Coordination of Active Objects by Means of Explicit Connectors

S. Ducasse and M. Günter
University of Bern, IAM-SCG,
Bern, Switzerland,
http://www.iam.unibe.ch/∼ducasse,mguenter

Abstract

Although coordination of multiple activities is a fundamental goal of object-oriented concurrent programming languages, there is only limited support for its specification and abstraction at the language level. This leads to a mismatch between conceptional designs, using high-level abstractions, and the implementation, using the low-level coordination constructs. Often coordination is hard-wired into the components they coordinate, which leads to evolution, maintenance and composibility problems.

We propose a model called FLO/C *that relies on the notion of connectors. A connector is an entity that enforces the coordination of the entities it coordinates. This model supports a clear separation between the coordinated active objects and their coordination. An active object only defines specific domain information and a connector only defines coordination between a group of active objects (its* participants*). The coordination is abstractly defined refering to components in terms of the object interface. Coordination and coordinated entities are independant and can evolve separately. Coordination can be composed and replaced easily.*

Keywords: *coordination, active objects, groups, synchronizers, components and connectors, laws, separation of concerns, message passing control.*

1. Introduction

Although coordination of multiple activities is a fundamental goal of object-oriented concurrent programming languages (OOCPL), there is only limited support for its specification and abstraction. There is no support for coordination at a high level of expression. This inability leads to a mismatch between conceptional designs, using high-level abstractions, and the implementation, using the low-level coordination constructs [7]. The situation complicates the composition of different coordination policies without changing the implementation of the coordinated entities. Furthermore, as the policies are coded into the coordinated entities their modification is difficult. As a possible solution, this paper will introduce explicit connectors as high-level coordination supporters.

In the area of software architecture design the distinction between components and connnectors was introduced to address the need of decoupling *domain specific design* from *collaboration design*[16]. Architectural connectors represent design decisions concerning the collaboration of software components. Allen and Garlan [2] present a specification language for connectors, which has descriptive and *analytical* properties such as component substituability or deadlock detection.

We introduce the FLO/C model[1], which takes up the component/connector distinction and applies it to the implementation level. Our explicit connectors *implement* the collaboration of components, therefore they are the ideal location for coordination code. Minsky et al. [13] and many other authors [4] recognized the need for such *explicit* entities, that represent and enforce interaction policies. FLO/C's connectors are *abstractly* defined, and only rely on the interfaces of the active objects they coordinate. Thus, they are *independent* from the implementation of the coordinated objects. This allows a clear *separation of concerns*: An active object only defines specific domain information and a connector only defines coordination between a group of active objects (its *participants*). A connector *restricts* the freedom of the coordinated objects by controlling message passing. The control done by a connector depends on the state and the history of the coordination. A connector specifies a *temporal ordering* such as precedence and atomicity of the exchanged messages amongst the objects.

In section 2 we discuss the coordination goals in the context of active objects and we present the lack of support for coordination in concurrent object-oriented languages and their consequences. Afterwards, the FLO/C model is presented using the gas station example [14].

2. Multi-Object Coordination Problems

We discuss why coordination support in traditional object oriented languages is insufficient. We briefly present our choice to represent concurrent objects, and we present our coordination goals.

[1] A prototype that fully implements this models in NEOCLASSTALK (a new SMALLTALK implementation providing explicit metaclasses) is available at the authors' web pages.

2.1. Explicit support for coordinating objects

Traditional OOCP languages offer little support for synchronisation of groups of concurrent objects [7]. Even modern languages like JAVA model coordination at a very low level of abstraction. *Threads* model activities, and communicate through unprotected, shared memory. While the set of provided constructs in theory is sufficient to solve any coordination problem, in practise only expert programmers are able to handle non trivial tasks. JAVA users tend to rely on design pattern collections [10] to solve common coordination problems. But even with such approaches, protocols used for establishing the coordination between a group of activities are hard coded into parts of the activities resulting in poor abstraction facilities avoiding composition and evolution of the coordination policies. The main problems can be summarized as follows:

- **No separation of concerns.** Expressing coordination abstraction is difficult because the code that manages the coordination is intimately tied to the implementation of the coordinated objects [11].
- **Absence of abstraction.** The fact that no abstractions are supported offers a low level of reasoning. There is no declarative means to specify coordination.
- **Lack of composability.** Composing different coordination policies is difficult without changing the code of the coordinated objects.
- **Lack of flexibility.** The coordination being not explicitly and abstracly expressed, it is difficult to modify and to customize the coordination policies.
- **Do it yourself.** This problem refers to the fact that the programmer must implement all the mechanisms that will support the coordination. This task is particularly difficult. Doing so the programmer should first focus on the tools and mechanisms instead of just expressing the desired coordination.

2.2. Modelling Activities as Active Objects

"coordination is managing dependencies *between* activities*"* [12]. Usually, the activities are modelled as threads or processes. Since these concepts cross object borders, different approaches tried to map threads to objects, thus enforcing object encapsulation: Actors [1], Actalk [5] and more recently ATOM [15] allow the definition of activity enhanced objects, so called *active objects*, that possess their own thread(s) and communicate asynchronously. Because ACTALK has been designated to be a minimal open testbed for active objects [5], we have chosen a variant of its active object notion for our model.

2.3. Coordination with Multi-Object Joint Actions

We can build a complete programming model out of two pieces - the computational model and the coordination model [6]. FLO/C uses active objects (respectively their methods) to express computation and connectors to implement coordination. Carriero and Gelernter state that a coordination language must provide the *glue* to bind separate active pieces into software systems. Such glue must allow these independent pieces to *communicate* and *synchronize* with each other. In the context of the multi-object coordination:

Communication. Connectors must provide ways for active objects to communicate with each other or eventually with *groups* of other active objects (e.g. multi-casting).

Synchronization. Here, one task is the *mutual exclusion* of object groups, the other the *conditional synchronization.* The problem is that the conditions might depend on the state of more than one active object.

We propose a coordination abstraction that extends the state machine model of a single object. A single active object can accept requests for computation, check if it is in the right state and then compute, thereby changing its state. Furthermore, upon failure of the state checks, a request can be *denied* (balking guard), or *blocked* in order to be tried again later.

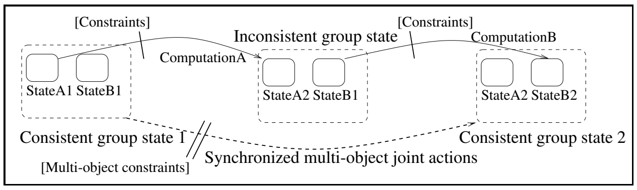

Figure 1. Group state change.

For groups of objects, the group state is defined by the states of its objects. An object group can accept requests according to its global state. State changes include computation on different group members. We call this abstraction **synchronized multi-object joint actions**, or simply *joint actions*. Joint actions preserve the group's consistency because the constraint checking and the computations execute atomically; the group state is protected from third-party access (see figure 1).

The following constructs are needed to compose joint actions: Declaration of both styles of *guards* on several objects. Declaration for a sequence of single computations that lead to a consistent group state. The sequence can be composed using *pull-based flow* or *push-based flow* [10].

FLO/C provides constructs to easily compose such multi-object joint actions that are used to realize *mutual exclusion* and *conditional synchronization*. Conditional synchronization is already reflected by the guards for the state transition. Mutual exclusion of a resource is modeled as object groups, each containing the resource, and each accessing the resource by multi-object joint actions. Note furthermore that multi-object joint actions can be used to model *pessimistic transactions*: Guards check if all participants are in a proper state or ask them directly if they can commit to a certain transaction. Then protected computation on different objects do the commitment. However, in FLO/C joint-actions do not address optimistic transactions, real-time support, and real distribution.

As we show in the next sections, FLO/C provides ways to specify such joint-actions plus a low level asynchronous communication based on rules like in CLF [9]. The following table summarizes the coordination abstractions addressed in FLO/C and presents the rule operators that support them.

Multi-object joint action			Communication	
purpose	*styles*	*operator*	*styles*	*operator*
guard	balking	permittedIf	push	impliesLater
	blocking	waitUntil		
computation ordering	push	implies		
	pull	impliesBefore		
third-party access protected			asynchronous	

3. A Model for Coordinating Active Objects

Conceptionally, the FLO/C model distinguishes between two entities modeled as active objects: *components* and *connectors*. Components model domain specific properties, while connectors model interaction between components.

[8] presents in detail FLO/C's specific features. Due to space limitations we present FLO/C's concept of components and connectors via the example of a gas station simulation [14] because it is non-trivial and shows most of the FLO/C features.

3.1. The Gas Station Example

A gas station has several pumps where car drivers can pump fuel. A car driver decides to pay an amount of money to the cashier. Only then, the car driver can pump fuel. Car drivers and cashier are autonomous entities that act concurrently on the pumps which are also autonomous. Therefore we model all of them as active objects. Car drivers interact with the cashier to pay for fuel, and they interact with pumps to get fuel, while the cashier interacts with the pumps to prepare them for pumping.
The example illustrates several coordination problems:

1. **Client-server interaction**: The customer accesses the cashier to get authorization to access a pump. Money and fuel representations flow between the participants.

2. **Shared resources**: The pumps are shared by customers.

3. **Race**: As discussed in [14], when two customers pay to get fuel from the same pump, the one who is faster can eventually get the fuel for both.

While the car drivers are responsible for their proper behavior (when to pump, how much to pay), the connectors enforce the interaction policies (correct amount of fuel and race regulation).

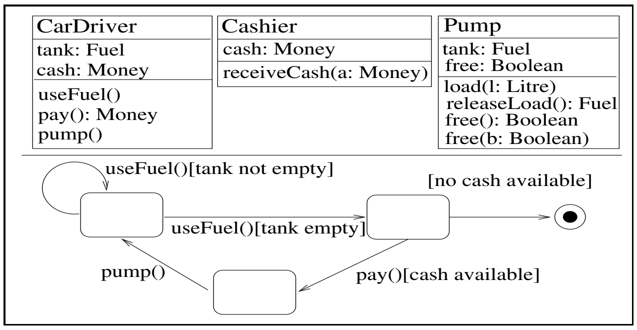

Figure 2. Component classes of the gas station and UML state diagram of the car driver.

3.2. Components

A component is an active object [5] or a group of components that are composed by connectors. Note that such a composite group must provide an interface like the objects of the base object model[2].

As shown by Figure 2, the cashier can receive and store money. The pump is a fuel server that can be loaded for an amount of fuel. Its method releaseLoad returns the loaded fuel. The car driver object stores money and fuel. Its autonomous behavior is represented by its methods invoking each other as illustrated in figure 2. It can "drive around", using up its fuel. If it has no fuel but still money, it can use this money to pay for new fuel. Then it pumps as much as possible and drives on.

Note that the car driver does not have to know, how to pay a cashier, or how to pump on a certain pump. It only knows, that it wants to pay and pump. Therefore, the CarDriver implementation can run on its own. However, if it is not connected, it gets no new fuel, thus stopping soon. It is the connector's responsibility to implement the concrete interactions, namely the correct transfer of money and fuel.

3.3. Connectors

Connectors are specialized active objects that are responsible for the *interaction* between the other components. A connector is independent of its participants, and the participants remain unaware of the connector. But a connector controls its participants by following *interaction rules* over the message passing (see section 3.4).

Roles. A connector refers to the components of the interaction, called its *participants*, by means of the *roles* they play. A group of components can play one role, while one component can play different roles in the same connector.

[2] In our implementation, the interface is only a set of selectors.

Furthermore, a component can participate in different connectors.

The following definition of the connector GasStationConnector contains three participant roles: cashier, customer and pump (line 2). E.g. the car drivers play the role customer in the gas station interactions. The rest of the definition is explained in the next section.

```
Connector subclass: #GasStationConnector;
  withRoles: 'customer cashier pump';
  withBehavior: '
 1 [ customer pay
        implies cashier receiveCash: result.
                connector  calcFuelFor: result ]
 2 [ connector calcFuelFor: a
        implies pump_select_Next_as_myPump load: result ]
 3 [ customer  pump
        impliesBefore myPump releaseLoad ]
 4 [ pump releaseLoad
        implies customer_select_REC tank: result ]'
```

Connector lifetime. Connectors are instantiated and destroyed dynamically. Once instantiated, a connector can only be activated if it has at least one participant per role. When activated it controls active objects to enforce its interaction rules (see 3.4). During its active phase, a connector can add or remove new participants. A new car driver can be added to the participants of the connector via its customer role. A connector *terminates* explicitly or when not enough participants play its roles.

The next lines shows how a connector is instantiated.

```
|gasStation customers pumps cashier|
gasStation := GasStationConnector new.
customers := Vector with: aCarDriver1  with: aCarDriver2.
pumps := ......
gasStation objects: customers playRole: 'customer'.
gasStation objects: pumps playRole: 'pumps'.
gasStation objects: cashier playRole: 'cashier'.
...
gasStation activate.
customers do: [:c | c useFuel].
...
```

Connector Behavior. To implement an interaction pattern (including coordination), a connector intercepts the participant messages, and it processes its own. It decides if participant methods (and which ones) should be invoked. The basis for decisions is a connector specific *set of rules* and the history of the interactions.

3.4. Interaction Rules

Like many other coordination approaches based on rules [9], [13], FLO/C uses *interaction rules*. Rules on message sending and dispatching yield the expressive power needed for coordination tasks. The advantage of rules are their high level of abstraction, their incrementality through composition and the ability to reason on them.

The connector GasStationConnector implements the interactions between the different components of the gas station. It enforces four rules that are designed to manage *role groups* (several customers and pumps at once). The four rules compose two sets of multi-object joint actions. The following enumeration explains the rules of the presented GasStationConnector declaration rule by rule.

Rule 1: The customer invokes its method pay which returns the amount of money the customer wants to pays. This starts a set of joint actions. The first rule ensures that the cashier gets the money[3]. As a second sequential consequence, the connector[4] calculates the amount of fuel the customer payed for. Conceptionally this could have been done by the cashier as well, but in other cases it is not obvious where to put such conversion code. So the example shows how connectors can host the conversion in such cases.

Rule 2: When the calculation of the fuel amount is done, this rule loads the pump for the resulting amount. The rule must select a particular pump since there can be several components playing the pump role. The _select_Next appendix to the role let the pumps take turn when being loaded. Note that, when there are less customers than pumps, this guarantees that two customers cannot select the same pump. Since the selection of a particular pump is needed later the _as_myPump appendix to the role stores each selection in the relative role myPump.

Rule 3: This rule starts a second set of joint actions. Before the customer executes its pump method, the pump, that was selected for it in rule 2, releases its load. This releasing action triggers the next rule.

Rule 4: The tank of the pumping customer is filled with the amount of fuel released by the pump, again using the result keyword. Note that only now, the pump method of the customer is executed.

Managing Races. The rules 1,2 and rules 3,4 form two sets of joint actions. The first one handles the payment and preparation of the pump, the second one the pumping of fuel. The global process is divided, because it is the customers free choice, when it wants to pay, and when it wants to pump. As we said in 2.3 the joint actions are atomic but because of the gap between them a race can occur when there are more customers than pumps. Then it is possible that two customers pay to pump from the same pump and the customer that pumps first will receive the fuel for both.

To prevent this kind of problem, the following connector ensures that a pump is not loaded twice. It uses the pump's free message as a lock. When a pump aready is loaded, further loading requests must wait.

[3] The strong sequential ordering properties of the implies operator offers the keyword result for right hand sided arguments.

[4] A connector can trigger messages to itself using the default role connector.

```
Connector subclass: #PumpLockConnector;
  withRoles: 'pump';
  withBehavior: '
1 [ pump load: a
        implies pump_select_REC free: false ]
2 [ pump load: a
        waitUntil pump_select_REC free ]
3 [ pump releaseLoad
        implies pump_select_REC free: true ]'
```

The PumpLockConnector only defines the role pump. Rule 1: When a pump is loaded, it is not free any more. Rule 2: The loading of a pump must wait until it is free. Rule 3: When the pump has released the load, it is free again.

The connector bridges the gap between the two joint actions of the GasStationConnector. It comes in, when the payment interaction ends, and ends, where the pump interaction finishes. By adding the PumpLockConnector to the example we can demonstrate how joint actions can be extended. The new guard in rule 2 locally protects the loading of the pump. But it also extends the payment joint actions of the GasStationConnector since the loading of the pump is a part of it. Therefore rule 2 adds a new constraint to these joint actions and rule 1 adds a new action to it. Rule 3 on the other hand extends the pumping joint actions.

The extended payment joint actions will therefore explicitly wait for the selected pump to be free, and explicitly reserve it when the payment succeeds. The extended pump joint actions will explicitly release the pumps after successful pumping of fuel. Therefore each pump only loads fuel for one customer at once and *no race condition can occur*.

3.5. Example evaluation

The FLO/C solution works for an arbitrary number of customers and pumps, thus it demonstrates FLO/C's *flexibility*. Furthermore it demonstrates how FLO/C's group managing specificators yield expressive power.

Problem	**Solution**
Concurrent client-sever interactions.	*Implies*-operators carry data in the arguments or even propagate the return value of the precondition. Conversion can be done in *connector methods*.
Managing of shared resources.	Specificators map resources to participants, joint-actions protect resources from inconsistent access.
Avoiding races.	Joint-actions work together with user-defined locks.

We dynamically added a connector, to enforce a new interaction policy, which guarantees race-freeness. This demonstrates the *incrementability* of FLO/C. Moreover it illustrates *separation of concerns*, which is also demonstrated by the fact, that e.g. the car driver objects are autonomous. Furthermore, the example showed the FLO/C solution techniques to different non-trivial *coordination problems*. Note that we have implemented traditional and recent coordination examples [8].

3.6. Details of Interaction Rules

Operators Semantics. FLO/C uses five operators to specify interaction rules. The permittedIf and the waitUntil operator express guards; the permittedIf operator supports balking style, the waitUntil operator supports blocking style. The implies and impliesBefore operators enforce computational ordering; The implies operator supports push style, the impliesBefore operator supports pull style. All four operators protect the objects involved in the rule: left and right hand side of the rule are executed atomically. The low-level communication tasks are met by the impliesLater operator that features *asynchronous consequence sending*. A formal description is available in [8].

Collaboration of Connectors. FLO/C composes simultaneous triggering rules at run-time, and fuses them to multi-object joint actions in a uniform way.

The sending of a rule triggering message (request) to a connected active object leads to the interception of the message. Then the FLO/C model's global reaction covers three phases; (1) the *consequence collecting* phase (all the connectors attached to the active object start to collect consequences and return a list consisting of three different kind of messages: the sequential ones (including the intercepted message), their guards, and their asynchronous consequences) Note that FLO/C detects and breaks cycles. (2) the *protected execution* phase (It executes multi-object joint actions starting with the *internal reservation* of all the participants, then the guards are executed. If all the guards succeed, the sequential consequences are executed) and (3) the *unprotected sending* phase (the asynchronous consequences of all the methods that were previously executed are sent asynchronously).

Group Management. In the right part of a rule, a role refers to a group of objects and per default sending a message to a role broadcasts it to all the group objects. The appendix _select_ to a role (called *specificator*) selects particular objects like the receiver of the controlled message (REC), all the objects except the receiver (Others) or the next object in the group (Next) (see rules 2 and 4 of GasStationConnector). Note that FLO/C also allows user-defined selection policies.

Relative Roles. For coordination of shared resources it is often convenient to refer to a selected object in another rule. FLO/C supports the definition of relative role names using the role appendix _as_*relativeRoleName*. Without going into detail, this associates the selected object with the receiver of the request that triggers a set of joint actions.

In the GasStationConnector connector, rule 2 defines a relative role named myPump that refers to the next available pump. Note that the receiver of the request that triggers the payment joint actions is a *customer* an not the connector (compare rules 1 and 2). Thus rule 2 stores the selected pump for a particular customer in myPump and the particular selection is used when the customer will pump fuel later like in rule 3.

4. Conclusion

With FLO/C we introduced an object oriented model for coordinating active objects. Where in modern programming languages (like JAVA), coordination is implemented in low-level constructs, FLO/C offers explicit, rule based connectors for coordination. In order to treat coordination at a higher level FLO/C introduces five operators, featuring two conditional synchronization policies and three communication policies. The paper demonstrates the sufficiency of the expressive power of this minimal set. FLO/C divides programming in computation (done in active objects) and coordination (done in connectors). Thus it directly maps architectural design, and enforces the separation of concerns. Composite active objects allow the mapping of hierarchical design and improve the scalability of the model. Connectors as explicit rule-based coordinators profit of the incrementability of rules. They collaborate through a uniform rule fusion protocol.

Our model addresses the same problem space as the synchronizers of Agha and Frølund [7]. Both introduce independent support constructs for multi-object coordination of active objects. However, we extend synchronizers in the following dimensions:

1. The coordination is no longer limited to the state of the connector itself. A connector can take into account the *state of its participants* and the *history* of the coordination. Moreover a connector is not limited to the synchronization of objects. A connector *enforces* coordination of objects by invoking participant services.

2. A connector is a complete *dynamic entity* that can be dynamically created and destroyed. FLO/C is completely dynamic, it can establish and cancel connections at run-time, and it allows new conectors to be created on the fly.
3. A connector manages *groups* of coordinated objects and supports dynamic addition or removal of participants.
4. The proposed model is *uniform* and open: a connector is an active object and not a specific construct, and the model allows one to extend its rule semantics.

FLO/C has been fully implemented in NEOCLASSTALK a new SMALLTALK implementation providing explicit metaclasses. All the material presented in this paper is freely available at the authors' web pages.

4.1. Future work

We implemented the FLO/C model on a single processor machine, using this fact to simplify the implementation. Future work will address real distribution. We claim that the FLO/C model and distributed systems infrastructure such as CORBA can form a basis for a real distributed FLO/C implementation. FLO/C's separation of concerns will pay off even more when used in a distributed environment. Active objects reside in different physical locations. Connectors form bridges over a network. FLO/C model extensions for distribution include additional declarations of *location* and *mobility* of active objects an connectors. We also need to address the low-level coordination tasks (e.g. conversion, real-time support) we omitted in this work. The handling of communication failures and roll-backs of synchronized joint actions also need considerable further efforts.

In [8] a formal specification of the presented model has been defined. An interesting future work could be an automatic translation of architectural design with formal connectors [3] to FLO/C code, as well as query languages, to prove properties of FLO/C examples (like in [14]).

References

[1] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
[2] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. of ICSE'94*, 1994.
[3] C. Arapis. *Dynamic Evolution of Object Behaviour and Object Cooperation*. PhD thesis, University of Geneva, 1992.
[4] F. Arbab. The IWIM model for coordination of concurrent activities. In *COORDINATION'96*, 1996.
[5] J. Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *ECOOP'89*, pages 109–129, 1989.
[6] N. Carriero and D. Gelernter. *How to Write Parallel Programs: a First Course*. MIT Press, 1990.
[7] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. of ECOOP'93, LNCS 707*, 1993.
[8] M. Günter. Explicit connectors for coordination of active objects. Master's thesis, University of Berne, 1998.
[9] S. F. J.-M. Andreoli and R. Pareschi. The coordination language facility: coordination of distributed objects. *TAPOS*, 2(2):77–94, 1996.
[10] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
[11] C. V. Lopez and G. Kiczales. D: A language framework for distributed programming. Technical Report TR SPL97-010P9710047, Xerox Parc, 1997.
[12] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1), Mar. 1994.
[13] N. H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In *Proc. of Coordination'97*, pages 81–97, 1997.
[14] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. Osterweil. Applying static analysis to sofware architectures. In *Proc. of ESEC/FSE'97, LNCS 1301*, pages 77–93, 1997.
[15] M. Papathomas, G. Blair, and G. Coulson. A model for active object coordination and its use for distributed multimedia applications. In *LNCS 924*, pages 162–175, 1995.
[16] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.