Coordination in Concurrent Object–Oriented Programming

Extended Abstract

Thilo Kielmann
University of Siegen
Dept. of Computer Science and Electrical Engineering
Hölderlinstr. 3
D-57068 Siegen, Germany
kielmann@informatik.uni-siegen.de

Abstract

Coordination viewed as the notion of "managing dependencies among activities" [8] is the key concept for modelling concurrent systems. In this paper, we investigate coordination from the viewpoint of programmers and programming language designers with respect to object–oriented programming. Finally, we evaluate existing object–oriented coordination models.

1   Introduction

Coordination as the key concept for modelling concurrent systems is discussed in a wide range of publications. Nevertheless, there is only very little consensus about this notion as shown by the following citations. Coordination is:

- *"managing dependencies among activities."* [8]
  This mostly general definition declares coordination to be some kind of "superconcept" of all possible concurrency models.

- *"the involvement of information exchange among active agents."* [6]
  This statement focusses on the communication between active agents as the principal topic of concurrent systems. This is strengthened by the following:

- *"the additional information processing performed when multiple, connected actors pursue goals that a single actor pursuing the same goals would not perform."* [7]

- *"the integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal."* [9]
  This view reflects the field of parallel and distributed algorithms.

- *"the process of building programs by gluing together active pieces."* [4]
  This quite different statement describes the view of a programmer dealing with concurrent systems.

In an effort to give a concise definition, one can say: *coordination* is managing the activities of individual agents connected in a universe in which computations are modelled. *Agents* are active, self–contained entities performing *activities* which can be divided in computational and communicative operations.

According to [8], there are three basic situations in which coordination takes place. First, there is *cooperation* where different agents share common rules of behaviour – like laws which are obeyed by everyone. An example might be a producer/consumer setting in which all participating agents use a certain protocol in order to achieve fair scheduling.

Second, there is *collaboration* where agents work together in order to solve a common problem. This is the situation of executing distributed or parallel algorithms.

And finally, there is *competition* where one agent's gains are another's losses. An example is the acquisition of shared resources.

2   Coordination Models and Languages

"A coordination model is the glue that binds separate activities into an ensemble." [6] In other words, a coordination model provides a framework in which the interaction of individual agents can be expressed. This covers the following aspects: creation and destruction of agents, communication between agents, spatial distribution of agents, as well as synchronization and distribution of activities over time. Thus, coordination is the most general term, the basic concept behind all considerations of concurrent systems.

For the evaluation of certain coordination models different aspects have to be considered:

- expressiveness, namely suitability for certain application ranges, granularity, and scoping.

- orthogonality in separating concerns of coordination from those of computation.

- semantics, the possibilities for reasoning in the model.

A coordination language is *"the linguistic embodiment of a coordination model."* [6] Thus, a coordination language orthogonally combines models for coordination and computation, providing both of them with a unified syntactical form. The presumably most famous example of coordination models is Linda [4] which encountered several linguistic embodiments like C–Linda or Scheme–Linda. Unfortunately, there is a lot of ongoing research where coordination models are implicitly and thus only vaguely defined in specific concurrent programming languages.

For the purpose of evaluating coordination languages one has to consider the representations of both the coordination and the computation models, possible interferences between them, and implementation issues, primarily efficiency.

3   Coordination in Object–Oriented Systems

The object paradigm represents the "state of the art" in software development techniques. In sequential programming, objects are the building blocks, encapsulating their internal state through well–defined interfaces. Here, objects are simply passive data containers.

For the concurrent case, there are two flavours of program modelling. The first possibility is to deal with passive data objects and separate execution threads. Here, coordination of participating threads is based on classical synchronization schemes like mutual exclusion or condition variables.

The second variant is to deal with active objects which behave like passive ones but additionally contain their own thread of control. Thus, active objects are able to perform activities even without being called by another object. This is a much more attractive approach because it simplifies coordination models by reducing them to homogeneous sets of autonomous, self–contained entities. These entities can be viewed exactly as the agents introduced above.

Therefore, the object paradigm serves as a well-suited basis for computation as well as for coordination purposes. Thus, the primary design goal of an object–oriented coordination language must be the seamless syntactical integration of both models while preserving their orthogonality. In the following, we investigate two different object–oriented coordination models with quite different properties.

3.1 Obliq

Obliq [3] has been introduced as a scripting language rather than as a coordination model. Nevertheless, it has some interesting features which contrast the *actorSpaces* model introduced below.

In Obliq, multiple sites containing objects and threads can communicate over a local or even world–wide network. An Obliq site corresponds to an address space, objects contain data in form of their state, and threads perform communication. Threads and objects can move across sites. Sites communicate via globally known nameservers which provide locations for given object names.

Every action performed by an Obliq computation (such as method invocation, delegation, object updating or cloning) is performed on a per–object basis. Obliq provides no abstractions for dealing with groups or sets of objects.

Synchronization can be performed on a per–object basis, too. Operations on objects can be synchronized by monitor–like constructs, namely mutual exclusion and condition variables. Deadlocks caused by recurrent method invocations are avoided using the notion of self–inflicted operations which circumvent the object's mutual–exclusion protection.

Locality or remoteness of object references and invocations are transparent to Obliq programs, but different sites can be accessed by programs via the nameservers. This makes Obliq programs "network–aware", allowing programmers to deal with locality problems explicitly. The transparent access to remote objects provides a uniform object space across all participating sites. Although this leads to a simple model, it also implies that computation and coordination cannot be separated easily, because method invocations as the only means of activity are used for both purposes.

3.2 *ActorSpaces*

The actor model [1] unifies objects and concurrency. Actors are autonomous and concurrently executing objects which operate asynchronously. Actors may send each other messages. In response to receiving a message, an actor may take the following types of actions: it may send messages to other actors, it may create new actors, and it may specify its own new behaviour (state) to be used when processing the next incoming message.

In the *actorSpace* model [2], communication is performed in a pattern–directed manner. The sender of a message directs it to a set of receiving actors by denoting a pattern which has to be matched by the receivers. *ActorSpaces* are passive containers for actors, providing a context for matching patterns on actors and their attributes. *ActorSpaces* may overlap or even be contained in others.

While computations are described in the actors' implementations, coordination issues like synchronization constraints or resource management policies are separated into terms of *actorSpaces*. On one hand, this allows separation of concerns between computation and coordination, on the other hand, it enables expressing coordination issues based on groups of actors, instead of single objects.

4 Conclusions

As we have seen, coordination can be viewed as the abstract "superconcept" of concurrency models. Furthermore, we have identified the object paradigm as a well–suited basis for autonomous, communicating agents. Current object–oriented coordination models cover a wide range of different flavours. But they are more or less confined to traditional schemes like message passing or synchronization constraints based on object states.

Forthcoming object–oriented coordination models will have to adapt different communication and coordination schemes suited for modelling more sophisticated settings like the "restaurant of dining philosophers" [5] or the interaction of neurons or gas molecules.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, Cambridge, Massachusetts, 1986.

[2] Gul Agha, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. In *Research Directions in Concurrent Object–Oriented Programming*, chapter I, pages 3–21. MIT Press, Cambridge, Mass., 1993.

[3] Luca Cardelli. Obliq: A Language with Distributed Scope. Research Report 122, Digital Equipment Corporation, Systems Research Center, 1994.

[4] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs.* MIT Press, Cambridge, Massachusetts, 1990.

[5] Paolo Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, New Orleans, 1990.

[6] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, 1992.

[7] Thomas W. Malone. What is Coordination Theory. Working Paper No. 2051-88, MIT Sloan School of Management, Cambridge, Mass., 1988.

[8] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[9] B. Singh. Interconnected Roles (IR): A Coordinated Model. Technical Report CT-84-92, Microelectronics and Computer Technology Corp., Austin, TX, 1992.