13}*

**Extra Variables for synchronization purposes**

*Exemplars: {2,6,9}*

**Synchronization Code embedded in method bodies?**

Yes

No

**Fig. 1 Specification of Synchronization**

Two approaches have been proposed to solve this problem (see figure 1). A first approach is to provide a functional access to instance variables from SC. Although it is an interesting approach, it turns out to be incomplete because it is impossible to rep-
resent all of the possible states of an object merely by means of instance variables (e.g. consider the history-only sensitiveness of states [8]).

A second approach uses extra variables to maintain an abstract state of the object that is separated from the current class implementation and is used purely for synchronization purposes. However, it becomes useless when the number of possible object states is considerable.

To summarize, in the previous discussion we have identified the primary causes of why prior proposals do not allow for an effective reusability, extensibility, substitutability and polymorphism of SC, that being because most of them involve inheritance anomalies:

- The presence of instance variables in SC. ([4, 7, 8, 14]).
- Methods and SC located in the same class. ([4, 7, 8, 10, 14]).

Then the main goals to be achieved when composing SC in COOLs through inheritance are:

i. The absence of instance variables in synchronization constraints.

ii. A clean separation between methods and synchronization constraints.

4 Our proposal

The main concepts of our proposal rely on abstract state information and abstract classes. Whereas abstract states allow the absence of instance variables in SC, abstract classes allow the separation of SC and basic object behavior in different components, that is, the two goals we are looking for in order to achieve composability in COOLs.

In the proposal, SC restrict method invocations when the object is in a certain state, which we call abstract states. Thus, the abstract states of an object are those exceptional states or conditions under which certain methods cannot be executed. If instance variables are needed in abstract states then a functional access is provided. In those cases where instance variables are insufficient to maintain the object state, the solution lies in introducing variables only for synchronization purposes
Fig. 2  A graphical representation of our framework

The outline of our framework is illustrated in figure 2. When a message arrives the state of the object must be checked in order to decide if the message can be accepted. This is made by means of PreConditions that are based on abstract states. PreConditions are like guards and they implement a mapping from the abstract state of the object to a finite set of conditions. If PreConditions are not satisfied, then the message is stored in the request queue which is continuously checked by an object manager as in [3, 13]. When a message satisfies PreConditions, then it can performs PreActions and PostActions to handle synchronization variables related to abstract states. PreActions and PostActions are actions to be executed respectively before and after real method invocation. For a specific method, the execution of both PreCondition and corresponding PreActions is atomic in order to preserve data integrity.

As an illustrative example, in figure 3.a we show the classical bounded-buffer example.

```c
class b_buf {
public:
    void put (int elem);
    int get (void);

  synchronization:
    int length (void);
  Abstract States:
    FULL: (length() == MAX);
    EMPTY: (length() == 0);
  PreConditions:
    put: not FULL;
    get: not EMPTY;

  implementation /*
    ... //implementation of methods */
};
```

Fig. 3a  Bounded-buffer example
The reader may observe that the SC do not contain instance variables thanks to the use of functional access in abstract states. It must also be noted that the synchronization policy is completely separated from object behavior. So, we can think in separating both aspects at different classes. In figure 3.b, SC are defined in an abstract class that may be composed through inheritance with any class that implements the abstract methods specified in the abstract class. Besides the inheriting class may redefine the name of methods in order to make the abstract class as much polymorphic as possible.

Abstract classes for SC lead to the achievement of the two other topics mentioned in section 2: no assumptions about the possible reuse of the object (sequential or concurrent) and the potential reuse of different parts of the object, SC and behavior in this case. Since these parts may be used separately to be components of other objects, composability of software in COOLs is achieved.

5 Conclusions

This paper analyzes several issues in composability in concurrent object-oriented languages through inheritance, namely reusability, extensibility, substitutability and polymorphism of synchronization constraints. Moreover, we identify why prior proposals do not make an efficient integration of these properties.

The presented mechanism achieves these features and favours software adaptability and composability of SC because of the use of abstract state information to separate the involved concerns: synchronization and basic object behaviour.

6 Acknowledgements

We would like to thank Michael Papathomas for helpful comments about the extended version of this paper.

References


