Adaptability in Concurrent Object Oriented Languages

FERNANDO SÁNCHEZ, JUAN MANUEL MURILLO, JUAN HERNÁNDEZ, ALBERTO GÓMEZ
Department of Computer Science
University of Extremadura
Avda. Universidad s/n, 10071, Cáceres
SPAIN

Abstract: - Adaptability and composability have become two of the most important research areas in concurrent object-oriented systems in recent years. Whereas adaptability tries to cope with system evolution by adding/replacing components, composability tries to facilitate system building by placing together different components or pieces of software. Nevertheless, at the present time, concurrent object-oriented languages do not provide enough support for the development of true adaptable and composable software because either i) the different aspects that appear in these systems, synchronization and behavior, are mixed in the same component or, ii) if they are properly separated in different components, once these components are woven, the resulting executable piece of software is too rigid to be adapted or reconfigured at run-time. In this paper, we distinguish different levels of synchronization and present two models mainly thought for a clear and consistent separation of the synchronization aspect from the behavioral aspect. Whereas both models allow behavioral code to be written in a standard language like Java, they provide a different language for the specification of synchronization components and a third language for the specification of the composition rules between the synchronization and behavioral components. This composition language allows synchronization policies to be added, replaced or reconfigured at run time, which is the main contribution of the proposed models. The techniques here presented have been satisfactorily integrated in Java.

Key-words: Concurrent object-oriented programming, adaptability, composability, synchronization, coordination.
CSCC99: Proceedings: pp.7511-7516

1 Introduction

Frequently, large and complex systems have to cope with a continuous change of requirements during their life. Consequently, these systems tend to grow and change after they have been developed. In order to adapt to changing requirements, the demand has risen for adaptable software. This research area has been named the adaptability of software, which can be defined as the ability to deal with new/changing requirements with minimum effort. In this definition, the meaning of effort must be understood as the number of modules that are created or modified when changes in the requirements are necessary.

Adaptability is not a new area of interest. The different programming methodologies proposed so far have had, among their main goals, the development of easily maintainable and modifiable software.

In this sense, object-oriented programming has proposed abstraction mechanisms such as classes, objects, interfaces, methods (procedures, functions) etc. However, many systems have properties that don't necessarily align with the object-oriented system's functional components. Synchronization constraints, persistence, distribution protocols, replication, coordination, real-time constraints, etc., are aspects of a system's behavior that tend to cut-across groups of functional components. Consequently, programming them using current object-oriented languages results in spreading aspect code through many components, and the source code becomes a tangled mess of instructions for different purposes (See A_1-A_4, C_1-C_4 in Fig. 1).

Under this situation, dependencies between components are increased, opportunities for programming errors are introduced, and components become less reusable. In short, the source code is difficult to reason about, to develop and to evolve. The introduction of new requirements affecting a single aspect may involve the modification of other aspects, thus expanding the number of components

that must be changed or modified. And this situation is clearly against the definition of adaptability given before[1].

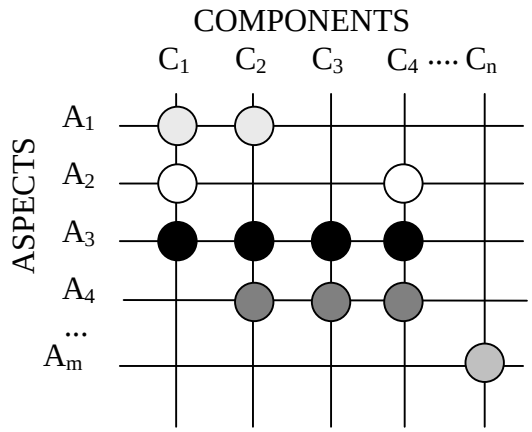

Fig.1 Relationship between aspects and components

In our research group we are working with the aspect of synchronization. In this paper, several levels of synchronization will be distinguished and two models to cope with them will be briefly presented.

2 Separation of Aspects

Nowadays, the separation of concerns or aspects from functional code has been recognized as an effective approach for increasing adaptability, that is, for solving the problem stated in the previous section [1, 4, 12]. This ideal situation is illustrated by C_n and A_m in fig. 1.

During last years, synchronization has been one of the most analyzed aspects in order to be separated from functional code.

The synchronization of an object can be seen at three different levels of abstraction that cover all the possible spectrum of object interactions (fig. 2).

- Firstly, the level of *synchronization among internal threads*, that is, mutual exclusion among threads, what will be the next thread to be executed, etc.
- Secondly, the level of *synchronization for incoming messages* according to the current state of the object, that is, if, when a message arrives, it can be executed, rejected or delayed.
- Finally, the level of synchronization between different objects or *multi-object coordination* can be distinguished.

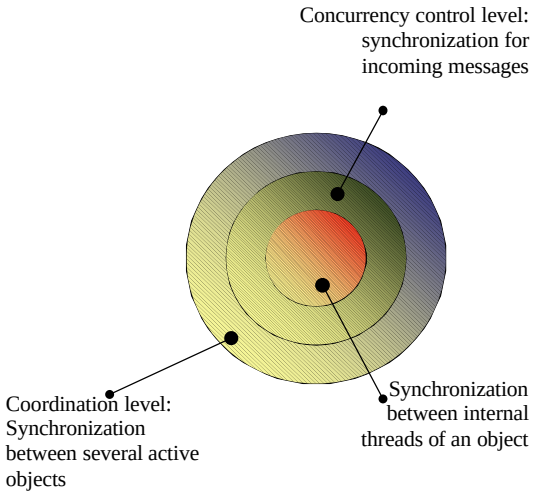

Fig. 2 Different levels of synchronization

In the following we refer to synchronization for the two internal levels in fig. 2 and coordination for the external one.

In the last few years there have been several proposals trying to separate the synchronization [2, 7] and coordination [3] aspects from the basic behavior of the object. A number of proposals deserve special mention which under the name of aspect-oriented programming (AOP) try to express each of a system's aspects in a separate and natural form that will be automatically woven to form an executable code using automatic tools [5].

All of these proposals have obtained a certain degree of success, and they have proposed different programming techniques in order to separate both concerns. But once these concerns are composed, what is the nature of the resulting entity after the composition process? Is it flexible enough to allow the synchronization and coordination aspects to be adapted not only at compile time but also at run time? That is, is it flexible enough for adding, replacing or reconfiguring synchronization and coordination components at run time? To our knowledge, the answer is no. Our contention is that this feature is very important in critical control systems where, due to unexpected environment changes, urgent and not pre-established decisions must be taken at run time. From security and economic point of views, it is not

[1] Inheritance anomaly [9] is a good example of the stated problem in adaptability: a change in the synchronization aspect may affect both the behavior aspect and data aspects.

admissible to stop the application, adapt it to the new environment and re-run it.

In the next section we show two models we have developed: the *disguises model* [13] to cope with the synchronization aspect and *coordinated roles* [10] to deal with the coordination aspect.

In both models the aspect of distribution is taken into account. [7, 6] are recent proposals for separating the aspect of distribution, but again run time adaptability is not addressed.

3 Our approach

Fig. 3 shows the main difference between our approach and others.

In conventional object-oriented languages, the functional aspect is mixed with any other aspect both at compile time and at run time. This explains why software developers have to deal with numerous revisions during the product's lifetime when new requirements appear.

This rigidity is less severe by using the aspect-oriented approach proposed in [5], where the interference between different aspects is reduced at compile time. This approach allows system developers to concentrate their attention on the design of individual aspects and to minimize the interdependence between aspects. But now the problem is moved from compile to run time because of the nature of the woven product.

What we have addressed in our approach is to maintain aspects separated both at compile time and at run time.

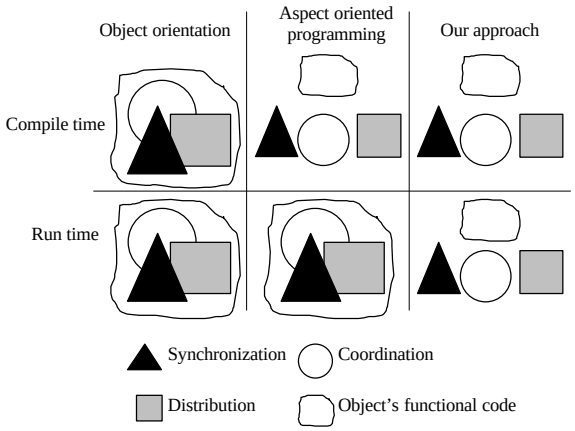

Fig.3 Separation of aspects in different approaches

The main benefits of this approach are:

- The set of components describing the basic behavior of the object may be reused in environments others than concurrent, coordinated or distributed.
- Synchronization, coordination and distribution components are not attached to any specific behavior component. Conversely, they are independent from each other. Consequently, these components may be applied to any other behavior components with different functionality but the same synchronization, coordination or distribution rules (polymorphism of components).
- As there is independence between aspects at the implementation level, it is possible to specify them in specific languages more suitable than general-purpose languages.
- In the same way, the composition mechanism for composing these concerns with basic functionality may be different from those used for composing individual concerns. This allows us to use a more adequate composition mechanism for run-time adaptability than inheritance or delegation.

Next we present the main ideas behind the two proposed models. Although here presented separately, currently they are being integrated.

In both, computational reflection [8] is the technique used to achieve our goals.

3.1 Disguises Model

The Disguises Model works at the level of protecting the object from concurrent invocations. So, it must allow the specification of mutual exclusion constraints and it must provide mechanisms for controlling message acceptance based on the state of the object.

Three different levels of execution can be distinguished in the model (figure 4).

The base level is the level in which objects exchange messages. Objects at this level have no knowledge about synchronization. In order to evaluate the synchronization constraints, incoming messages are intercepted and redirected to the object manager who is in charge of applying the synchronization polices currently applied to the object.

The *disguise level* contains the different synchronization policies that control the execution of methods in the object. These policies are independent one each other. They are implemented in a particular

language more appropriate for synchronization than languages for general purposes such as Java or C++. This language uses the concept of abstract state information [11], so they do not refer to implementation details of objects at the base level. This is the way to obtain polymorphism of synchronization components.

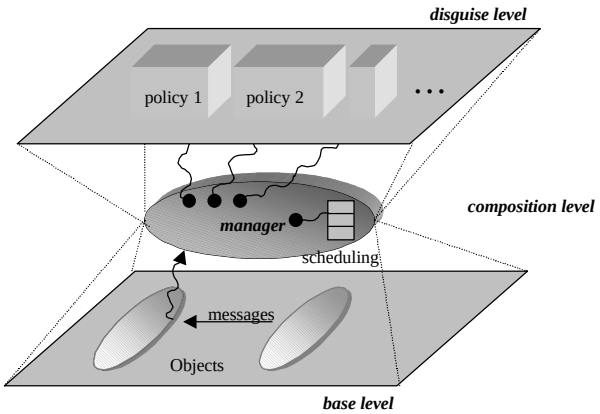

Fig. 4 Different levels of execution

The *composition level* contains the object manager. Before allowing the execution of a message, the object manager verifies if the object satisfies the synchronization constraints currently defined in every policy. If so, the message can be executed, optionally performing pre-actions and post-actions before and after the real execution of the method in order to update synchronization constraints that do not depend on the state of the object. If synchronization constraints are not satisfied then the message will have to wait until they are satisfied. The object manager also decides what is the next message to be executed according to the scheduling policy currently applied.

Inheritance is the mechanism to extend synchronization and behavior components in an independent way. However, a composition language is provided to compose synchronization with behavior. This language interacts with the object manager. Composition can be made statically (compile time) as well as dynamically (run time).

The programmer or a qualified operator can interact with this manager to add, replace, modify or delete synchronization policies by using the composition language.

Currently there is an available version of the model for Java. Figure 5 shows the different languages in the model: Java for functional components, SLAS, the Specification Language for the Aspect of Synchronization, and CLAS, the Composition Language for the Aspect of Synchronization.

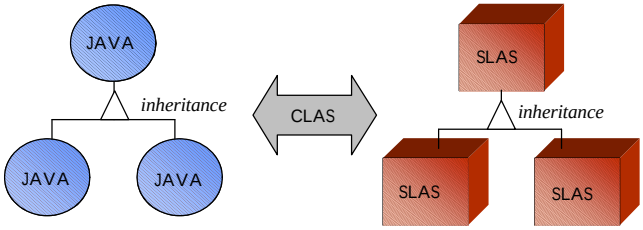

Fig. 5 Different languages in the model

3.2 Coordinated Roles

Next, Coordinated Roles is presented, a general-purpose coordination model specifically designed to be integrated with active objects models. The main goals behind its design are flexibility, composability, polymorphism, distribution, and dynamic change of coordination patterns.

The design of Coordinated Roles is motivated by the behavior of large business organizations. These are usually ruled by the leader hierarchy criterion. The structure of coordination patterns in our model is similar. A pattern is organized as a hierarchy of coordination components. Every component can have under its charge several groups of work and coordination components. The task of every component is to impose the laws of internal behavior on every group it has under its charge and to coordinate the whole behavior of all the groups and leaders that depend on it. In the lowest level of the hierarchy there are always groups of workers.

Every coordination component in the hierarchy implements a part of the whole coordination pattern to be imposed on working groups and other coordination components. Every work group or coordination component controlled by another one becomes a *role* in the imposed pattern. The coordination components are called *coordinators of roles* or simply *coordinators*.

The mission of a coordinator is to monitor the events that occur in every component under its control. To code a coordinator, the programmer simply specifies the events it reacts to, jointly with the actions it executes in every case.

Coordinators are specified in a self-contained language specific for coordination purposes that makes no reference to the classes of the objects being

coordinated, providing the polymorphism feature of coordination components.

A coordinator can monitor the events that occur in a coordinated component using the Event Notification Protocols (ENP) mechanism, which takes advantage of the run time system of the objects. Through this mechanism, an object, *A*, (the coordinator) can request from another, *B*, (the coordinated) to communicate to it the occurrence of an event *E*. The run time system of B accepts and processes the request in such a way that, when *E* occurs, it is communicated to *A*. Several objects can ask for the notification of the same event. Note that as the notification is controlled by the run time system of B, B has no special code to be coordinated. This feature increases the reusability of such a component in environments others than coordinated.

The events we can ask for notification of through ENP are the reception of a message (*RM events*), start (*SoP events*) or the end (*EoP events*) of the processing of a message, or when a certain abstract state is reached (*SR events*). The first three events are introduced for the utility they have exhibited in the coordination models based on the interception of messages [3], while the fourth one solves the lack of expressive power of those models. All the notifications can be asked for in an asynchronous or synchronous way. With the asynchronous protocols the run time system of the coordinated object simply communicates the occurrence of the event and continues performing its actions. With the synchronous protocols, the run time system of the coordinated object communicates the occurrence of the event and then waits for the response of the coordinator. When the coordinator receives the notification it executes the coordination actions associated with the event. Next, the coordinator sends the response to the coordinated object. This response can be positive letting to continue to the coordinated object with the action that triggered the event, or negative, forcing the coordinated object to abort this action.

In addition to the simple events described above, a coordinator can monitor compound events. A compound event is a group of simple events to which a sequence of actions that will only be executed when all the events of the group have occurred is associated. To declare a compound event, the programmer gives it a symbolic name, associates with it a series of notification requests as asynchronous or synchronous simple events and, finally, specifies the sequence of operations to be executed when all notifications have taken place. Compounds events let us express mutual interdependencies among different objects (an object *A* cannot execute the action *P* unless object *B* executes *Q* and vice versa).

As it has been mentioned before, a coordinator, *A*, can monitor other coordinator, *B*. In particular, *A* can monitor in *B* all the events that *B* monitors in the objects under its control. In this way, when a notification is received in *B*, it is propagated to *A*. This feature lets the composability of complex coordination patterns by means of fine-grained coordination components.

In Coordinated Roles both, coordinators and coordinated components, are implemented as active objects. In order to build coordination hierarchies and to impose coordination patterns on objects the programmer simply instantiates coordinator and worker objects and afterwards, he builds the coordination hierarchy placing some coordinators (or workers) under the control of others. This process is done through an special composition syntax which allows the dynamic change of coordination patterns.

The fact of each coordination policy being a set of active objects, possibly running in different computers, makes possible the distribution of such policies.

Currently, a prototype has been developed for ATOM [11] and it is running on Linux. Experience has been gained and an improved version is being developed for Java.

3.2 Distribution

In both models, disguises model and coordinated roles, we are integrating the separation of the distribution code. This means that objects developed without distribution code can be accessed from and coordinated with remote objects using different distribution protocols such as JavaRMI, Corba or ILU.

The same idea behind synchronization and coordination holds: the possibility of statically and dynamically interchanging distribution protocols.

4 Conclusions

In this paper we have shown how the separation of aspects principle improves software adaptability. In particular we have focused on the synchronization aspect. We have identified three different levels of

synchronization that have led to the development of two different models, disguises model for mutual exclusion and message acceptance control, and coordination roles for multi-object coordination.

Although here presented separately, currently they are being integrated in one single model. The ultimate idea is to provide a final framework for the development of reusable software components.

Acknowledgments

This work has been developed with the support of Junta de Extremadura, under project IPR98A041and by CICYT under project TIC98-1049-C02-02.

References:

[1] M. Aksit, B. Tekinerdogan, L Bergmans. *Achieving Adaptability through separation and composition of concerns*. Max Mühlhäuser editor, Special Issues in Object-Oriented Programming, Workshop Reader of the ECOOP'96, Linz, Austria.

[2] L. Bergmans. *Composing Concurrent Objects*. Ph.D. Thesis, University of Twente, 1994.

[3] S. Frø lund. *Coordinating Distributed Objects*. *An Actor-Based Approach to Synchronization.* The MIT Press, 1996.

[4] W. Hursch, C. V. Lopes. *Separation of Concerns.* Northeastern University, February 1995.

[5] G. Kiczales et al. *Aspect-Oriented Programming*. Proceedings of ECOOP'97, Jyvaskyla, Finland, June 1997.

[6] J. Kleinöder, M. Golm. *MetaJava: An Efficient Run-time Meta Architecture for Java*. Proceedings of International Workshop on Object Orientation in Operating Systems, October, 1996, Seattle, Washington.

[7] C.V. Lopes. *D: A Language Framework for Distributed Programming*. Position paper in Aspect-Oriented Programming Workshop of ECOOP'97, Jyvaskyla, Finland, June 1997.

[8] P. Maes, *Concepts and experiments in computational reflection*. Proceedings of OOPSLA'87, Vol.22 of ACM SIGPLAN Notices, pp 147-155, ACM Press, 1987.

[9] S. Matsuoka, A. Yonezawa. *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*. in Research Directions in Concurrent Object-Oriented Programming Languages. Ed.: G. Agha, P. Wegner and A. Yonezawa, MIT Press, April 1993, pages 107-150.

[10] J.M. Murillo, J. Hernández, F. Sánchez, L.A. Álvarez. *Coordinated Roles: Promoting Reusability of Coordinated Active Objects Using Events Notification Protocols.* To be published in Proceedings of Coordination'99, Amsterdam, 26-28 April, 1999.

[11] M. Papathomas, J. Hernández, J.M. Murillo, F. Sánchez, *Inheritance and Expressive Power in Concurrent Object-Oriented Programming*. Proceedings of the LMO Conference, Roscoff, France, Oct. 1997.

[12] F. Sánchez, J. Hernández, M. Barrena, J. M. Murillo, A. Polo. *Issues in Composability of Synchronization Constraints in Concurrent Object-Oriented Languages*. Max Mühlhäuser editor, Special Issues in Object-Oriented Programming,, Workshop Reader of the ECOOP'96, Linz, Austria.

[13] F. Sánchez, J. Hernández, J. M. Murillo, E. Pedraza. *Run-time adaptability in COOLs*. Technical Report n°1/97. Computer Science Department. University of Extremadura. September 1997.