Managing Evolution of Coordination Aspects in Open Systems

Juan Carlos Cruz, Sander Tichelaar

Software Composition Group, University of Berne

E-mail: {cruz,tichel}@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~scg.

Abstract

Most of the work on coordination technology so far has focused on the development of special coordination languages and environments that provide the basic mechanisms for realizing the coordination layer of an open system. It is clear that the idea of managing separately the coordination aspect from the computation in a language has a lot of advantages in the development of those systems. Nevertheless, most of the coordination languages do not take care that additionally to managing coordination requirements, they must manage other kinds of "openness" related requirements in Open Systems. The most important requirement being to support the evolution of the coordination requirements themselves. This problem manifests during the software development process by the development over and over again of solutions to similar coordination problems. To tackle this problem, and instead of proposing a new language, we are attempting to develop an open set of adaptable and reusable software components that realize various useful coordination abstractions. With these components we provide explicit separation of coordination from computation, and facilitate reuse and evolution of coordination aspects in Open Systems.

1 Introduction

Over the last two decades software development has changed significantly. Its evolution has been motivated by the goal of producing open rather than closed, proprietary systems. Open Systems are basically "open" in terms of topology, platform and evolution [18]: they run on networks which are continuously changing and expanding, they are built on the top of a heterogeneous platform of hardware and software pieces, and their requirements are continuously evolving. *Evolution* is the most difficult requirement to meet, since not all the application requirements can be known in advance[12].

Development of Open Systems is a difficult task, because in addition to requirements concerning openness, they have special computational constraints. One of the most important constraints concerns the *coordination* of the interaction of the different software entities that compose those systems. Coordination in this context concerns the management of possible interdependencies between the activities performed by the entities of the system. Carriero and Gelernter [3] have proposed to manage separately the coordination aspect from the computation in the development of Open Systems. They propose to use what they call Coordination Languages. The idea of separating these two aspects during the system development seems to have a lot of advantages, mainly related with a simplification in the complexity of the development. Nevertheless, most of the coordination languages do not take care of satisfying the "openness" related requirements Open Systems have to. This is particularly true in which concerns the management of the evolution of the coordination aspect itself. This problem manifest during the software development process by the development of over and over again of solutions to similar coordination problems. How to manage in a single approach the complexity of the coordination aspect and its evolution is not still clear. To tackle this problem, and instead of proposing a new Coordination Language, we propose to use a component-base approach. We will develop an open set of adaptable and reusable coordination components that will realize solutions to standard coordination problems. They can be specialized and parameterized to solve specific coordination problems. With these components, we provide explicit separation of coordination from computation, and facilitate reuse and evolution of coordination aspects.

We are developing a prototype of this set of coordination components using the Java language, an object oriented language well-suited to modelling software entities in a distributed setting. We are validating our approach by using those components in building a set of sample applications that characterize Open Systems. Our initial experiences in building this set of components has shown us, that coordination solutions are not easily shrink-wrapped into off-the shelf components. One reason is that coordination solutions affect typically more than one component. A second is that the coordination solutions are subject to a complex set of requirements for Open Systems like [8]: interoperability, concurrency, distribution, reliability, security, etc. A systems developer who develops coordination solutions, therefore has to deal with many of the above requirements. And every application has its own unique mix of interacting components and specific requirements.

In this paper we present some results of our experimental work building a such set of coordination components. Section 2, discusses coordination aspects in Open Systems. Section 3, introduces an example of a coordination component, and explains how this component was designed using a component-based approach. Section 4 gives an overview of related work. Finally, Section 5 concludes with a discussion: evaluating our contributions and pointing-out lessons learned in this experimental work.

2 Coordination Aspects in Open Systems.

Coordination problems arise in the organisation of interactions of a group of entities that collaborate and cooperate to accomplish some task and to satisfy some goals. It is because entities cooperate that they can accomplish more elaborated actions, but it is also because of their multiplicity that they must coordinate their actions and resolve conflicts.

Malone and Crowston [10][11] have defined coordination as:

> *The act of managing interdependencies between activities performed by entities in order to achieve some goals.*

They have identified three kinds of dependencies out of which all other important dependencies can be expressed, either by their specialization or by their combination. They are:

- *Flow*: occurs when an activity produces something that is used by another one. Flow dependencies can be viewed as a combination of three other kind of dependencies: *Prerequisite* constraints (an item must be produced before it can be used), *Accessibility* constraints (an item that is produced must be available for use), and *Usability* constraints (an item that is produced should be "usable" by the activity that uses it).
- *Sharing*: occurs when multiple activities all need the same (limited) resource
- *Fit*: occurs when multiple activities produce things that have to fit together.

Some dependencies can be viewed as specialization of others. For instance *Task assignment* can be seen as an special case of *Sharing*, where the "resource" being shared is the time of the entities that do the tasks.

This list of interdependencies is not intended to be exhaustive. To our viewpoint the importance of Malone's work is that it can be used to propose a systematic approach to identify coordination problems in open systems. The main idea consists to identify the existence of these interdependencies between the multiple activities that the system realizes [5] (e.g.communication, management of resources and services, task assignments etc.).

2.1 Coordination Problems

We have selected a representative set of coordination problems that we have yet identified. For each one we give some hints about how these problems can be managed using a coordination process.

2.1.1 Management of Shared Resources

Managing shared resources is one of the most common coordination problems that occurs in open systems. Whenever multiple activities share some limited resource (e.g. storage space, an actor's time, etc.), a coordination process which controls the allocation of and/or access to, the resource is needed. The coordination process must: serialize the incoming concurrent requests, select (by using a pre-defined allocation and/or access policy like for example: first-come/first serve, priorities, etc.) the request to be served, control that all clients will eventually obtain the resource, control security aspects (access rights, etc.), take care of possible software and hardware failures during the allocation process, etc. A particular situation in which this coordination problem occurs is when multiple processes share a unique disk space.

2.1.2 Transfer of Information Between Activities

Producer/consumer relationships occur in software systems whenever one activity produces some information that is used by another activity. A coordination process which controls the transfer of information between activities is needed. The coordination process must: guarantee the physical transfer of information between the entities (from one entity space to another), control their synchronization, and eventually control the replication of information in case of a replicated transfer (multicast or broadcast). A particular situation in which this coordination problem occurs is when computing the topology of a system.

2.1.3 Activity Synchronization

Execution of activities in a concurrent setting must be synchronized in order to either communicate, or to perform some common operations (i.e accessing shared resources). A coordination process controlling the synchronization must constraint the order of execution of these activities so that only acceptable execution sequences can ever be used. Two basic forms of synchronization are necessary when working with concurrent activities: 1) *Condition-Synchronization* (needed when an activity wishes to perform an operation that can only safely be performed if another activity has itself taken some action or is in some defined state) and 2) *Mutual Exclusion* (needed to protect critical sections). A particular situation in which condition synchronization is needed is when buffers are used to exchange data between a producer and a consumer. The producer produces items the consumer needs. The consumer cannot consume what the producer has not produced, and the producer cannot produce if the consumer does not consume. Moreover, if simultaneous deposits or extractions are possible from the buffer then mutual exclusion must be ensured so that two producers, for example, do not corrupt the "next free slot" pointer of the buffer. A coordination process that controls the synchronization between the concurrent activities must basically: serialize concurrent execution requests, and select (by using a pre-defined synchronization policy like for example: mutual-exclusion, multiple readers/only one writer, etc.) the request to be executed.

2.1.4 Group Decisions

It is very common that a group of entities working together are confronted with the problem of making coordinated decisions. These decisions cannot be made in isolation by one of the members, possibly because no individual has enough authority, competence or information to decide. A coordination process that controls the group decision must: ensure that all the entities will receive an invitation to participate in the group decision, implement a decision scheme (e.g. consensus, voting, authority, etc.), announce the final decision to the group, and decide what to do in case of failures of the entities which participate (to terminate the process, etc.). A particular situation in which this kind of coordination problem can occur is in the election of a new server from a group of replicated servers. Replication of servers is a common strategy used to guarantee the fault tolerance of a provided service.

3 A Component-Based Approach to Manage Evolution of Coordination Aspects

In a component-based approach [12], each application is viewed as only one instance of a generic class of applications, each one being flexible composed of software components. New application requirements are addressed by removing, replacing and reconfiguring the software components without disturbing other parts of the application. Why not to manage the coordination aspects of those applications using the same approach? Why not to develop a set of adaptable and reusable coordination components that encapsulate standard solutions to coordination problems? That is what we tried to in our work. The initial results of this experimental work has shown that coordination solutions are not easily shrink-wrapped into off-the shelf components. The reasons for this are analyzed in [17]. They are:

- *information dependencies*: Many coordination solutions are dependent of application dependent information. Information that is typically needed by coordination solution is: “this method updates the state and this one only reads”, or “these three method calls constitute a critical section”, etc.
- *service dependencies*: In some cases the computational components need to provide coordination related services like to store state relevant information in order to guarantee recovery in case of failures.
- *request control*: Coordination solutions need to control the ordering and adaptation of requests. They typically need techniques like message reification, message passing control or method wrapping, etc.
- *complex system requirements*: Coordination solutions are subject to a complex set of requirements for Open Systems like [8]: interoperability, concurrency, distribution, reliability, security, etc. A systems developer who develops coordination solutions, therefore has to deal with many of the above requirements. Every application has its own unique mix of specific requirements.

In the design and development of the coordination components we have investigated how framework technology can help us to deal with this complex set of system requirements in the packaging of coordination solutions. Particularly, we had look at the design guidelines for building “tailorable frameworks” as presented in [6].

Example

As an example of what is being done, we show access policy components for a shared resource in a toy banking system. In this case the resource is an account database which is shared by multiple teller machines. The teller machines need to get information from the account database in order to check a client’s account. They also need to update account information if they have given money to a client. To keep the database consistent we need an access policy to regulate the multiple requests.

As a solution for this regulation problem we introduce an access policy component. This solution not only provides access regulation, but also explicitness of architecture and flexibility. We can imagine a banking system which initially starts with a basic FIFO policy, but which, as more and more teller machines get connected to the system, needs a more sophisticated policy to handle the increasing number of incoming requests. By creating a structure which allows us to change policies transparently, the banking system can be adapted without having to change other parts of the system. Our solution is shown in Figure 1. In its design we used ideas from the Command and Strategy pattern from Gamma et al. [7], and from the Active Object pattern [13].

The class `Wrapper` is the interface to the resource for the rest of the system. For every command which is invoked by an incoming request (1), a `ConcreteCommand` object is created (2). This explicit representation of commands allows us to buffer the commands, change their order, execute them in parallel, or whatever a policy needs to do with them. This is, for instance, needed for a readers/writer policy, which allows multiple readers to access the database simultaneously. These commands are then given (3) to the policy which is connected (through parameterization) to the interface. This policy handles the commands, i.e. determines when and in which order the requests can access the resource. If the command is allowed, it executes (4).

A major problem with the transparency of these policies, is that policies may need application-dependent information. A readers/writer policy for instance needs knowledge about a `getBalance` command being a reader command and an `updateBalance` command being a writer command. This problem obviously violates the transparency of the proposed solution: by integrating the storage and retrieval of this application-dependent information in the `Command` or the `Policy`, the change and reuse of policies is much more difficult. We tackle this problem by introducing a context object. This object contains the application-dependent information of the policy, so it can tell the policy if a certain command is a reader or a writer command.

Reusability and Flexibility

In the presented example it is possible to completely separate the coordination policy from the application specific part of the application. The information dependencies between the generic policy and the application specific part of the application are made explicit in the context objects. This means that the policy is independent of the application and the application independent of what policy is used. The context objects are used to configure a generic policy for a specific application. In this way the policies are reusable in different applications and within applications we can easily switch policies.

Limitations

The example we have shown is limited in a couple of ways. Firstly, it only deals with a single resource. More complex coordination problems involve distributed solutions, e.g. replicated resources or distributed transactions. This is work in progress. Secondly, there are dependencies that cannot be dealt with in this way. For instance, the set of actions that make up a transaction, is local for every transaction, making it impossible to store this information in some kind of general transaction context object [17].

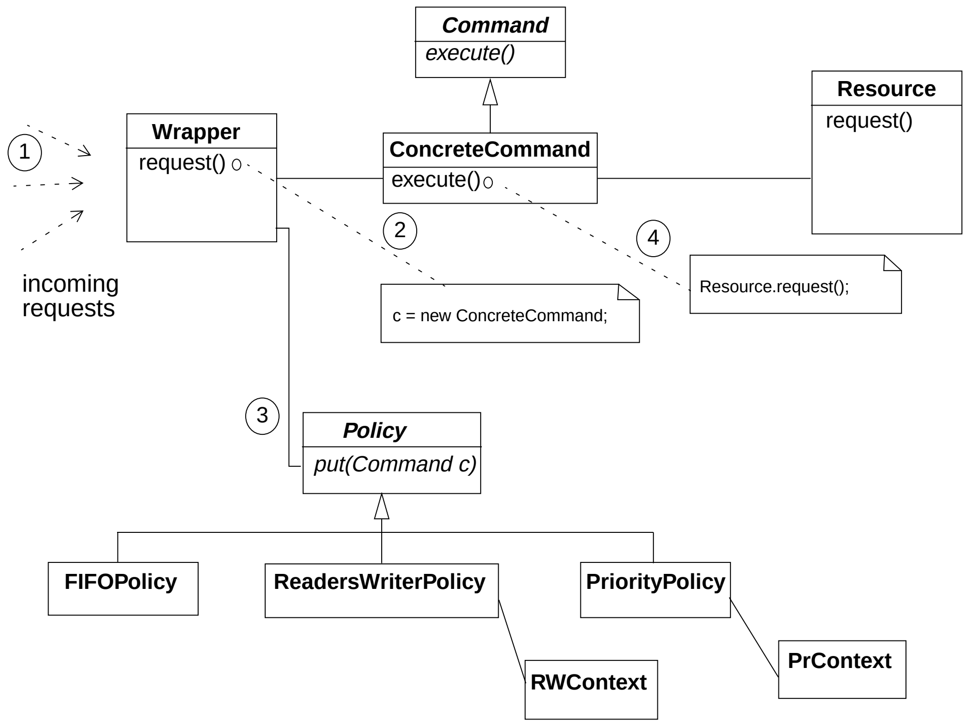

Figure 1 Access policy for shared resource.

4 Related Work

Most of the work done in the area of coordination have been directed to the development of coordination languages (e.g. Gamma [2], Linda [3], etc.) used to describe coordination aspects in open systems. The fact that the coordination model they implement is based on a particular paradigm makes these languages limited in scope. Each one of these coordination languages proposes its own set of coordination abstractions that realizes a particular paradigm for realizing coordination. Coordination problems, however, do not always fit to a particular paradigm. Language based approaches are inherently limited in scope. Instead of proposing a new language, we are attempting to provide an open set of coordination components that realize various useful coordination abstractions. With our components we provide explicit separation of coordination from computation and facilitate reuse and evolution of the coordination aspect.

The idea of using and composing coordination components is not new, as evidenced by work such as Manifold [1] and ConCoord [9]. These languages allow composition of existing coordination components with communication operations. Dynamically new communication configurations between components can be reached in this way. Coordinator processes do their job with no knowledge of the internal details of the computations done by the processes they coordinate. The communication configuration determines the coordination flow of an application. In our work we try to explore other forms of composition of coordination components different to communication (e.g. policies, etc.). Furthermore, we build components that are themselves configurable and composable. Our components can be adapted to particular situations by plugging in instances of the elements that define their parameterized structure.

In a different domain, the Adaptive Communication Environment (ACE)[14] is an interesting reference. One of the most important contributions of this work is the set of coordination patterns [13][15][16] that it proposes and that can be used to realize coordination components. We have presented examples of coordination components that have used some ideas from some of these design patterns. The object-oriented network programming toolkit includes a series of wrappers, class libraries and frameworks for developing communication software. The main problem with these work is its closeness. Synchronization and communication abstractions in the ACE toolkit are reduced to those provided by some common operating systems (e.g. Mutex, Condition, Semaphore, RW, pipes, FIFOs, sockets, etc.), thus limiting its scope. New coordination abstractions representing solutions to more complex coordination problems cannot be included in the environment by users. Our approach is more open in this way.

5 Conclusions

In this paper we have shown that is possible to provide extendable and reusable coordination solutions using a component-based approach. With these coordination components, we provide explicit separation of coordination from computation (as promoted by coordination languages), and facilitate reuse and evolution of coordination aspects. We show that although it is possible to define generic components for managing the aspect of coordination using composition, a complete separation of coordination and computation is not possible due to dependencies of coordination solutions of information and services that can only be provided by the computational components.

References

[1] F. Arbad, "The IWIM Model for Coordination of Concurrent Activities", in [4] pp. 34-56

[2] Jean-Pierre Banatre and Daniel Le Metayer, “Gamma and the Chemical Reaction Model”, *Proceedings of Coordination’95 Workshop*, IC Press, London, 1995.

[3] N. Carriero and D. Gelernter, “Linda in Context”, *Communications of the ACM*, vol. 32, no. 4, April 1989, pp. 444-458.

[4] P. Ciancarini, C. Hankin (eds.), "First International Conference on Coordination Models, Languages and Applications COORDINATION’96", Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag.

[5] George Coulouris, Jean Dollimore and Tim Kindberg, *“Distributed Systems Concepts and Design”*, Addison-Wesley, 1994.

[6] Serger Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, Patrick Steyaert, “Design Guidelines for Tailorable Frameworks”, *Communications of the ACM*, vol. 40, no. 10, October 1997, pp. 60-64

[7] Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides, *“Design Patterns”*, Addison-Wesley, MA, 1995.

[8] C. Hewitt, “Offices are Open Systems”, *ACM Transactions Off. Inf. Syst.*, vol. 4, no. 3, 1986, pp. 270-287.

[9] A.A. Holzbacher, "A Software Environment for Concurrent Coordinated Programming", in [4] pp. 249-266.

[10] Thomas Malone, Kevin Crowston, “The Interdisciplinary Study of Coordination”, AC*M Computing Surveys*, vol. 26,no. 1, March 1994

[11] Thomas Malone, "Free on the range: Tom Malone and the implications of the digital age", IEEE Internet Computing, Vol 1, no. 3, May-June, 1997

[12] Oscar Nierstrasz and Laurent Dami, “Component-Oriented Software Technology”, *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 3-28.

[13] R. Greg Lavender and Douglas C. Schmidt, “Active Object: an Object Behavioural Pattern for Concurrent Programming”, *Proc. Pattern Languages of Programs*, September 1995.

[14] Douglas Schmidt, “The ADAPTATIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Development of Communication Software”.

[15] Douglas Schmidt, “Acceptor: A Design Pattern for Passively Initializing Network Services”, *C++ Reports*, SIGS, Vol. 7, no. 8, Nov/Dec 1995.

[16] Douglas Schmidt, “Connector: A Design Pattern for Actively Initializing Network Services”, *C++ Reports*, SIGS, vol. 8, no. 1, January 1996.

[17] Sander Tichelaar, Juan Carlos Cruz, Serge Demeyer, “Coordination as a Variability Aspect of Open Distributed Systems”, SCG-Report, IAM, University of Berne, February 1998.

[18] Dennis Tsichritzis, “Object-Oriented Development for Open Systems”, *Information Processing 89 (Proceedings IFIP’89)*, North Holland, San Francisco, Aug. 28-Sept. 1, 1989, pp. 1033-1040.