Explicit Connectors for Coordination of Active Objects

Masters Thesis
of the Faculty of Science
University of Berne

by **Manuel Günter**
20.3.1998

Supervisors:
Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz
Institute of Computer Science and Applied Mathematics

Contents

1 Introduction **11**
- 1.1 Overview . . . 11
- 1.2 Structure of the Thesis . . . 13
- 1.3 How to Read the Thesis . . . 14

2 Multi-Object Coordination Support **15**
- 2.1 Problems with Traditional Coordination Approaches . . . 15
- 2.2 Constraints and Goals . . . 16
  - 2.2.1 Modeling Activities as Active Objects . . . 17
  - 2.2.2 Coordination Goals . . . 17
  - 2.2.3 Constraints of the Rule Based Approach. . . . 20
- 2.3 Coordination Goals and Further Extensions. . . . 20

I The Model **21**

3 The Paradigms of the FLO/C Model **23**
- 3.1 Components and Connectors . . . 23
  - 3.1.1 Components . . . 23
  - 3.1.2 Connectors: an Overview . . . 24
  - 3.1.3 Separation of Concerns between Connectors and Components. . . . 24
- 3.2 Explicit Connectors . . . 25
  - 3.2.1 Static Properties . . . 25
  - 3.2.2 Dynamic Properties . . . 25
  - 3.2.3 A First Example of a Connector Declaration . . . 26

4 Connector Behavior **29**
- 4.1 Syntax and Elementary Semantics of Interaction Rules . . . 29
- 4.2 The Semantics of the Operators . . . 30
  - 4.2.1 Balking Conditional Synchronization: permittedIf . . . 31
  - 4.2.2 Blocking Conditional Synchronization: waitUntil . . . 31
  - 4.2.3 Push Style Temporal Ordering of Execution: implies . . . 32
  - 4.2.4 Pull Style Temporal Ordering of Execution: impliesBefore . . . 32
  - 4.2.5 Asynchronous Communication: impliesLater . . . 32
  - 4.2.6 Evaluation . . . 33
- 4.3 Collaboration of Connectors / Fusion of Rules . . . 33
- 4.4 Group Management . . . 35

4.4.1 Role Semantics . . . 35
4.4.2 Roles in the Dining Philosopher Example . . . 35
4.4.3 Specificators . . . 36
4.4.4 Relative Roles . . . 37
4.5 Special Features . . . 37
4.5.1 Self Controlling of Connectors . . . 37
4.5.2 FLO/C's Additional Exception Mechanism . . . 38
4.5.3 Propagation of Computation Results . . . 38
4.6 Categories for the Expressive Power of Connectors . . . 39

5 An Illustrating Example: The Gas Station 41
5.1 The Participants . . . 41
5.2 The Connectors . . . 42
5.2.1 Managing Races . . . 43
5.3 A Complete Simulation . . . 44
5.4 Example Evaluation . . . 45

6 Component Hierarchy 47
6.1 Declaration of Composite Object Classes . . . 48
6.2 Inheritance of Composite Object Classes . . . 52
6.3 Evaluation + Limitations . . . 53

7 Discussion of the FLO/C Model 55
7.1 FLO/C Fulfills its Requirements . . . 55
7.2 Limited Pull-Style Support . . . 56
7.3 The Polling of the waitUntil Operator . . . 56
7.4 Tradeoff between Connector and Component Responsibilities . . . 57

II Formal Approach 59

8 Formal Specification of FLO/C 61
8.1 Notations . . . 61
8.2 System Entities . . . 62
8.3 System State Transitions . . . 63
8.3.1 Local Transition Firing Order . . . 64
8.3.2 Two Requirements for FLO/C's System Transitions . . . 65
8.4 Functions . . . 66
8.5 Analyzing Possible Execution Orders of an Example System . . . 68
8.5.1 The Start Transition and a Function Trace . . . 69
8.5.2 Execution of the Consequences . . . 70
8.5.3 Evaluation of the Example . . . 71

9 Properties of the Formal FLO/C Model 73
9.1 Execution Properties of the Operators . . . 73
9.1.1 The implies Operator. . . . 74
9.1.2 The impliesBefore Operator. . . . 76
9.1.3 The Guard Operators. . . . 77

9.1.4 The impliesLater Operator. 77
9.2 Liveness Properties of the FLO/C Model 78
9.2.1 Deadlock 78
9.2.2 Livelock 78
9.2.3 Operator Loops 80
9.3 Limitations of the Formal Approach 80
9.4 Summary 81

III Implementation 83

10 Implementation Overview 85
10.1 Layering of the Implementation 85
10.2 Technical Data of the FLO/C Implementation 86
10.3 Choice for an Open and Reflective Environment 86
10.3.1 SMALLTALK 86
10.3.2 NEOCLASSTALK 87
10.4 Meta-Level Programming 88
10.4.1 Explicit Meta-Classes 88
10.4.2 Meta-Objects 88

11 The FLO/C Kernel 89
11.1 Implementation of Active Objects 89
11.1.1 ACTALK's Asynchronous Message Passing System 89
11.1.2 FLO/C's Asynchronous Message Passing System 89
11.1.3 Message Interception 91
11.2 Controller 92
11.3 Connector 93
11.3.1 Responsibility of Class MetaConnector 94
11.3.2 Responsibility of Class Connector 94
11.4 Collaboration of Controllers and Connectors 95
11.5 Composite Objects 96
11.5.1 Responsibilities of the Composite Object Class and Meta-Class 96

12 Visual Programming Tools 99
12.1 The FLO/C Workspace 99
12.2 The Composite Object Class Browser 101

13 Performance Optimizations 105
13.1 Problems with the First Implementation 105
13.2 Optimizing the Rule Lookup 106
13.3 Caching Consequence Messages 107
13.3.1 Working Principle 107
13.3.2 Value of Arguments 108
13.3.3 Caching in a Dynamic Implementation 110
13.4 Evaluation of the Optimization 111

14 Implemented Examples **113**
14.1 The Vending Machine . . . 114
14.1.1 Description . . . 114
14.1.2 Solution . . . 114
14.1.3 Evaluation . . . 118
14.2 Synchronized Movements . . . 119
14.2.1 Description . . . 119
14.2.2 Solution . . . 119
14.2.3 Evaluation . . . 120
14.3 An Unstable Server for a Client . . . 120
14.3.1 Description . . . 120
14.3.2 Solution . . . 120
14.3.3 Evaluation . . . 121
14.4 The Decrementor . . . 122
14.4.1 Description . . . 122
14.4.2 Solution . . . 122
14.4.3 Evaluation . . . 123
14.5 Workers and Tools . . . 123
14.5.1 Description . . . 123
14.5.2 Solution . . . 124
14.5.3 Explicit Locks and Processor Yields . . . 125
14.6 The Binary Adder with Logical Switches . . . 125
14.6.1 Description . . . 125
14.6.2 Solution . . . 125
14.6.3 Evaluation . . . 127
14.7 The Dining Philosophers . . . 127
14.7.1 Description . . . 127
14.7.2 Solution . . . 128
14.7.3 Evaluation . . . 129
14.8 Administrator and Workers . . . 129
14.8.1 Description . . . 129
14.8.2 Solution . . . 130
14.8.3 Evaluation . . . 132
14.9 The Electronic Vote . . . 132
14.9.1 Description . . . 132
14.9.2 Solution . . . 133
14.9.3 Evaluation . . . 134
14.10 The Sleeping Barber . . . 134
14.10.1 Description . . . 134
14.10.2 Solution . . . 135
14.10.3 Evaluation . . . 139
14.11 Evaluation of the Examples . . . 139

IV Finale **143**

15 Related Work **145**
15.1 Coordination 145
15.1.1 Explicit Entities for Multi-Object Coordination 145
15.1.2 Factoring Out Per-Class Coordination 146
15.2 Architectural Design 147
15.2.1 Formal Approaches for Architectural Design 147
15.2.2 Connectors at Run Time 147
15.3 Active Object Models 148

16 Conclusion **149**
16.1 Future Work 150

Acknowledgments

First and foremost I want to thank my supervisor Dr. **Stéphane Ducasse** for his inspiring energy and enthusiasm and his lovely wife Florence for her patience listening to my French.

I enjoyed writing my thesis within the Software Composition Group (SCG) founded by Prof. Dr. **Oscar Nierstrasz**. I thank him for bringing new impulse to the Institute of Applied Mathematics of Berne. Within a short time he formed a successful international research team that is comfortable, informative and productive to work with.

The following thanks go to the SCG group. I want to thank

- Franz for sharing the C++ student crash course with me.
- Isabelle for occasional cakes.
- Jean-Guy for the SCG basketball domination.
- Juan for his marvelous presentation of FLO/C during the lecture.
- Markus for fueling the discussions at the group meetings.
- Matthias for teaching me how to juggle.
- Patrick for being "on my side".
- Rob for sharing cokes with me.
- Sander for visiting the derniere of a fine musical ensemble.
- Serge the SMALLTALK guru for finally having told me about the connection between dictionaries and hash.
- Tamar for English support.

Furthermore I want to thank my fellow students:

- Daniel for uplifting discussions on the structure of scientific revolutions.
- Tob for playing the ace of spades (when I got the queen).

A special thanks goes to my family for their endurance in supporting me.

Chapter 1

Introduction

1.1 Overview

The trend to interconnect software systems increased the pressure on object oriented (OO) programming languages to *include* concurrency control features. Thread packages that are not integrated into programming languages tend to be hard to use [Lea97]. Thus for example JAVA, a recent OO programming language introduces a small and consistent set of object synchronization primitives. However, Bloom [Blo79] already pointed out the need of *factoring out* synchronization code (in non OO context). This was also reflected in OO synchronization approaches like the generic synchronization policies [McH94] or the D-language [LK97]. Such approaches feature separation of concerns and high level synchronization abstractions instead of built-in primitives. However their declarations affect only single class declarations therefore they do not cover the field of *multi-object* synchronization, where different kind of objects share resources concurrently. The management of such dependencies between otherwise independent objects is called *coordination*. Our FLO/C model is situated in this context: we want to introduce an OO model that factors out high-level coordination abstractions. Our model is situated in the same area as the synchronizers of Agha and Frølund [FA93]. We both use rules to enforce interaction behavior. However, in FLO/C the rules are supported by explicit coordinator objects that allow us to achieve *run-time flexibility* like the dynamic exchange of coordination policies.

Instantiating coordination in specialized objects has a tradition in the domain of *software architecture design*. There the distinction between *components* and *connectors* was introduced to address the need of decoupling *domain specific design* from *collaboration design*[SG96]. Architectural connectors represent design decisions concerning the collaboration of software components. Allen and Garlan [AG94] present a specification language for connectors, which has descriptive and *analytical* properties such as component substitutability or dead-lock detection. Unfortunately, when such valid designs are implemented in a traditional programming language, design decisions concerning object interactions get lost because languages have no appropriate construct for connectors.

A similar problem occurs, when high level coordination abstractions such as transactions are implemented in languages which feature only low level synchronization constructs as in JAVA. The domain specific code is interleaved with synchronization code, which is hard to validate, to document, and sometimes impossible to reuse (inheritance anomaly [MA93]). Furthermore, the composition of software pieces that are coordinated at a low level of abstraction can lead to hard predictable liveness problems like the nested monitor problem.

In this masters thesis we introduce the FLO/C model which takes up the software architecture

design idea of separating components from connectors and applies the idea to the implementation level of concurrent object-oriented programming. Our explicit connectors [DBFP95] *implement* the interaction of components, therefore they are the ideal location for *multi-object coordination* code. 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 constraints on the *method execution order*. FLO/C's connectors are *abstractly* defined, they only rely on the interfaces of the objects. Thus, they are are *independent* from the implementation of the coordinated components. This allows clear *separation of concerns*: A connector defines coordination between a group of components which it refers to by the *roles* they play in the interaction. Components only define their proper functionality therefore they are independent of other components' interaction behavior. The components in FLO/C encapsulate the concurrency of activities in an object oriented way by encapsulating their own threads. FLO/C's components are represented as active objects similar to those in Briot's ACTALK [Bri89], which use asynchronous message passing.

Thus, FLO/C takes up ideas of architectural software design to contribute the following achievements to the area of object oriented concurrent programming:

- The FLO/C model maps architectural connectors to run-time entities, thus preserving the design at the implementation level.
- The FLO/C model offers constructs to declare high-level coordination.
  - High-level coordination patterns such as transactions, multi-object constraints and synchronized joint actions are feasible without expert knowledge. Instead of keywords hidden and sprayed all over the code FLO/C instantiates run-time entities that handle the component interactions.
  - Coordination declarations can be added incrementally with a minimal risk of liveness problems.
  - Coordination code is not wired to a components' class declaration.
  - A group of components can be coordinated as one.
  - Coordinated components can be composed to a component again, thus supporting object hierarchies.
- The separation of concerns in FLO/C supports the reuse, maintenance and evolution of the coordinated- as well as of the coordination code.
- Since coordination code is encapsulated in connector objects, FLO/C features run-time flexibilities like dynamic exchange of coordination policies.

In part II of the thesis we present a formal model of FLO/C as a basis to analyze the coordination of a given FLO/C program. The formalism also provides requirements for a FLO/C model implementation and is used to prove liveness properties of FLO/C systems.

FLO/C has been implemented in NEOCLASSTALK a new SMALLTALK implementation providing explicit meta-classes [Riv96a]. We implemented eleven coordination examples (six taken from the coordination literature) to evaluate the model. All the material presented in this thesis is freely available at the author's web pages. http://www.iam.unibe.ch/~mguenter/ .

1.2 Structure of the Thesis

The next chapter will explain why the current support for multi-object coordination in OO languages is insufficient. It will also define what we understand by the term "coordination" and to which aspects we will constrain our work. The rest of the work is divided in four parts. Part I describes the model, part II takes a formal approach, part III describes the implementation of the model and its validation by coordination examples and part IV concludes.

The following list summarizes the context of each part and chapter.

Part I - the FLO/C model This part describes the FLO/C model from an object-oriented point of view.

- **3** Here we justify why the FLO/C model distinguishes two kind of objects: the components and the connectors.
- **4** After the conceptual view we introduce the behavior of FLO/C mainly focusing on connectors, because they are the entities that implement coordination.
- **5** This chapter is dedicated to an example where FLO/C is used to coordinate objects that simulate the concurrent activities at a gas station. The example will illustrate how FLO/C uses active objects to implement domain specific behavior and connectors to model interaction as well as coordination. A connector is added to solve the race-condition inherent in the example and described in [NACO97].
- **6** Here we will present how encapsulated *object hierarchies* can be declared in the FLO/C model.
- **7** limitations of the model, like its push-style preference are discussed.

Part II - FLO/C's formal base model Here we formally specify FLO/C's active object coordination.

- **8** We formally describe how FLO/C rules guide the interactions between concurrently running objects. We demonstrate how a given FLO/C program's coordination can be analyzed.
- **9** This chapter contains proofs about execution properties of a system that is constrained with different types of FLO/C rules. Furthermore, it proves properties of the FLO/C model as such e.g. the impossibility of deadlocks.

Part III - implementations Here we explain the implementation of the FLO/C model using SMALLTALK.

- **10** This chapter gives an overview of the SMALLTALK language extensions, and the meta programming techniques used to implement FLO/C.
- **11** Here we present the architecture of the implementation kernel.
- **12** We present two visual tools that support the programming in FLO/C.
- **13** This chapter describes our various efforts to optimize the performance of the FLO/C implementation.
- **14** An important part of our implementation work consists of a collection of FLO/C solutions to different coordination problem examples. We present this collection here.

Part IV - finale This part summarizes the achievements of FLO/C.

15 Here we compare our approach to recent approaches in similar areas.

16 The achievements are summarized and the future development of FLO/C towards distributed systems is sketched.

1.3 How to Read the Thesis

The thesis addresses most of the authors work when developing FLO/C. Therefore, it contains a lot of material that might distract the reader from the core of the FLO/C model. Here is our suggestion for a short-cut through the thesis.

Finding the core of the thesis. The chapters that immediately follow are strongly recommended. First they treat the context, constraints and goals of the model and then they describe it. From section 4.4.4 on the rest of chapter 4 describes features less central to the model therefore these sections can be skipped. When reading chapter 4 we recommend to peek into the example presentation of chapter 5. The chapter 6 is about the object hierarchies of FLO/C and should be read but the limitations (chapter 7) at the "far end" of the model can be omitted. We recommend to skip the formal specification of FLO/C in part II but to read its summary (section 9.4). Then the implementation part III can be skipped except from the beginning and the last section of chapter 14 which describe and evaluate ten implemented coordination examples. Finally the conclusion of section 16 is crucial. Here is a short notation describing where the core of FLO/C is found:
([2 - 4.4.3] , [6 - 7[, 9.4 , [14 - 14.1[, 14.11 , 16)

Chapter 2

Multi-Object Coordination Support

Malone and Crowston suggested the following definition for coordination:

> Coordination is managing *dependencies* between *activities* [MC94].

In this chapter we discuss why coordination support in traditional object oriented coordination language is insufficient (section 2.1). In section 2.2 we list the constraints we put on our model to keep it simple for implementation but general enough for extensions. We restrict FLO/C to use a simple model of *active objects* (section 2.2.1). Furthermore, FLO/C does not cover the complete area of coordination described by the above definition. Section 2.2.2 refines which coordination tasks FLO/C addresses. Then section 2.2.3 discusses the constraints coming from FLO/C's simple rule semantics using the meta-level. Finally in section 2.3 the coordination goals of the FLO/C model are summarized, and further extensions discussed.

2.1 Problems with Traditional Coordination Approaches

Traditional OOCP languages offer little support for the synchronization of concurrent objects [FA93]. JAVA for example, which is a recent OO language that was designed for use in concurrent settings, models coordination at a very low level of abstraction. In JAVA*Threads* model concurrent activities. They communicate through unprotected, shared memory. Dependency managing constructs are the synchronized keyword, to protect data, and the messages notifyAll(), wait() to synchronize activities on activity raised events. While the set of constructs is in theory sufficient to solve any coordination problem, in practise only experts are able to handle non trivial tasks. The lack of higher level coordination support is documented by the fact that JAVA users tend to rely on design pattern collections like Doug Lea's "Concurrent Programming in Java" book [Lea97] to solve common coordination problems. But even with such pattern based approaches, protocols used for establishing the coordination between different groups of activities are hard-coded into parts of the activities, resulting in poor abstraction facilities like composition and evolution of the coordination policies [LK97]. When discussing *atomic transactions* involving different kinds of objects, Lea says:

> One of the principal disadvantages of transactional techniques is that conformance to a transaction protocol impacts the *method signatures* and *implementations* of every participant class. This often spreads in turn through most classes in an entire application or framework. Worse, it is often troublesome at best to make a set of classes using one set of standardized transaction interfaces to work with those using another. And because

> "transactionality" tends to infiltrate the details of ground-level code, these problems resist smoothing over via Adapters and the like [Lea97]. (p.259)

Later on in the same book, another multi-object synchronization abstraction is discussed. The *synchronized joint actions* are guarded methods that involve conditions among multiple participating objects. In respect to the presented JAVA solution Lea says:

> The combination of direct coupling and the need to exploit any available constraints to avoid deadlock accounts for the high context dependence of many joint action designs. *This in turn can lead to classes with so much special purpose code* that they must be marked as final [Lea97]. (p.284)

We can conclude that without explicit coordination support, complex concurrent designs can cripple basic object oriented features as object interfaces (quote 1) and inheritance (quote 2).

We categorize the main problems using traditional approaches for coordination, and give a hint how FLO/C will address them.

- **Absence of abstraction.** As we just showed, object oriented languages need high-level coordination abstractions. There must be means to declaratively specify coordination. FLO/C uses *rules* to declare per-object coordination.
- **No separation of concerns.** Expressing coordination abstraction is difficult because the code that manages the coordination is closely tied to the implementation of the coordinated objects. This hampers the maintenance as well as the reuse of the code. FLO/C introduces *connectors* as run-time entities that are independent of the object they control. By reusing components separately of all synchronization, connectors can even avoid the infamous inheritance anomaly [MA93].
- **Do it yourself**. This problem refers to the fact that the programmer has to manually implement all the mechanisms that will support the coordination. This task is particularly difficult and error-prone. Doing so the programmer should first focus on the tools and mechanisms instead of just expressing his wished coordination. With its commitment to architectural software design FLO/C offers means to directly map a valid design onto code.
- **Lack of flexibility.** When coordination is not explicitly and abstractly expressed, it is difficult to modify and to customize the coordination policies. Furthermore, if the coordination is per-class, objects cannot change their coordination policy at run-time. FLO/C's connectors can coordinate dynamically changing groups of active objects.
- **Lack of composability.** Composing different coordination policies is difficult without changing the code of the coordinated objects. Furthermore the composition of already synchronized components can lead to liveness problems (nested monitors, deadlocks). FLO/C is a model with minimal liveness problems as shown in section 8. It supports the composition of coordination policies (connectors) as well as the composition of components.

2.2 Constraints and Goals

The next three subsections discuss the focus of FLO/C in the wide area of coordination covered by the definition of Malone and Crowston. We argue why and how we model "activities" and "coordination", and what kind of rules we use to coordinate such activities.

2.2.1 Modeling Activities as Active Objects

Usually, the activities are modeled as threads or processes. Since these concepts cross object borders, several different approaches mapped threads to objects, thus enforcing object encapsulation: Actors [Agh86], ACTALK [Bri89] and more recently ATOM [Pap96], CodA [McA95] 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 [Bri89], we have chosen a variant of its active object notion to model an activity.

In ACTALK, an active object consists of a *behavior object*, an *activity object* and a *queue*. The behavior object is a normal object, encoding a domain specific behavior. It is the logical target of messages coming from other active objects and the sender of new messages. But all messages to the behavior object are redirected to its queue (asynchronism). The activity object running in its own thread accesses the queue. It decides what messages should be executed (what method of the behavior object should be called). The activity object is responsible for intra-object synchronization. It is the only object to access the behavior object in a synchronous way.

In our approach the activity does not fork new threads. Therefore, the active object does not have *intra*-object concurrency. This model directly maps the coordination of activities to coordination of active objects. Thus, *a* FLO/C *system features as many concurrent activities as it contains active objects.* Not to have intra-object concurrency is not a severe limitation, because FLO/C offers a way to compose active objects into a composite object that behaves like an active object again (see section 6). Such an object features intra-object concurrency, since it contains several active objects.

Our active objects differ from the ACTALK model in their initialization and termination. At instantiation time, an object is still passive (no own thread) until it receives an explicit activation message. Once activated it is possible to gracefully terminate an active object.

2.2.2 Coordination Goals

According to Carriero and Gelernter [CG90], we can build a complete programming model out of two pieces - the computational model and the coordination model. 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. For the multi-object coordination of FLO/C this includes the following tasks:

- **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.** There are two tasks 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.

From the point of view of a single active object, it can accept requests for computation, check if it is in the right state and then compute, thereby changing the state as shown in figure 2.1. Furthermore, upon failure of the state checks, a request can be *denied* (balking guard), or *blocked* in order to be tried again later[1].

[1]The retry-policy is a design dimension itself.

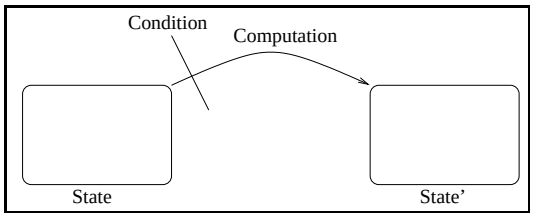

Figure 2.1: A single object's conditional state change.

Generalized to groups of objects, the state of the group is defined by the states of its objects. A group of objects can accept requests according to its global state. Therefore, guards on several objects must succeed, in order for several computations to lead to a consistent global state again (see figure 2.2). State transitions can be *pushed* or *pulled*. A push style request pushes the next computations in order to render the object group consistent again. A pull style request pulls the computations that must be accomplished before in order to reach a consistent group state. We said that according to Malone and Crowston coordination is managing dependencies between activities. Note that the order of state transitions and the guards reflect such dependencies. A single computation *depends* on other computations and on different objects' (guards') states.

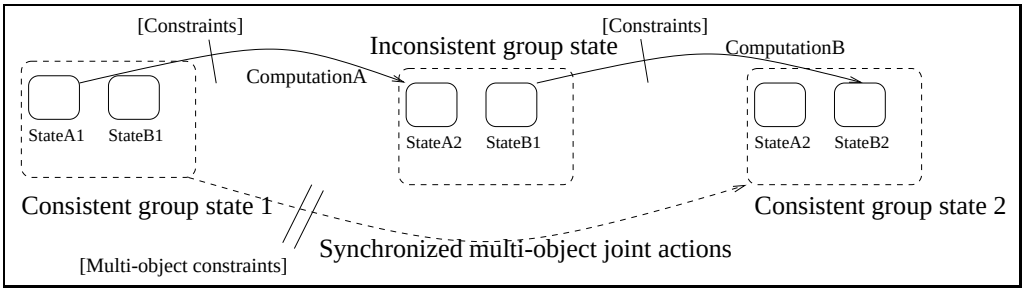

Figure 2.2: Multiple object's conditional state change.

We call this abstraction ***synchronized multi-object joint actions***, or simply *joint actions*. We can derive the following coordination requirements for the abstraction:

- In order to stay consistent, the group must be access protected when evaluating the guards.
- The group must be access protected when executing state changing computations.
- There must be ways to *temporary order computation*, since intermediate computations could rely on other computations' results or object states.

FLO/C shall provide constructs to easily compose such multi-object joint actions, which we believe to be powerful enough to solve any coordination problem in this area. When we compare to the coordination definition of Carriero and Gelernter, the multi-object joint actions can be used to achieve their coordination tasks *mutual exclusion* and *conditional synchronization.* Conditional synchronization is already reflected by the guards for the state transition. Mutual exclusion of a resource can be modeled as object groups, each containing the resource, and each accessing the resource by multi-object joint actions. Since we said that such actions protect the group from third-party access, figure 2.3 shows that mutual exclusion is for free.

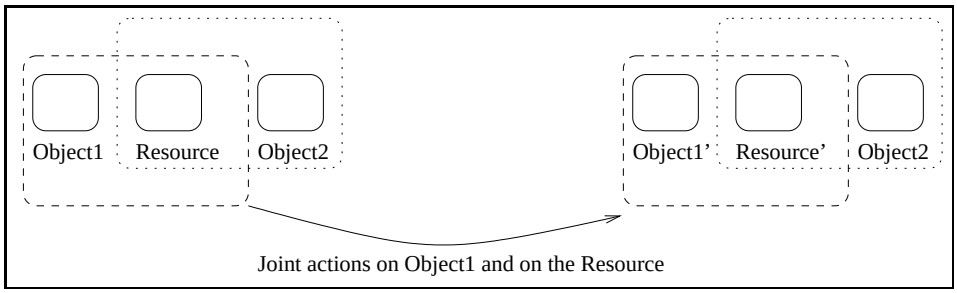

Figure 2.3: Mutual exclusion of two objects on a resource.

Multi-object joint actions		Communication
aspects	*styles*	*aspects*
guards	balking	transition request
	blocking	data-flow
computation ordering	push	multi-casts
	pull	
access protected		asynchronous

Figure 2.4: Coordination abstractions.

Here object1 exclusively computes on the resource which is in the same group, excluding object2 from accessing the resource.

Therefore, according to the definitions of Carriero and Gelernter the multi-object joint actions abstraction will suffice to express multi-object coordination.

Note that single-object synchronization is just a special case of the abstraction, where the group only contains one object. 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.

FLO/C provides ways to specify such joint-actions plus a low level asynchronous communication mechanism to offer the possibility to program light-weight solutions. Table 2.4 summarizes the coordination abstractions addressed in FLO/C.

However, the two mechanisms do not cover all possible "dependencies" covered by the Malone and Crowston's definition of coordination. The FLO/C model does not address:

- **Optimistic transactions.** FLO/C does not offer mechanisms to automatically preserve domain specific object states, therefore it cannot perform *roll-backs* on executed computations.

- **Low-level, close to the machine responsibilities.** Real-time support, or conversion of language specific data-formats (usability dependencies) are not supported.

- **Real Distribution.** Currently, FLO/C does not feature the physical distribution of active objects, although the model is designed for such an extension, which we consider future work.

2.2.3 Constraints of the Rule Based Approach.

FLO/C achieves coordination of components by rules over their message passing and message execution. Connectors control the message passing of their components and ensure that a set of rules holds. The schema of the control process is simple. Before a component is about to execute, the connector checks, if this is consistent with the rules. The connector can inhibit the execution, trigger other executions and propagate other messages. This connector behavior can be achieved by changing the meta-object protocol for message passing [Duc97a]. FLO/C thus uses the constructs of the base object model, changing only the message passing protocol. Real-time features are therefore only available if they are supported by the base model. On the other hand, FLO/C can be put on any OO language with an open meta- object protocol like Open C++ [CM93], Meta-Java [Gol97], Clos [KdRB91] and NEOCLASSTALK (SMALLTALK) [Riv96b].

2.3 Coordination Goals and Further Extensions.

FLO/C is an object oriented model for multi-object coordination. It eliminates fundamental problems (nested monitor problem, inheritance problem) that occur when manually coding high-level coordination abstractions, such as transactions and joint actions. It does so by factoring out coordination and interaction code as stand-alone objects. FLO/C is a highly dynamic run-time object-oriented model using one powerful coordination abstraction and allows composition and encapsulation of component groups in a transparent way. While we constrain FLO/C's collaboration support to collaboration of concurrent threads written in a single language, we believe that the model can be used as a basis for extensions towards real distributed components implemented in different languages as discussed in section 16.1.

As the next section will show, FLO/C enables the developer to preserve design decisions in the implementation, because it maps to the component/connector view of software architecture design.

Part I

The Model

Chapter 3

The Paradigms of the FLO/C Model

In software architecture design, Allen and Garlan [AG94] differentiate between components and connectors. Each piece of software carries implicit expectations how to interact with it. Architectural connectors offer the possibility to factor out these constraints and reason about them.

In the area of parallel programming Carriero and Gelernter [CG90] differentiate between computation and coordination. Furthermore, in concurrent programming Bloom [Blo79] pointed out the need to factor out synchronization code.

The FLO/C model *unifies* these concepts. Therefore connectors *implement all interaction between the components*, thereby coordinating them. Synchronization code is contained solely in connectors.

The unification of the concepts is possible because the distinction between interaction and coordination is a fuzzy one. Note that if a designer thinks to know the difference and wishes the further separation of pure interaction from coordination code, then the simple collaboration protocol of connectors allow him/her to use separate connectors for each category.

In this section, we present FLO/C's concept of components and connectors. We will mainly focus on connectors, because their responsibility for interaction includes the *coordination* of the components.

3.1 Components and Connectors

3.1.1 Components

FLO/C components are responsible for modeling *domain specific* entities. Unlike for example components of open coordination component frameworks [CTN98] *no* FLO/C component is concerned about its coordination with others. Since FLO/C is object-oriented, its component model is based on an *active object* model [FA93, Bri89] (see section 2.2.1), which on its turn is based on an object model. A component is an active object or a group of components that is composed by connectors. Such composite groups must provide an interface like the objects of the base object model[1]. In section 6 we will explain FLO/C's concept of component hierarchies.

Since connectors are responsible for the interaction between components, the components should keep *no references* to each other. Only if this requirement holds, the connectors can make the individual interaction constraints transparent. Components are considered as complete and stand-alone units that are separated from each other and run in different threads. However, components also

[1] In our implementation using SMALLTALK, the interface specification is only a set of selectors.

have responsibilities, they model domain entities by carrying their functionality and data. This responsibility can be delegated to passive helper objects of the base object model. Since there is no intra-object concurrency, these helper objects need no synchronization, the communication between an active object and its passive helpers is synchronous. In order to keep consistency, the helper objects must be contained by the component, so that the outside world can only interact with them through the component. According to Lea [Lea97](p.44) such structural isolation of helper objects avoids synchronization problems and is a style often seen in traditional concurrent programming. Note that the object model of FLO/C is hybrid; it contains active and passive objects. One advantage is that a FLO/C program can interact with any object of the base object language.

3.1.2 Connectors: an Overview

The understanding of FLO/C's connectors is essential to this thesis, since they encapsulate coordination. Therefore, their description requires its own chapters (section 3.2 and the following). FLO/C's connectors are specialized active objects that are responsible for the *interaction* between the other components. But they do not only offer communication channels for components. Connectors rather *enforce* interactions between some components. Since FLO/C's connectors are active *objects*, they can be manipulated at run-time, therefore we classify them as *explicit* connectors. Connectors *implement* design decisions concerning the interaction of components.

3.1.3 Separation of Concerns between Connectors and Components.

On one hand different recent approaches of coordination [LK97][McH94] acknowledge the fact that separation of concerns (coordination vs. domain specific behavior) improves the components reusability. In object oriented programming on the other hand the law of demeter [LH89] was introduced to improve reusability and software evolution. The law restricts method calls in methods. A method should only call methods of self, of its argument objects or methods of objects that were created by the method itself.

In the FLO/C model we go even further and demand that the components at no point in time keep references to each other. Therefore, they cannot send messages to each other but only to self and to helpers that must be private and passive. Thus no component is "polluted" by assumptions on how it must coordinate with other independent software pieces. It contains only its domain functionality, using close helpers. A component can be used to collaborate in another environments by attaching it to different connectors. We claim that our extreme version of the law of demeter will ease reuse and code evolution even more than the original one.

The reader may wonder how components may coordinate when they cannot communicate. Connectors are the answer. They implement the components' interactions by controlling the components internal activities and coordinating them.

An example. To illustrate the separation of concerns between components and connectors, we informally describe the example that we will present in section 5.

A gas station has several pumps where customers can pump fuel. To do so, a customer decides to pay an amount of money to the cashier. Only then can the customer pump fuel. Customers, cashier and pumps are modeled as active objects, because they represent domain entities.

Customers 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. Moreover, as discussed in [NACO97] race conditions between customers can occur: A fast customer can pump the fuel that

another one payed for. In section 5 we model the various interactions and the race prevention by two specialized connectors.

While the customers 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). Therefore, e.g. the customer is not polluted by assumptions about cashiers or pumps.

3.2 Explicit Connectors

After this intuitive view of connectors, we focus on the specific properties of explicit FLO/C connectors, mainly that: (1) a connector refers to the components of the interaction, called its *participants*, by *roles* of the interaction theme[2], and (2) a connector controls its participants by following *interaction rules* over the message passing.

3.2.1 Static Properties

A connector implements an interaction theme. The components that are involved in this theme are the connectors *participants*. The participants play *roles* in the theme. The connector description refers to the 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. Furthermore, a component can participate in different connectors. Since we model connectors as active objects their description is located in a class.

Figure 3.1 illustrates a simplified arrangement of components and connectors in the gas station example. A PaymentConnector instance connects two CarDriver and a Cashier instance in order to implement the payment interaction between customers and a cashier.

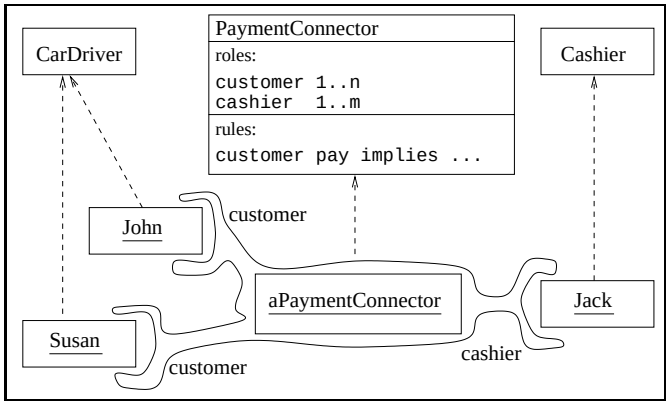

Figure 3.1: A system snapshot of a connector and its components.

3.2.2 Dynamic Properties

Connectors in FLO/C are user-defined active objects so they are instantiated and destroyed dynamically. They are dynamically attached to active objects, or detached from connected ones. A connector

[2] A single connector can implement several interaction themes, but it is better style to use one connector per interaction theme.

is attached to a component by association of the component with one of the roles defined in the connector. A connector is independent of its participants, and the participants remain unaware of the connector.

Connector Lifetime. After its definition a connector passes by three stages: (1) instantiated, (2) activated and (3) terminated.

1. An instantiated connector can be attached to active objects, but it does not yet influence them. When the connector has at least one participant per role, it can be *activated*.

2. When activated a FLO/C connector enforces the interaction behavior on its participants. Therefore, it will activate [3] its participants if they are still passive (see section 2.2.1) and itself. While activated, the connector can add (and activate) or remove participants *dynamically*. For each new participant, the activated connector can run a user defined initialization script.

3. There are several ways to terminate a connector. When participants are detached, it can happen that there are no more players of a particular role. This causes the connector to terminate, in order to prevent inconsistency. On the other hand a connector can explicitly be caused to terminate, allowing two options. Either the connector also terminates all its participants or it leaves them active.

Note that during all the stages, connectors can refuse to attach participants for different reasons (e.g. participant is missing proper interface to play its role, participant is not an active object).

Connector Behavior. In order to implement an interaction pattern (including coordination), the connector intercepts the internal messages of its participants (reactive), and it processes its own ones (proactive in the sense of CLF rules [AHM96]). It decides, if methods of the participants (and which ones) should be invoked. The basis for decisions is a connector specific *set of rules* and the history of the interactions.

3.2.3 A First Example of a Connector Declaration

The definition[4] of the connector presented in figure 3.1 is the following one.

```
(1) Connector subclass:   #PaymentConnector;
(2)   withRoles: 'customer cashier';
(3)   withBehavior: '
(4)     customer payment: amount. implies cashier receiveCash: amount. endRule'
```

A connector definition extends the SMALLTALK class definition with connector specific information (lines 2 to 4). Line 2 defines the roles of the connector, here customer and cashier. Lines 3 to 4 define the behavior. Note that this really simple connector defines only a single interaction rule. It specifies how to deliver money from the customers to the cashier. The example of section 5 will show connectors that implement more complex types of interactions. Furthermore, in section 14 eleven coordination problems are implemented in FLO/C. The examples document the division of responsibilities between connectors and components.

[3] Activate = provide the object with a thread and start message redirection to its message queue.

[4] For now, to ease the understanding, we slightly changed the syntax of the connector definition. In section 14 when we have presented the concept of explicit meta-classes we will use the syntax that is also used in our implementation.

In this chapter we outlined why and how FLO/C enforces a strong separation of concerns. FLO/C integrates the separation concept of architectural software design and the one of concurrent programming in a single paradigm: The separation between (hybrid) stand-alone active objects and explicit connectors.

The next chapter will show how the user can specify a connector's behavior, and how the behavior collaborates with the behavior of other connectors.

Chapter 4

Connector Behavior

A connector coordinates the interaction between its participants. Like many other coordination approaches based on rules (CLF [AHM96], Coordination Policies [MU97] and Synchronizers [FA93]), 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 incrementability through composition and the ability to reason about them.

In section 3.2 we said that the behavior of a connector is described in its class by user-encoded *interaction rules* on message passing. First we will introduce the syntax of the rules and then we will present the semantics of the rules and how they can enforce coordination.

4.1 Syntax and Elementary Semantics of Interaction Rules

As shown by the following syntax, a rule is composed by a *precondition*, an *operator* and *consequences*. A *precondition* identifies which message should be intercepted. A message consists of a *role* that refers to a receiver of the message, a *selector*, and *calling arguments*.

When a message sent to a participant is intercepted by the connector[1], it handles the *consequences*. They consist of a list of messages. Roles optionally use a *specificator* for the management of object groups (see 4.4).

Rule	::=	Precondition Operator Consequences
Precondition	::=	Message
Consequences	::=	Message$^+$ endRule
Message	::=	Role *Selector Args* .
Role	::=	*Rolename* | *Rolename*_select_*Specificator* | *Rolename*_select_*Specificator*_as_*Rolename*
Operator	::=	impliesLater | implies | impliesBefore | permittedIf | waitUntil

Note that roles can be seen as formal parameters for participants. Furthermore, the arguments in the rules are formal (only symbols).

[1] The connector only conceptually intercepts messages, the implementation uses other entities to do so.

Rule triggering. Rules are declared in connectors that observe their participants. Each time a participant is about to execute a method, its connectors check if there is a rule to trigger, and if so they handle the consequences. All rule triggering involves the following symbol replacement steps where the formal arguments of the messages in the rules are replaced by actual arguments.

1. If a participant is about to execute a method, the connector checks, if it has a rule for this participant for the given method selector. In order to do so, it looks up what role this participant plays, and matches it with the role in the precondition of the rules.

2. If there is a rule that triggers, the connector evaluates the consequence messages. In order to do so it must replace the roles and formal argument of the consequence messages. For each consequence message it looks up what participant plays the role. Furthermore, it values the formal arguments of the consequence messages with the actual arguments it has got from the rule-triggering method. The operator of the rule declares, what the connector has to do with the consequence messages (see section 4.2).

In section 3.2.3 we saw a connector with the following rule.

```
customer payment: amount. implies cashier receiveCash: amount. endRule'
```

customer payment: amount is the precondition message, where customer is the role, payment: the message selector and amount the formal argument. After the operator implies the rule contains one consequence message. The consequence message also uses the argument amount. Therefore, when a message triggers the rule then its actual argument is used for the consequence message. For example if an object o1 plays the role customer and an object o2 plays the role cashier and o1 receives the message payment: 10 then a consequence message will be o2 receiveCash: 10.

The operator defines the semantics of the rule, therefore understanding the operators is crucial for the understanding of FLO/C.

4.2 The Semantics of the Operators

The precondition message of a rule is a request for an object to do some computation. The consequences are messages that reflect a certain reaction to this event. According to section 2.2.3 the reaction is limited to inhibition, method execution and reissuing requests. On the other hand, the operators should offer the possibility to compose the coordination abstractions specified in section 2.2.2. Let us recall table 2.4 that summarized the required coordination abstractions:

<table>
<tr><td colspan="2">Multi-object joint action</td><td>Communication</td></tr>
<tr><td>aspects</td><td>styles</td><td>aspects</td></tr>
<tr><td rowspan="2">guards</td><td>balking</td><td rowspan="4">transition request
data-flow
multi-casts</td></tr>
<tr><td>blocking</td></tr>
<tr><td rowspan="2">computation ordering</td><td>push</td></tr>
<tr><td>pull</td></tr>
<tr><td colspan="2">access protected</td><td>asynchronous</td></tr>
</table>

Since multi-object joint actions are a high-level abstraction, FLO/C uses four operators to compose them: 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 pull style, the impliesBefore operator supports push style. All four operators protect their participants (objects of the precondition message and the consequence message) from third-party access, as argued for in section 2.2.2

The low-level communication tasks are met by the impliesLater operator that features *asynchronous consequence sending*. It can be used to send request that start multi-object joint actions. It can also carry data, but only in requests' arguments. Multi-casting is not directly supported. It is part of the *group-management* of FLO/C, which is described in section 4.4.

Let us now present each of the five FLO/C operators in more detail. For each operator we factor out what coordination aspects it addresses. A mini-example, consisting of a single rule, is intended to give a taste of the semantics and the use of the operator. Note that these one-line examples are for demonstration purpose only, they are not fully worked-out. Some rules are inspired by the "dining philosopher" problem. In section 14.7 we present the complete FLO/C solution to this problem.

4.2.1 Balking Conditional Synchronization: permittedIf

The permittedIf operator guards the computation on one object by a condition potentially involving another object. The consequence part of the rule is interpreted as a predicate message[2] set. Upon reception of a request matching the precondition of the rule the predicates are evaluated. If a predicate fails (the evaluation returns false), the precondition message is ignored (no computation), and an exception message is sent to the target object of the precondition message. A more detailed view of the exception mechanism is presented in section 4.5. As mentioned previously, when checking the guard the participating objects (targets of the precondition and the consequence messages) are access protected. This is obviously necessary, because neither the guard object nor the guarded object should change between the success (evaluation to true) of the guards and the computation of the request. Consider the rule:

```
point moveTo: p. permittedIf screen isInRange: p. endRule
```

The example rule enforces a point to stay in the range of a screen.

4.2.2 Blocking Conditional Synchronization: waitUntil

The waitUntil operator enables the declaration of blocking style guards. The consequence part of the rule is a predicate message set. Upon reception of a request matching the precondition of the rule, the predicates are evaluated. If a predicate fails, the execution of the precondition message is *delayed*. The delaying policy is to asynchronously resend the precondition message, which is equivalent to polling. Section 7.3 discusses this limitation. Again, the guard and the guarded object are access protected during computation. Consider the rule:

```
chopstick pickedUp. waitUntil chopstick isFree. endRule
```

In this mini-example a chopstick can only be picked up if it has not already been picked by someone else, which is reflected by the isFree predicate message. If the predicate isFree is false, the pickedUp request is delayed. The guard will be checked later again, until the chopstick can be "picked". Thus, the waitUntil operator is not just blocking, but also polling. This has its disadvantages for liveness, as discussed in section 8 and disadvantages performance wise as discussed in section 7.3.

[2] Predicate messages are expected to return a boolean value.

4.2.3 Push Style Temporal Ordering of Execution: implies

The implies operator can order executions by telling what else has to be executed *after* a certain method was executed. The execution of the precondition *pushes* the execution of the consequences. The participant objects are access protected, because we want to model multi-object joint actions, where the computation depends on the state and computation of other objects as argued in section 2.2.2. Consider the following rule:

```
philosopher eat. implies chopsticks release. endRule
```

After a philosopher has eaten (eat was executed for a philosopher), the chopstick is released, in order to be available for another philosophers. Note that the implies operator makes the result of the precondition message execution available for the consequence messages, thereby supporting dependencies on computation of other objects. This is described in detail in section 4.5.

4.2.4 Pull Style Temporal Ordering of Execution: impliesBefore

The impliesBefore operator can order executions by telling what has to be executed *before* a certain method is executed. Before the precondition message executes, the consequence messages are *pulled*. Again, the participants are access protected. Consider the rule:

```
philosopher eat. impliesBefore chopsticks pickedUp. endRule
```

This example rule enforces that the chopsticks must be picked up before a philosopher can eat.

We will refer to the consequences of the ordering operators implies and impliesBefore as the *sequential consequences* since they order messages into a sequence.

4.2.5 Asynchronous Communication: impliesLater

In section 2.2.2 we argued for an asynchronous communication mechanism that can request multi-object joints and transport data. However, this would not necessarily have to be a rule operator, but could be done in a statement from within the components. FLO/C uses a rule operator because of two reasons: 1) FLO/C's paradigm is to factor out *all* interaction between components as argued in section 3.1.3. 2) FLO/C should be uniform therefore we will address light-weight request propagation with rules, too. A rule based asynchronous communication support, can again be push or pull driven. The impliesLater operator is (its name already gives a clue) a *push* style operator. After the execution of the precondition message, a consequence message is asynchronously sent, which can carry data in its arguments. The data is carried from the arguments of the precondition message to the arguments of the consequence messages by argument matching. As seen in section 4.1 the argument matching mechanism also applies to the other operators. In section 7.2 we will explain, why pull style asynchronous communication is not featured in FLO/C. The impliesLater operator is not intended to compose a multi-object joint action. Its purpose is light-weighted communication. Therefore, it offers *no access protections*. Consider the rule:

```
producer produced: a. impliesLater consumer consume: a. endRule
```

Assume that after each completion of a product, the producer stores it using the message produced:. The connector containing the presented rule will intercept the message, match the arguments, and *asynchronously send* the consume: message to the consumer. Thus, the message carries the fresh product in the argument. Later, the consumer receives the request to consume the new product.

4.2.6 Evaluation

The following table maps the five preceding operators to their coordination purpose.

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

The introduced operators allow request propagation and the declaration of the basic elements of synchronized multi-object joint actions that trigger upon requests. As seen in section 2.2.2 this is the coordination abstraction that we want to achieve with the FLO/C model. As shown there, the coordination abstraction is powerful enough to model transactions, multi-object constraints and mutual exclusion on shared resources in a straight forward way.

Up to now it is not obvious how multi-object joint actions are *composed* with rules using these operators. A composition is achieved by declaring several rules using guards or ordering operators on the same objects. The rules can even be located in different connectors. The following section shows how FLO/C composes simultaneous triggering rules at run-time, and fusions them to multi-object joint actions in a uniform way.

4.3 Collaboration of Connectors / Fusion of Rules

This section presents how multi-object joint actions are composed upon a request for a single execution. All triggering rules and their consequences are explored and fused into one global behavior. Figure 4.1 illustrates this process.

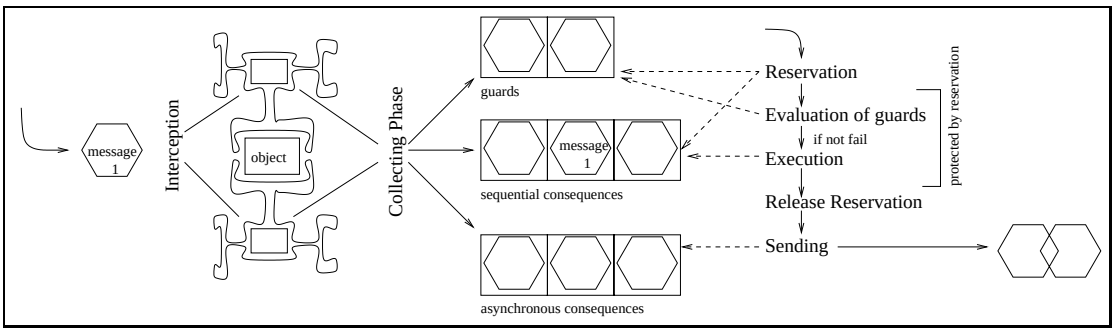

Figure 4.1: A message triggering multiple rules.

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 (2) the protected execution phase and (3) the unprotected sending phase.

In the *consequence collecting phase*, all the connectors attached to the active object start to collect consequences. Sequential consequences[3] get special treatment. Because they will be directly

[3]Consequences of the implies and the impliesBefore operator.

executed, *each of their messages must be checked for further consequences*. Thus, the total of the consequences of consequences of an intercepted message is collected to *one* ordered list of sequential consequences[4]. The final list consists of three different kind of messages: the sequential ones (including the intercepted message), their guards, and their asynchronous consequences[5]. Note that the consequences can be cached instead of collected as long as the connectors do not dynamically change (see section 13.3).

The next phase is the *protected execution phase*. This is in fact the phase that executes the multi-object joint actions. It starts with the *internal reservation* of all the participants, that are target of a guard message or a sequential consequence message (participants of the joint actions). The FLO/C model guarantees that at any time, at least one complete reservation succeeds. During the joint-actions, the reservation protects the participants from being accessed by other active objects.

Now the guards are evaluated. According to what type of guard failed, the triggering message is delayed or deleted, thus the joint-actions are aborted. If *all* guards succeed, the sequential consequences (the interdependent computations) are executed. Since all consequences of these executions are already calculated, no connector needs to check the execution for consequences again. When the execution phase is finished, the object reservations are cancelled.

At last, in the *unprotected sending phase*, the asynchronous consequences of all the methods, that were previously executed, are sent asynchronously in order to trigger new multi-object joint actions.

Consistent declaration of joint actions. To declare a set of multi-object joint actions, guard rules can be declared to ensure the right state of the object group in order to change state. A consistent way to change the single objects in order to reach a consistent group state can be reflected by rules using the ordering operators. Then, rules using the impliesLater operator can trigger new multi-object joint actions. Thus the participants of joint actions are implicitly declared by rules: (1) Each rule using a guard or an ordering operator declares all involved object to be participants and (2) the ordering operators connect actions to joint actions. Therefore, extending a given joint action is simply done by using new rules that define additional guards (possibly on other objects, extending the group), or order new computation on new objects in between the existing computation chains.

Pessimistic transactions. Pessimistic transactions are composed by ordering the commitments in push or pull style, starting from a message that is intended to trigger the transaction. For every execution that has a constraint, a guard rule is used. At run time, upon interception of the triggering message FLO/C will check all the guards first, then atomically execute the transaction if the guards were successful.

Cycles in the sequential consequences. The sequential ordering operators as well as the impliesLater operator can be used to declare message cycles. However, cycles in sequential consequences would be vicious therefore they are automatically broken at run-time. Otherwise such cycles would cause an endless collecting phase. Here the shortest possible rule declaration of a loop:
obj1 m1. implies obj1 m1. endRule Note that cycles are allowed and even useful in rules with *asynchronous* consequences.

This section gave an overview of FLO/C's rule fusion mechanism. Many aspects (recursive lookup of sequential consequences, the cycle breaking mechanism) were not be explained in detail

[4] In fact, the indirect consequences form a tree (messages to execute before, and messages to execute after a given one) that is linearized by an in-order traversal.

[5] Asynchronous consequences are the consequences of rules with the impliesLater operator.

because section 8 will formally specify the semantics of the operators and the fusion of rules and address the open questions of this section. Moreover, it will enable us to prove execution and liveness properties of FLO/C systems.

While we have now discussed how FLO/C rules enable coordination, we have neglected how connectors manage their components through roles. The elaborated role group management of FLO/C is discussed in the next section.

4.4 Group Management

FLO/C connectors refer to their participants through roles. This indirection not only allows connector definition to be independent from components, it also allows one-to-many relationships. A *group*[6] of objects can play a single role known to a connector. This improves the flexibility of FLO/C connectors. It allows connector definitions that are able to coordinate a dynamically changing number of participants. Furthermore, the number of rules decreases drastically, if not every participant needs its own role and thus its own rules. Therefore, the expressive power of FLO/C improves with what we call its *group management features*. We present the refined semantics for roles, concerning the management of participant groups.

4.4.1 Role Semantics

FLO/C considers a role as representing a non-empty, ordered[7] group of active objects. We will refer to such a group as the *role group*.

A role can be used in two different parts of a rule: in the precondition part or in the consequence part (see 4.1).

- In the precondition part of a rule, the role is used to determine if the rule triggers. The rule is triggered when the target object of an intercepted message *is a member* of the role group (and the message selectors match).
- In the consequence part, when a role is used as receiver of a message, it represents *all* the objects of the group. The message is *multi-casted*, i.e. applied to *all* members of the role group.

Note that the distinction between precondition- and consequence role is transparent if only one participant is associated with each role. However, if a role group contains several participants, a restriction to multi-casting limits the expressive power. Therefore, FLO/C introduces *specificators* to select a subset of a role group.

4.4.2 Roles in the Dining Philosopher Example

The well-known dining philosopher problem features two roles, namely philosophers and connectors. Philosophers sit around a table and mutually exclude each other on shared chopsticks. Without role groups each philosopher and each chopstick would have to be represented by a separate role. Therefore, either a connector has to feature as many roles as there are philosopher and chopsticks or there are as many connectors as there are philosophers. Both solutions lack flexibility and duplicate a lot of

[6] These are groups from the point of view of a connector. Do not confuse such groups with the participants of multi-object joint actions, which are only conceptual groups.

[7] The order of the group reflects the temporal order of the attachment.

code. With the group management of FLO/C only one connector can handle an arbitrary number of philosophers and chopsticks. While the complete and fair FLO/C solution to the dining philosopher problem is documented in section 14.7 we present a simplified solution here:

```
(1) chopstick pickedUp. waitUntil chopstick_select_REC isFree. endRule
(2) philosopher eat. impliesBefore
            chopstick_select_LeftNRight_as_myChopsticks pickedUp. endRule
(3) philosopher think. impliesBefore myChopsticks putDown. endRule
```

Here we assume, that the philosophers default behavior is to alternate between eating and thinking. Furthermore the chopsticks can be notified when they are picked or released by a philosopher (messages pickedUp and putDown). A chopstick can be asked if it is currently picked up by sending isFree.

The presented set of rules is capable of enforcing mutual-exclusion on the chopsticks for *an arbitrary number of participants*. The rules roughly enforce the following: (1) Only one philosopher can use a chopstick, (2) Before a philosopher eats it will pick up the chopstick to its left and the one to its right. (3) Before a philosopher starts to think it will release the chopsticks it currently holds. In rule one we notice the syntactical extension _select_REC to the role. Here we use a role specificator.

4.4.3 Specificators

Each role in the consequences can optionally use a specificator. When an intercepted message triggers a consequence containing such a role, its specificator selects a subset of the role group, and the consequence message is multi-casted to this subset, instead of the whole role group. A specificator defines a *policy* to select objects from the role group. It uses the intercepted message as a context. In fact, the reified rule-triggering message is the context. It contains dynamic history information about the multi-object joint actions it belongs to, thus supporting complex selection policies.

In guards, for example, it is often necessary to check the condition of just one certain participant. This is what we presented in rule (1) of the dining philosopher example.

```
(1) chopstick pickedUp. waitUntil chopstick_select_REC isFree. endRule
```

Thus the rule says that a chopstick cannot be picked up, until it is free. When the message pickedUp is intercepted for *any* of the chopstick objects, this rule is triggered. As there are more than one objects playing the role of a chopstick, we need to select the chopstick that received the message pickedUp. By using the context information, the *receiver specificator* REC selects the receiver of the request for the message pickedUp.

FLO/C provides the following built-in specificators:

Rolename_select_REC	Selects the receiver of the intercepted call.
Rolename_select_Others	Selects all players of the role, excluding the receiver of the intercepted call.
Rolename_select_RND	Selects a random player of the role.
Rolename_select_Next	Selects the object of the group that is next[8] to the previously selected one. If none was ever selected yet, it selects the first one of the group. The successor of the last object is the first.

Note that as FLO/C is an open-implementation, users can define their own specificators, as long as they only refer to the provided context information.

4.4.4 Relative Roles

In our dining philosopher example specificators should not only select appropriate chopsticks but *associate* a participant with a subset of a role group, so that the association is available later. In the example a philosopher picks two chopsticks and only releases them before (s)he starts to think. We do not want to store the selection in a philosopher, because this is against our paradigms stated in section 3.1. It would be possible to explicitly store the selection in the connector by declaring additional rules and connector methods to do so (see next section). However, FLO/C also offers *relative roles* as syntactic sugar to express the association between a shared resource and its current owner.

The optional role appendix _as_*relativeRoleName* causes the specificator to *associate* its particular selection with the receiver of the intercepted message, and to map the association to *RelativeRoleName*. This name can now act as a *relative role* in other rules, where it is used in consequences. When these consequences are triggered later, the receiver of the *currently* intercepted method is used to look up the particular selection.

Our presented dining philosopher example used a relative role myChopsticks in rule 2 and 3.

```
(2) philosopher eat. impliesBefore
        chopstick_select_LeftNRight_as_myChopsticks pickedUp. endRule
(3) philosopher think. impliesBefore myChopsticks putDown. endRule
```

Before a philosopher philo1 eats, it picks its chopsticks that are selected by the user-defined specificator LeftNRight. The selection is mapped to the relative role name myChopsticks. Thus myChopsticks is an association of a chopstick with philo1. The second rule, triggered later, expresses that before the *same* philosopher starts to think, *his/her* chopsticks, referred by the relative role myChopsticks, are put down on the table.

Note that it makes no sense to use relative roles in the precondition. The use of specificators in the precondition would also be useful but is not implemented yet. Furthermore, consequences with relative roles should[9] always trigger after their corresponding rule with the _as_*name* appendix.

4.5 Special Features

In order not to confuse the reader, we factored out some special but useful features of FLO/C into this section. It explains connector rules that refer to the connector itself, the exception mechanism and the keyword result for the propagation of sequential computation results.

4.5.1 Self Controlling of Connectors

The connector can use its own state or even computation to influence the behavior of its controlled object. In fact, the connector is an active object that can bear state and operations and that controls itself. Note that together with its operations connectors can therefore develop proactive behavior as found in CLF [AFP96].

The connector is accessible for its own rules by the keyword connector. The keyword is used like any other role. If we want to introduce a connector that represents the fact that its participants

[9] Our implementation cannot check this because it is not statically known.

are not allowed to display on a screen at a given moment (e.g. for a screen-lock, or screen saver), we could introduce a connector with setter methods visibilityOn and visibilityOff and a testing method isVisibilityOn. The connector would carrying the rule:

```
object display. permittedIf connector isVisibilityOn. endRule
```

Then we connect the relevant visible objects to the role object and the connector enforces the behavior, using its own state. Thus a connector can represent the state of the coordination in its own state and it can host computation that belongs to the coordinated group but not to a participant of the group. A simple example for such code is *conversion* of values between participants.

4.5.2 FLO/C's Additional Exception Mechanism

The base model's built-in exception mechanism cannot express exceptions that are raised by failing guards. Therefore, in FLO/C each active object understands the messages methodWasForbidden: and methodWasDelayed:. When a permittedIf guard fails, the target object of the joint actions request receives the methodWasForbidden: message with the method selector of the forbidden message as argument. If a waitUntil guard fails, the methodWasDelayed: is sent. Note that the exception message is not raised in the sender of the request. Since all requests are sent asynchronously none of the senders expects a return value. Instead it is the starting point of the joint actions that is notified since it is here that a request was refused. It is also here that any exception handling takes place.

The previous example could be extended to register each illegal display request.

```
object display. permittedIf connector isVisibilityOn. endRule
object methodWasForbidden: m. implies connector registerIllegalRequest: m. endRule
```

When sending display to an object that is controlled by the connector which currently enforces that the visibility be turned off, the guard will fail, and the object will not display. Instead methodWasForbidden: with the selector #display as argument is sent to the object that received the display request. Now this triggers the second rule, which calls a method of the connector that can registers the forbidden request.

Note that the exception mechanism can be used to break livelocks as we will demonstrate in section 7.3

4.5.3 Propagation of Computation Results

In the multi-object joint actions abstraction, computation may depend on other computations. In FLO/C style programming, method's return values are usually transmitted via an indirection: At the end of a calculation, and object writes the result using an accessor method. This accessor method then triggers a rule, and the result is transfered via the arguments (see section 4.1), thus reaching the next object about to compute. In section 7.4 we will discuss this FLO/C idiom for asynchronous propagation. In this section we will present the keyword result which eases the propagation of results of sequential *push*-style computations. For the *pull*-style of execution ordering which does not offer. the result keyword, the idiom is quite ugly In pull style the last calculation is triggered first, pulling the results of its predecessors. For example a computation comp2 on o2 could be triggered, implying first a computation comp1 on o1, this computation would write its result using an accessor message res:, which in turn would trigger a rule that causes, the result to be used in the computation comp1with:

arg. The following rules would implement this pull-style ordering:

```
o2 comp2.  impliesBefore o1 comp1. endRule
o1 res: r.  implies o2 comp2With: r. endRule
```

Now in push style, FLO/C offers the convenient keyword result, to propagate the method results. The keyword result can be used in the consequences, instead of an argument of a message. When the consequences start to execute, FLO/C will value the keyword argument with the result from the execution of the precondition (see section 13.3.2). This simplifies the previous example to the push-style rule:

```
o1 comp1. implies o2 comp2With: result. endRule
```

In the section 7.2 we will discuss why the keyword is not available for pull-style ordering, and why push-style is generally better supported in FLO/C.

4.6 Categories for the Expressive Power of Connectors

We presented the coordination support and group management features plus the special extensions of FLO/C showed in the previous section. Now we want to classify these features according to their expressive power.

Since FLO/C relies on a computationally complete object model, connectors need only little expressive power to bridge the gap[10] between components in order to render FLO/C computationally complete again. The impliesLater operator that models asynchronous communication between components will provide such a bridge. Therefore, this section we are interested in completeness of the model, but in analyzing, how the other FLO/C features raise the abstraction level, and increase the expressive power of a connector. As we will see in section 8 the waitUntil operator differs from the others, because it can cause liveness-problems, thus increasing the expressive power of a connector.

As seen in section 4.4 the use of role groups also increases the expressive power of a connector. Furthermore, a connector can refer to its own state. At an even higher abstraction level, a connector can be *programmed*, using its knowledge about its participant, and history information it possibly saved. For example FLO/C's fair solution to the "dining philosopher" problem (see section 14.7) logs the time, when each philosopher has eaten the last time and uses this information to hinder greedy philosophers. At the highest level of abstraction a connector uses the dynamics of the FLO/C model. It dynamically creates[11] connectors on the fly in order to control subtasks. An example of such a technique is illustrated in the "sleeping barbers" example of section 14.10. Note however that connector programming removes interaction responsibilities from rules, thus hiding some interaction behavior inside of its methods. But this conflicts with the paradigm of using a clear policy of rule fusion in order to keep transparent the global behavior of a FLO/C system.

We can split the connector features in three orthogonal categories: involved operators, role usage and usage of the keyword connector (self usage). The following table shows the resulting categories with each entry being more expressive than the one before:

[10] Components should not keep references to each other as stated in section 3.

[11] This is done in a method of the connector, thus it involves connector programming.

involved operators	*role usage*	*self usage*
only impliesLater	single objects only	not using connector
no waitUntil	group of objects	using connector's state
all operators	using specificators	using connector programming
	using relative roles	dynamically install new connectors

Now we have completed the presentation of the FLO/C model. The next chapter will present the model in its application on an example.

Chapter 5

An Illustrating Example: The Gas Station

In this chapter we present a complete FLO/C solution to an example bearing coordination problems. The gas station example [HL85] is well-known in the software architecture design community. Recently it was used to demonstrate automatic analysis of race-condition problems [NACO97].

We want to present this example here because it is small enough for complete presentation but large enough to emerge non-trivial coordination problems. We can illustrate FLO/C's coordination solutions and its separation of concerns as well as most of FLO/C's special features on the example. Note that in chapter 14 there are many other examples implemented in FLO/C.

Problem description. We already described the example in section 3.1. It is a simulation of car drivers that tank their vehicles at a gas station. First they pay the cashier, then they pump fuel from one of several pumps. The example bears 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: If two customers pay to get fuel from the same pump, the one who is faster can get the fuel for both.

5.1 The Participants

In the real world cashiers an drivers are autonomous entities that have their own will and act concurrently. Active objects are well fit to cover such aspects. The pump is an independent resource that is shared, therefore it is an active object itself and not a helper object of another active object. Now we have identified the actors of the example. Following the separation of concerns in FLO/C each of the participants has its proper behavior but no assumption on how to interact with the other participants. Figure 5.1 shows the UML class diagram of the participants. It gives an overview of the functionality of each participant.

The cashier can receive money and (s)he stores it. The pump acts as a fuel server. The cashier can prepare it to release an amount of fuel (using load: amount). The server method releaseLoad will then return the loaded fuel. The car driver object stores money and fuel. Its methods call each other

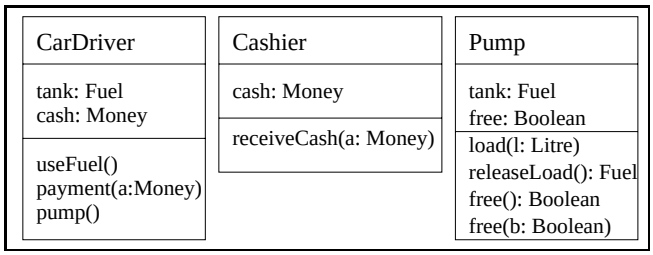

Figure 5.1: Participant classes of the gas station example.

thus forming the life cycle[1] illustrated in figure 5.2. 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.

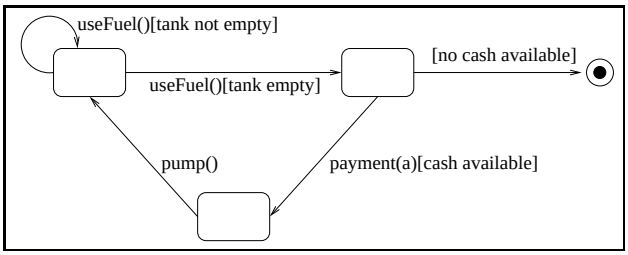

Figure 5.2: UML state diagram of the car driver.

Note that the car driver does not have to know, how to pay a cashier, or how to pump on a certain pump. (S)He only knows that (s)he wants to pay and pump. Therefore, the CarDriver implementation can run on its own, his/her behavior is independent. However, when (s)he is not connected, (s)he gets no new fuel, thus stopping soon. It is the following connector's responsibility to implement the concrete interactions, namely the correct transfer of money and fuel at the appropriate time.

5.2 The Connectors

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

The GasStationConnector defines the three roles customer cashier and pump. The connector behavior is defined by four rules.

Rule 1: When the customer pays the amount a, the cashier gets the money. As a second sequential consequence, the connector[2] calculates the amount of fuel, the customer payed for. Conceptu-

[1] In fact this graph represents how the methods call methods of the same class. We refer to this self-calling graph as "life-cycle" because in FLO/C it is often cyclic.

[2] A connector can trigger messages to itself using the default role connector (see section 4.5).

ally, 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: This calculation leads to the association of a pump to a customer through the relative role myPump (see section 4.4.4) [3]. The use of the Next-specificator guarantees that there are no two customers selecting the same pump, when there are less customers than pumps. The pump is loaded with the amount of fuel that the connector has calculated using the keyword result (see section 4.5).

Rule 3: This rule starts a new set of multi-object 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.

5.2.1 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 real pumping. The global process is divided, because it is the customers free choice, when it wants to pay, and when it wants to pump. Because of the unprotected gap between the two sets of joint actions, this connector is unable to prevent a race. If there are more customers than pumps, it is possible that two customers pay to pump from the same pump. 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 instance variable as a lock. When a pump already is loaded, further loading requests must wait.

```
( ) Connector subclass: #PumpLockConnector;
( )  withRoles: 'pump';
( )  withBehavior: '
(1)    pump load: a. implies pump_select_REC free: false. endRule
(2)    pump load: a. waitUntil pump_select_REC free. endRule
(3)    pump releaseLoad. implies pump_select_REC free: true. endRule'
```

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 sets of joint actions of the GasStationConnector. It comes in, when the payment actions end, and ends, where the pump actions finish.

By adding the PumpLockConnector to the example we can demonstrate how synchronized multi-object 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.

[3] Note that the pump is associated to a customer an not to the connector. This is because the association is bound to the target of the request that lead to these 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 load fuel for one customer at once and *no race can occur*.

5.3 A Complete Simulation

It follows a sketch of a SMALLTALK script, correctly running the gas-station example with two customers, a cashier and a pump.

```
|customers pumps cashier gasStation lock |
                        "Create active object groups:"
customers := OrderedCollection with: CarDriver new  with: CarDriver new  with: CarDriver new.
pumps := OrderedCollection with: Pump new.
cashier := Cashier new. "Only one cashier."

                        "Instantiation of the two connectors:"
gasStation := GasStationConnector new.
lock := PumpLockConnector new.

                        "Attachment of the participants to the connectors:"
gasStation objects: customers playRole: 'customer'.
gasStation objects: pumps playRole: 'pump'.
gasStation object:   cashier playRole: 'cashier'.
lock objects: pumps playRole: 'pump'.

                        "Activate connections now:"
gasStation activate.
                        "Enable race prevention now:"
lock activate.

                        "Start customer behavior."
customers do: [:c | c useFuel].

 ... later ...
                        "Terminate one connector. This also ends the participants,"
                        "which will then lead to the end of 'lock':"
gasStation end.
```

The script instantiates the participants and connectors, prepares the role groups, connects them to the connectors and then activates the connectors. The activation of the connectors enforces the coordination policy. Then the active objects' proper behavior is started (by sending useFuel to them). Note that the race prevention (the lock connector) could also be enabled after the start of the participant behavior. Such a script would first show collaboration but maybe also race. Then, when the lock connector is activated, the race is suppressed.

5.4 Example Evaluation

The FLO/C solution works for an arbitrary number of customers and pumps[4], thus it demonstrates how FLO/C allows *flexible solutions*. Furthermore it demonstrates how FLO/C's group managing specificators yield expressive power.

We can dynamically add a connector, to enforce a new interaction policy, which guarantees race-freeness. This demonstrates the *incrementability* of FLO/C. The example illustrates FLO/C's *separation of concerns*. The autonomous entities are factored out in active objects that behave concurrently. Their interaction code is factored out into the GasStationConnector and a race preventing policy is factored out into the PumpLockConnector.

Furthermore, the example showed the FLO/C solution techniques to the following non-trivial *coordination problems*.

Problem	**Solution**
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.

[4]With a minor extension, it would also work for an arbitrary number of cashiers. But this would not add interesting complexity to the example, therefore we presented the "single cashier" version.

Chapter 6

Component Hierarchy

The FLO/C model as described up to now was not modular. All active objects and their connectors reside at the same conceptual layer. Since one of FLO/C's goals is to offer the possibility to implement architectural design decisions (see section 3.1.2), it must offer a way of declaring the composition of active objects into a composite active object. Since active objects should keep no reference to each other (see section 3.1.3), this way of composition is ruled out. From a composition construct for FLO/C we require the following:

- A composite active object *encapsulates* active objects, which are interconnected by connectors that are encapsulated, too.
- From outside, a composite active object should behave like a plain active object[1]. This also includes that it can be connected and composed again.

Composite active objects represent a group of active objects that fulfill a well-defined task doing so by collaborating in non-trivial ways. If the collaboration is trivial (e.g. host and a helper), then the whole task can be implemented in a plain active object that eventually uses private and passive helper objects (see section 3.1.3).

For the reader's convenience, we will refer to composite active objects as *composite objects*. FLO/C offers the possibility to declare composite object classes. This bears the following advantages.

- The composite object class declaration describes the encapsulated component by its class. Therefore, at run-time the components and connectors objects cannot be referenced, they are truly encapsulated.
- When instantiating a composite object, the connectors and components are automatically created, connected and initialized. Otherwise this work has to be done by the programmer, leading to long and redundant code as sketched in the code example 5.3.

Figure 6.1 shows an instance of a composite object. The instantiation process delivers only an *interface object*, but in the background it creates and activates the participating components and connectors as a black box (conceptual composite object). For the world outside, the interface object *is* the composite object, because it is the instance of the composite object class. The composite object's behavior, however is located in its participating components and connectors. The interface object can

[1] An active object that is not composed by other active objects.

act as if it was the complete composite object, because an *interface connector*[2] triggers the appropriate collaboration of the hidden components.

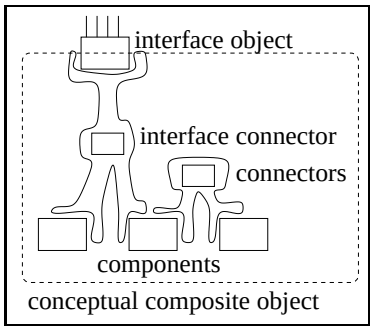

Figure 6.1: Instance of a composite object.

The next section shows how composite object classes are declared in FLO/C.

6.1 Declaration of Composite Object Classes

In order to instantiate a new composite object, the user has to define a composite object class, providing it with the particular composition information. These static informations are:

- The *interface* of the composite object. It is located in the interface object, and is defined like the interface of an active object.
- The participating *component classes*. They are needed to instantiate the hidden components of a concrete composite object.
- The participating *connector classes*.
- The *connection schema*. It declares for each connector, which component will play which role. The role is referred by its name, the components and connectors by numbers. The numbers represent the position of their class in the composite object declaration.

An example: the binary adder simulation. To document the composition of active objects, we preview the binary adder simulation of section 14.6. There, composite objects implement logic units that take two binary entries and compute a logical function like and, or and xor. The elements receive their inputs asynchronously through the messages inA: bool and inB: bool. They synchronize on both channels then sending the result of the logical function by invoking outA: res. In the example we build a binary adder. Therefore, we compose units in order to calculate the carry bit and the sum of two entry booleans and one input carry bit. Then we can compose a Sum and a Carry unit to form an Adder unit, which in turn can be wired to a complete binary adding machine by connecting outputs to carry inputs. Figure 6.2 shows how units are wired together to form an adder element and a complete binary adder.

[2]The interface connector is a user-defined connector that uses the default role interface. At the instantiation of a composite object, the interface object is automatically attached to play this role.

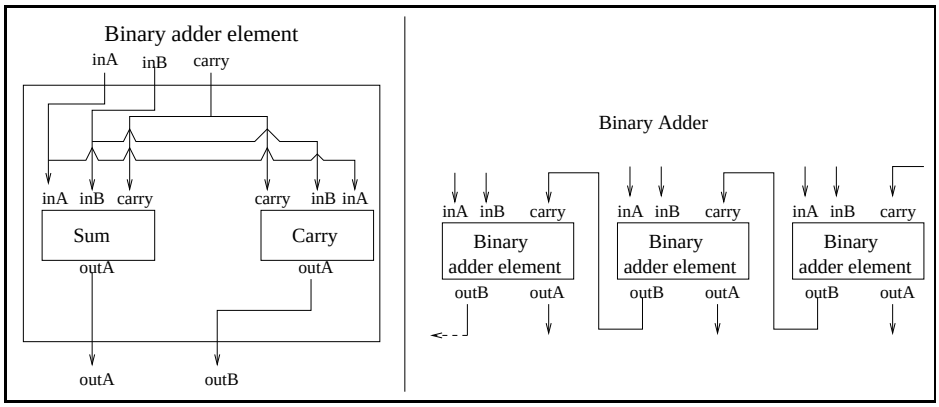

Figure 6.2: An adder element with sum- and carry units and the wiring between adder elements

The sum- and the carry calculating units are composed of synchronized logical components. Given that the logical components already exist, we can declare both calculating units as composite objects. Let us have a look at the carry calculating unit, since it is a little bit more complicate. We can follow the previous list declaring the static properties: The *interface* of the new composite object contains three methods. We use a naming convention and call them inA:, inB:, carry: and outB:[3]. Inside the carry element the carry bit must be calculated with logical operations. Whenever two input bits are set to one (the incoming boolean is true), the carry bit is one. Therefore, we can use and operations between each entry and demultiplex their outputs with or operations to calculate the carry bit. Figure 6.3 describes how the carry bit can be calculated. Furthermore, the procedure for the sum-bit calculation is designed.

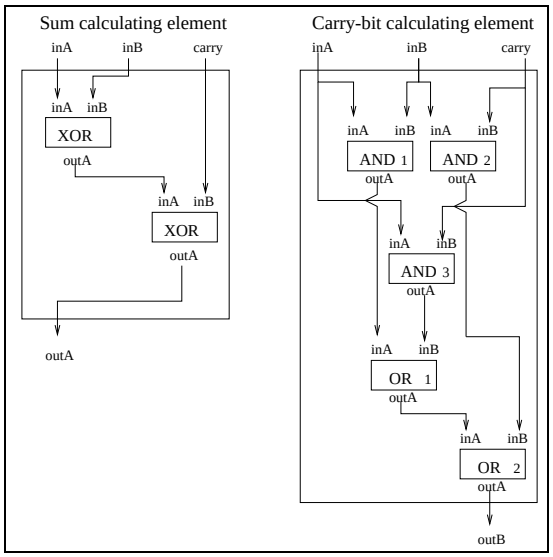

Figure 6.3: Using logical operations to calculate sum- and carry bit.

Now we have identified the *components' classes* for the carry-bit calculating composite object. We use three times the AND class and two times the OR class. We furthermore use two *connector classes*. An interface connector directs the input from the interface to the participants and the output from the

[3]We follow the design in figure 6.2 therefore we don't call the message outA:.

participants to the interface. An inner connector connects in- and outputs of the participants between each other. Since the logical units are synchronized already, the connectors do not have additional coordination responsibilities. They use asynchronous consequences for data flow. Figure 6.3 labels the elements with numbers. We use them for the role names. Thus the wiring shown in the figure shows which connections must be established. The ins and outs of the components must be connected appropriately. The following code shows the declaration of a composite object to implement a carry-bit unit.

```
(1) MetaCompositeObject new
(2)  superclass: CompositeObject ;
(3)  withComponentClasses: '
( )   AndElement AndElement AndElement OrElement OrElement ' ;
(4)  withConnectorClasses: '
( )   CarryInterfaceConnector CarryConnector ' ;
(5)  withConnectionSchema: '
( )   connector: 1    role: and1    object: 1
( )   connector: 1    role: and2    object: 2
( )   connector: 1    role: and3    object: 3
( )   connector: 1    role: or2     object: 5
( )   connector: 2    role: and1    object: 1
( )   connector: 2    role: and2    object: 2
( )   connector: 2    role: and3    object: 3
( )   connector: 2    role: or1     object: 4
( )   connector: 2    role: or2     object: 5' ;
( ) installAtName: #CarryElement
```

The following list will explain this composite object class declaration by the line numbers indicated in the code.

(1) Here we instantiate a new composite object class from the composite object meta-class. FLO/C uses meta-classes because all new composite object classes have common object instantiation code. The instantiation protocol must create all the inner active objects in the background. Using meta-classes is more an implementation than a model issue. For more information about meta-classes see 10.4 and about MetaCompositeObject see 11.5.

(2) All composite objects inherit from the base class CompositeObject.

(3) Here we declare the participants classes. As already mentioned, we use three AND components and two OR components.

(4) The declaration of the connector classes.

(5) The connection schema holds the information which connector refers to which object by which role. Here both connectors use a simple naming convention for their role names. The numbers indicate positions in the declarations of line 3 and 4. Thus the first line of the connection schema reads: The connector that was declared first (a CarryInterfaceConnector) maps the role and1 to the object that was declared first (an AndElement). Note that the interface connector does not have to know the role or1. Figure 6.4 shows the connectors and their connections.

On line 4 we refer to two connector classes that connect the participants inside of the CarryElement. We will now present them. The CarryInterfaceConnector and the CarryConnector contain rules that propagate output results from units to input messages from others. Again we follow the

design presented in figure 6.3. Here the declaration of the connectors. The interface connector propagates messages to and from the interface.

```
MetaConnector new
 superclass: Connector ;
 withBehavior: '
  interface inA: a.   impliesLater and1 inA: a.  and3 inA: a. endRule
  interface inB: a.   impliesLater and1 inB: a.  and2 inA: a. endRule
  interface carry: a. impliesLater and2 inB: a.  and3 inB: a. endRule
  or2 outA: a.        impliesLater interface outA: a. endRule' ;
 installAtName: #CarryInterfaceConnector
```

The inner connector handles the composite object's intern communication.

```
MetaConnector new
 superclass: Connector ;
 withBehavior: '
   and1 outA: a. impliesLater or1 inA: a. endRule
   and2 outA: a. impliesLater or2 inB: a. endRule
   and3 outA: a. impliesLater or1 inB: a. endRule
   or1  outA: a.  impliesLater or2 inA: a. endRule ' ;
 installAtName: #CarryConnector
```

The final arrangement of connectors and components for the carry-bit calculating element, and for the sum-bit calculating element are sketched in figure 6.4

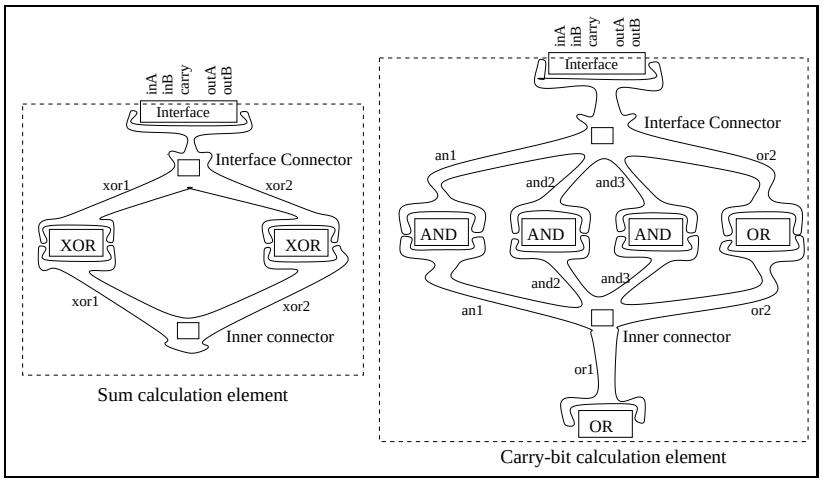

Figure 6.4: Composite objects to calculate sum- and carry bit.

Once we have declared composite objects to calculate the carry-bit and the sum bit, we can compose them again to an AdderElement. Like shown in figure 6.2, the inputs to this element are redirected to both the sum- and the carry calculating component, their output are propagated separately. Having the class AdderElement, we can connect instances of it at run time to form a binary adder for arbitrary numbers of bits. Figure 6.5 sketches this layered object hierarchy.

In the presented solution an additional object hierarchy layer is used to compose the SumElement and CarryElement together to form the AdderElement. However, the adder element has almost the same interface than the carry element, and half of its task is the same. Therefore, it can be convenient to *inherit* the declarations of the CarryElement when declaring the AdderElement. The next section

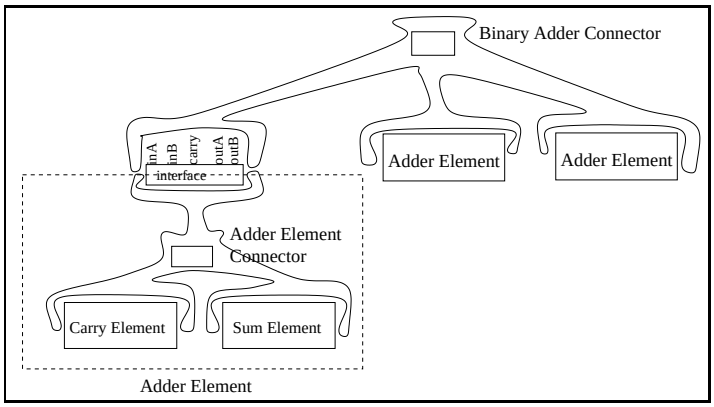

Figure 6.5: A binary adder object hierarchy.

shows how FLO/C supports inheritance of composite object declarations.

6.2 Inheritance of Composite Object Classes

The inheritance of composite objects can bring two benefits.
(1) A possibly elaborated connection schema can be reused.
(2) The inheritance hierarchy offers a way to structure code.

According to [WZ88], an inheritance mechanism must take care, that a subclass' instance can alway safely be used where a superclass's instance was expected. As far as the interface is concerned this is already ensured by the standard inheritance mechanism. A subclass of a composite object class automatically understands all messages that the superclass understands.

Therefore, the only further restriction we will meet is that *the whole connection schema of the superclass is inherited, and cannot be changed*. Thus, the inheritance is incremental; no component or connector can be removed (this would be dangerous, because connections could be defined for them in the inherited connection schema). But object-, and connector classes can be *replaced* by others (provided that they are able to play the role in the inherited connections). The user can also include new object-, and connector classes and define new connections between new or inherited objects and new or inherited connectors.

The following example demonstrates a definition of an inherited composite object class using the previous example.

```
( ) MetaCompositeObject new
(1)   superclass: CarryElement ;
(2)   withComponentClasses: '
( )    super super super super super XorElement XorElement' ;
(3)   withConnectorClasses: '
( )    AdderInterfaceConnector super  SumConnector';
(4)   withConnectionSchema: '
( )    connector: 1   role: xor1   object: 6
( )    connector: 1   role: xor2   object: 7
( )    connector: 3   role: xor1   object: 6
( )    connector: 3   role: xor2   object: 7 ';
( )   installAtName: #AdderElement
```

Explanation of the example following the line numeration of the listing.

1. The new class is a subclass of CarryElement (compare to its declaration in section 6.1).

2. The first five participating object classes of this new class are specified in the superclass (keyword super). They are used to calculate the carry-bit. Two new participant are added in order to add the functionality of calculating the sum bit. See figure 6.3 for the "wiring" of the sum calculating logical units. Thus two XorElements are the participants of this composite object.

3. In order to demonstrate overloading, the first connector of the superclass, namely the CarryInterfaceConnector is overloaded by a AdderInterfaceConnector. This connector must be an extension of the CarryInterfaceConnector in the sense that is must define the same (or more) roles. The second connector is inherited from the superclass. It connects the components that calculate the carry. The third connector is a new one, it connects the components that calculate the sum.

4. Here the new role connections are defined. Note that all connections declared in the superclass are inherited. The new connectors refer to the new components by the roles xor1 and xor2. The inherited connector (on position 2) does not have to know the new participants.

Using inheritance to declare the AdderElement saves us from defining an additional object hierarchy layer to compose the sum and the carry calculation.

6.3 Evaluation + Limitations

When we instantiate an object of a composite object class, the instance will offer *well defined access to a fully instantiated and connected group of active objects*. A composite object can be used like a plain active object. It can receive messages and it can be connected again. Furthermore, it can be a participant of a new composite object. This is important; it offers the possibility of hierarchically compose components. It maps ideally to bottom-up or top-down design processes.

Composite object classes can save code, when FLO/C's dynamics is not needed. If the "gas station" example (see section 5) had been implemented in a composite object class named GasStationExample, it can be used like this (compare to example 5.3).

```
|station|
station := GasStationExample new.
                "The active objects are created, activated and connected."
                "Then, the interface object is stored in the variable 'station'."
station run.
                "This message to the interface will trigger a user defined interface"
                "connector that lets all customers start to use their fuel."
 ... later ...
station end.
                "All active objects terminate."
```

When we compare this activation code to the code example 5.3, we can conclude that composite object declaration can improve the quality of code, since it encapsulates the tedious and error-prone activation protocol.

Limitations. The declaration of the connection scheme is tedious and inflexible. FLO/C's visual tools help to ease this limitation (see section 12).

The inheritance mechanism of composite objects would be of higher value if there was an inheritance mechanism for connectors, too. In the sequential FLO language, such an inheritance was evaluated [Duc97b]. Therefore, connector inheritance was no primary goal of FLO/C.

In the current version of FLO/C composite object classes use a *fixed number* of components (one per each class). Therefore, composite objects do not exploit the possibilities introduced with specificators (see section 4.4). It would be an easy and small extension to add protocol that enables component class declaration to be *parameterized*. At instantiation time, fore each component classes a parameter would indicate the number of instances. For example GasStationExample new: # (3 2 1) would instantiate a composite object hiding three drivers, two pumps and one cashier.

FLO/C's composition concept is influenced by the base language it is built upon. Our implementation uses SMALLTALK. Composition variants other than only keeping references, e.g. full containment as seen in languages like C++ leave FLO/C's scope, they are not discussed here.

Chapter 7

Discussion of the FLO/C Model

7.1 FLO/C Fulfills its Requirements

The FLO/C model is an extension of an object oriented base model, namely of actors (see section 2.2.1). It factors out object interaction in stand-alone run-time entities called connectors. This increases reusability and preserves architectural design decisions in the implementation (see section 3). The FLO/C model's main purpose is to provide high-level coordination support. Therefore it features asynchronous communication and multi-object joint actions. As stated in section 2.2.2, these two abstractions allow to easily and consistently implement the following coordination abstractions:

- Conditional synchronization on multiple objects.
- Mutual exclusion of multiple objects.
- Pessimistic transactions.

Therefore FLO/C supports coordination at a high-level in an object-oriented way. Furthermore, FLO/C's connectors provide group management not only to keep connector declarations independent of the components, but also independent of the *number* of involved components. FLO/C exploits its dynamics culminating in programmable connectors that install and destroy other connectors at run-time.

We conclude that the FLO/C model solves problems of traditional concurrent object oriented programming (see section 2.1). It offers easy ways to create extendible solutions to non-trivial coordination problems. Furthermore it allows to map design decisions concerning the coordination and interaction inside the implementation, which is not possible in traditional approaches. In the conclusion of chapter 16 contributions of FLO/C are discussed in more detail.

However, the FLO/C model also encountered some limitations. We will summarize them now, and discuss whether they can be fixed. Note that here we do not want to address the restrictions of the model that were put on in advance in section 2.2. There we said that constraints like the exclusion of data-format conversion (see section 2.2.2 are due to time limitations of this work but can be treated in future work. The next two sections will concern the emerging limitations of the introduced operators that are due to the constraints of our reflective model that uses rules on message passing (see section 2.2.3). The last section of this chapter discusses the division of responsibilities between components and connectors in FLO/C. Problems arise when a component has too little or too much internal behavior.

7.2 Limited Pull-Style Support

FLO/C is based on message passing control that triggers rules upon interception of a precondition message and then handles several consequence messages. This constraint (see section2.2.3) favors push-style approaches. While it is easy to realize the impliesLater operator that way, it is much harder to think what an asynchronous pull-style operator should do. An impliesSomewhenBefore operator would have to store the precondition message, send the consequence message, and then after this was executed *resend the precondition message again*, but this time not controlling it any more. Unlike in the case of the impliesLater operator, this cannot be done in one step, it involves storing of messages and waiting for the appropriate time for resending. Therefore such an operator would call for an unacceptable large extension of the message control mechanism, therefore it was omitted in FLO/C.

The other pull-style limitation concerns the keyword return. Although consequences of the impliesBefore operator are sequentially executed *before* the precondition message, the precondition message cannot directly access the results. The reason for that is illustrated in the following two rules.

```
objectB secondCalcWith: result. impliesBefore objectA1 firstCalc1 endRule
objectB secondCalcWith: result. impliesBefore objectA2 firstCalc2 endRule
```

The calculation in objectB could rely on the fact that several other calculations in other objects were done before. This is no unusual case. However, here the secondCalcWith: is specified to use the results of *both* its predecessors. It is not determinable in this situation, which result will be used, therefore such a construction is ambiguous and omitted in FLO/C.

Since limited pull-style support is available and the push-style support is well elaborated, we believe this limitation to be minor.

7.3 The Polling of the waitUntil Operator

Upon failure of a waitUntil guard, the whole set of multi-object joint actions will be tried again later. FLO/C models this by asynchronously resending the request. For guards who fail most of the time, this leads to considerable overhead. Worse, a system that is otherwise idle can eventually poll all the time, because the guard will never change. Thus the system can livelock. The FLO/C model presents no built-in solution for that problem. Note however that exactly the same problem occurs with Frølunds synchronizers [Frø96]. The following extension of FLO/C could ease the problem: When a waitUntil guard fails, the request is not automatically resent, but put in a special queue of the *guard object that implements the failed predicate message*. Now each time a guard object changes, it asynchronously resends all the messages in its special "wait until" queue. In such an approach, requests are only retried, when they actually have a chance to succeed. Therefore, less retries are necessary. However, the design can still livelock and this time it can even deadlock.

If a developer knows that a certain message can livelock (s)he can use FLO/C's exception mechanism (see section 4.5.2) and connector programming to introduce a connector that takes evasive actions, if the guarded object is looping on a request for too long. For example as shown in the following set of rules, the connector can kill the guarded active object, which will break the livelock.

```
(1) obj possibleLivelock. waitUntil guard couldAlwaysBeFalse. endRule
(2) obj possibleLivelock. implies connector resetCounter. endRule
(3) obj methodWasDelayed: p. implies connector killIfTooMany. endRule
```

The first rule represents the possible livelock loop. For every time the guard fails, rule (3) checks if the loop should be broken by killing obj[1]. If however the guard eventually evaluates to true, the method possibleLivelock can execute and rule (2) notifies the connector that the loop has ended.

Nevertheless, polling is a computationally intensive form of blocking. Since it is not amongst FLO/C's goals to model high-performance but to provide high-level abstractions, we believe the polling waitUntil operator to be a tolerable solution.

7.4 Tradeoff between Connector and Component Responsibilities

The paradigm of the FLO/C model is separation of concerns by factoring out interaction code into connectors (see section 3.1). In traditional programming we find no or little interaction code factored out. Section 3 pointed out that this is clearly an undesirable state. The other extreme would be to factor out *all* interaction code of components. However, such components would be reduced to data containers and collections of stand-alone methods[2]. Thus we lose the benefits of object-oriented design and programming.

In our FLO/C model we steer a middle course for this tradeoff between components and connectors responsibilities. Active objects (components) hold a restricted collection of operations. The operations in a collection can call each other (they contain restricted interaction code). Furthermore they can use private and passive helper objects' methods. But they cannot call methods of other active objects. An active object represents a stand-alone domain entity with its data and functionality. Active objects do not keep references to each other, therefore they are not polluted with assumptions of other active objects' behavior.

Nevertheless, any inner activity of an active object can obscure its interactions with other active objects. For example we often[3] use a particular FLO/C idiom to asynchronously propagate computation results. The method that produces the result (e.g. produce) stores it on a particular instance variable by calling an accessor method (e.g. self produced: res). The propagating connector uses the knowledge of this internal behavior of the active object. When the producer stores the result, the connector triggers a rule that asynchronously propagates the result.

```
producer produced: a. impliesLater consumer consume: a. endRule
```

But in this idiom the interaction chain between produce and consume: res is obscured, because the calling of produced: a is hidden. Unfortunately there is no other way to asynchronously propagate computation results in FLO/C. But since a single method call is a simple interaction and since the possible targets of the call are limited to self we believe that a comment in the connector suffices to ease the problem. We also recommend that rules that trigger a relevant computation are placed immediately before the rule that asynchronously propagates the result. Note that thanks to the keyword result this problem does not occur for sequential propagation (see section 4.5).

The idiom demonstrates just one case where connectors hard-wire assumptions on internal behavior of active objects. Therefore, the design problem to identify the proper objects in OO programming is augmented to the design problem to *identify the proper active objects* in FLO/C. One relevant problem is to decide which parts of a behavior a component should activate *itself* (self calls or helper calls) and which parts should be activated by the connector. When the designer puts too much self and helper method calls into a component, we showed that important parts of the interaction design can

[1] Each connector has the possibility to kill its participant. The simplest one is by killing itself.

[2] Methods that solely use their local variables and their arguments.

[3] Chapter 14 presents ten coordination examples implemented in FLO/C.

get lost. This is exactly what we wanted to avoid with FLO/C. Therefore, we propose that an active object should have a *simple* self-calling graph (for example one life-cycle). Note that the "sleeping barbers" example of section 14.10 will present a design of such simple self-calling graphs of active objects. All methods that are important for constraining or activating other active objects must be implemented as stand-alone methods. If a particular components internal interaction behavior is not simple, the component must be implemented as composite object (see chapter 6) where it is internally split into active objects and connectors.

We claim that the identification of active objects and their proper behavior is an open issue that demands future research (see section 16.1).

Part II

Formal Approach

Chapter 8

Formal Specification of FLO/C

The following section precisely specifies how FLO/C models the concurrency of active objects and how it uses rules on message passing to coordinate active objects. With this formal base, we want to prove properties of the operators as well as of FLO/C systems as such. We describe the activities of a FLO/C system by its concurrent, rule guided message passing. The semantics of a method execution is considered as computation thus it is a part of the underlying object model and not a subject here. Since here we are interested in interaction between the objects, we simplify a method execution to an atomic system step which uses no time[1] and returns no value. While we are not interested in the execution of methods as such, we look at *all possible orders* in which the executions of methods can occur in a given system. Concurrency is expressed as possible variations of execution orders. The formal specification of FLO/C systems enables us to analyze all possible execution orders of a given system independently of the semantics of the methods. This will be demonstrated on an example in section 8.5. The effort of the formalization pays off in chapter 9 where we can prove properties like deadlock freedom of the FLO/C model. The formalism allows the definition and prove of properties of operators (see section 9.1), as well as other liveness properties that hold for any FLO/C system (see section 9.2).

The formalism simplifies the complete FLO/C model in the following aspects: a single rulebase instead of connectors, abstraction of method execution, omitting of exception handling and dynamic connector activities. The impact of these limitations is discussed in section 9.3. Finally section 9.4 collects the results of part II into one table.

8.1 Notations

Let $I\!N$ be the set of all natural numbers, and Σ the set of all symbols[2]. We declare the FLO/C system entities in terms of tuples and subsets of these sets. Lists are expressed as binary tuples $(head, RestList)$, where $[]$ is the empty list. Our formalism only allows *finite* lists. We use the system entity definitions to define a FLO/C *system state*. Then, we can describe the dynamic behavior of FLO/C as eight possible kinds of atomic *system state transitions* that can occur in arbitrary order. The transitions use helper functions that lookup FLO/C rules and do the list processing. Thus, after having formally described the FLO/C system we can start to reason about it.

[1] Therefore, our model does not cope with methods that use an infinite amount of time to terminate.

[2] Symbols are unique character strings.

8.2 System Entities

Message and message lists. Let $\mathcal{M}$ be the set of all messages. A *message* is a tuple of a natural number i and a symbol s:
$m = (i, s) \quad m \in \mathcal{M}, i \in I\!N, s \in \Sigma.$
Explanation: The number represents the target of the message, the symbol represents the message selector[3].

A *message list* M is either the empty list or a tuple consisting of a message and a message list.
$M = (m \in \mathcal{M}, M') \mid []$

Consequences. Let $\mathcal{C}$ be the set of all possible consequences to a message. A *consequence* is a five-tuple consisting of a symbol and four message lists.
$c = (t, W, P, Y, A) \quad t \in \Sigma, \quad W, P, Y, A$ message lists.
Explanation: t is the consequence triggering selector, W is the list of waitUntil consequences, P is the list of permittedIf consequences, Y is the list of sequential consequences (consequences of the operators implies and impliesBefore) and A is the list of the asynchronous consequences (operator impliesLater).

Queue. A Queue Q is either the empty list or a tuple consisting of a symbol and a queue.
$Q = (s \in \Sigma, Q') \mid []$
Explanation: The queue stores method selectors that were sent to it.

Active objects. Now, we can define the set $\mathcal{O}$ of all active objects. An *active object* is a four-tuple consisting of a natural number, a queue, a consequence and a set of natural numbers.
$o = (i, Q, c, L) \quad o \in \mathcal{O}, i \in I\!N, Q$ queue, c consequence, $L \overset{finite}{\subset} I\!N$
Explanation: An active object o is identified by i. It has a message queue[4] Q, and a consequence c that represents the currently treated message. The set L represents the currently locked (sometimes we also say "reserved") other objects, using their identifiers. For simplicity we will sometimes note o_i for an object with the identifier i.
Definition. an active object is *idle* when the lists in the consequence and the locks (reservations) are empty:
$o = (i, Q, c, L)$ is idle $\leftrightarrow c = (t, [], [], [], []) \wedge L = \emptyset$

Rule-base. The set of FLO/C operators Ω is defined as follows:
$\Omega = \{\rightarrow, \leftarrow, \rightsquigarrow, |, \|\}$
Explanation: each of the symbolic elements of the set represents a FLO/C operator:

$\rightarrow$	implies	$\leftarrow$	impliesBefore
$\rightsquigarrow$	impliesLater	$\mid$	permittedIf
$\|$	waitUntil		

Let $\mathcal{R}$ be the set of all rules. A rule is a triple, containing a message, an operator and a message.
$r = (m_p, op, m_c) \quad r \in \mathcal{R}, m_p\, m_c \in \mathcal{M}, \; op \in \Omega$

[3] We omit the arguments, because they don't influence the execution order, thus are of no interest in liveness problems.
[4] The queue holds only the message selector, because the target object of messages in the queue is the object holding the queue.

Explanation: Every rule has a precondition message m_p to which a consequence message m_c is related, according to the operator op.

System. The set $\mathcal{S}$ of all FLO/C system states consists of a finite set of active objects O and a finite set of rules R. The set of active objects is restricted: the object identifiers have to be unique.
$S = (O, R) \quad S \in \mathcal{S}, O \overset{finite}{\subset} \mathcal{O}, R \overset{finite}{\subset} \mathcal{R}$ Furthermore:
$o_i \in O \wedge o_i' \in O \Longrightarrow o_i = o_i'$
Explanation: Since messages find their target object by identifying natural numbers, these identifiers must be unique.
Definition: Initial states of a system. In order for $S = (O, R)$ to be a valid *initial* state all active objects in O must be idle:
$S_{init} = (O, R), \quad O \overset{finite}{\subset} \mathcal{O}, (i, Q_i, c_i, L_i) \in O \Rightarrow L_i = \emptyset \wedge c_i = (t_i, [], [], [], [])$ Note that the queues do not have to be empty. In fact an initial state can only show behavior if there are messages in the queues.

8.3 System State Transitions

We describe possible system state transitions from the state S to S' by eight *transition rules*[5] of the form $S \overset{\langle \text{action} \rangle}{\Leftrightarrow} S'$. System transitions are atomic, but can fire in arbitrary order.

A system transition changes objects of the system state. It can fire, when one or several objects meet the precondition. Most transitions change only one object of the system state, only one transition changes two objects. Therefore, to simplify the notation of transitions, we will put only the relevant objects and constraints in the transition formula, instead of the whole system state. Instead of $(O, R) \overset{\langle \text{action} \rangle}{\Leftrightarrow} (O', R)$ we will write e.g. $o \overset{\langle \text{action} \rangle}{\Leftrightarrow} o'$ assuming that $o \in O$ is replaced by $o' \in O'$. In order to specify the relevant objects' state change in the transition, the objects are noted in their full tuple form.

A system state transition models an activity of an active object. The concurrency of FLO/C's active objects is expressed by the indeterminism which transition rule is firing on which object. The system transitions follow a global scheme: all active objects can pull messages from their queues. Then, they collect the consequences of this message and try to execute them. Consequences involve guards, sequential consequences and asynchronous consequences.

Thus in the first system transition, a message is taken from the queue. To start the consequence execution, it is necessary that all relevant objects involved in the consequences are not already reserved. Then, the function $f_{rules}(i, s)$ calculates the global consequences, and the necessary objects are reserved. Note that the function $f_{rules}(i, s)$ performs the only access of the rule base R. It delivers a *consequence* (five-tuple). After the definition of the state transitions, the function and its helpers will be discussed in detail in section 8.4.

$$\begin{gathered} (i, (s, Q), (t, [], [], [], []), \emptyset) \wedge \\ \forall j (j, Q_j, c_j, L_j) \in O \quad (f_{reserv}(f_{rules}(i, s)) \cap L_j = \emptyset) \\ \overset{\langle \text{Start reaction} \rangle}{\Leftrightarrow} \\ (i, Q, f_{rules}(i, s), f_{reserv}(f_{rules}(i, s))) \end{gathered} \tag{8.1}$$

[5] Not to be confused with the FLO/C rules.

Once the consequences are calculated, the guards are evaluated. Both types of guards (waitUntil and permittedIf) can evaluate to true or false. It follows the two cases of the waitUntil operator: The message (j, g) from the waitUntil list is handled. If it evaluates to true, the message is simply taken out of the list (since it was evaluated). If it evaluates to false, the original selector t, that was used to calculate the consequences, is appended to the queue. This way the selector will be handled again later (the request is blocked but not deleted). The consequences however are deleted (filled with empty lists).

$$(i, Q, (t, ((j, g), W), P, Y, A), L) \overset{\langle\, g \text{ on } j \text{ is true}\rangle}{\Leftrightarrow\!\!\rightarrow} (i, Q, (t, W, P, Y, A), L) \tag{8.2}$$

$$\begin{array}{c}(i, Q, (t, ((j, g), W), P, Y, A), L)\\ \overset{\langle\, g \text{ on } j \text{ is false}\rangle}{\Leftrightarrow\!\!\rightarrow}\\ (i, f_{append}(Q, t), (t, [], [], [], []), \emptyset)\end{array} \tag{8.3}$$

The two cases of the permittedIf operator: If the guard evaluates to true, the reaction can go on like for the waitUntil operator. If it evaluates to false, all the consequences are simply deleted.

$$(i, Q, (t, W, ((j, g), P), Y, A), L) \overset{\langle\, g \text{ on } j \text{ is true}\rangle}{\Leftrightarrow\!\!\rightarrow} (i, Q, (t, W, P, Y, A), L) \tag{8.4}$$

$$(i, Q, (t, W, ((j, g), P), Y, A), L) \overset{\langle\, g \text{ on } j \text{ is false}\rangle}{\Leftrightarrow\!\!\rightarrow} (i, Q, (t, [], [], [], []), \emptyset) \tag{8.5}$$

The sequential consequences (implies and impliesBefore operators): When all guards are evaluated (their lists are empty), the execution of the sequential consequences can start.

$$(i, Q, (t, [], [], (s, Y), A), L) \overset{\langle \text{ execute } s \text{ on } i\rangle}{\Leftrightarrow\!\!\rightarrow} (i, Q, (t, [], [], Y, A), L) \tag{8.6}$$

The release of the object reservations: When all guards and sequential consequences are executed, the reservation of the involved active objects is released.

$$(i, Q, (t, [], [], [], A), L) \overset{\langle \text{ release reservations}\rangle}{\Leftrightarrow\!\!\rightarrow} (i, Q, (t, [], [], [], A), \emptyset) \tag{8.7}$$

The sending of the asynchronous consequences (impliesLater operator): When all guards and sequential consequences are executed, the asynchronous consequences can be sent, by appending them to the proper queues.

$$\begin{array}{c}(i, Q, (t, [], [], [], ((j, s), A)), L) \quad (j, Q_j, c_j, L_j)\\ \overset{\langle \text{ send } s \text{ to } j\rangle}{\Leftrightarrow\!\!\rightarrow}\\ (i, Q, (t, [], [], [], A), L) \quad (j, f_{append}(Q_j, s), c_j, L_j)\end{array} \tag{8.8}$$

Note: this is the only system transition rule involving two objects of the system state since it involves the sender and the receiver object of the message.

8.3.1 Local Transition Firing Order

The state transitions represent the fusion of rules as discussed in section 4.3. From the point of view of a single active object o, the transitions can only happen in certain orders. This is illustrated in the state diagram of figure 8.1.

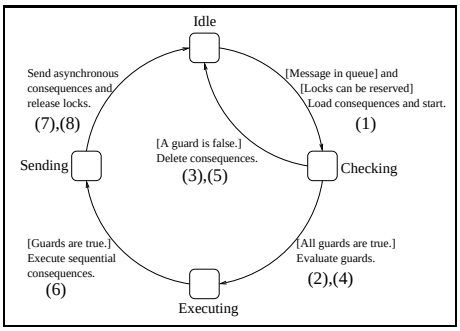

Figure 8.1: Process state changes of an object.

Object is idle. Transition (1) represents the collection and the reservation phase as an atomic step starting from the idle object state. The function $f_{rules}(i, s)$ returns all the consequences, and the function $f_{reserv}(c)$ is used for the reservation of the relevant objects. Transition (1) can only trigger, when there is a message in the queue of o and when the reservation can succeed.

Object is checking the guards. Transitions (2),(3),(4),(5) represent the guard evaluations. When reasoning about the system, it is not generally determinable, if a guard will evaluate to true or false. Thus the four transitions can generate all possible cases. Note that there is no order between the waitUntil and the permittedIf guards. In some cases it could be more useful to evaluate all permittedIf guards before starting with the waitUntil guards, since they are more restrictive. As FLO/C is implemented in an open way (see section 11.2), such behavioral extensions are feasible.

Object is executing methods. Transition (6) represents the executions taking place in the execution phase. It can only trigger when *all guards are evaluated* (when the two guard lists are empty).

Object is sending and releasing. When all executions have taken place (when the sequential consequence list is empty), the reservation set is emptied in transition (7). Note that from this moment on, other transitions can execute methods of this object. Transition (8) sends an asynchronous consequence message. It can trigger only after all sequential consequences are executed, but eventually before transition (7). Note that an object can receive an asynchronous message at any time. Only after the last asynchronous consequence is sent, the object can treat a new message of the queue.

We see that the transitions handle message reception and execution and sending in the order already described in section 4.3 in the model part of this work. In fact system transitions (1) to (7) in the transition cycle (see figure 8.1) represent one set of *synchronized multi-object joint actions* (described in section 2.2.2). Transition (8) represents the propagation of requests.

The state changing from a per-object point of view is simple. However the state changes of different objects can conflict with each others.

8.3.2 Two Requirements for FLO/C's System Transitions

In order to prove statements about properties of a FLO/C systems, we must add two requirements to the firing of transitions. The requirements enforce *progress* of FLO/C's internal mechanisms that are described by the system transitions.

Requirement 1: Progress of objects. In a FLO/C system with n objects, each object is given a chance to participate in a transition after at least m transitions, $m >= n$.

If this requirement does not hold, an active object can stay passive, even if system transitions could fire. Therefore, a FLO/C system intrinsically guarantees progress, if progress is possible. Note that $m < n$ would be an impossible demand. If all n objects are able to participate in an object transition, after m transition there would still be at least one object left that could not participate in a transition although it was able. However, $m = n$ is possible, and it is the way the implementation of FLO/C treats the transitions (see section 11.2).

If the object's lists are not completely empty then for almost any object state there is a transition that can fire. The only transition that uses the states of more than one object as precondition is the transition (1). The transition is trying to lock a number of other active objects, but maybe they are locked already by another transition (1). Therefore, different transition (1) can conflict, possibly starving each other.

Requirement 2: Fairness for transition (1). In a system state S with n objects and r rules, if an object o is only missing the locks in order to participate in transition (1) it will fire transition (1) after at most $r + 3(n \Leftrightarrow 1)$ system transitions.

We saw in section 8.3.1 that transitions do reservation, consequence treatment and release of reservation. When each other object beside of o has completed such a pass (one set of synchronized multi-object joint actions), at most r consequences were treated[6]. The original message was executed at most $n \Leftrightarrow 1$ times, at most $n \Leftrightarrow 1$ reservations and releases occurred. Therefore after at most $r + 3(n \Leftrightarrow 1)$ system transitions object o is waiting the longest for every reservation, it has to do.

Evaluation of the requirements. When an implementation of the FLO/C model meets the two requirements it can rely on the results of section 9. Requirement 1 ensures progress of an active object that is about to treat some consequences. The factor m represents the speed of the slowest of the objects. After at most m transitions the slowest object will try to do something. The requirement 2 is more elaborate. In order to meet it, an implementation can use an algorithm like described in [CM84] for the *drinking philosopher problem*. The algorithm describes how objects can acquire sets of shared resources in a fair way.

Note that the requirement reflects the fact that the active objects *internal mechanisms* do progress. It does not say that the external behavior, namely the execution transitions always do so. However, in section 9 we can use the guarantees of the requirements to prove liveness issues of the external behavior. In section 11.2 we will describe how our implementation meets the requirement stated here.

The next section will explain how functions do the rule lookup.

8.4 Functions

The transition (1) uses functions in order to read from the rule base. We start two well known list manipulation functions: appending and concatenation that are used as helper functions.

List manipulating functions. The function to append an element to the end of a list:

[6]There cannot be more than r consequences because no rule is triggered twice. This is because only one request per object is treated and because no rule can trigger for two different objects.

$$\begin{aligned} f_{append}((x, L), y) &= (x, f_{append}(L, y)) \\ f_{append}([], y) &= (y, []) \end{aligned}$$

The function for concatenation:

$$\begin{aligned} f_{concat}(L_1, (y, L_2)) &= f_{concat}(f_{append}(L_1, y), L_2) \\ f_{concat}(L_1, []) &= L_1 \end{aligned}$$

Simplified list notations. Some of the following functions iterate through lists and append or concatenate the results. For the readers convenience we will use a simplified notation for lists, using the square brackets as list delimiters and the comma as a concatenation operator. For example the list $f_{concat}(f_{append}(L_1, e_1), L_2)$ where L_i are lists and e_1 is an element is noted: $L_1, [e_1], L_2$

Helper functions to identify the necessary locks. The following function calculates the set of objects that must be reserved for a given consequence five-tuple. The resulting set contains all the objects, that are target to a guard or a sequential execution. This is because these are the participants of one set of joint actions. As argued in section 2.2.2 they should not be changed by third-party access during intermediate states.

$$f_{reserv}((c, W, P, Y, A)) = f_{targets}(W) \cup f_{targets}(P) \cup f_{targets}(Y)$$

The function uses the following helper function to collect the target identifiers:

$$\begin{aligned} f_{targets}((i, s), L) &= i \cup f_{targets}(L) \\ f_{targets}([]) &= \emptyset \end{aligned}$$

Functions for the recursive rule lookup. We start top down, with the function $f_{rules}(i, s)$ that takes an object identifier and a selector and returns a complete consequence five-tuple:

$$\begin{aligned} f_{rules}(i, s) = (s, &f_{consFor}(f_{findSync}(i, s), \|), f_{consFor}(f_{findSync}(i, s), |), \\ &f_{findSync}(i, s), f_{consFor}(f_{findSync}(i, s), \rightsquigarrow)) \end{aligned}$$

Basically, the function $f_{findSync}(i, s)$ *recursively* looks up the sequential consequences list, while $f_{consFor}(M, op)$ works on the resulting list, finding the consequences for the given operator for any message in the list. Thus, the resulting tuple contains the complete lists of consequences of the consequences, ordered by operators. Note how $f_{consFor}(M, op)$ accesses the rule base R.

$$\begin{aligned} f_{consFor}(((i, s_1), L), op) &= f_{asList}(\{(j, s_2) \mid ((i, s_1), op, (j, s_2)) \in R\}),\ f_{consFor}(L, op) \\ f_{consFor}([], op) &= [] \end{aligned}$$

The function $f_{asList}(M \overset{finite}{\subset} \mathcal{M})$ converts the given set of consequence messages into a list. FLO/C does not specify a particular order here, as long as: $m \in M \Leftrightarrow m$ is in list $f_{asList}(M)$.

The sequential consequences (implies and impliesBefore) and their consequences of consequences can be seen as a binary tree. Each node (a message) can have a branch to a list of nodes

that express what should happen before the node (its impliesBefore consequences) and a branch to a list that expresses what should happen after the node (its implies consequences). $f_{find}(i, s, H)$ linearizes the tree by an in-order traversal. The set H contains history information of the traversal (a set of the father nodes). Once a new node is traversed that is already in the H set, a cycle is detected and the recursion stops. The function $f_{findSync}(i, s)$ starts the recursion, with an empty set as history information.

$$
\begin{aligned}
f_{findSync}(i,s) &= f_{find}(i,s,\emptyset) \\
f_{find}(i,s,H) &= \begin{cases} [] & \text{if } (i,s) \in H \\ f_{collectFor}(i,s,\leftarrow,H)\,,\ [(i,s)]\,,\ f_{collectFor}(i,s,\rightarrow,H) & \text{otherwise} \end{cases}
\end{aligned}
$$

In the $f_{find}(i, s, H)$ function we see how the father node (i, s) is appended after all impliesBefore consequences, then all implies consequences are concatenated to build the well ordered result.

The $f_{collectFor}(i, s, op, H)$ collects the consequence list of either of the two branches (depending on *op*). It uses the $f_{findList}(M, H)$ function that recurs to $f_{find}(i, s, H)$, passing each message of its list as argument. Note that $f_{collectFor}(i, s, op, H)$ updates the history information.

$$
f_{collectFor}(i,s,op,H) = f_{findList}(f_{consFor}(((i,s),[]),op), H \cup (i,s))
$$

$$
\begin{aligned}
f_{findList}(((i,s),L),H) &= f_{find}(i,s,H)\,,\ f_{findList}(M,H) \\
f_{findList}([],H) &= []
\end{aligned}
$$

The next section presents the functions' application on a given FLO/C rule-base.

8.5 Analyzing Possible Execution Orders of an Example System

Before we start to use the formalism to prove liveness properties, we present a step-by-step example. We will evaluate all possible state transitions of a given FLO/C system.

Let $S_{init} = (O, R)$
$O = \{o_1, o_2\} = \{(1, Q_1, c_1, L_1), (2, Q_2, c_2, L_2)\}$ and o_1, o_2 are idle

$$
\begin{aligned}
R = \{&(1,\text{move}) \rightarrow (2,\text{move}), \\
&(2,\text{move}) \leftarrow (1,\text{move}), \\
&(2,\text{move}) \mid (1,\text{allow}), \\
&(2,\text{move}) \rightsquigarrow (1,\text{turn}), \\
&(2,turn) \rightarrow (1,\text{turn})\}
\end{aligned}
$$

Thus, the initial system state consists of two idle objects (no reservation, empty consequence lists), and five rules. The objects know the methods move and turn, one of them additionally knows the predicate message allow. Roughly, the rules compose two sets of joint actions. The first set orders the moving executions: o_1 moves before o_2. The moving is constrained by the $allow$ method of o_1. The joint actions can be requested by sending $move$ to o_1 as well as o_2 because the rules form a sequential cycle. This example will show in detail, how the cycle is broken. After the successful joint actions, a request to turn will be sent to o_1 which starts a new set of joint actions that turns both objects.

In order to start the analysis we need to know the contents of the queues of both objects. If the queues are empty, no transition can apply at all. Therefore, let Q_2 be empty but $Q_1 = (\text{move}, [])$. Now we can generate all possible orders of system transition. Furthermore, we will show a full trace of the role lookup functions.

8.5.1 The Start Transition and a Function Trace

From S_{init} only the transition (1) can trigger on o_1. It collects the consequences of the message move, using the function $f_{rules}(1, \text{move})$, which in its turn essentially uses $f_{findSync}(1, \text{move})$, which collects and orders all implies and impliesBefore consequences.

$$f_{findSync}(1, \text{move}) = \\ f_{collectFor}(1, \text{move}, \leftarrow, \emptyset), [(1, \text{move})], f_{collectFor}(1, \text{move}, \rightarrow, \emptyset)$$

This is a list starting with the impliesBefore consequences and their further implications, followed by the actual message $(1, \text{move})$ followed by the implies consequences and their further implications. Let's trace the impliesBefore part first. It will return an empty list, because there are no impliesBefore consequences for $(1, \text{move})$, but for completeness, we shall spawn this trace.

$$\begin{array}{ll} f_{collectFor}(1, \text{move}, \leftarrow, \emptyset) & = \\ f_{findList}(f_{consFor}([(1, \text{move})], \leftarrow), \{(1, \text{move})\}) & = \\ f_{findList}([], \{(1, \text{move})\}) & = [] \end{array}$$

More interesting is the implies consequences branch of $(1, \text{move})$ it will not only have a consequence, but this will have a consequence again, that even leads to a cycle.

$$\begin{array}{ll} f_{collectFor}(1, \text{move}, \rightarrow, \emptyset) & = \\ f_{findList}(f_{consFor}([(1, \text{move})], \rightarrow), \{(1, \text{move})\}) & = \\ f_{findList}([(2, \text{move})], \{(1, \text{move})\}) & = \\ f_{find}(2, \text{move}, \{(1, \text{move})\}) & = \\ f_{collectFor}(2, \text{move}, \leftarrow, \{(1, \text{move})\}), [(2, \text{move})], f_{collectFor}(2, \text{move}, \rightarrow, \{(1, \text{move})\}) & \end{array}$$

The $f_{collectFor}(2, \text{move}, \rightarrow, \{(1, \text{move})\})$ branch will return an empty list, for the same reason already seen before: there are no implies rules in R for the message $(2, \text{move})$. Therefore, let's investigate the other branch:

$$\begin{array}{ll} f_{collectFor}(2, \text{move}, \leftarrow, \{(1, \text{move})\}) & = \\ f_{findList}(f_{consFor}([(2, \text{move})], \leftarrow), \{(1, \text{move}), (2, \text{move})\}) & = \\ f_{findList}([(1, \text{move})], \{(1, \text{move}), (2, \text{move})\}) & = \\ f_{find}(1, \text{move}, \{(1, \text{move}, (2, \text{move}))\}) & = [] \end{array}$$

The cycle in the sequential consequences is broken, because the history list argument $\{(1, \text{move}), (2, \text{move})\}$ contains the function argument $(1, \text{move})$. Now we can collect the results:

$$\begin{array}{rl} f_{findSync}(1, \text{move}) = & [], [(1, \text{move})], [], [(2, \text{move})], [] = \\ & [(1, \text{move}), (2, \text{move})] \end{array}$$

In order to get the result of the main function $f_{rules}(1, \text{move})$, we need to feed the result list of $f_{findSync}(1, \text{move})$, namely $[(1, \text{move}), (2, \text{move})]$ into the $f_{consFor}()$ function using the operator that we are interested in. Because up to now, we only took care of the impliesBefore and implies consequences. But all of them can cause consequences of other operators that must be taken

into account. The $f_{consFor}()$ function will check for each entry of the sequential consequence list $[(1, \text{move}), (2, \text{move})]$, if they imply further consequences for the given (up to now neglected) operator.

$$\begin{array}{lcl} f_{consFor}(f_{findSync}(1, \text{move}), \|) & = & [] \\ f_{consFor}(f_{findSync}(1, \text{move}), |) & = & [(1, \text{allow})] \\ f_{consFor}(f_{findSync}(1, \text{move}), \rightsquigarrow) & = & [(1, \text{turn})] \end{array}$$

Thus the complete consequence and the reservation set are:

$$\begin{array}{ll} f_{rules}(1, \text{move}) & = (\text{move}, [], [(1, \text{allow})], [(1, \text{move}), (2, \text{move})], [(1, \text{turn})]) \\ f_{reserv}(f_{rules}(1, \text{move})) & = \{1, 2\} \end{array}$$

Note that both objects are engaged in a sequential consequence (both have to move), therefore they must both be reserved.

With the complete results of the functions we can determine, if the first system transition can fire or not. It can, because the both objects are idle (therefore hold no reservation). This leaves us with the new system state S' holding the object set $O' = \{o'_1, o_2\}$ where
$o'_1 = (1, [], (\text{move}, [], [(1, \text{allow})], [(1, \text{move}), (2, \text{move})], [(1, \text{turn})]), \{1, 2\})$

8.5.2 Execution of the Consequences

In the current state S' there is still no transition for o_2 possible. The only possible transitions for o'_1 are the ones that evaluate the guard (transitions (4) and (5)). From our abstract view of the system we cannot tell if the guard would fail or not, therefore we have to consider both possibilities. Let us first assume that the guard would fail. This is represented by the transition (5).

$$(1, [], (\text{move}, [], [(1, \text{allow})], [(1, \text{move}), (2, \text{move})], [(1, \text{turn})]), \{1, 2\})$$

$$\langle \text{ allow on } 1 \text{ is false} \rangle \atop \Leftrightarrow\!\!\rightarrow$$

$$(i, [], (\text{move}, [], [], [], []), \emptyset)$$

Clearly, after this transition no other transition can fire anymore. But maybe, instead of transition (5) transition (6) could have fired. This represents the case, where the guard evaluates to true.

$$(1, [], (\text{move}, [], [(1, \text{allow})], [(1, \text{move}), (2, \text{move})], [(1, \text{turn})]), \{1, 2\})$$

$$\langle \text{ allow on } 1 \text{ is true} \rangle \atop \Leftrightarrow\!\!\rightarrow$$

$$(1, [], (\text{move}, [], [], [(1, \text{move}), (2, \text{move})], [(1, \text{turn})]), \{1, 2\})$$

Since both guard queues are empty, the execution of the sequential consequences can start (transition (6)). This transition can fire twice, while no other transition can fire.

$$(1, [], (\text{move}, [], [], [(1, \text{move}), (2, \text{move})], [(1, \text{turn})]), \{1, 2\})$$

$$\langle \text{ execute move on } 1 \rangle \atop \Leftrightarrow\!\!\rightarrow$$

$$(1, [], (\text{move}, [], [], [(2, \text{move})], [(1, \text{turn})]), \{1, 2\})$$

$$(1, [], (\text{move}, [], [], [(2, \text{move})], [(1, \text{turn})]), \{1, 2\})$$

$$\langle \text{ execute move on } 1 \rangle \atop \Leftrightarrow\!\!\rightarrow$$

$$(1, [], (\text{move}, [], [], [], [(1, \text{turn})]), \{1, 2\})$$

In the resulting system state, two transitions can fire. The asynchronously sending transition and the reservation release transition. It is unimportant, which one fires first. When the asynchronous sending happens first, the release will be the only further option[7] and vice versa.

$$(1, [], (\text{move}, [], [], [], [(1, \text{turn})]), \{1, 2\})$$

$$\langle \text{ send } \mathsf{turn} \text{ to } 1 \rangle$$
$$\Leftrightarrow\rightarrow$$
$$(1, [\text{turn}], (\text{move}, [], [], [], []), \{1, 2\})$$

$$(1, [\text{turn}], (\text{move}, [], [], [], []), \{1, 2\})$$

$$\langle \text{ Release reservations} \rangle$$
$$\Leftrightarrow\rightarrow$$
$$(1, [\text{turn}], (\text{move}, [], [], [], []), \emptyset)$$

Only now, when the object reservations are released, the sent turn message can be treated in transition (1). The consequences contain no guards, no asynchronous consequences. They only order the two turn executions.

$$(1, [\text{turn}], (\text{move}, [], [], [], []), \emptyset)$$

$$\langle \text{ Start reaction} \rangle$$
$$\Leftrightarrow\rightarrow$$
$$(1, [], (\text{turn}, [], [], [(1, \text{turn}), (2, \text{turn})], []), \{1, 2\})$$

The only possible transitions are the execution of the messages, and then the release of the reservations.

$$(1, [], (\text{turn}, [], [], [(1, \text{turn}), (2, \text{turn})], []), \{1, 2\})$$

$$\langle \text{ execute } \mathsf{turn} \text{ on } 1 \rangle$$
$$\Leftrightarrow\rightarrow$$
$$(1, [], (\text{turn}, [], [], [(2, \text{turn})], []), \{1, 2\})$$

$$(1, [], (\text{turn}, [], [], [(2, \text{turn})], []), \{1, 2\})$$

$$\langle \text{ execute } \mathsf{turn} \text{ on } 2 \rangle$$
$$\Leftrightarrow\rightarrow$$
$$(1, [], (\text{turn}, [], [], [], []), \{1, 2\})$$

$$(1, [], (\text{turn}, [], [], [], []), \{1, 2\})$$

$$\langle \text{ Release reservations} \rangle$$
$$\Leftrightarrow\rightarrow$$
$$(1, [], (\text{turn}, [], [], [], []), \emptyset)$$

After this transition, no further transition is possible.

8.5.3 Evaluation of the Example

The tracing of the possible state transitions reveals the possible execution orders. There are two main paths, they represent the truth value of the allow guard, and there is an effect-less variation between

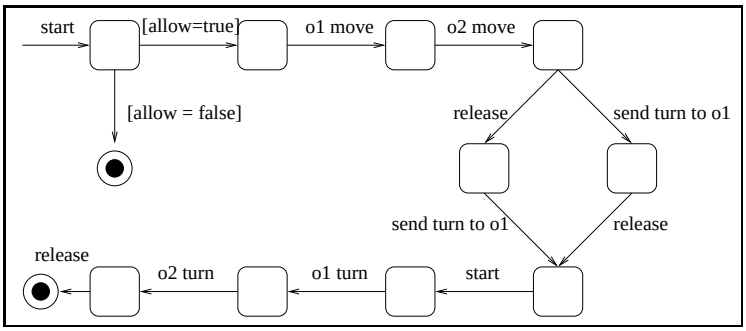

Figure 8.2: The system transitions as state diagram.

asynchronous sending and release of the reservation. This leads to the transition order represented in figure 8.2.

The guard transitions show at which stage a guard method is executed, and the execution transitions show other method executions. Thus the transitions reveal all possible execution orders:
$(1, \text{allow}^{false})$ or
$(1, \text{allow}^{true}), (1, \text{move}), (2, \text{move}), (1, \text{turn}), (2, \text{turn})$.
We see that heavy use of sequential operators (opposite to the use of impliesLater operator) almost completely sequentializes the behavior. The small bit of concurrency that is still left in the example would show, if at the starting state *both* objects would have hold a move message in their queue. Still, the behavior is similar: One of the two possible transition (1) would fire, which would lead to the execution or failure of one of the move joint actions. It is not determinable if o_1 or o_2 fire transition (1) first. But both movement requests will lead to separate joint-actions that move the objects, and to separate joint actions that turn the objects.

[7] This is because both objects of the system are still reserved.

Chapter 9

Properties of the Formal FLO/C Model

As the former example showed, the formal approach did already prove itself for analyzing possible execution orders of a given FLO/C system. This section will prove some *execution* behavior of the different operators, and then of the model as such. It will rely on the requirements for system transition firing put up in section 8.3.2.

9.1 Execution Properties of the Operators

Here we are interested in statements about the *execution order* given a rule with a certain operator, and statements about *actuality of object states* of objects involved in rules with a given operator. We analyze all those properties by reasoning on the order of state transitions. Usually, we assume the presence of an ordered list of transitions T that is either long enough for the analysis, or that ends because it lead to a system state, where no further transition is possible. The list contains the system transitions in an order they can occur for a given system. T contains the execution transition of the precondition of the rule with the operator of interest. Given such a T we are then able to conclude statements of the form: "since T contains the execution of the precondition message it must (or must not) contain the execution of consequence message in between a calculable distance of the preconditions execution transition. To ease the notation, the elements of a transition list T are labeled (e.g with T_p, T_q, etc).

Ordering property. The ordering property of an operator expresses that if the precondition message was executed in a given transition sequence, then *the consequence message execution must also be in that sequence within a certain distance*. Note that the property can also be specialized to define if the precondition execution has to be before or after the consequence execution.

Actuality property. Given an operator with the ordering property, the actuality property expresses that between the execution of the precondition- and the consequence message both these participants *are protected from third-party access*. We call this "actuality" because in between the two executions the state of the participants stay "actual".

We will show that both guard operators (permittedIf, waitUntil) and sequential ordering operators (implies, impliesBefore) have both properties, and that the impliesLater operator has neither. Note that these results reflect the purpose of the operators. The operators to compose synchronized multi-object joint actions must fulfill the properties, while the light-weighted communication operator should not (see section 2.2.2).

To prove the ordering property we will use the definition of the function $f_{rules}(i, s)$ which orders the consequence lists. Together with requirement 1 (presented in section 8.3.2) that ensures progress, the transition firing order will then prove the ordering property. The proof of the actuality property is based on the fact that the reservations are not released between the execution of the precondition and a consequence.

Before we start the proofs of the properties we have to analyze FLO/C's cycle breaking mechanism for sequential consequences, because they make the reasoning on such consequences more complex. For a given rule-base R containing r rules, we define the set of all *indirect sequential consequences* of a message ISC(i, s_1) recursively as the union of the set of all direct sequential consequences and their indirect sequential consequences. The size of the set of indirect sequential consequences is limited by $r + 1$, which is the number of consequences that are available in the rule-base R and the original message. Therefore, the following algorithm collects them in at most r recursions.

```
collectIndSeqCons(message,ISCSet,Rules) [
list := collectDirectCons(message,R)
for list do: [ anElement |
   if anElement is not in ISCSet then: [
      ISCSet add anElement
      collectIndSeqCons(anElement, ISCSet, Rules)
      ]
   ]
]
```

The algorithm is started with ISCSet containing message, since the starting message is also part of the joint actions.

Properties of the indirect sequential consequences. The set of indirect sequential consequence ISC(i, s) for a message (i, s) contains exactly the same messages than the sequential consequence list Y loaded by the function $f_{rules}(i, s)$, since they both recursively collect each possible sequential consequence. However, the ISC set is (per definition) unordered. Let $f_{rules}(i, s) = (t, W, P, Y, A)$ then we can say: $(j, s_2) \in Y \Leftrightarrow (j, s_2) \in$ ISC(i, s)

The ISC set can be used to determine, if two messages sequentially imply each other. Thus we can check if two messages form a sequential cycle. Namely when both are in each others indirect sequential consequence set: $(i, s_1) \in$ ISC$(j, s_2) \wedge (j, s_2) \in$ ISC(i, s_1). Now we have a mean to discuss the properties of the operators, separating the special case of sequential cycles.

9.1.1 The implies Operator.

Given a FLO/C system with a rule-base R with r rules where one of them has an implies consequence for the message (i, s_1) that is not cyclic to (i, s_1). Let T be possible state transition path that contains the execution of (i, s_1):

$$R = (..., (i, s_1) \rightarrow (j, s_2), ...)$$
$$(i, s_1) \notin \text{ISC}(j, s_2)$$
$$T = (..., (o_k \overset{\langle \text{ execute s1 on } i \rangle}{\Leftrightarrow} o'_k)_p, ...)$$

Note that ISC(i, s_1) contains (j, s_2) because of the rule with the implies operator. Therefore, if ISC(j, s_2) also contains (i, s_1), (i, s_1) and (j, s_2) are cyclic to each other. Note also that o_k is the

object that is running the joint actions. Now we can start to reason about what happens with the consequence message (j, s_2) in T.

Theorem 1 ***Ordering property of implies.*** *If (i, s_1) is not an indirect sequential consequence of (j, s_2) (in which case there would be a cycle), the* FLO/C *model guarantees that the execution of the consequence (j, s_2) must follow in T:*

$$\exists q \text{ with } T_q = (o_k^* \overset{\langle \text{ execute s2 on } j \rangle}{\Leftrightarrow\!\!\rightarrow} o_k'^*) \text{ and } p < q \leq p + rm$$

Note that the execution of (j, s_2) must not *immediately* follow the one of (i, s_1) for two reasons: Other sequential consequences can be in the Y list of o_k, ordered between them, or transitions of unrelated objects can be ordered in between the two executions.

Prove. If $T_p = (o_k \overset{\langle \text{ execute s1 on } i \rangle}{\Leftrightarrow\!\!\rightarrow} o_k')$ this means that o_k has loaded the consequences for a message in its queue, which has also lead to the adding of $(1, s_1)$ into the sequential consequences list. Therefore, at this transition the helper function was called as follows:
$[f_{collectFor}(i, \text{s1}, \leftarrow, H), (i, \text{s1}), f_{collectFor}(i, \text{s1}, \rightarrow, H)]$. Now the implies branch $(\rightarrow)$ of this list must return a list containing (j, s_2) because this sequential consequence is in the rule base R:
$f_{collectFor}(i, \text{s1}, \rightarrow, H) = [..., (j, s_2), ...]$. The only possibility that the function could return the empty list is the case where the history set H already contains (j, s_2), and thus a cycle was broken. But since (i, s_i) is not an indirect sequential consequence ISC(j, s_2), this case is ruled out. Therefore, we can say that the complete list of the sequential consequences Y of o_k, after the transition T_p must have looked like that: $Y_k = [..., (i, s_1), ..., (j, s_2), ...]$. Since (i, s_1) was executed at system transition p, we can conclude that o_k stays able to fire further execution transitions. Now requirement 1 requires that at least every m transitions o_k is given a chance to fire again. The Y list can carry at best $r \Leftrightarrow 1$ consequences between (i, s_1) and (j, s_2), therefore execution of (j, s_2) is guaranteed to take place in between mr transition steps:
$\Rightarrow T_q = (o_k^* \overset{\langle \text{ execute s2 on } j \rangle}{\Leftrightarrow\!\!\rightarrow} o_k'^*)$ and $p < q \leq p + rm$
Therefore, the rule $(i, s_1) \rightarrow (j, s_2)$ enforces that *after* the execution of (i, s_1) the execution of (j, s_2) *must* follow after a finite amount of transition steps. We call this the *ordering property* of the implies operator.

If we would not demand the cycle freeness between the two messages, the reasoning would not be that easy. The execution of (i, s_1) at step p would not necessarily lead to the later execution of (j, s_2), because in a given system (j, s_2) might already have been executed before, causing the execution of (i, s_1) at p. Nevertheless, with more detailed analysis of the cases, we can conclude that either one or the other must follow, therefore the two messages always form pairs with maximum distance of rm transition steps.

Let us now discuss what can happen in between such pairs, that are caused by an implies operator.

Theorem 2 ***Actuality property of implies.***

Given

$$R = (..., (i, s_1) \rightarrow (j, s_2), ...)$$

$$T = (..., (o_k \overset{\langle \text{ execute s1 on } i \rangle}{\Leftrightarrow\!\!\rightarrow} o_k')_p, ..., (o_k'' \overset{\langle \text{ execute s2 on } j \rangle}{\Leftrightarrow\!\!\rightarrow} o_k''')_q, ...)$$

q Nearest occurrence of (j, s_2) to p

For any transition T_o in between p and q holds that if it is executing something on o_j or o_i it must be a transition (more precisely the execution transition(6)) *on the object* o_k[1].

Prove. If this was not so, a transition step $T_o, \quad p < o < q$ would be a transition on another object $o_l \quad l \neq k$. But then o_l would hold locks for which ever o_i or o_j it is changing in transition T_o. Since both of these locks must also be held by o_k there is a contradiction to the precondition of the transition (1), which requires the exclusive reservation of all necessary locks, therefore this situation is impossible.

Now we can conclude that o_i and o_j are protected from access by executions caused from transitions applying to another object than o_k. In fact, the only thing that can happen to o_i and o_j when they are executed sequentially is that they can receive asynchronous messages, or that the joint actions going on on o_k include different actions (method executions) on them. Therefore, as expected, the implies operator can be used to build synchronized multi-object joint actions, where the participants are protected against interleaved state change. The rule $(i, s_1) \rightarrow (j, s_2)$ enforces that at the execution of (j, s_2) the object o_i is still in the state the method (i, s_1) left it. We call this property *actuality* of (i, s_1) for (j, s_2).

9.1.2 The impliesBefore Operator.

This operator behaves symmetrically to the implies operator. Given a FLO/C system with a rule base R with r rules, where one of them has an impliesBefore consequence for the message (i, s_1), which is not cyclic to (i, s_1).

Theorem 3 ***Ordering property of impliesBefore.***

Let T be a possible state transition path that contains the execution of (i, s_1)

$$R = (..., (i, s_1) \leftarrow (j, s_2), ...)$$
$$(i, s_1) \notin \text{ISC}(j, s_2)$$
$$T = (..., (o_k \overset{\langle \text{ execute s1 on } i \rangle}{\Leftrightarrow\!\!\rightarrow} o'_k)_p, ...)$$

The FLO/C model guarantees that the execution of the consequence (j, s_2) must be in T some when ***before*** the execution of (i, s_1).

$$\exists q \text{ with } T_q = (o^*_k \overset{\langle \text{ execute s2 on } j \rangle}{\Leftrightarrow\!\!\rightarrow} o'^*_k) \text{ and } p \Leftrightarrow rm \leq q < p$$

The proof of this property is analog to that of the implies operator. The impliesBefore operator has the same *ordering* property only inverting the order of the precondition and the consequence.

Theorem 4 ***Actuality property of impliesBefore.***

The very same argumentation as with the implies operator leads to the conclusion that the impliesBefore operator also has the *actuality* property.

[1] Like mentioned before o_k is the object that held (i, s_1) and (j, s_2) in its Y list.

9.1.3 The Guard Operators.

The guards that belong to a set of joint actions are all executed before the start of the execution of the sequential consequences. Given

$$R = (..., (i, s_1)(| \text{ or } ||)(j, s_2), ...)$$
$$T = (..., (o_k \overset{\langle \text{ execute s1 on } i \rangle}{\Leftrightarrow\!\!\!\rightarrow} o'_k)_p, ...)$$

The FLO/C system guarantees, the *positive* evaluation of the guard (j, s_2) must be in T some when before the execution of (i, s_1).

Theorem 5 ***Ordering property of the guard operators.***

$$\exists q \text{ with } T_q = (o^*_k \overset{\langle \text{ s2 on } j \text{ is true} \rangle}{\Leftrightarrow\!\!\!\rightarrow} o'^*_k) \text{ and } p \Leftrightarrow rm \leq q < p$$

Again, the proof follows the pattern used for the implies operator. Only this time, the consideration of cycles is irrelevant, because guards cannot form a cycle, they are not collected recursively. Since the execution of (i, s_1) was in T, (i, s_1) must have been in the list of sequential consequences $f_{findSync}()$. But then, one of the guard lists must have contained (j, s_2) since the guard lists are calculated with $f_{consFor}(f_{findSync}(), guardOp)$ and (j, s_2) is a predicate message of (i, s_1) in R. Before (i, s_1) can be executed, the guard lists must be empty (see definition of transition (6)). In order for (i, s_1) to stay in the execution list Y (not being erased) and being executed, the guard lists must be emptied by transitions that represent positive guard evaluation. We showed that the guard lists contain at least the message (j, s_2). Therefore, if the execution of (i, s_1) is in T, the positive evaluation of (j, s_2) must be there before.

Theorem 6 ***Actuality property of the guard operators.*** *The* actuality *property holds for the guard operators.*

The proof follows the previous patterns of actuality proves. It relies on the fact that the guard objects are reserved. The actuality property is important for guards, since the execution of (i, s_1) should only take place, when the guard is true, which cannot be guaranteed when o_j can be changed from outside in between the guard evaluation and the execution of the guarded message.

9.1.4 The impliesLater Operator.

Theorem 7 *Neither the ordering nor the actuality property holds for the impliesLater operator.*

Given

$$R = (..., (i, s_1) \rightsquigarrow (j, s_2), ...)$$

$$T = (..., (o_i \overset{\langle \text{ execute s1 on } i \rangle}{\Leftrightarrow\!\!\!\rightarrow} o'_i)_p, ...)$$

This only guarantees that (j, s_2) will be sent some when later.

$$\exists q \text{ with } T_q = (o^*_i, o^*_j \overset{\langle \text{ send s2 to } j \rangle}{\Leftrightarrow\!\!\!\rightarrow} o'^*_i, o'^*_j) \text{ and } p < q \leq p + rm$$

Thus (j, s_2) is sent after at most rm steps. Again, this is because the sum of sequential and asynchronous consequence messages to a message cannot be bigger than the number of rules r. However, when (j, s_2) eventually gets treated by transition (1) on o_j, it is possible that a guard forbids its execution. Therefore, the ordering property does not hold. Furthermore, the actuality property cannot hold, because s_2 might queue behind other messages to o_j. One of these other messages could trigger transitions that change the state of o_j or even o_i .

9.2 Liveness Properties of the FLO/C Model

Here we discuss three liveness issues: *deadlocks*, *livelocks* and *message loops*. We prove liveness properties that hold for any FLO/C system depending on which kinds of operators are used there. In order to prove the properties we analyze the local transition firing order (see section 8.3.1) together with the progress ensuring requirements for transitions (see section 8.3.2).

9.2.1 Deadlock

If no transition can fire in a given system S, it is called *dead*. Although this is a pessimistic sounding attribute of a system, there are legitimate versions of such states, like the graceful termination. In this section however we are interested in the states, where not "everything" was treated, but the system is dead. Messages that reside in lists can be seen as requests. If such requests don't lead to any external behavior, the system is deadlocked.
Definition: Emptiness. An object o_i is *empty*, when it is idle (the consequence lists are all empty, it holds no locks) and it has no message in its queue:
o_i is empty $\leftrightarrow o_i = (i, [], (t, [], [], [], []), L)$
Definition: Deadlock. A deadlock is a system state, where no *execution*[2] occurs anymore but not all objects are *empty*.

Clearly this is an undesirable state. When an object is not empty, it means that it is still engaged in some coordination. However:

Theorem 8 *According to this definition every* FLO/C *system is deadlock free*

Prove. (by contradiction of the negation.) If a FLO/C system deadlocks, there is at least one object o_i that is not empty. Now we have to consider all different cases of non-emptiness.

1. First we consider the case where one of the consequence lists is not empty. If this is a guard list, then we can conclude with requirement 1 that transactions (2 to 5) will fire, leading to an execution (evaluation of a guard). Thus the system is not deadlocked. Otherwise when the guard lists are empty but the sequential consequence list Y is not, then the execution transition (6) will fire. If The guard lists and Y are empty but the asynchronous consequence list A contains at least a message then transaction (8) can fire, which is not an execution but leads to the next case.

2. If there is an object o_j with messages in its queue, according to requirement 2 the transition (1) will fire after a finite time of transitions, loading the consequences of for example (j, s_2). The transition will load at least the message (j, s_2) itself into the execution list Y. According to the case 1, this must lead to either the execution of a guard or (j, s_2) itself. Therefore, no deadlock can occur, **qed**.

9.2.2 Livelock

A livelock is the case where a request keeps on trying, but is never handled but always delayed. In the FLO/C model, this translates to methods that are always underway but never executed.
Definition: livelock A FLO/C system S livelocks, if there exists a particular message (o_i, s) that forever resides in a queue or any of the consequence lists P, W or Y without ever being executed.

[2] Executions are: executions by transition (6) and evaluation of guards by transitions (2)(3)(4)(5).

Note that every message in one of these lists is waiting to be handled, whereas for example the asynchronous consequence list A or the reservation set L do not contain requests in that sense, therefore they are not of interest here.

Theorem 9 *According to this definition* not every FLO/C system is livelock free.

Consider the rule $((i, s) \parallel (i, \text{alwaysEvaluatingToFalse}))$. In a system containing this rule, and where s is in the queue of o_i, there is a livelock. The consequences of (i, s) contain a waitUntil guard that always evaluates to false. According to transition (3), the request s is put back in the queue and the execution lists are emptied. Therefore, this particular request s for object i will never be executed, but it will stay in the system, either in Q_i or in Y_i.

Using the waitUntil operator can livelock a system, but what happens when we don't use it?

Theorem 10 *Every* FLO/C *system S that contains no rules using the watitUntil operator $\parallel$) is livelock free.*

Prove. Let (i, s) be the message that stays in the system, but never gets executed. We differ the cases where s is in the queue of o_i and when (i, s) is in an execution list.

1. (i, s) can not be in W, since there are no waitUntil consequences therefore W is always empty.
2. If (i, s) is in P, the execution list of the permittedIf consequences then after at most rm transitions, the transitions (4) or (5) applies on the message (i, s). In both cases, the message gets evaluated, therefore there is no livelock.
3. If (i, s) is in the list of the sequential consequences Y, there are two cases to be considered: if the P list of the guard lists is empty, after at most rm transitions, the transition (6) applies, so (i, s) is executed. Note that the list of the waitUntil consequences W must be empty, because there are no rules with the $\parallel$ operator. If however the P list is not empty, we must separately investigate the cases, where a guard fails, and where all guards succeed.
   - If (i, s) is permitted by a permittedIf guard, then, as seen in the case with empty guard lists, (i, s) will be executed after at most rm system transitions.
   - If (i, s) is *not* permitted by a permittedIf guard (transition 5), then the consequence lists are emptied, and (i, s) is not executed. But now, after at most rm system transitions this particular (i, s) message does not reside in the system any more.
4. In the last case o_i is idle and s is in the queue of o_i at position p. After $r + 3(n \Leftrightarrow 1)$ system transitions (requirement 2) the transition (1) on o_i will load the consequences for the next request. For the $p \Leftrightarrow 1$ other messages in the queue the consequences are treated before it is the turn of (i, s). The treatment of the consequence for one request is done in at most $m(r + 3)$ transitions because there can be r consequences that need one transition each, and the original message plus the loading and the releasing transition. Therefore, after a finite amount of transitions, each s in Q_i will be removed from the queue, and put in the Y_i list, where it gets treated as discussed above in case 2, not leading to a livelock, **qed**.

This theorem gives us a way to ensure that a given FLO/C program is livelock free. We must simply use no rule with a waitUntil operator. In section 14.2 we present an example that synchronously moves graphical objects. The example does not use the waitUntil operator therefore it is livelock

free. In examples that do use the waitUntil operator we must analyze the possible execution orders as demonstrated in section 8.5 when we want to be sure about liveness. We claim that the presented formalism can serve as a base for the development of methods and tools to analyze the liveness properties of a given FLO/C system. We consider such development to be future work (see section 16.1)

9.2.3 Operator Loops

The previous prove does not preclude the queues from getting fuller and fuller. Consider a FLO/C system with four rules:
$R = ((1,s) \rightsquigarrow (1,s)), ((1,s) \rightsquigarrow (2,s)), ((2,s) \rightsquigarrow (1,s)), ((2,s) \rightsquigarrow (2,s))$
Although the liveness of this system is guaranteed ($(1,s)$ and $(2,s)$ will be executed after at most $5m$ transitions), the queues of o_1 and o_2 grow with each execution. Each time after a message is executed, two asynchronous consequences are sent. Therefore the asynchronous consequence can multiply requests. Only one other operator has (a limited) possibility, to reproduce the request by resending it, namely the waitUntil operator. Therefore, both the waitUntil and the impliesLater operators can be used to form an *asynchronous loop*, e.g. by the following rules:
$((1,s) \rightsquigarrow (1,s))$ respectively
$((1, \text{alwaysEvaluatingToFalse}) \parallel (1, \text{alwaysEvaluatingToFalse}))$

The permittedIf operator is unable to produce a loop, the implies and impliesBefore could loop, if it was not for the cycle breaking mechanism.

Theorem 11 *A FLO/C system without rules containing the waitUntil or the impliesLater operator, will be dead after a finite amount of transitions.*

Prove. None of the other operators cause the putting of messages into a queue. The n messages already in the different queues take at most $n(r+3)$ transitions to be treated, then the objects are empty.

Therefore, FLO/C systems that only use rules with the operators permittedIf implies and impliesBefore do not loop, and as seen in section 9.1 have strong concurrency restricting properties. Note that these are the operators inspired by the synchronous FLO model [Duc97b].

The next section shows the simplification made between the full FLO/C model as described in Part I and this formal FLO/C approach, and how these simplifications impact on the previous results.

9.3 Limitations of the Formal Approach

The main goal of the formal approach was to unambiguously describe the rule fusion and give a mean of reasoning about possible execution paths. Non of the upcoming limitations will break the previous results, but still affect them:

- In the full FLO/C model, rules are spread in connectors, and address their participants not the objects themselves. Participants can be groups of objects. In the full model, rules can have more than one consequence. All these generalizations can be neglected, because a given set of connectors and participants of the full model can easily be translated to the simplified form of the rule base used in the formal approach. In fact, when the implementation of FLO/C caches consequences, it also reduces away the connectors and role groups (see section 11.4 and 13.3).

- When a message is executed in the formal model, this is done atomically. However, the method could send other messages. If these messages do not trigger rules it is okay to ignore that fact, since according to the paradigm stated in section 3.1, the object only calls helper methods

on itself. If an execution sends a sub-message that is precondition to a rule, FLO/C sends the sub-message *asynchronously*. That way, the state transitions can treat the sub-message consistently without breaking the actuality properties (see section 9.1). However, this provides another way to write endless loops of asynchronous messages, without using the waitUntil or the impliesLater operator.

- The exception handling mechanism of the full FLO/C model (see section 4.5) is not integrated into the formal model. The exception handling mechanism catches not permitted messages, thus allowing the programmer to let an object, possibly a connector try to save the situation. The following rules show how the concept can be abused to form a loop, even worse a livelock.

```
object livelock. permittedIf guard alwaysReturnFalse. endRule
object methodWasForbidden: m impliesLater object livelock. endRule
```

Upon sending of livelock to whatever object plays the role object, the guard gets evaluated, and fails. Then the exception mechanism calls methodWasForbidden: #livelock. This in turn will lead to the resending of livelock. In fact, the two rules simulate the rule:

```
object livelock. waitUntil guard alwaysReturnFalse. endRule
```

which livelocks. The only difference is that the waitUntil version uses less message sending and can be optimized in an implementation of the model. Furthermore, both operators have an independent exception mechanism.

- The *dynamics* of the full FLO/C model is not reflected in the formal model. The fact that connectors (dis-)connect at run-time, would translate in changing the rule base of a FLO/C system between system transitions. Therefore, between two consequence-loading transitions (transition (1)) of the same object[3], changes in the rule base might not be properly reflected. Namely a new rule could be introduced, but still the old consequence are executed. In practise when dynamically changing connectors while participants are working is indeed unsafe in the sense that the programmer does not know, when exactly the rules are enforced on the still working participants.

When we specially take into account the dynamics of the connectors, the helper message calls, and the exception handling situation of a given full model FLO/C system, we are still able to use the previous conclusions. The last section of this chapter will resume them.

9.4 Summary

The introduced formal FLO/C model as specified in section 8 offers ways of analyzing the *execution behavior* of a given FLO/C system. This has been done for an illustrating example in section 8.5. The formal model reflects the indeterminism that comes from the concurrency of active objects, and from the inability to predict the concrete evaluation result of a guard.

In section 9.1, we analyzed execution properties that describe the semantics of operators. Given that a precondition message of a rule was executed, we were able to prove statements about the execution of the consequence message, depending on the operator of the rule. The *ordering property*

[3]= In the midst of some joint actions.

limits the distance between the execution of the precondition and the execution of the consequence in terms of system transitions. The *actuality property* ensures that in between the two executions, no third party request can change the state of the involved objects.

Section 9.2 started to analyze liveness properties that hold for *every* FLO/C system, although there are some simplifications to the full FLO/C model (described in part I of this thesis) as discussed in section 9.3.

The following table shall summarize the found and proven execution properties, with respect to which entity they hold, and what limitations must be considered.

entity	property	restrictions
model	*Deadlock free*	
	Livelock possible	
	Livelock free, if no waitUntil operator used.	Exception handling for permittedIf operator.
	Only the impliesLater and the waitUntil operators can cause *asynchronous message loops*	Helper messages and self calls that trigger new rules.
implies & impliesBefore operators	Actuality and ordering property.	Dynamic connector activities.
permittedIf & waitUntil operators	Actuality and special ordering property: guard evaluated to true.	Dynamic connector activities.
impliesLater operator	No actuality or ordering property.	

Part III

Implementation

Chapter 10

Implementation Overview

The FLO/C model as presented in part I and formally analyzed in part II is fully implemented using SMALLTALK. This chapter gives an overview of the implementation. First, in section 10.1 we show how the implementation is layered to give an impression of the global structure of the implementation and in section 10.2 we present the size of different parts of the implementation. Then section 10.3 evaluates the choice of the base languages SMALLTALK and NEOCLASSTALK. Section 10.4 will explain the meta-class and meta-object concepts and point out their use for the FLO/C implementation.

10.1 Layering of the Implementation

During the development of FLO/C in SMALLTALK we had to implement different services ranging from low-level issues like an asynchronous message passing protocol to high-level issues like a visual programming tool for FLO/C. Figure 10.1 shows how the implementation is layered. Each layer uses the services provided by the layer below.

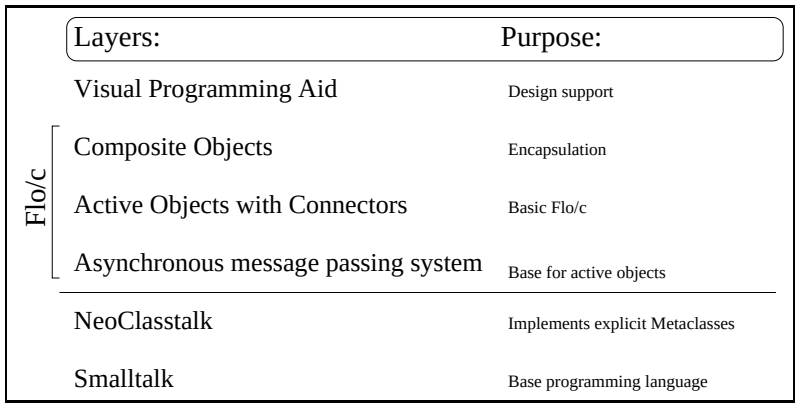

Figure 10.1: The layering of the FLO/C implementation.

On top of SMALLTALK we used NEOCLASSTALK, which is a reflective extension that allows the declaration of explicit meta-classes. NEOCLASSTALK eases the control of the message passing which enabled us to we implement active objects similar to Briot's ACTALK [Bri89] but put them on top of a more elaborate asynchronous message passing system. We did this to study different interception policies for message passing. The basic FLO/C implementation defines the classes and meta-classes used to program in FLO/C. Composite objects are an extension, FLO/C can be programmed without

using them. The visual FLO/C tool is a VISUALWORKS 2.0 application that allows one to graphically connect active objects and to send messages to them.

The basic FLO/C implementation and the higher support features deserve their own discussions in separate chapters. The next section presents the technical numbers of the implementation.

10.2 Technical Data of the FLO/C Implementation

We classify the implementation source code into four categories: kernel-, helper-, visualization- and example code. *Kernel* code includes all classes that represent FLO/C abstractions, their meta-classes and their meta-object classes, for example Connector, MetaCompositeObject and Controller. *Helper* code includes the classes that are used to decompose functionality - for example group-management and data containers. Classes for this category are for example Call (reified message), RndSpecificator and InteractionRule. The *visualization* code includes the VISUALWORKS 2.0 application to support visual FLO/C programming. The *example* code holds the examples described in section 14. It contains the FLO/C programs written to solve particular coordination problems in order to prove FLO/C's expressive power.

category	code	classes	methods	code/class	methods/class
kernel	64 KB	13	217	4.9 KB	16.7
helper	39 KB	22	155	1.8 KB	7.1
visual	70 KB	9	167	7.8 KB	18.6
examples	83 KB	70	226	1.2 KB	3.2
total	256KB	114	765	2.3 KB	6.7

For SMALLTALK style programming [SKT96] [Bec97] some classes carry too many methods. The FLO/C implementation packs a lot of functionality into the few kernel classes. The classes of the visualization code are also overloaded. This problem is not FLO/C related but identified in [How95]. The high number of methods in the visualisation classes reflects the many user-interaction ways. The size of the code is also due to graphics (icons and layout). Note that only the kernel uses explicit meta-classes, namely two. The examples use 31 connectors and 11 composite object classes. More statistics for these classes will be presented in section 14.11. After this quick overview of the implementation, the next section justifies the choice of the base languages.

10.3 Choice for an Open and Reflective Environment

This section justifies our choice of the base language SMALLTALK [GR83], VISUALWORKS 2.0 [Par95] to be more precise. It explains why the extension NEOCLASSTALK [Riv97] was needed.

10.3.1 SMALLTALK

SMALLTALK was chosen as base language for the FLO/C implementation for the following reasons:

- The language is reflective [Riv96b]. Big parts of its system behavior is coded in SMALLTALK and available in the repository for inspection and changes, therefore SMALLTALK can be adapted to the needs of FLO/C.
- The language supports rapid prototyping, because it allows short life cycles. This is ideal to prototype new models like FLO/C that can change quickly. Furthermore, the debugging features are excellent (e.g. recompiling flawed methods at debugging time).

- SMALLTALK truly decouples message passing from method execution. Therefore, it is easier to control message passing.

VISUALWORKS 2.0. For our implementation we used VISUALWORKS 2.0 [Par95] as SMALLTALK environment. VISUALWORKS 2.0 is a fully object-oriented environment for constructing applications, using SMALLTALK as the scripting language. It enables application developers to build graphical user interfaces rapidly. Furthermore, VISUALWORKS 2.0 provides convenient linkages to many popular databases. The implementation of a visual programming aid for FLO/C took the most advantage of VISUALWORKS 2.0 by using its application framework based on the model-view-controller paradigm [How95].

Limitations of SMALLTALK. SMALLTALK offers only low-level concurrency constructs. Processes (threads) can be forked and they have a priority. Semaphores are provided to synchronize processes. Unfortunately SMALLTALK is not preemptive. If a process is active and running, it must yield the control explicitly. The concurrency supporting code and the scheduler code is available for browsing, however most of the code relies on built-in primitives, thus cannot be modified easily.

The fact that FLO/C is implemented in the VISUALWORKS 2.0 environment with little concurrency features and residing on a single processor machine, simplified the implementation. We will show in section 16.1, how the architecture could be extended to be used in real distributed systems.

10.3.2 NEOCLASSTALK

NEOCLASSTALK [Riv96a] is a SMALLTALK extension that is even more reflexive than SMALLTALK. NEOCLASSTALK is inspired by the OBJVLISP object model proposed by Cointe [Coi87][Coi90]. NEOCLASSTALK version 1.2 is available for free on the world-wide web under the URL `http://wfn.emn.fr/dept_info/neoclasstalk/`. The following list shows some of the important improvements of NEOCLASSTALK over SMALLTALK.

- In NEOCLASSTALK each object can dynamically change its class without taking care of the number of instance variables.
- In NEOCLASSTALK dynamically changing the class hierarchy is simplified.
- NEOCLASSTALK features multiple inheritance.
- In NEOCLASSTALK the message application is reified. This enables method execution control.
- NEOCLASSTALK does not distinguish between classes and meta-classes. It offers *explicit meta-classes*.

For the implementation of FLO/C we used explicit meta-classes in order to declare enhanced connector classes and provide component classes with a special instantiation protocol. We also used the message application reification in order to control th message passing.of our active objects. The rest of the features (e.g. multiple inheritance and dynamic class changes) were not used.

Note that NEOCLASSTALK does not improve the concurrency or coordination support of SMALLTALK.

The next section will explain why and how meta-classes are used, and what meta-objects are.

10.4 Meta-Level Programming

Meta-objects and meta-classes have almost only the word "meta"[1] in common. We treat both concepts in this section because they both go beyond "regular" programming. Therefore, these concepts must be introduced before we can explain the implementation of FLO/C's kernel in section 11.

10.4.1 Explicit Meta-Classes

In OBJVLISP everything is an object and instance of a class. Therefore, classes are objects too. They are instances of a classes, namely of meta-classes. Just like any class the meta-class declares operations and instance variables that their instances have in common, the only difference being that the instances are classes themselves [Coi87]. SMALLTALK's meta-classes are *implicit*. They are accessed via the class protocol of their instances and each implicit meta-class has only one instance. In NEOCLASSTALK meta-classes are declared like any other class. A standard meta-class understands the new message, since this message is sent to a class. In section 6.1 we saw that composite object classes have an unconventional instantiation protocol. This can be implemented by declaring a meta-class that instantiates all composite object classes (see section 10.4).

Connectors of one class all share the same rules. Therefore it is a waste of memory to store them in each instance. A meta-class of connector classes can be used to declare a rule-store for each connector class. In SMALLTALK this could have been done in a class-variable, using the *implicit* meta-classes of SMALLTALK.

10.4.2 Meta-Objects

Kiczales [KdRB91, Kic92] proposed the separation of base-language programs from *meta-language* programs. The meta-language program can customize particular aspects of the *underlying systems' implementation* so that it better meets the needs of the base-language program. A *meta-object* instantiates an aspect of the underlying system's implementation. In FLO/C meta-objects are used to instantiate aspects of the message passing system. The most important meta-object of FLO/C is the *controller* (see section 11.2). It instantiates the message handling policy of a single active object [Fer89]. Since FLO/C is based on message passing control (see section 2.2.3), an extension or customization of the FLO/C implementation can be achieved by changing the FLO/C controllers. Meta-objects for message passing control were successfully used in other approaches like for example Open C++ [CM93].

After this first contact with the "meta" concepts used in the FLO/C implementations, the next section will explain the architecture of the FLO/C kernel thereby showing the *use* of the concepts.

[1] *Meta* "beyond"

Chapter 11

The FLO/C Kernel

The FLO/C kernel includes the the two main entities of this thesis: connectors and active objects. The active objects use asynchronous message passing and they collaborate with connectors. The next section explains the implementation of these two aspects of the active object architecture. The *controller* as part of an active object acts as meta-object. It implements FLO/C's message control strategy, therefore controllers will be discussed in detail. Then we explain the implementation of connectors. Finally we introduce the architecture for the composite objects.

11.1 Implementation of Active Objects

Before implementing FLO/C, we experimented with different asynchronous message passing styles. We knew that we wanted to intercept the messages at different stages of their propagation (received, handled, sent). Therefore we changed the ACTALK model [Bri89] refining the asynchronous sending.

11.1.1 ACTALK's Asynchronous Message Passing System

In ACTALK, an active object consists of a *behavior object*, an *activity object* and a *queue*. The behavior object implements the domain specific behavior. It sends and receives messages. But all messages to the behavior object are redirected[1] into the queue of the active object (asynchronism). The activity object that is running in its own thread, accesses the queue. It decides what messages should be executed (what method of the behavior object should be called). Figure 11.1 shows an active object in ACTALK.

11.1.2 FLO/C's Asynchronous Message Passing System

As we mentioned in section 2.2.1 our implementation is similar to the ACTALK architecture but extends the asynchronous message passing. Furthermore the live-cycle of our active objects differs from ACTALK, for example our implementation features graceful termination of active objects.

In our implementation the *behavior* object's class must be subclass of ActiveObject. In order to be used as an active object it has to be encapsulated in a hull (class Hull) that connects it to the message *queue* (class InQueue[2]) and the *activity* which we call *controller* (class FloInController). Our active objects can use a *router* (class Router) to send asynchronous messages. The active object sends the

[1] Briot used message interception by overloading the Object»doesNotUnderstand: message.

[2] The queue is implemented using class SharedQueue provided by SMALLTALK.

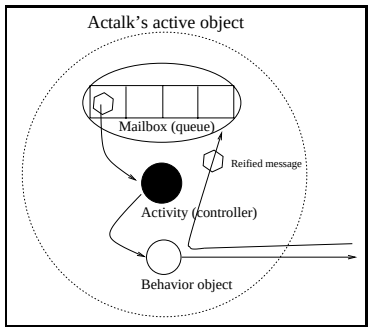

Figure 11.1: An active object in ACTALK.

message to the router which will deliver it to the appropriate queue. We introduced the router in order to have more control points where we can get hold of messages and to be able to gracefully terminate all active objects at once. The router turned out to be unnecessary for FLO/C, but it is still useful since it can be extended to simulate real distribution. It can do so by delaying asynchronous messages or even loosing some. It can also simulate name-space problems.

Until this point our implementation does not need message capturing, because the asynchronous messages are sent by methods of the class ActiveObject that use the router. Therefore our asynchronous message passing system is stand-alone. In order to render the system compatible with passive objects, we added message capturing (see section 11.1.3). Any message sent to the behavior object will be intercepted and put in the in-queue. Together with a FLO/C compatible controller our active object implementation is fit to model FLO/C's active objects.

We show now step-by-step how asynchronous messages flow through our active object implementation. Figure 11.2 illustrates these steps and numbers them. The following enumeration follows the numbers in the figure.

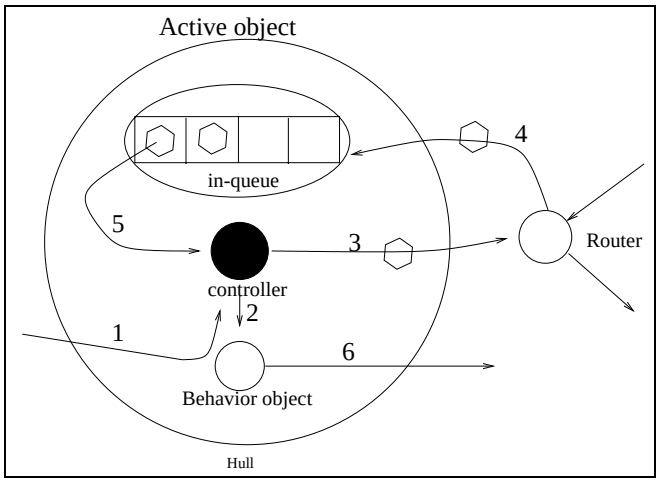

Figure 11.2: The lean active Object.

1. All messages that are sent to the behavior object are intercepted and redirected to its meta-object. Message capturing will be the subject of section 11.1.3.

2. A simple version of the controller can just execute the message on the object, and return the

result of the evaluation. This is equal to synchronous message passing.

3. But the controller can also use asynchronous sending. It uses the asynchronous sending features of the active object. If the controller uses this way to re-send *every call* coming in at 1), the active object shows heavy asynchronous behavior. It is not able to provide return values.

4. The router will lookup the target of a reified message, and put the message in the appropriate queue.

5. The controller pulls the messages out of the queue one by one. If it is a simple controller, it will just execute the message on the object.

6. The object sends synchronous messages to other objects. If the target object is an active object, point 1) applies there.

Note that here are two ways to send a message to an active object. One is to directly send it to the inner object (which will lead to message capturing). The other way is to put a reified message into the queue (ev. by the aid of a router). Note that all messages to active objects pass the controller, before being executed. Therefore, the controller is the entity, where FLO/C can put "the foot in the door". But first we discuss how the messages intended for the behavior object are redirected to the controller.

11.1.3 Message Interception

In order to intercept messages sent to objects, NEOCLASSTALK allows the dynamic recompilation of classes. The implementation we use recompiles a given class so that when a message is looked up in this class, the message is redirected to the meta-object by calling its method:
Controller»control: receiver method: compiledMethod withArgs: args
The argument receiver holds a reference to controlled object, the compiledMethod holds the method that was called. Per default each active object has a controller that simply performs the compiled method:

```
control:receiver  method: compiledMethod withArgs: args
   ^compiledMethod valueWithReceiver: receiver arguments: args.
```

The method valueWithReceiver: executes the compiled method without a method lookup, therefore it is not controlled again. The controller simply executes the controlled message. Therefore whether a not yet activated active object is controlled or not is transparent.

It is the responsibility of a connector to activate the active object (see 11.3.2). When establishing a new connection the connector checks if the new participant is already active. If not it checks if it is already controlled. It recompiles its class and replaces the default controller by a FLO/C controller. This new controller does not simply execute the compiled method but it can also negotiate with other controllers and use the asynchronous message passing system (see section 11.2).

The message passing control we use is of low granularity, since *each* message sent to a controlled object will go to the meta-object. Worse, every object of the same class will be controlled. Note that NEOCLASSTALK would also enable the implementation of other message capturing techniques that allow interception on a per-message and even per-instance basis.

By introducing a dummy controller, the controlling of the unwanted objects is made transparent. Furthermore the moment of establishing control is separated from the active object activation and both are reversible. A clear advantage of the message capturing we use is that the control is nicely

integrated in the NEOCLASSTALK environment. The browsers indicate that a class is controlled, but the source code is presented as if it was an uncontrolled classes.

Problems. Beside of performance overhead we encountered two problems using this message passing control in the FLO/C implementation:

- As we will see in section 11.2 we sometimes need a way to send messages to controlled objects that should *not* be intercepted. To do so our implementation provides a back-door for the controller. It can directly look up the compiled method of the class.
- When changing or file-in/out a controlled class it gets compiled again, therefore it is not controlled anymore. Usually a connector will recompile it when needed. However, if a *superclass* was changed, the superclass must also be controlled. Therefore the connector must browse the inheritance hierarchy of a participant when connecting it. Thereby it must take care not to control classes of the FLO/C kernel. Otherwise this can lead to endless loops. For example if the Connector class was controlled a message to the connector would be redirected to a controller. Per default (as seen in the next section) the controller will send a request to connectors to see if there are rules for this message. But these messages will be redirected to the controller again and so forth.

Note that message passing control can also be implemented in pure SMALLTALK. The different techniques are described and classified in [Duc97a]. We will now proceed to the meta-object that controls the message passing: the controller.

11.2 Controller

The FLO/C controller represents the activity object of the ACTALK model. The activity object loops in its own thread, trying to pull messages out of the queue, blocking when the queue is empty. FLO/C enhances the activity object and calls it controller. The controller works together with the connectors that connect the active object. Therefore it has to keep references to them. The controller can get messages in two ways. Either it pulls one out of the queue, or it receives one that was directly sent to the active object and intercepted. In the later case, the controller checks with the connectors if there are rules triggering on this message. *If not, the message will be executed right away and its return value is returned.* Such an execution represents synchronous interaction with helper objects as mentioned in section 3.1. It represents the inner computation of the conceptual active object. Such computation is done in a sequential way, because from the outside it is treated as one atomic operation. If the intercepted message triggers rules it is put in the queue. That way the controller can assume that all relevant messages (non-helper messages) come from the queue.

If a message is pulled from the queue it is considered to be an interaction that involves active object interaction. Whenever the controller treats a message from the queue it asks all of the connectors that connect the active object whether they have consequences for this message. The controller *fusions the consequences* like described in section 4.3. The formal specification of part II of this work has already described most of the tasks of the controller. We will subsequently refer to this part for more details. The controller implements the system transitions presented in the formal part in section 8.3. The controller's tasks are:

- **Collection of consequences.** The controller must collect consequences by asking the connectors. This includes a recursive collection of the *sequential consequences* and the cycle breaking

mechanism. The implementation follows the specification of the function $f_{rules}(i, s)$ seen in section 8.3. Note however that the controller must communicate with other controllers to do so, because it does not know all connectors involved in the consequences. The controller does not keep references to all other controllers. Instead it knows the target objects of the consequence messages and can thus find their meta-objects (controllers).

- **Reservation.** To reserve the appropriate objects for execution, the controller also contacts the other controllers to negotiate reservations. This can be done with elaborate algorithms as proposed in section 8.3.2. However, we simplified our FLO/C implementation to achieve a small and elegant solution. On the down side our solution does not exhaust the full amount of concurrency possible in the FLO/C model. Every time a controller handles a message it does so without yielding the processor (see SMALLTALK limitations section 10.3.1). This is equivalent to the reservation of *all* active objects, thus it constrains concurrency in an unnecessarily severe manner. When the controller has finished the consequences, it yields the processor. Since all controllers run at the same priority, and SMALLTALK uses first-in first-out queues to store runnable processes that wait for the processor, *each controller gets a chance to handle a message*. Therefore, our implementation fulfills the requirements for the state transitions postulated in section 8.3.2. Thus the liveness properties guaranteed in section 9 hold for the implementation: e.g. no deadlock and no livelock without the waitUntil operator. The downside of the simplified implementation is reduced concurrency, which renders some of the examples too "well-behaved". As we will state in section 14 we added some processor yields to some examples in order to increase concurrency and produce visible conflicts in the examples. However, this embellishment is not due to a limitation of the model but a limitation of the implementation.

- **Execution of consequences.** The sequential consequences must be executed. The controller must execute the guards and sequential consequences appropriately. A problem here is that the executions should not be controlled again, since in the collecting phase all consequences were already recursively collected. Therefore the controller needs a way to execute methods on other objects without interception and redirection to other controllers. Furthermore, when sequential consequences are executed, the controller has to take care of the return values. As seen in section 4.5 the keyword return can indicate that a consequence argument must be valued by a previous return value.

- **Asynchronous sending of consequences.** To asynchronously send consequences or a delayed message, the controller can rely on the asynchronous message passing features of the active object it belongs to.

This list shows that the controller is the hidden "heart" of the FLO/C implementation that has to provide the role fusion which relies on communication and negotiation with other controllers and with connectors. The next section shows what responsibilities are left to the connectors and how they are implemented.

11.3 Connector

We saw that connectors have roles and interaction rules. Both are specified when declaring a connector class. Note that in our implementation only the rules are declared explicitly. The roles are parsed from the rule definition thus they are declared implicitly. In order for all connector classes to understand FLO/C specific declarations, we introduce a meta-class named MetaConnector. By

sending the message MetaConnector»withBehavior: string to this class, a new connector class can be instantiated with the behavior declared in the argument string. The string contains the rule declarations.

Every connector class inherits from an abstract class Connector in order to share behavior like the initialization protocol. Since every connector is also an active object, Connector inherits from the class ActiveObject. Instances of connector classes can then be individually connected to coordinate active objects. Figure 11.3 shows the inheritance/instantiation graph of the FLO/C connector architecture.

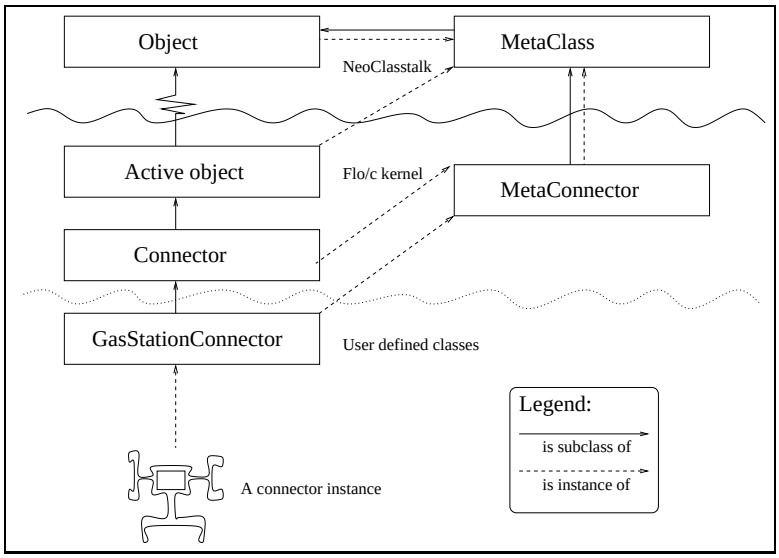

Figure 11.3: The architecture of the connector implementation.

To clarify the design, the next two subsections show the responsibilities of the connector class and meta-class.

11.3.1 Responsibility of Class MetaConnector

The meta-class MetaConnector declares what all connector classes have in common:

- MetaConnector declares instance variables that can hold representations of *rules* and *roles*.
- Instantiation methods providing the rule- and role declarations are implemented in MetaConnector. Thus a little parser for the rule syntax defined in section 4 is implemented there. In fact the method MetaConnector»withBehavior: string leads to the parsing of string which extracts the rule and role definitions of a connector class.
- Since the rules are stored in the connector class, the MetaConnector implements the *rule lookup* and some general role management features.

11.3.2 Responsibility of Class Connector

This abstract superclass of all connector classes implements the commonalities of connector instances.

- Every connector has a table with its *actual participants* and a mapping to the current role they play.

- The class Connector provides operations to *add and remove* new participants, which includes activating them if necessary and activate the message passing control as seen in section 11.1.3. It also includes compatibility checks: A participant is only connected as player of a role, if it can understand all possible messages that this role uses in the rules. The methods to connect a single active object or a role group group are Connector»object:playsRole: and Connector»objects:playRole:. The method to disconnect an object from a connector is Connector»releaseActiveObject:.

- The class Connector provides operations to *initialize* a connector, for example mapping itself to the default role connector (see section 4.5). Upon activation of the connector, there is also default code to initialize the participants (Connector»startingState). When a connector starts the coordination of participants it may want to force the participants to an initial state. A user-defined connector can do this by overloading the startingState method.

- The class Connector features also code for the graceful *termination* of connectors, either terminating (method Connector»end) the participants along with itself or letting them alive (method Connector»decouple).

- The Connector class includes the entry point for the role lookup that is used by controllers. It uses *symbol replacement* mechanisms in order to exchange participants by roles when calling the rule-lookup method of the meta-class. Then it must replace roles by participants again, and also exchange the formal arguments with the actual arguments before it can send back the consequences to the controller. The replacement of arguments and roles was already sketched in section 4.1.

11.4 Collaboration of Controllers and Connectors

The FLO/C system is managed by the collaboration between controller themselves and between controllers and connectors. As we will see in chapter 13 the controllers have a cache so that they do not have to contact the connectors every time. We will now list the interactions between the FLO/C controlling entities, which in turn will establish coordination between the active objects. As illustrated in figure 11.4 we will list these interactions step by step from the point of view of an active object o1.

1. As already described in section 11.2 the controller of o1 pulls a message out of its queue.

2. It asks its cache if the consequences of that message are already known.

3. If the consequences are not known, all connectors are contacted, passing the message from the queue as argument. The connectors check if the object o1 plays a role. If so they look up if rules should trigger. If so they translate the symbolic consequences message (coded with roles) to consequence messages for active objects and deliver these message lists. The controller will then recursively traverse the lists and contact the connectors again in order to find *all* indirect sequential consequence messages. Because of the cycle breaking described in section 8.4 this collection phase terminates.

4. The controller of o1 contacts all controllers of active objects that are targets of *sequential* consequence messages or *guard* messages in order to negotiate reservation. In our implementation this does not happen explicitly (see section 11.2). Instead the controller does not yield the processor to another thread.

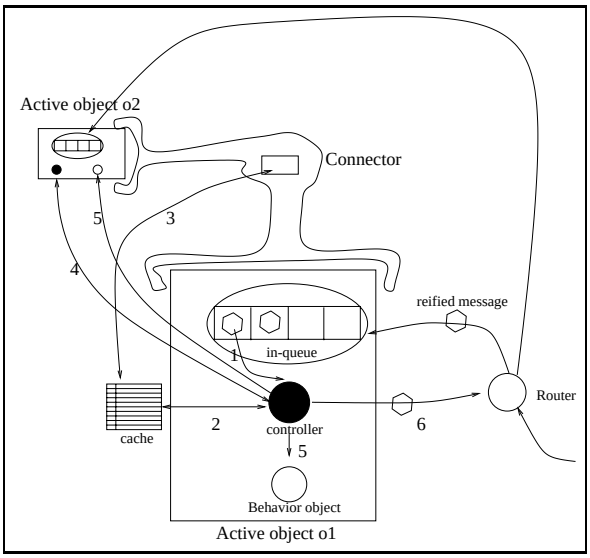

Figure 11.4: An overview of the interaction between the control entities.

5. The controller executes guards and - if they succeed - it executes the sequential consequences on the objects involved. Since these are all consequences of a message in the local queue, at least one sequential consequence is the message itself.

6. The asynchronous consequence messages are sent.

The next section treats the next higher layer of FLO/C. It shows the meta-class design to implement composite objects which is similar to the one used to implement the connectors.

11.5 Composite Objects

To support composite objects, FLO/C includes the meta-class MetaCompositeObject, that is responsible for the creation of new composite object classes as introduced in section 6. All composite objects inherit their common behavior from the class CompositeObject, which is a subclass of ActiveObject, since composite objects are also active objects. See figure 11.5 for the inheritance/instantiation graph of the composite object framework.

11.5.1 Responsibilities of the Composite Object Class and Meta-Class

Again the meta-class implements instantiation methods. They allow the declaration of the additional static properties needed in a composite object class, like the participating component- and connector classes and the connecting schema (see section 6.1). These static properties are stored in the composite object class. When a composite object is instantiated, methods of the meta-class take care to properly instantiate objects of the components- and connector classes and to connect them according to the schema. Furthermore the inheritance mechanism (see section 6.2) for composite objects is implemented in the meta-class. Component- and connector classes are inherited or extended. Illegal subclasses (for example subclasses that have less components) cannot be created.

The abstract superclass CompositeObject has only little responsibilities. Again it takes care of the *initialization* and *destruction* of composite objects. These operations should not only treat the in-

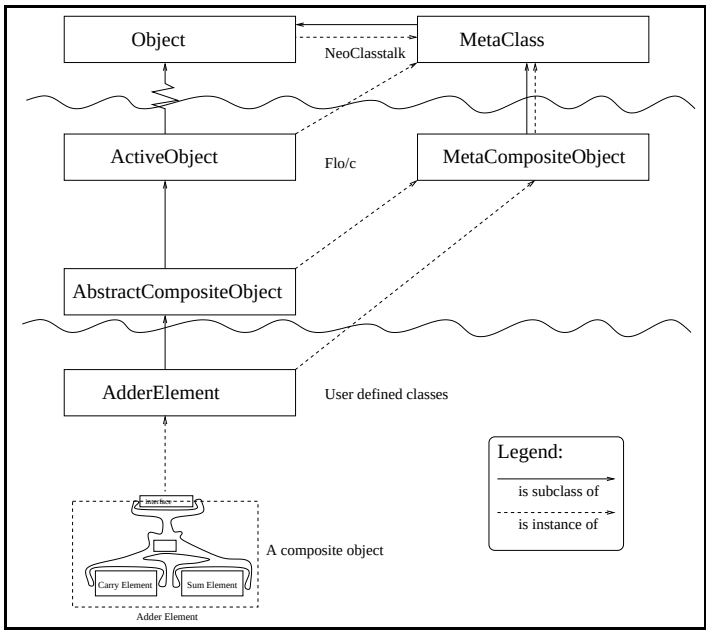

Figure 11.5: Inheritance/instantiation graph for the composite object framework.

terface but also the components and connectors of the composite object. Furthermore the default role interface must be properly mapped to the interface object. For example, the first participating connector is considered the interface connector. During the initialization the interface connector will be connected to all participants. Thus the composite objects can be gracefully terminated by terminating this connector.

The next section will discuss the highest layer of the FLO/C implementation which contains tools that support the visual connection and composition of active objects.

Chapter 12

Visual Programming Tools

The FLO/C implementation separates the instantiation, attachment and activation of a connector providing flexibility to the FLO/C programmer. However, especially the attachment of active objects to connectors is tedious and error-prone. As already seen in the instantiation script for the gas station example (section 5.3) each connection needs a statement:
myConnector object: participant1 playsRole: aRole.

Furthermore when testing active objects, the dynamics of FLO/C cannot fully be exploited. A test-script sending messages to the object does not reflect the concurrency of the active objects. It is complicate to test the group management of connectors that allows attachment and detachment of active objects on role groups on the fly.

To address these problems we implemented two visual programming tools. The FLO/C *workspace* enables to visually compose connectors and components and send messages to them asynchronously. The *composite object class browser* allows one to visually create a composite object class.

12.1 The FLO/C Workspace

This tool[1] solves the following problems:

- Inability to exploit concurrency of the active objects and the asynchronous message passing systems when using and testing connected active objects.
- Inability to exploit the dynamics of connectors' live time and the group management.
- Complexity of textual representation when using a statement per connection.

The tool is not intended to design active objects and connectors since we believe that this is easier to do textually by declaring classes. Therefore the meta-class MetaConnector we presented in section 11.3 is designed to support connector class declaration in the standard browsing environment. Rather the FLO/C workspace is intended to visually *connect* and *run* active objects. Therefore the user of the tool can instantiate active objects and connectors. (S)he can connect the active objects to connectors choosing a role for them. The tool will check if the active objects fit[2] the roles. The user can activate the connectors and asynchronously send messages to the active objects, by typing messages in an entry field of the graphical user interface (GUI). (S)he can kill the running objects or connectors or add new

[1] In the implementation it is called "example editor".
[2] =Provide the proper interfaces.

ones on the fly. Therefore the dynamics of the asynchronous messages, the connector lifetime and the group management are exploited. The user can arrange the visual representations of the objects and connectors and examine the connection schema. We believe that visual connections are far easier to understand and manage than the textual representation is. The structure of a screen setting can be saved in order to use it to visually declare a composite object (see section 12.2) and other settings can be loaded. Note that the inner state of the active objects is not saved, only their activation state, their connections and positions on the screen.

Figure 12.1 shows a screen-shot of the FLO/C workspace displaying a configuration of the gas station example introduced in section 5. On the top-left of the graphical user interface there are two lists containing all active object classes and connector classes. By clicking on an entry, a new instance is created in the display field at the bottom. Active objects are graphically represented as circles that are labeled by their class name. Connectors are represented by a neuron-like shape. Connections can be drawn between components and connectors. They are labeled by the role name. Connectors and components can be selected by clicking on their representation. The representation will show the selection by inverting a rectangular area around the representation. The selected representations can be arranged using the mouse. The activation state of connectors and active objects is color coded: black means passive, red active. The color blue indicates that a connector is ready for activation (has at least one participant per role).

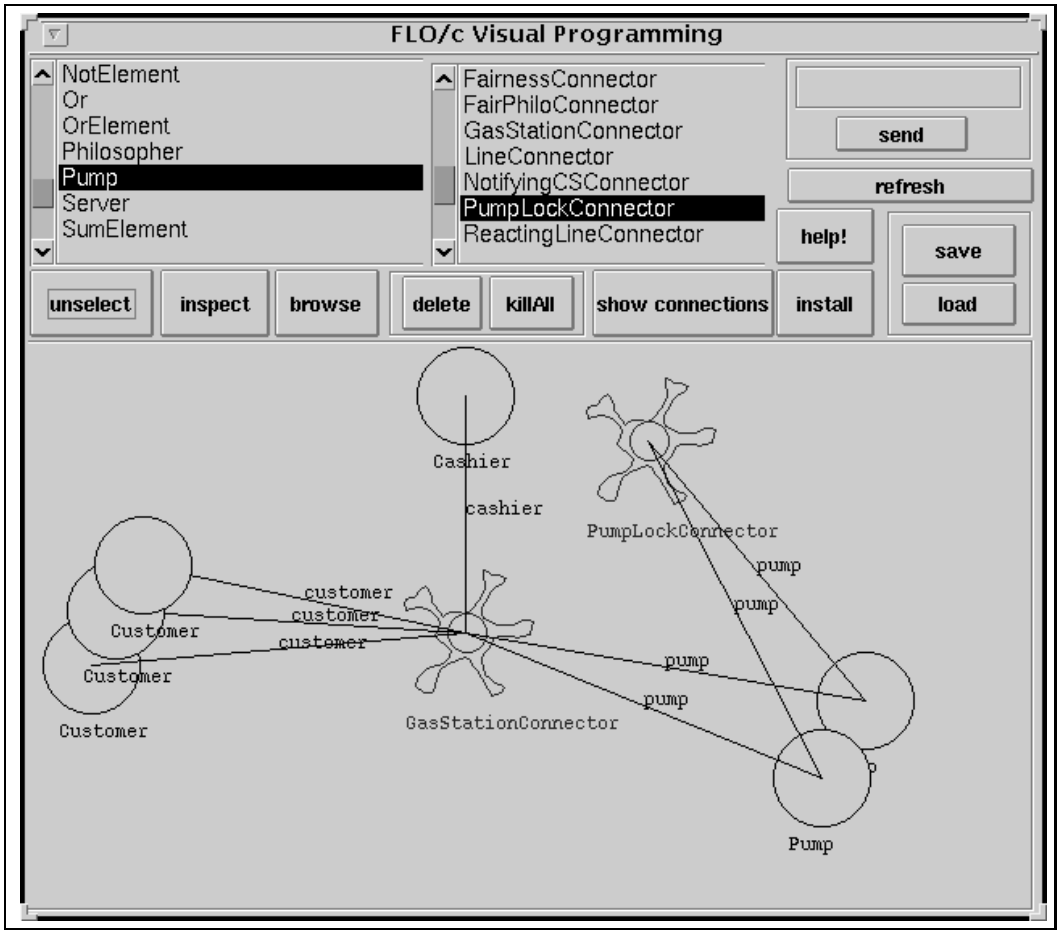

Figure 12.1: A screenshot the FLO/C workspace.

The main features of the tool are best explained following the buttons of the GUI from left to right.

- Unselect. When the user selected an active object but does not want any selection (s)he can press this button.
- Inspect. To inspect the inner state of the selected active object or connector.
- Browse. To browse the class of the selected item in a system browser. This allows for example to read the rules of a connector.
- Delete. The selected object is gracefully terminated and deleted from the screen.
- KillAll. All displayed objects are terminated and deleted from screen.
- ShowConnections. The connections of the selected connector can be inspected.
- Install. The selected and connected connector is activated.
- Load/Save. The current example is stored on- or loaded from a class of the users choice. The description of the setting is therefore compiled into a class method of the chosen class. This way of storing is also used by VISUALWORKS 2.0 to store window layouts.
- Refresh. Updates the display.
- Send. This button asynchronously sends the contents of the entry field (above the button), to the selected active object. With this feature a user can run and test a setting of components and connectors, exploiting the concurrency of the active objects. The entry field evaluates the entry string. Therefore it understands literals like numbers, booleans, strings and arrays (e.g. #(1 2)) as arguments (e.g. setName: 'Manuel').

12.2 The Composite Object Class Browser

The composite object class browser can inherit half of the code[3] used for the FLO/C workspace. This is possible because the purposes are similar. The tool allows to declare new composite object classes thereby facilitating the following problems:

- Composite object classes involve other class definitions, but ordinary browsers display only one class a time. Therefore composite object classes are hard to browse.
- The connecting schema of a composite object is hard to code and understand in textual form (compared to the code presented in section 6.1).
- When inheriting from a composite object class it is difficult to see what participant classes must be overloaded and what connections must be declared. Compare to the code example in section 6.2.
- It is difficult to create "by hand" a composite object class out of a code script that uses or tests the involved components and connectors (such a code script for the gas station example is presented in section 5.3).

The composite object class browser solves these problems. When starting it, the user has to chose a superclass[4] and the name of the new composite object class (s)he wants to create. The composition

[3] Not including the code from the VISUALWORKS 2.0 GUI framework.

[4] Every composite object class must have the base class CompositeObject.

graph of the superclass is then displayed on the GUI. When a user only wants to inspect a composite object class, (s)he enters it as superclass and does not compile the result. If the user wants to create a new class (s)he can use the GUI to add new participants and connector classes or extend inherited ones, then compile the result. Furthermore examples that were tested by the FLO/C workspace tool can be imported and thus encapsulated into a composite object class.

Figure 12.2 shows a snapshot of the GUI at the moment a a user creates a new class InheritedAdderEl which inherits from CarryElement. The user wants to add new functionality (sum-bit calculation) to the inherited functionality (carry-bit calculation). Thus (s)he wants to do the same as we did in the example of section 6.2. The moment the snapshot is take (s)he already added two XorElement component classes and is about to connect them to the new ConnectionOfSum connector.

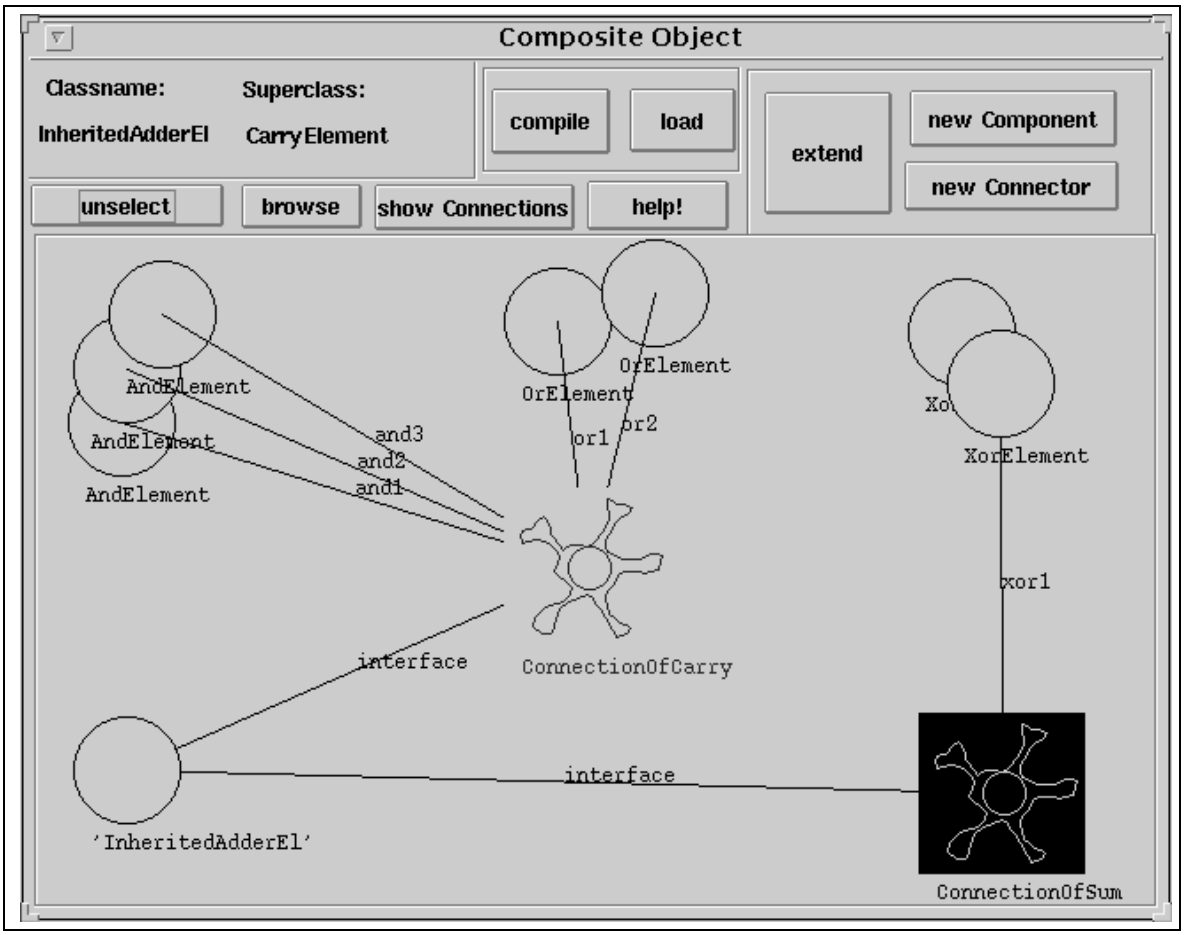

Figure 12.2: A screenshot of the composite object class browser.

The main features of the composite object class browser are best explained following the buttons of the GUI from right to left.

- New Component and New Connector. These buttons allow the user to select a new component- or connector class.
- Extend. The selected component or connector class is replaced by another one of the user's choice. The tool will ensure, that extensions are able to support the connections they are already engaged in. Connectors that replace another connector must know all the necessary roles.

Replacing components must understand all the messages they can receive when they play roles. The tool checks this by analyzing the rules of the connectors.

- Load. If the user has tested and saved a successful arrangement of active objects and connectors, using the FLO/C workspace, (s)he can load it with this button, using it to compose the new composite object.

- Compile. This button starts the compilation of the new composite object class. The interface methods are *generated automatically*. The tool does this by analyzing all connectors that use the default interface role. When a rule contains a message containing this role, the selector is used to compile a dummy method on the new class that serves as interface to the composite object as seen in section 6.

- The other buttons do the same as in the visual example tool. They can be used for gaining introspection of the connectors and components involved.

Evaluation of the tools. The two tools' main goal is to visualize the parts of FLO/C programming that are not well supported by the VISUALWORKS 2.0 environment: The multi-object *connections*. All the programming examples that we will see in chapter 14 were tested (or developed in case of composite objects) with these tools. Even users who have no idea of the FLO/C model can play with the examples that are stored with the FLO/C workspace.

The application framework of VISUALWORKS 2.0 enabled the rapid development of the tools. However, the tools do not cover all possible visual support for the FLO/C implementation. The FLO/C workspace does not monitor the inner behavior of the components, it does not save its states and it cannot monitor the message passing. The composite object class browser is for example not capable to open up nested composition hierarchies and it cannot display the inheritance hierarchy. Such extensions are possible but are omitted because of the time limitations of this thesis.

Before we start to discuss the implementation of some examples of coordination problems, we want to focus on low-level issues of the FLO/C implementation such as its performance optimization.

Chapter 13

Performance Optimizations

There are still a lot of untreated subjects in the complete FLO/C implementation. Examples of omitted implementation issues are how the specificators work (which also involves the implementation of a class Role) or how the VISUALWORKS 2.0 application framework was used to implement the visual tools, or the termination protocol of active objects. Because of space limitations we only discuss the different optimization made in the FLO/C implementation, since this consumed a considerable part of the implementation effort and involves other topics of the implementation (such as dynamic connections).

The message interaction, the consequence lookup, the asynchronous message passing system and last but not least the scheduling of threads adds performance overhead to a FLO/C program. The next section presents where we had to take measures.

13.1 Problems with the First Implementation

In the first straight forward implementation (not featuring group management or connector programming) even simple examples used execution times in the range of several seconds. Later it turned out that a lot of the overhead was not caused by the FLO/C implementation but by a flaw of the garbage collector of NEOCLASSTALK [1]. Still, the first FLO/C implementation had to be optimized in many ways.

FLO/C's activities can be divided in four categories. These categories reflect the communication flow inside and between active objects (see section 11.1.2 and following).

- The message interception.
- The consequence lookup.
- The negotiation for reservation of objects.
- The asynchronous message passing.

The message interception is not optimized since little time is lost there. In section 11.2 we already explained that our FLO/C implementation uses a minimal solution for reservation of active objects, so there is no time lost there either. The asynchronous message passing was optimized by omitting an additional out-queue and its controller, which halved the number of threads. Note that the asynchronous message passing system could be further optimized, leaving away the routers. However, we

[1] We informed the developer of NEOCLASSTALK of the bug, which is now fixed in the newest NEOCLASSTALK release.

want to keep the routers since they are useful to simulate distribution (see section 11.1.2). This is also the reason why we don't care if the asynchronous message passing system is slow.

These considerations lead to the conclusion that the *consequence lookup* must be optimized. For each message taken out of the queue, the controller has to ask each connector attached to that active object, if rules are triggered and which. Then it has to fusion the consequences and send them and react appropriately (see section 11.2). Time profiling revealed that our first implementation did indeed waste most of its execution time in the consequence lookup.

The next two sections treat our two optimization approaches. First we tuned the lookup itself. Then we cached the consequences in the controllers, in order to avoid a full lookup. This provided a major speed-up but lead to some nasty problems.

13.2 Optimizing the Rule Lookup

The controller recursively collects the consequences by repeatedly asking connectors for consequences of a particular message. The controller has to treat consequence messages according to the operator of the triggering rule. In the old implementation the controller requested the consequences for each operator separately. This had the advantage that the controller didn't have to sort the consequences by operators. On the other hand the lookup methods had to scan the rules for as many times as there are operators. Therefore, we implemented a new method: Connector»giveConsequencesOf: aCall. This method takes a reified message as argument (instance of the helper class Call) and returns a list containing instances of class RightSideOfRule, which carry the consequences of a single rule *and* the operator. Thus the controller receives a list of such RightSideOfRule instances. On one hand it has to sort this list by operators. On the other hand a connector is only asked once to scan its rule base for a given message instead of once per operator. Since the rule base is in general much larger than the consequence list of a single message, this optimization achieved a speedup by almost factor 5.

In the first implementation an InteractionRule held an instance of Call as precondition message, an operator and a list of Call instances as consequences. The rules were stored in the instance variable interactionRules declared in MetaConnector, which held an instance of the SMALLTALK container class OrderedCollection. The rule lookup was implemented using an overloaded equality operator of the Call. Equality was determined by comparing the target objects and the message selectors If both are equal then the calls are equal and thus a precondition is matched. This lead to well structured consequence lookup code:

```
triggeringRules := interactionRules collect: [:rule | rule precondition = message].
```

However, in this code, the whole rule base is scanned to find rules that trigger. To optimize this, we use nested instances of the SMALLTALK class Dictionary, which is already optimized for lookup. The instance variable ruleDictionary which is declared in MetaConnector[2] holds roles as keys and again dictionaries as values. These dictionaries hold selectors as keys and *a list of instances of* RightSideOfRule as values. The structure of ruleDictionary is therefore:
$(\text{role} \rightarrow (\text{selector} \rightarrow (\text{operator}, \text{consequence message}*)*))$.
Thus the lookup is significantly improved even if the dictionaries were not designed for fast retrieval of the values. If the preconditions in a rule-base contain r roles, and if for each role there is an average of $s_{average}$ selectors that trigger a rule, then the old lookup has to check equality for $n = r * s_{average}$ times. The new lookup uses an average of $m = r + s_{average}$ checks. Note that the use of selectors instead of roles as the first key is also an optimization possibility. However, the introduction of role

[2]The rules are stored in the connector's class, referring to participants trough roles.

groups decreases the ratio: average roles per selector ($o_{average}$) to almost 1. This is because if two objects play different roles, their selectors are probably named differently. With the ratio $o_{average} \approx 1$, the nested lookup with selectors as first keys does not pay off since then $s * o_{average} \approx s + o_{average}$. The nested lookup with roles as first keys speeds the implementation up with the factor $s_{average}$ which is near 2 (see section 14.11 for the average number of different selectors per role in the preconditions of all FLO/C examples).

The draw-back of the optimization is that the structural integrity of the old implementation is lost. The new consequence loading code uses a nested dictionary access, and returns an empty list as soon as it is sure that no consequences can be found in the dictionary. Note that these code examples of the rule lookup are shortened for the readers convenience. The new consequence lookup code is:

```
MetaConnector>>listOfRightSidesOfRulesForMessage: aCall
 ^(ruleDictionary at: aCall object ifAbsent: [^OrderedCollection new])
    at: aCall selector ifAbsent: [OrderedCollection new]
```

The two nested at:ifAbsent: calls use the knowledge of the structure stored in ruleDictionary but this is bad style of object oriented programming. It violates the law of demeter [LH89]. The code examples document how our optimization sacrificed clean structure for performance improvement. The problem of not being able to have both is addressed in [Kic97] where Kiczales et al. propose aspect oriented programming for this purpose. This is nevertheless not directly related to our work.

Evaluation. The single-pass lookup optimization brought a constant speedup factor of approximately 5. The restructuring of the rule store speeded even small examples up by factor 2. As seen before, it consists of the advantage of dictionaries over other collection types and a factor $s_{average}$ which is the average of different selectors per role in a rule base.

While these optimizations concerned only the connector classes, the caching of consequence messages will mainly impact the controller's implementation.

13.3 Caching Consequence Messages

We will first explain how the cache works, then present three particular implementation problems and finally we evaluate the overall performance improvement of the optimization.

13.3.1 Working Principle

When a controller pulls a message from the in-queue, it has to communicate with all connectors that are attached to the controlled object. This involves a recursive process that orders the sequential consequences. Obviously this is a time consuming part and the next time the same message is pulled from the queue, it has to be done again. However, when the connections have not changed this is unnecessary work, because the consequences will be the same. Therefore, we implemented a class Cache[3]. It contains a dictionary with message selectors as keys and instances of the class ExecutionLists as values. These instances contain lists that hold the consequence messages sorted by operators, ready to be handled in a single pass. Note that we only need the selector as key (and not also the object) because the message comes from the queue of the particular object that the controller controls. Figure 13.1 shows the dictionary of a cache.

[3] In [KdRB91] this mechanism is called "memorization".

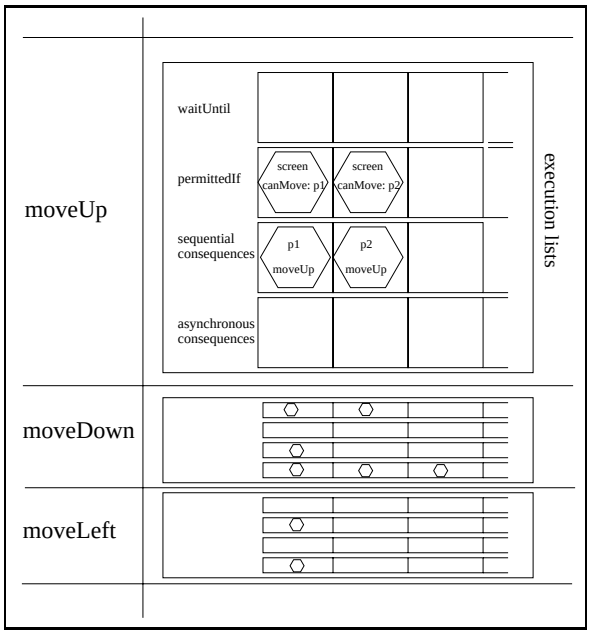

Figure 13.1: The cache containing different execution lists.

Each time a controller pulls a message from the queue it calls the following method on its cache, asking it if this message triggers any rules at all:

```
Cache>>checkAndUpdateFor: call
 "This method returns whether the call triggers rules or not. It also updates the cache"
 | res execLists|
 res := dictionary at: call selector ifAbsent: [
      "This is the fist time we receive this call..."
      execLists := ExecutionLists new.
      self addList: execLists forCall: call.
 ].
 ^res isNil not
```

If the cache does not know the selector of the pulled message, it calls addList:forCall:. This will call back the controller to collect the consequences from the connectors. Then it stores the result in the dictionary and returns the answer to whether there are consequences or not. As described in section 11.2 the controller will use this answer to decide if the message is a helper message (no rules triggering) and can be executed sequentially, or if it triggers rules and must be executed using the execution lists.

We saw that the dictionary stores message selectors as keys. But what happens to the arguments of a message?

13.3.2 Value of Arguments

After the controller knows that there are rules triggering on a message pulled from the queue, it can get the execution lists from the cache, since the cache has lazily loaded them. But when the controller starts to execute the lists, the consequence messages must have actual arguments provided from the message pulled from the queue (see section 4.1). The problem is that the cache holds the actual

arguments of a *previous* request. Normally the connector values the arguments of the consequence using the actual arguments of the message that was provided for lookup (see section 11.4) but with the cache the connector is bypassed.

In order to solve this problem we introduced a class ArgumentHolder which holds the *current* value of a symbolic argument. When looking up the consequences for the first time the cache sends the connector argument holders instead of actual arguments The connector then returns consequence messages valued properly with these argument holders instead of actual arguments. Furthermore, the cache keeps the argument holders separately in the ExecutionLists instance. When these lists are about to be executed, the cache uses the actual argument values that it gets from the message pulled out of the queue in order to set the value of the separately stored argument holders appropriately. Since all arguments of the cached consequences use the same objects as value holders, the setting of the separate holders also sets all arguments in the cached consequences. Thus the argument holders in the consequence messages refer the current arguments. When executing cached consequences the controller tells each argument holder in the consequences to provide this current actual value.

Figure 13.2 illustrates the use of the argument holders.

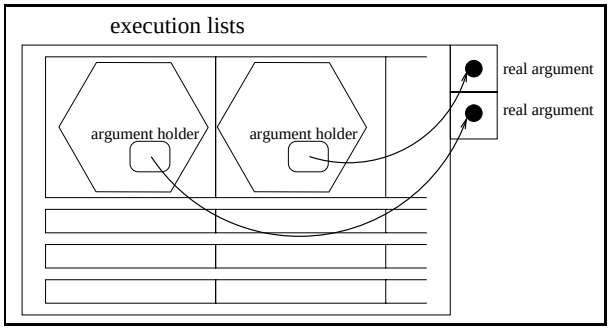

Figure 13.2: Argument holders in execution lists.

We use the concept of argument holders to solve another problem, namely the propagation of computation results with the keyword result (see section 4.5).

Our implementation subclasses the class ArgumentHolder to declare the class ResultHolder. When a rule uses the keyword result as argument, the connector replaces it with a result holder that is linked to the reified message which result should be used. When the controller executes a message it writes its return value to the result instance variable of the message. Result holders use this fact. Since they keep a reference to the reified message in which result their interested in, they can access the result instance variable when requested to provide the actual argument value. This happens when a message gets executed in the controller by the following method

```
Controller>>execute: call
 | res  realArgs|
 realArgs := cache giveArgsOfHoldersIn: call.
          "Ask the arguments holder to provide real args."
 res := self executeCompiledMethodFor: call object
          selector: call selector
          arguments: realArgs.

 call result: res. "This is for the symbolic 'result' argument"
 ^res.
```

```
Cache>>giveArgsOfHoldersIn: call
 ^call arguments collect: [:a | a giveArgument].
```

Thanks to the use of these holders neither the controller nor the cache has to know the difference between normal arguments and result values. The holders are simply told to provide their value using the message giveArgument, when the value is used for execution. Figure 13.3 illustrates the use of the result holders.

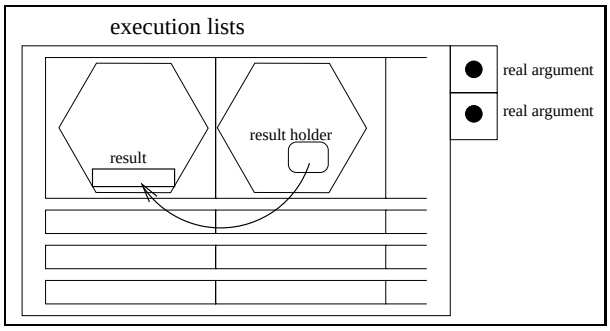

Figure 13.3: Result holders in execution lists.

13.3.3 Caching in a Dynamic Implementation

The FLO/C model is very dynamic. However, caching is something static therefore we must discuss the problems that arise when caching in a dynamic implementation.

In our FLO/C implementation connectors may be attached and detached from an active object dynamically. Such an action involves the notification of the controller of the active object, which will have to reset the cache. But the situation is even more complicated. Because of the recursive lookup of the sequential consequence messages, the caches of *every object that is reachable in the net of connectors* must be reset. This is because in any reachable object, a message could cause sequential consequences that finally affect the new connection. Therefore, the cache contents of any reachable object could be incomplete, when a new connection is established.

The FLO/C implementation provides broadcast facilities to propagate a cache update. Whenever a new connection is established, the connector informs all controllers of its participants, which in turn inform all known connectors and so forth.

Dynamic role specificators. Each controller controls one active object and has one cache. Therefore, the caches store the consequences per-object. Consider the rule:

```
philosopher eat. impliesBefore chopstick_select_RND pickedUp
```

The rule enforces that each time before a philosopher eats, one of the chopsticks is picked up. As seen in section 4.4.3 the RND-specificator selects randomly one of the objects that plays the role chopstick. The cache of the controller which pulled the rule triggering message out of its queue should not store the choice of the chopstick permanently. Else the same philosopher would always choose the same chopstick, which is clearly not the desired behavior when using the RND-Specificator. Therefore, our FLO/C implementation allows the connectors to indicate that consequences should not be stored in the cache, but collected from the connectors every time. Note that most of the specificators (receiver, next,

other) are not dynamic. Given a particular object in the precondition message, the receiver (itself) or the next (according to the inner order in the role group) is always the same object. Therefore, their consequence message can be cached.

Another solution to the problem of dynamic roles would be to transfer knowledge of roles to the controller. However in our implementation the controllers are per-object. The notion of roles and their mapping to objects is clearly the responsibility of the connectors.

13.4 Evaluation of the Optimization

Optimizing the *rule lookup* on the connector side brought a speedup of factor bigger than 5 and an additional factor that is proportional to the number of selectors per object in the rule-base, normally bigger than 2. On the downside it obscured the design. Optimizing the lookup on the controller side by introducing a consequence message cache promised great performance improvements but got us in trouble with the dynamics of FLO/C. In order to measure the performance improvement with caching, we measured execution time of a large example. We used the binary adder example already mentioned in section 6.1 and further discussed in section 14. The example consists of four composite objects that are connected to form a binary adder. The composite objects are nested, they consist of logical elements that can calculate logical operations like AND, OR and XOR. Inputs and the results of the elements are propagated entirely by connectors using asynchronous message passing. Because of the exhaustive use of nested composite objects, there are a lot of rules that just propagate a message to and from an interface object. Therefore, cached execution lists are small, which means that the cache will not show its full potential of performance improvement. This would be the case if there were a lot of connectors connected to every single object, defining many of rules for it. Therefore, we can interpret the resulting speedup in this example as a lower bound for the average speedup achieved by caching.

We ran the example on a Sparc Ultra 4. It contains 137 active objects therefore using 137 threads. In one run 156 asynchronous messages are used to propagate the 8 entry booleans to the 4 outputs. Since the system is purely push-flow based [Lea97], the first run encounters empty caches in every active object, but in the second run all consequence messages are loaded from the cache. The following table provides execution times in milliseconds.

first run (empty cache)	subsequent run (cached)
8648	1451
	1094
8804	1452
	1549

The heavy use of asynchronous message passing renders the example quite slow (see section 13.1), nevertheless the caching provided an average speedup of factor 6 that scales up with the numbers of rules per object.

We can conclude that the optimization work payed off since it speeded up the first implementation by an average factor higher than 60.

Chapter 14

Implemented Examples

Our FLO/C implementation is an object oriented language extension that allows one to explicitly express multi-object coordination. It also enforces the paradigm of strict separation between communication and computation (see section 3.1). The following sections illustrate how FLO/C programming is used to solve different toy-example problems which demonstrates the expressive power of the FLO/C model. The examples are partially taken from other coordination literature or designed to show either a particular problem of coordination or a particular feature of FLO/C. Because of the time limitations of this work we were not able to implement a full-scale real world example. However, we hope that by presenting our solutions to canonical examples taken from the literature, and by the diversity of the eleven[1] implemented examples, we can convince the reader that FLO/C is a simple and effective model to implement multi-object coordination.

Usually the components used in an example are small and easy to understand. Therefore, we only present their public interfaces. We will focus on the connectors and the rules they use to solve an interaction and coordination problem. Note that now we use the connector declaration syntax of the implementation. It has some small differences compared to the example of section 5: Connector classes are instantiated from the meta-class MetaConnector and the rule declaration of the method MetaConnector»withBehavior: string also implicitly declares the roles a connector knows.

The following sections describe the ten examples:

1. The **"vending machine"** proposed by Agha and Frølund was the first implemented example in FLO/C. It is explained in more detail than the other examples. Our solution does not use the higher FLO/C features like connector states and role management.

   The next five examples are the author's creations.

2. The **"synchronized movements"** is a small but paradigmatic example for FLO/C style coordination since it consists of only one set of synchronized multi-object joint actions (see section 2.2.2).

3. The **"unstable server"** example uses FLO/C's exception mechanism, and demonstrates *pull-style* client-server relations.

4. The **"decrementor"** example features a complex asynchronous interaction pattern.

[1] There are ten examples in this section an one in section 5.

5. The **"workers and tools"** example shows the use of relative roles to implement *mutual exclusion on shared resources*. It also discusses the subtle difference between the FLO/C implementation and the FLO/C model and its consequences to the example implementations.

6. The **"binary adder"** example illustrates nested composite object hierarchies and massive asynchronous message passing.

   The rest of the examples are taken from the coordination literature. They use FLO/C's high level features like dynamic connector establishment.

7. The **"dining philosophers"**. We present a fair solution to this well-known problem.

8. The **"workers-administrator"** example is an architectural pattern from parallel programming.

9. The **"electronic vote"** example was introduced to illustrate Minsky's coordination approach using so called policies.

10. The **"sleeping barber"** example models an enriched producer-consumer problem.

14.1 The Vending Machine

14.1.1 Description

The vending machine example was used in [FA93] as an example of a concurrent part-whole hierarchy. A vending machine consists of concurrent parts and the whole is subject to consistency requirements. Since their proposed vending machine architecture is trivially small we extended it by factoring out more participants with particular functionality. Our vending machine holds items behind an outlet door. It concurrently accepts: money, requests to open the door and requests to cancel the deal. When the door is open, it accepts requests to deliver the item. After the item in the outlet was taken, the machine puts a new one into the display. Additionally (not described in [FA93]) there are different policies of returning the money that was payed too much.

Coordination Aspects

Beside of the client-server relationships, there are different constraints between the outlet and the money accepting device (e.g. unlock outlet when enough money added), as well as between the outlet and the money returning device (no money back when the door is open). The coordination managing object must enforce that the machine cannot reach an inconsistent state or even is cheated on purpose when accessed concurrently, therefore it features *multi-object constraints*. The fact that the locking mechanism is located separately from the money entry can cause a variety of inconsistent states. For example a user can pay and then try to open the door and press the change releasing button at the same time. Or the user can pay, then open the door but not closing it again thus causing troubles for the reloading of a new item into the display.

14.1.2 Solution

We divided the machine in different independent units. The coin outlet, the money store, the door lock and the door simulate the control and monitoring of physical devices. Furthermore, a money manager represents the price policy of the machine. All these units are implemented by independent active objects. They carry only their proper functionality and use no assumptions how they collaborate in a

vending machine[2]. The goal is that they communicate and synchronize only through connectors. The connectors also enforce the global consistency of the different units. We want the vending machine to exploit as much asynchronism as possible because the different units represent real-world objects that behave concurrently. For example when a user has taken an item the next one should be able do add money immediately and should not be blocked by the process that puts a new item into the display.

The next section will show the active objects that represent parts of the machine. Then the connectors that implement the interactions are presented. Finally follows an evaluation of the example. Figure 14.1 shows the components and connectors of the vending machine.

The Active Objects

The active objects that represent real-world objects provide three kinds of services. They can be asked to provide an aspect of the state of the physical object (e.g. is the door open now) or they can signal that something of interest happened to the physical object (e.g. the change giving button is pushed) or they can trigger an action of the mechanical devices (e.g. lock the door). The following list shows the active objects and its interfaces.

coinEater. This active object signals when coins are inserted into the mechanical device triggering the message add: number. It also signals when the user pushes the "change" button with the message changeButtonPushed.

store. This object represents the place where the coins are stored. It can release coins for a requested amount with the method physicalRelease: number.

moneyManager. This is the location where the price of the item, the entered amount of money and the change amount is stored. The object stores these numbers in instance variables and provides accessor methods to get and set them. With the add: amount message the moneyManager internally reflects when money is added. The calculateIfEnough can be used to ask if the already added money is enough to purchase an item. The method calculateChange calculates the current amount of change and the method consume subtracts the price from the entered money, thus reflecting that an item has been sold.

doorLock. This active object represents the mechanic device that can lock the door. The following messages query the current state of the door: locked, unlocked. The door can be loked and unlocked using the methods: lock, unlock.

door. The door can signal if it is opening or closing. It can also trigger the mechanism to put a new item into the display. Query methods are: opened, closed, itemAvailable, itemTaken. The following methods reflect a change of the door's state: openDoor, closeDoor, takeItem. Furthermore, the door can issue a request for the physical device to put a new item into the display with the method produceItem.

Note that the door lock is separated from the door, because they represent two different physical devices. The door could easily be further separated from the "item display", but the example is rich enough to show characteristics of FLO/C.

None of these objects know each other or use methods of each other. We now introduce connectors that implement their collaboration. They will also take care to use asynchronous message passing if possible, to support the example's intrinsic concurrency.

[2]This is enforced by the fact that at no point in time a participant has any reference to another (see chapter 3).

The Connectors

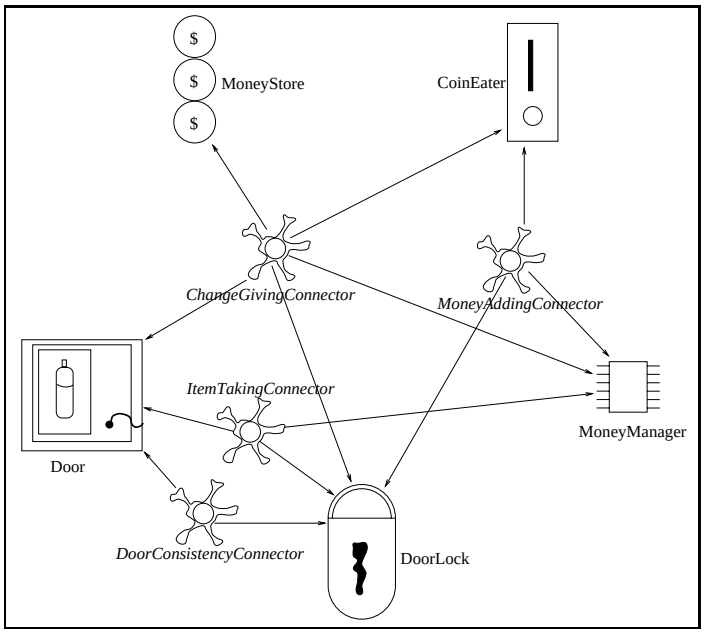

Figure 14.1: The vending machine architecture.

We split the interactions of the objects in four connectors. Three of them address a logically connected line of actions (user enters money, user takes item and user requests change), while the one we present right now just reflects the consistency of the simulation.

The DoorConsistencyConnector. The door and doorlock represent real-world entities that signal their state. This connector enforces the consistency of the signals of the door, and between the door and its lock. If "impossible" signals occur an exception is raised. Here is the declaration of the connector DoorConsistencyConnector:

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   door openDoor. permittedIf doorLock unlocked. door closed. endRule
(2)   door closeDoor. permittedIf doorLock unlocked. door opened. endRule
(3)   door takeItem.  permittedIf door opened. endRule
(4)   door methodWasForbidden: m. implies connector illegalSignal: m. endRule ';
( ) installAtName: #DoorConsistencyConnector
```

If the vending machine works correctly, there can't be a signal indicating that the door opens (or closes) when the door is locked. To get the open signal, the door must be closed, and vice versa (Rule 1 and 2)[3]. The item (located behind the door) cannot be taken, when the door is closed (Rule 3). If "impossible" signals occur, an exception is raised and the connector handles it in its method illegalSignal: methodSelector (Rule 4).

[3]Note that each of these rules has *two* consequence messages.

The MoneyAddingConnector. This connector expresses how the adding of money from outside is reflected inside the machine, therefore it connects the coin eater, the money manager and the door lock.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   coinEater add: amount. impliesLater moneyManager add: amount. endRule
(2)   moneyManager add: amount. impliesLater doorLock unlock. endRule
(3)   doorLock unlock. permittedIf moneyManager calculateIfEnough. endRule';
( ) installAtName: #MoneyAddingConnector
```

The first rule notifies the money counter. The second rule asynchronously tries to unlock the door. The door lock unlocks, of course, only if *enough* money is added (Rule 3). Note that this connector, with its asynchronous operators, brings concurrency into the simulation.

The ItemTakingConnector. The following connector implements the interactions that take place when a user takes an item. It connects the door, the money manager and the door lock.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   door takeItem. implies moneyManager consume. endRule
(2)   door closeDoor. implies doorLock lock. endRule
(3)   door closeDoor. impliesLater door produceItem. endRule
(4)   door produceItem. permittedIf door itemTaken. endRule
(5)   doorLock unlock. waitUntil door itemAvailable. endRule ';
( ) installAtName: #ItemTakingConnector
```

When the user takes an item, the price must be subtracted from the money entered (Rule 1). The second rule ensures that the door is locked as soon as the door is closed. Rules 3 and 4 take care that a new item appears, when the old one is sold. Rule 5 delays the unlock process until the item is produced by the two former rules. The process of putting a new item into the display is triggered asynchronously, because it can be time consuming but should not block all other activities of the vending machine. However, rule 5 prevents that the user can open the door before the new item is displayed. Else a user could open the door before the next item is on the display and might close the door again in disappointment. Then the vending machine must think that the item has been taken thus it consumes the money of the user (Rule 1). Therefore, rule 5 prevents that the user can loose money without getting an item.

Note that the closing of the door implies the door to lock, and the locking of the door will cause the next connector to move into action.

The ChangeGivingConnector. This connector is responsible for the change that the machine gives back. It connects all active objects of the machine together. If the user pushes the change button, the door is locked (rule 1). Rule four ensures that the door is closed. Else, locking the door would be ineffective. The user could open the door, push the change button to get the full change, then take the item (since the door is still open) and cheat the machine that way. Rule 2 causes the money manager to calculate the change. When the money manager has calculated the change it stores the amount using its accessor method change. In rule 3, this accessor triggers the store to release the correct

amount. Note that the method calculateChange and rule 3 show the FLO/C programming idiom for asynchronously propagate computation results (see section 7.4). A part of the chain of actions is hidden in the fact that calculateChange calls change: amount which triggers rule 3.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   coinEater changeButtonPushed. implies doorLock lock. endRule
(2)   doorLock lock. implies moneyManager calculateChange. endRule
(3)   moneyManager change: amount. impliesLater store physicalRelease: amount. endRule
(4)   coinEater changeButtonPushed. permittedIf door closed. endRule ';
( ) installAtName: #ChangeGivingConnector
```

We see that the change giving process is triggered each time the door locks. That way, the user also gets change each time he takes an item and closes the door (see the previous connector). Other ways of change giving (e.g. only when button pushed) are also possible.

14.1.3 Evaluation

We used only stateless connectors to implement the example. This is possible, because we use object states as guards. (Door»opened? MoneyManager»calculateIfEnough?). They represent the abstract state of the group of objects that compose the vending machine. Operations that should execute on a stable state are connected using *sequential ordering operators* (implies and impliesBefore) operators, which guarantees that the state of the participants is not changed by some concurrent action. For example, the following scenario is not possible: The user adds money then presses the change button. If the messages could interleave in a bad way, first the door will lock (because of the change button), then the door will unlock (because the last adding will be handled asynchronously). Now the vending machine releases the full change, but the user can open the door! This cannot happen because the change giving joint actions update the money manager (rules 1 an 2 in ChangeGivingConnector) and the unlocking of the door is guarded by the money manager (rule 3 in MoneyAddingConnector).

The example of [FA93] is only a sketch of a vending machine that even shows limitations of the synchronizer construct that were used. Their machine can not automatically give change, because the synchronizers lack of a possibility to send messages to the active objects. The example we present is not trivial. Therefore, it shows the expressive power of FLO/C in offering different design dimensions:

- The *level of concurrency* is controllable by the connectors. In the example we tried to use a lot of asynchronous propagation. This lead to interleaving of processes. By replacing the asynchronous with synchronous operators, we could sequentialize the whole simulation.

- The designer has *full control over the behavior* of the machine, and is not limited by FLO/C to one solution. For example the change giving policy can be changed to: no change, when item is taken, only when button is pushed. Or the moment when the money is consumed can be varied. For example the money could be consumed as soon as the door opens (not when the item is taken).

- The designer can break communication patterns into easy *understandable units* (as seen in the example: chains of implications, and their guards), or (s)he can take other considerations into account, like minimizing the number of connectors or rules and the use of connector states etc.

14.2 Synchronized Movements

14.2.1 Description

An arbitrary number of graphical elements should keep their relative position to each other, but each can be told to move concurrently. All objects should stay within a certain border.

Purpose of the Example

This is a canonical example for synchronized multi-object *joint actions* (see 2.2.2). Every participating object adds its own constraint. Once the preconditions to the joint actions are checked, the participants should move without treating other movement requests until all participants have reached the destination of their movement. Since this is a paradigmatic example for joint actions it's easy to solve in FLO/C.

14.2.2 Solution

In this example, graphical objects implement the method moveAddX: x addY: y that moves the object from its current position for the provided amounts x and y along the x and y axis of a screen. Furthermore, a testing message notOutOfRangeX: x rangeY: y is implemented to check if a movement would carry the object out of range.

The connector declaration to coordinate an arbitrary number of such visual objects looks like this:

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   graphObj moveAddX: x addY: y. implies
( )           graphObj_select_Next moveAddX: x addY: y. endRule
(2)   graphObj moveAddX: x addY: y. permittedIf
( )           graphObj_select_REC notOutOfRangeX: x rangeY: y. endRule' ;
( ) installAtName: #GraphicObjConnector
```

Rule 1 When one of the graphical objects receives the request to move for a certain amount on the x and y axis, the request is propagated to the next graphical object. Considered the cycle breaking mechanism, the sending of a move request causes each participant to move once in a *sequential order*.

Rule 2 However, each object must check if it can move without going out of the screen range.

Rule (1) and (2) form a set of joint actions. Therefore, (see section 4.3) upon a single movement request the guards for *each* participant are checked first. If all guards succeed all the objects move in a sequential and protected fashion. During the movements they may receive other movement requests, but do not treat them until all objects have reached their destination of the current movement. Note that instead of the permittedIf guard a waitUntil guard can also make sense, when there is a chance that a retry of the movement request will succeed later, for example when the constraining range will be widened or when participants are detached from the connector.

14.2.3 Evaluation

The solution consists of the declaration of one set of joint-actions. The actions are an ordered sequence of movements and every movement is guarded individually. The fact that the guards are collectively evaluated before the actions guarantees that all movements take place or none. Therefore, the solution also represents a *pessimistic transaction*: only when all participants can commit to the actions the movements take place and they are protected from-third party access.

Note that we can immediately guarantee that our presented solution is livelock free because it does not use waitUntil operators (see section 9.2).

14.3 An Unstable Server for a Client

14.3.1 Description

The example consists of clients that use servers to preprocess data. Each server is up at a chance of 50 percent. If it's up, it does the preprocessing on the argument. If it is down, the client has to do all the work on its own. Note that the client could also react in other ways, e.g. contacting another server. The concrete processing of the toy example: A client gets a string in its in: accessor. A NotifyingCSConnector causes a server to preprocess it (conversion to number), and delivers the result back to the client, which processes it (increments the number by one). If the server fails, the connector causes the client to do the conversion on its own. The final result is put in the out instance variable of the client.

Purpose of the Example

The connector connects *independent* active objects. It allows the client to use services of the server in *pull-style*. Just before a client needs the preprocessing of a server the connector pulls the result from the server. Furthermore the example demonstrates the exception handling of FLO/C.

14.3.2 Solution

The following two tables present the interface of the participants. The interface of the participant that plays the role of the client:

Client

in: string	Input.
preprocessed: number	Intermediate result.
localwork: number	Handle the local work of a request.
doAll	Handle both local work and server's preprocessing.
out: number	Output.

The interface of the participant that plays the role of the server:

Server

preprocess: string	Preprocess a string.
isAlive	Answer if server is available.

We declare the NotifyingCSConnector to connect an arbitrary number of objects that play the roles client and server. We will explain the solution following the order of the rules.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   client in: a. impliesLater client_select_REC localwork: a. endRule
(2)   client localwork: a. impliesBefore server_select_Next preprocess: a. endRule
(3)   server preprocess: a. implies client_select_REC preprocessed: result. endRule
(4)   server preprocess: a. permittedIf server isAlive. endRule
(5)   client localwork: a. implies client_select_REC out: result. endRule
(6)   client methodWasForbidden: m. implies client_select_REC doAll. endRule
(7)   client doAll. implies client_select_REC out: result. endRule  ' ;
( ) installAtName: #NotifyingCSConnector
```

Rule 1 Once a client received data through the argument to a in: message, the connector sends a message to the client, telling it to do it's local work. Note that the method localwork: expects to find the preprocessed data in the instance variable preprocessed.

Rule 2 Therefore, before the local work can start, a server is called to do the preprocessing ($\rightarrow$ pull-style). If there are several clients and servers, the servers are called one after the other according to an inner order (see section 4.4.3 for the definition of the Next-specificator). Thus they get equal amounts of requests.

Rule 3 Once the server has preprocessed a result, it is sequentially propagated back to the client that requested it, using the default role result. Note that rule 2 and rule 3 form joint actions to trigger the preprocessing and propagating it to the client. Therefore, when the client starts its local work, the preprocessed data is ensured to be ready in preprocessed.

Rule 4 However, the server only preprocesses data when it is up. This guard message extends the sequential joint-actions of the rules 2 and 3 that does the pull-style preprocessing service. Thus first this guard is checked (is the server up?) then the reserved server is told to preprocess and deliver the result to the client[4].

Rule 5 Finally the result of the locally and remotely processed request is written to the output.

Rule 6 However, the pulling of the preprocessing can be forbidden when the server is down. In this case an exception is raised that we catch here. For this example we decided to make the client capable of processing the whole task on its own. Other solutions, like retrying or using another server are possible by changing this rule.

Rule 7 If the result is processed by the client itself it must be out-putted, too.

14.3.3 Evaluation

Pulling client-server interaction is frequently used. The example shows a FLO/C programming pattern to implement sequential pulling-style interactions (rule 2-4). It also illustrates how the exception mechanism can be used to implement different failure handling policies.

[4]The example is simplified because we assume that the server will not go down once it is reserved.

14.4 The Decrementor

14.4.1 Description

The decrementor is introduced by the author as an example with non-trivial, time consuming interaction between active objects. A decrementor iteratively processes a natural number. Whenever the number is even, it is divided by two. When it is odd, an odd amount (typically 3) is added to it. Eventually, this process leads to the number one. E.g. $11 \rightarrow 14 \rightarrow 7 \rightarrow 10 \rightarrow 5 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

However, for some inputs (like for 6) the process goes on forever: $6 \rightarrow 3 \rightarrow 6 \rightarrow ...$
The decrementor returns the number of iteration steps it takes to reduce the starting number to one.

Purpose of the Example

The example shall show interesting interaction patterns using asynchronous push-style propagation. Therefore, our decrementor consist of three independent active objects, namely the *adder*, which does the adding on odd numbers, the *divider* which divides the even numbers and the *counter*, which counts the steps. The divider cannot process odd numbers therefore it is responsible to test if a given number is odd or even. One (little) multi-object coordination problem comes from the fact that the adder must also use the testing method of the divider. However when using FLO/C's guard operators this is no problem at all. The interaction pattern is "complex" because the connector cannot tell on its own if a number must be propagated to the adder or to the divider.

14.4.2 Solution

We identify three independent components: the adder the divider and the counter. Furthermore, we declared additional methods for the connector (see section 4.5). The components are implemented as active objects. The adder and the divider implement the method process: number to provide their special functionality (adding or dividing). They store their results using their accessor method result: number. Note that this will be used to implement FLO/C's idiom for the asynchronous propagation of computation results (see section 7.4). The divider implements the query methods canProcess: number and cannotProcess: number. The counter increments the current count through the message count. With resultAt: number it outputs the current count and resets the counter. The connector implements the query methods isOne: number and biggerThanOne: number.

In our solution we introduce two connector classes. The DecrementConnector asynchronously propagates results between the adder and the divider. The CounterConnector is responsible that the counter can correctly count the number of iterations used to decrement an input to 1. Here is the declaration of the DecrementConnector.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   adder process: a. permittedIf divider cannotProcess: a. connector biggerThanOne: a. endRule
(2)   divider process: a. permittedIf divider canProcess: a. connector biggerThanOne: a. endRule
(3)   adder result: a. impliesLater divider process: a. adder process: a. endRule
(4)   divider result: a. impliesLater divider process: a. adder process: a. endRule';
( ) installAtName: #DecrementConnector
```

Rule 1 The adder should only process an input, if it is odd (not processable by the divider) and if it is still bigger than one.

Rule 2 The divider should only process an input if it can (input is an even number) and if it is still bigger than one.

Rule 3 and rule 4 Whenever the adder or the divider has processed an input they store the result using an accessor method result:. This will trigger rules 3 and 4, thus propagating the result asynchronously to the adder and the divider again.

Since the connector does not know if the adder or the divider has to process the next result, it sends requests to both and defines guards for both to filter out the wrong processing. The asynchronous messages keep bouncing between the processing units until the argument they carry is decremented to 1.

The CounterConnector knows the roles counter and processor. While it triggers the counter object to keep track of processings it does not distinguish between adder and divider. They both play the role of processor.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   processor result: r. implies counter count. endRule
(2)   processor result: r. impliesLater counter resultAt: r. endRule
(3)   counter resultAt: r. permittedIf connector isOne: r. endRule' ;
( ) installAtName: #CounterConnector
```

Rule 1 Whenever a processing unit produces a result, the counter increments immediately.

Rule 2 The connector also tries to asynchronously trigger the method resultAt:. This method outputs the final count of the counter.

Rule 3 However, the result is only final if the input is decremented to 1.

14.4.3 Evaluation

We composed the adder, divider and counter into a decrementor composite object, in order to use it as worker that shows heavy internal communication, and thus is time consuming. The decrementor was used amongst other things to test the *workers-administrator* example (see section 14.8), where it played the role of a time consuming worker. Note that there the decrementor is protected against concurrent requests. In the example as presented here, multiple requests can bounce between the adder and the divider. The counter counts *every* processing step. It does not distinguish between different requests therefore the results will be wrong. But the different request are all reduced correctly therefore the requests will cause the correct numbers of outputs.

14.5 Workers and Tools

14.5.1 Description

In this example an arbitrary number of workers use an arbitrary number of tools. The workers work in independent live cycles. They must be provided with a tool, when they want to work. However each

tool can only be used by one worker at the time.

Purpose of the Example

The example illustrates *mutual exclusion* and the problem of *shared resources*. FLO/C proposes *relative roles* to express the temporal ownership of an active object (see section 4.4.3). Furthermore, we demonstrate the dynamics of role groups in FLO/C. In a running example, workers and tools can be added or removed on the fly.

14.5.2 Solution

The workers do either work or have a break. Every time before they start to work, a connector of class WorkersConnector automatically delivers a tool. The workers implement their working cycle themselves as the following code sketch shows. However, this could also be programmed in the connector.

```
Worker>>work
 "Doing something here"
 ...
 self haveABreak

Worker>>haveABreak
 (Delay forMilliseconds: self lazyness) wait
 self work
```

Here is the declaration of the WorkersConnector:

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   workers work. impliesBefore tools_select_RND_as_myTool free: false. endRule
(2)   workers work. impliesBefore workers_select_REC  tool: myTool. endRule
(3)   workers work. implies myTool  free: true. endRule
(4)   workers work. waitUntil myTool  free. endRule ' ;
( ) installAtName: #WorkersConnector
```

All four rules compose one set of multi-object joint actions (see section 2.2.2) that handles the state transitions when one worker decides to work.

Rule 1 Before a particular worker starts to work, a tool is explicitly reserved using the method ActiveObject»free: boolean [5]. The tool is chosen by random (RND-specificator) and the choice is stored in the relative role myTool (see section 4.4.4).

Rule 2 Furthermore, the worker receives a reference to the tool. This is in contrast to FLO/C's paradigm (see section 3.1) but used in order to underline the reservation of the tool.

Rule 3 After the work is done, the tool is released.

Rule 4 If the chosen tool is not free, the multi-object joint actions are delayed.

[5]The active objects provide a free: boolean accessor method and a testing method free. They are just a getter and setter method to a boolean instance variable and have no special functionality. Thus the user could also declare an instance variable of Tool to use it as explicit lock.

14.5.3 Explicit Locks and Processor Yields

This example reveals the simplification of the FLO/C implementation in contrast to the model. In the FLO/C model the explicit reservation of the tool and the guard in rule 4 would not be necessary, because the participants of the joint actions (a worker and a tool) are reserved by the underlying mechanisms (see 4.3). Our implementation however does no implicit reservation, but simply does not yield the processor. This leads to almost equal behavior as an underlying reservation (see section 11.2). However the work method does yield the processor by calling haveABreak which calls Delay»wait which subsequently suspends the current process. Therefore, the implicit reservation of our FLO/C implementation is broken, and the example has to add explicit reservations (using ActiveObject»free:). In other examples like the "dining philosophers" example, we will yield the processor on purpose in order to show how explicit reservation can be programmed with our FLO/C implementation. Using explicit reservation is sometimes necessary in the model, too. It is used when an object must be protected against a particular access for a particular time not corresponding to a single set of multi-object joint actions. The access of the pumps between the payment actions and the pumping actions seen in the gas station example of section 5 is such a case.

14.6 The Binary Adder with Logical Switches

14.6.1 Description

Logical elements (not, and, or, xor) as independent actors are composed to a binary adder that can add an arbitrary number of digits.

Purpose of the Example

We did not introduce this example to implement a program that can add fast, but to illustrate the composition of active objects into composite object classes (see chapter 6). The whole example works as an *asynchronous push-flow* system. At the bottom layer of the hierarchy, there are composite objects that each implement a logical operation. To do so they must *synchronize on their entries* (that are triggered asynchronously) to produce an output.

14.6.2 Solution

Since the composition of the adder is already described in chapter 6 we focus on the architecture of the logical elements.

Logical elements. A logical element is a composite object that encapsulates an inner element. The inner element is an active object that implements the logical operation (and,or etc). The inner object has two instance variables that represent the current input to the logical operation. The inner element implements the method fire that returns the result of the operation, using the current values of its instance variables. The instance variables are set by the methods: inA: bool and inB: bool.

The logical element contains two connectors, one of them (the TwoInputsConnector) propagates input from the interface to the inner element and delivers the result of the firing from the inner element to the interface. The other connector (the SynchronizeInputsConnector) synchronizes the inputs and the firing. For example the composite object declaration of the AndElement looks as follows:

```
MetaCompositeObject new
   superclass: NotElement ;
   withComponentClasses: '
      And ' ;
   withConnectorClasses: '
      TwoInputsConnector SynchronizeInputsConnector ' ;
   withConnectionSchema: '
      connector: 1   role: innerElement   object: 1
      connector: 2   role: innerElement   object: 1' ;
   installAtName: #AndElement
```

Note that the declaration of all further logical elements can reuse the AndElement declaration through *inheritance* (see section 6.2). The following code illustrates this on the example of the XorElement declaration which solely overrides the class of the inner object (Xor instead of And):

```
MetaCompositeObject new
   superclass: AndElement ;
   withComponentClasses: '
      Xor ' ;
   withConnectorClasses: '
      super super ' ;
   withConnectionSchema: ''
   installAtName: #XorElement
```

All logical elements use the same two connectors. TwoInputsConnector implements the internal propagation and SynchronizeInputsConnector the synchronization of the inputs. Here is the declaration of the class TwoInputsConnector:

```
( )MetaConnector new
( )  superclass: Connector ;
( )  withBehavior: '
(1)    interface inA: a. implies innerElement inA: a. endRule
(2)    interface inA: a. impliesLater innerElement fire. endRule
(3)    interface inB: a. implies innerElement inB: a. endRule
(4)    interface inB: a. impliesLater innerElement fire. endRule
(5)    innerElement fire. implies interface outA: result. endRule' ;
( )  installAtName: #TwoInputsConnector
```

Rule (1) and (3) of the TwoInputsConnector immediately set the instance variables according to the input on the interface. Upon each input rules (2) and (4) try to asynchronously trigger the firing. When the firing took place, rule (5) carries the result to the interface for output. However, the inner element should not trigger when only one input arrived already. This synchronization is addressed by the class SynchronizeInputsConnector as seen here:

```
( )MetaConnector new
( ) superclass: Connector ;
(1) instanceVariableNames: 'bSent aSent ' ;
( ) withBehavior: '
(2)   innerElement inA: a. implies connector aSent: true. endRule
(3)   innerElement inB: a. implies connector bSent: true. endRule
(4)   innerElement fire. permittedIf connector aSent. connector bSent. endRule
(5)   innerElement fire. implies connector aSent: false. connector bSent: false. endRule ' ;
( ) installAtName: #SynchronizeInputsConnector
```

This connector defines instance variables (1) to store if input has already arrived. Rule (4) allows output only when it has arrived on both channels (2)(3). Upon firing the connector's state is reset (5).

Composition of the adder. In section 6.1 we explained how the logic elements can be composed to a binary adder element. The following connector connects the binary adder elements to a *binary adder* with arbitrary entries:

```
MetaConnector new
 superclass: Connector ;
 withBehavior: '
  adder outB: a. impliesLater adder_select_Next inC: a. endRule' ;
 installAtName: #BinaryAddersConnector
```

The connector connects the carry bit output (outB:) to the carry bit input (inC:) of the next adder element.

14.6.3 Evaluation

The example shows nested use of composite objects as well as the inheritance of them. Furthermore, it is used for measuring the performance improvement of the FLO/C implementation using a consequence message cache (see section 13.4).

The example illustrates that asynchronous push-flow systems [Lea97](p.220) are naturally programmed in FLO/C. "Splitters" can be programmed by using role groups of rules with several consequences. "Mergers" can synchronize using the permittedIf operator as seen in the SynchronizeInputsConnector declaration.

14.7 The Dining Philosophers

14.7.1 Description

The dining philosopher problem [Dij72] is probably the most cited coordination problem example. An arbitrary number of philosophers are sitting on a round table, thus each one has two neighbors. A philosopher alternates between eating an thinking. In order to eat, a philosopher has to grab its left *and* right chopstick. The left chopstick of a philosopher is the right chopstick of his left neighbor. Only one philosopher at a time can hold a particular chopstick.

Purpose of the Example

The example illustrates three well known coordination problems: Shared resources (chopsticks), with mutual exclusion and the possibility of deadlock. Furthermore, a coordination language should offer the possibility to program a *fair* or starvation free solution.

14.7.2 Solution

The philosophers and the chopsticks are implemented as independent active objects. The philosophers' methods implement their live-cycle. After a philosopher has eaten it thinks for a random time and then starts to eat again. The methods of a philosopher are:

```
Philosopher>>eat
   (Delay forMilliseconds: self randomAmountOfTime) wait
   self think

Philosopher>>think
   (Delay forMilliseconds: self randomAmountOfTime) wait
   self eat
```

Our solution introduces two connectors to implement the interactions between chopsticks and philosophers and to enforce mutual exclusion and fairness. The DiningPhiloConnector is responsible for the relationship between philosophers and their chopsticks. In its initialization phase, it orders the philosophers and the chopsticks. Then, whenever a philosopher wants to eat, the user-defined LNR-specificator selects it's appropriate chopsticks. The connector changes the chopsticks' state to *explicitly* reserve them. Again this is only necessary, because of a Delay»wait call intentionally placed within the eat method (see section 14.5.3). Thanks to the explicit reservation, the connector still guarantees mutual exclusion.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   philosopher eat. impliesBefore chopstick_select_LNR_as_mySticks free: false. endRule
(2)   philosopher eat. waitUntil mySticks free. endRule
(3)   philosopher eat. implies mySticks free: true. endRule ' ;
( ) installAtName: #DiningPhiloConnector
```

Rule 1 Before a philosopher wants to eat, his chopsticks are chosen by the select-left-and-right specificator LNR[6]. They are explicitly locked setting free to false. Furthermore the relative role mySticks maps the particular selection of chopsticks to the philosopher (see section 4.4.4).

Rule 2 If one of the two sticks is occupied already, the philosopher must wait for eating.

Rule 3 After the philosopher has eaten its chopsticks are released.

In order to grant fairness, the FairPhiloConnector enforces, that no philosopher can grab chopsticks, when one of his neighbors is waiting for a longer time. The connector uses methods defined in

[6]The Left-and-right specificator uses the inner order of the role group philosopher to determine the neighbors. See also section 4.4.

its superclass FairnessConnector to register request times and to compare them, using the SMALLTALK message Time class»millisecondClockValue. The method registerTimeFor: participant registers the time of a request by a particular participant and the method unregisterTimeFor: participant resets the registration when the request was granted. The method notWaitingLonger: group than: participant returns false if there is a member of group who's request time is older than the one of the participant.

```
MetaConnector new
( ) superclass: FairnessConnector ;
( ) withBehavior: '
(1)   philosopher think. implies connector registerTimeFor: philosopher_select_REC. endRule
(2)   philosopher eat. waitUntil
( )         connector notWaitingLonger: philosopher_select_LNR
( )                                 than: philosopher_select_REC. endRule
(3)   philosopher eat. implies connector unregisterTimeFor: philosopher_select_REC. endRule';
( ) installAtName: #FairPhiloConnector
```

Rule 1 After a philosopher has finished thinking, he wants to eat. Therefore, the connector registers the time of this event.

Rule 2 Before an actual eat request is treated, this rule ensures that none of the neighbors (determined by the user-defined LNR-Specificator) is waiting longer for eating. Note that only the two neighbors of a philosopher compete for the chopsticks.

Rule 3 After a philosopher has eaten the connector has to register this fact. The waiting time of the philosopher is set back to zero.

14.7.3 Evaluation

The FLO/C solution to the "dining-philosopher problem" is so easy that we had to pep it up with Delay»wait calls. The solution is deadlock free because any FLO/C solution is (see section 8). It is livelock free as well. A given philosopher must at most wait until his two neighbors have eaten once. Then it will be the one that waited the longest. According to section 8.3.2 the model (or section 11.2 the implementation) will give the philosopher a chance to eat. Since both neighbors are blocked out the philosopher will succeed to pick the chopsticks and eat. Therefore, the solution is always live and fair. It demonstrates how FLO/C can be used for mutual exclusion on shared resources.

14.8 Administrator and Workers

The administrator and worker example [Gen81] is an example often cited in various literature. It concerns the problem how to design an architecture that can exploit the distribution of tasks. The administrator-workers pattern is such an architecture.

14.8.1 Description

An arbitrary number of clients need to use a service that more than one server (worker) can provide. An administrator distributes the requests of the clients to the workers and forwards the results back to the clients. The administrator can use different policies to decide which request goes to which worker (depending on the size of the request, or the state of the worker etc.)

Purpose of the Example

The example bears complex interaction. The administrator has to protect the worker from concurrent access and it has to manage dynamically changing connections between clients and workers. Therefore, it is a good test for the FLO/C model. Since the administrator-workers example is still a simple problem we want to solve it without using connector programming to explicitly keep track of the connections between workers and clients. Instead we want to code *all* interactions between clients and worker into rules of connectors.

14.8.2 Solution

In our solution a client issues a requests for a computation by calling its own method request: arg. It expects to be called back with the message result: res that carries the result in the argument res. The workers can process an argument with their method process: arg. When they finished a processing they write the result using their method result: res.

The work of the administrator is done in the AdministratorConnector. It connects the clients with the workers. Our solution is *push based* and involves a *call-back to the client* implemented by a connector. The AdministratorConnector dynamically creates a BacksendConnector for each request, which in turn will propagate the result back to the client, and then destroy itself. Thus, the AdministratorConnector does not have to keep track which worker currently works for which client. The policy to choose a worker is implemented in a specificator (FRE). This specificator simply chooses the first free (not running) worker. If none is found, it chooses a random one, and the request has to wait. In order to use another policy, the user can define another specificator.

Our solution involves two new connector classes. The class AdministratorConnector implements the administrator. It uses the roles worker and client to connect an arbitrary number of workers an clients. It uses the default role connector to call its own methods. One of these methods is createConnectionFrom: source to: target which uses the class BacksendConnector to establish a channel between a particular client and a particular worker.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)     client request: a. implies connector request: a from: client_select_REC. endRule
(2a)    connector request: a from: c. implies worker_select_FRE process: a.
(2b)            connector createConnectionFrom: worker_select_FRE to: c. endRule
(3)     worker process: a. implies worker_select_REC free: false. endRule
(4)     worker process: a. waitUntil worker_select_REC free. endRule
(5)     worker result:a. implies worker_select_REC free: true. endRule' ;
( ) installAtName: #AdministratorConnector
```

Rule 1 When a client issues a request for a computation on an argument, the connector calls a dummy method to redirect the request. Note that we want the rules of connectors to do all the communication work, and not methods of the connector.

Rule 2a The user defined free-specificator (FRE) chooses a free worker to process the request. Note that the specificator will find a free worker if possible. However, if all workers are occupied it will select an occupied one. A user can define a specificator by inheriting from class Specificator. (S)he must declare three properties of the specificator: (1) Is the specificator dynamic

or not[7]. (2) What is the abbreviation for the specificator when used in rules (e.g. FRE) and (3) What is the policy to select a subset of the role group. In order to define a policy the user overloads the method select:context:. This method receives the role group and the context of the current message, which is necessary for some specificators (see section 4.4.3), and it returns a collection containing the subset of participants. The following code defines the policy of the free-specificator.

```
select: roleGroup context: connector
 "Selects one free participant of the role group."
 "If none is free, a random one is returned."

 ^OrderedCollection new add:          "Return a subset with:"
    (roleGroup detect: [:o | o free]
                ifNone: [
                        roleGroup at: (Random new next * roleGroup size) rounded
                        ]
    )
```

Note that the free-specificator (FRE) is dynamic. Its choice cannot be cached.

Rule 2b The connector calls its own method createConnectionFrom:to: which establishes a channel for the result to be sent from the worker to the appropriate client.

Rule 3 and 5 The workers are reserved and released explicitly using the selector free:. Note that the free-specificator is designed to work with the free selector to determine if a worker is free.

Rule 4 If no worker is free, a request has to wait.

Rule 2 triggers the method createConnectionFrom:to: which establishes a channel for the callback of the client when the worker has produced the requested result.

```
AdministratorConnector>>createConnectionFrom: worker to: client
 |b|
 "Establish a connection to deliver the result."
 b := BacksendConnector new.
 b objects: worker playRole: 'source'.
 b objects: client playRole: 'target'.
 b install
```

Here we make heavy use of the dynamics of FLO/C. Once the connector is established, the administrator can forget about the request of the client. The declaration of the BacksendConnector is:

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   source result: a. implies target result: a. endRule
(2)   target result: a. implies connector decouple. endRule' ;
( ) installAtName: #BacksendConnector
```

[7]Dynamic specificators forbid caching as described in section 13.3.3

Rule 1 Once the worker produced the result, the client gets a call-back immediately, providing it with the result.

Rule 2 When the target received the result the connector terminates gracefully, without terminating its participant[8] (Connector»decouple).

14.8.3 Evaluation

The example shows the expressive power of dynamic connector creation and destruction. Instead of programming the worker selection policy into the connector, which is also a possible way of implementing the problem, a user-defined specificator selects the workers for a given request. Therefore, all client-worker interaction is coded in connector rules and the selection policy of the administrator is factored out in a user-defined specificator.

14.9 The Electronic Vote

14.9.1 Description

The electronic vote example was presented by Minsky et al. [MU97] to demonstrate the expressive power of their coordination language which also features explicit entities for coordination (see section 15).

They describe the following problem. Assume that there is a specific issue on which an open and heterogeneous group of agents (active objects) is asked to vote. Consider the following policy designed to ensure that the vote is fair and confidential:

1. An agent can vote at most once, and only within the time period alloted for this vote.
2. The counting is done correctly.
3. An agent is guaranteed that nobody else, not even the organizer of the vote, will know how (s)he voted.

We present a FLO/C solution to a small extension of the problem. In our solution each voter can initialize a vote and provide the issue and deadline. Multiple votes can run concurrently as long as they concern different issues. All voting results are propagated back to the voters.

Purpose of the Example

Voters have their own independent opinions. The example demonstrates the separation of concerns between policy enforcement and individual object behavior. The example involves proactive behavior of the participants that call a vote, and it involves time measurement. Furthermore, since the voters are independent they can try to cheat by voting twice on the same issue. The FLO/C approach has the advantage that active objects have no references to others (see section 3.1) and therefore cannot bypass the connectors to cheat.

[8] Else it would terminate a client and a worker.

14.9.2 Solution

The voters are the only actors in this example (beside of the connector). Class Voter has the following public interface.

Voter	
startVoteOn: string deadline: time	The voter request a vote.
voteOn: string	The voter is informed about a vote on an issue. It calls:
opinion: string	This method returns the voters opinion on an issue.
resultOf: string was: res	The voter gets the result of a vote.

When a voter is told to vote on an issue it builds its own opinion on the issue. The method looks like that:

```
Citizen>>voteOn: issue
 "The citizen asks itself, what its' opinion on as subject is."
 self opinionOn: issue.
```

In our implementation a citizen builds its opinion using its private method intelligence which returns a number. The methods code looks like this:

```
opinionOn: issue
 ^((Random new next *(issue size / self intelligence ) )  <0.5)
 "The longer the issue, the more likely it is rejected. Issues that have"
 "the same size as the number stored in 'intelligence' have a 1:1 chance."
```

We declare a class VoteConnector. A single connector of this class is designed to ensure proper voting capabilities for an arbitrary number of voters, to which it refers by the role citizen. The VoteConnector class defines methods to register voters and their vote. The connector contains rules that start its own methods using the default role connector (see connector programming in section 4.5). The methods mainly write single results into SMALLTALK dictionaries stored in the connectors own instance variables timeTable, voteTable and votersRegistration. They are not further presented here.

```
( ) MetaConnector new
( )  superclass: Connector ;
( )  instanceVariableNames: 'timeTable voteTable votersRegistration ' ;
( )  withBehavior: '
(1)    citizen startVoteOn: issue deadline: t. implies connector openVoteOn: issue until: t. endRule
(2)    citizen startVoteOn: issue deadline: t. impliesLater citizen voteOn: issue.
( )                connector countResultOf: issue. endRule
(3)    citizen opinionOn: i. implies connector register: result on: i.
( )                connector registerVoter: citizen_select_REC on: i. endRule
(4)    citizen opinionOn: i. permittedIf
( )                connector voter: citizen_select_REC hasNotYetVotedOn: i.
( )                connector timeNotElapsedFor: i. endRule
(5)    connector countResultOf: i. waitUntil connector timeElapsedFor: i. endRule
(6)    connector countResultOf: i. implies citizen resultOf: i was: result.
( )                connector closeVoteOn: i. endRule
(7)    citizen startVoteOn: issue deadline: t. waitUntil connector notVotingOn: issue. endRule ';
( ) installAtName: #VoteConnector
```

Rule 1 A citizen can call a vote on an issue and provide a deadline for it. The connector (accessed by the default role connector) must register the issue and the deadline immediately.

Rule 2 Then the connector broadcasts the subject of the vote to the citizens. It does so asynchronously, because it is not expecting them to return an opinion; it is the citizen's choice to do so. The second consequence of rule 2 sends the request to count the result of the vote to the connector. As seen in rule (5), this will not happen until the deadline is reached.

Rule 3 When a citizen has built its opinion, it is registered by the connector. The connector stores the citizen's vote separately from the citizen in order to keep the vote anonymous.

Rule 4 However, the opinion is only registered if the citizen has not already voted, and if the deadline has not passed.

Rule 5 As seen in rule (2) the connector asynchronously requests itself to add up the results, but it must wait for the deadline.

Rule 6 Once the deadline is passed and the votes are counted, the result is immediately propagated to all citizens, and the connector closes the vote.

Rule 6 Before the vote is not closed it is not possible to start a new vote on the same issue. Here we delay the request with a waitUntil guard. A permittedIf guard can be an option, too.

14.9.3 Evaluation

VoteConnector is an example of a programmed connector. It uses its own methods and instance variables to keep track of the votes. Our implementation extended the example described in [MU97] because it allows concurrent votes on different issues. It illustrates complicate interaction behavior with timing constraints between an arbitrary group of participants. The participants are constrained from behaving malevolent like voting twice (rule 4). The anonymity of the vote can be guaranteed because the rule (3) store the results separate from the voters.

14.10 The Sleeping Barber

14.10.1 Description

The sleeping barber example [BA82] features an arbitrary number of customers that need a haircut once in a while. These customers wait in a waiting room that is limited in size. When there are no customers around the barber sleeps. New customers wake up the sleeping barber. The barber invites the customer who waited the longest into the barber room. He cuts the customer's hair to the size that the customer desires. After sending the customer out, the barber checks for the next customer.

Coordination Aspects

This example represents a producer-consumer problem, where the barber "consumes" customers. The waiting room represents a customizable buffer of fixed size. Customers should be treated fair. Note that when the barber finished a customer, he must actively check, whether there are customers in the waiting room. If no customer is waiting the barber sleeps and the next customer has to wake him up. Therefore, the example features the two policies of polling and of notification.

The example contains the waiting room as explicit entity that represents a buffer. Note that we prefer to use the message queues of active objects instead of an explicit buffer entity. But we tried to stick to the description as close as possible therefore we introduced an active object playing the role of a waiting room.

14.10.2 Solution

In our solution, the customers have their own live cycle, they decide when to go to the barber. Then they try to enter the waiting room. If the waiting room is full, they are rejected by a connector and live on their lives (the hair is constantly getting longer) and try to go to the barber later. The public interface of the HairyGuy class:

HairyGuy	
live	Live and check if hair is too long.
growHair	Models the growing of the hair.
goCutting	Go to the waiting room of the barber.
enter	Enter the barber room.
sit	Sit on the barber chair.
desiredLength	Return the desired length of the haircut.
leave	Leave the barber room.

The customer's methods call each other in order to implement the customer's live cycle (see figure 14.2). Note that the methods involved in the interactions with the barber and the waiting room are called by connectors.

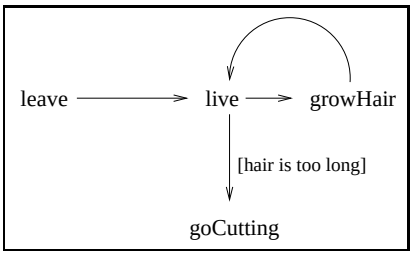

Figure 14.2: The self calling graph of a "hairy guy".

The methods of the barber call each other in one chain that represents the complete hair cutting process (see figure 14.3). A connector will connect the chain to a cycle.

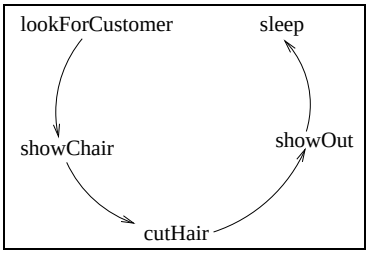

Figure 14.3: The self calling graph of a barber.

The waiting room is implemented as a first-in first-out buffer of limited size. Here is its public interface:

WaitingRoom	
newWaitingCustomer: h	A new hairy guy h enters the waiting room.
whoWaitedLongestEnters	The one who waited the longest enters the *barber room*.
notEmpty	Return whether there are waiting customers.
notFull	Return whether there is place for another customer.

The SleepingBarberConnector glues together the hairy guy (role customer) the waiting room (role waitingRoom) and the barber (role barber). Furthermore, it uses the default role connector to refer to its own state and operations. The SleepingBarberConnector implements all the interactions that happen outside the barber room (for example the interactions with the waiting room and the waking up of the barber). The connector is designed for one barber and one waiting room but an arbitrary number of customers.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) instanceVariableNames: 'innerConnector customerEntered ' ;
( ) withBehavior: '
(1)   customer goCutting. implies
( )     waitingRoom newWaitingCustomer: customer_select_REC. endRule
(2)   customer goCutting. permittedIf waitingRoom notFull. endRule
(3)   customer methodWasForbidden: m. impliesLater customer_select_REC live. endRule
(4)   customer goCutting. impliesLater barber lookForCustomers. endRule
(5)   barber lookForCustomers. implies  waitingRoom whoWaitedLongestEnters. endRule
(6)   barber lookForCustomers. waitUntil connector customerLeft. endRule
(7)   barber showChair. waitUntil connector customerEntered. endRule
(8)   customer enter. implies
( )     connector barberRoomWithCustomer: customer_select_REC
( )                                barber: barber.
( )     connector customerEntered: true. endRule
(9)   barber showOut. implies connector barberRoomWithoutCustomer.
( )     connector customerEntered: false. endRule
(10)  barber sleep. permittedIf waitingRoom empty. endRule' ;
( ) installAtName: #SleepingBarberConnector
```

Rule 1 When a customer enters the waiting room, the waiting room is notified to host the customer.

Rule 2 This is only allowed if the waiting room is not full.

Rule 3 If it is not allowed the methodWasForbidden: exception is caught here, and the customer continues its daily live.

Rule 4 If the customer successfully went to cut the hair, this means that (s)he entered the waiting room. Therefore, the barber is waken up to look for customers.

Rule 5 When the barber looks for customers the waiting room is told to bring forth the next customer.

Rule 6 The barber should not look for new customers until the one he treated last is gone. This prevents that the barber is waken up although it is not sleeping but treating a customer.

Rule 7 The barber should not show a new customer the chair until the customer entered the barber room. This rule and rule 6 synchronize the activity of the connector that implements the behavior inside the barber room (see later) and the one that implements the behavior outside (this connector here).

Rule 8 When a customer enters the barber room, the connector dynamically creates a special connector of the class BarberRoomConnector for the behavior there. Furthermore, in order to synchronize with this new connector an instance variable is set marking that the inner connector is handling the customer and the barber now (compare with rules 6 and 7)

Rule 9 The method barberRoomWithoutCustomer destroys the inner connector which duty is done when the barber has finished the haircut of a customer. Furthermore, the synchronizing instance variable customerEntered is reset.

Rule 10 The barber is only allowed to sleep when the waiting room is empty. Note that it is the barbers own live cycle that will send him to sleep after he treated a customer (see figure 14.3).

The SleepingBarberConnector controls the interaction that takes place between customers and the waiting room and it controls the live of the idle barber. When the barber is busy with a customer, a dynamically created inner connector of class BarberRoomConnector takes over the responsibility of the barber and this particular customer. It follows the live cycle of the barber and triggers the corresponding customer behavior. The BarberRoomConnector knows the roles barber and customer. Note that the connector is not designed for role groups since there is never more than one customer or barber inside the barber room.

```
( )MetaConnector new
( ) superclass: Connector ;
( ) withBehavior: '
(1)   barber showChair. implies customer sit. endRule
(2)   barber cutHair. implies customer desiredLength. endRule
(3)   customer desiredLength. implies customer hair: result. endRule
(4)   barber showOut. impliesLater customer leave. endRule' ;
( ) installAtName: #BarberRoomConnector
```

Rule 1 When the barber shows the chair for the customer, the customer will sit on it.

Rule 2 When the barber cuts the hair, the customer's desired length is requested.

Rule 3 The return value of the desired length is used for the new hair length of the customer, thus his/her hair is cut to the desired length.

Rule 4 When the barber shows the customer out, (s)he will eventually (asynchronous propagation) leave.

The two connectors glue together the live cycle of the customers and the barber. The figure 14.4 shows the state transitions (method executions) which are triggered by the live cycle of the barber and the customers themselves and it shows the transitions that are triggered by rules of the SleepingBarberConnector connector. Transitions belonging to the live cycle are represented as arrows that leave the state representation (a box). However, if a dashed arrow points to the starting of the transition arrow then the transition is caused by a propagating rule. The figure refers to the rules by the number we used in the declaration of SleepingBarberConnector before.

The behavior *inside* the barber room is modeled by the BarberRoomConnector. The figure 14.5 shows the state transitions of the barber and its customer. It also shows which state transitions belong to the live cycle of the barber and the customer, and which ones are triggered by the connector.

Note that the individual live cycles presented in the figures 14.2 and 14.3 can still be identified in the figures 14.4 and 14.5.

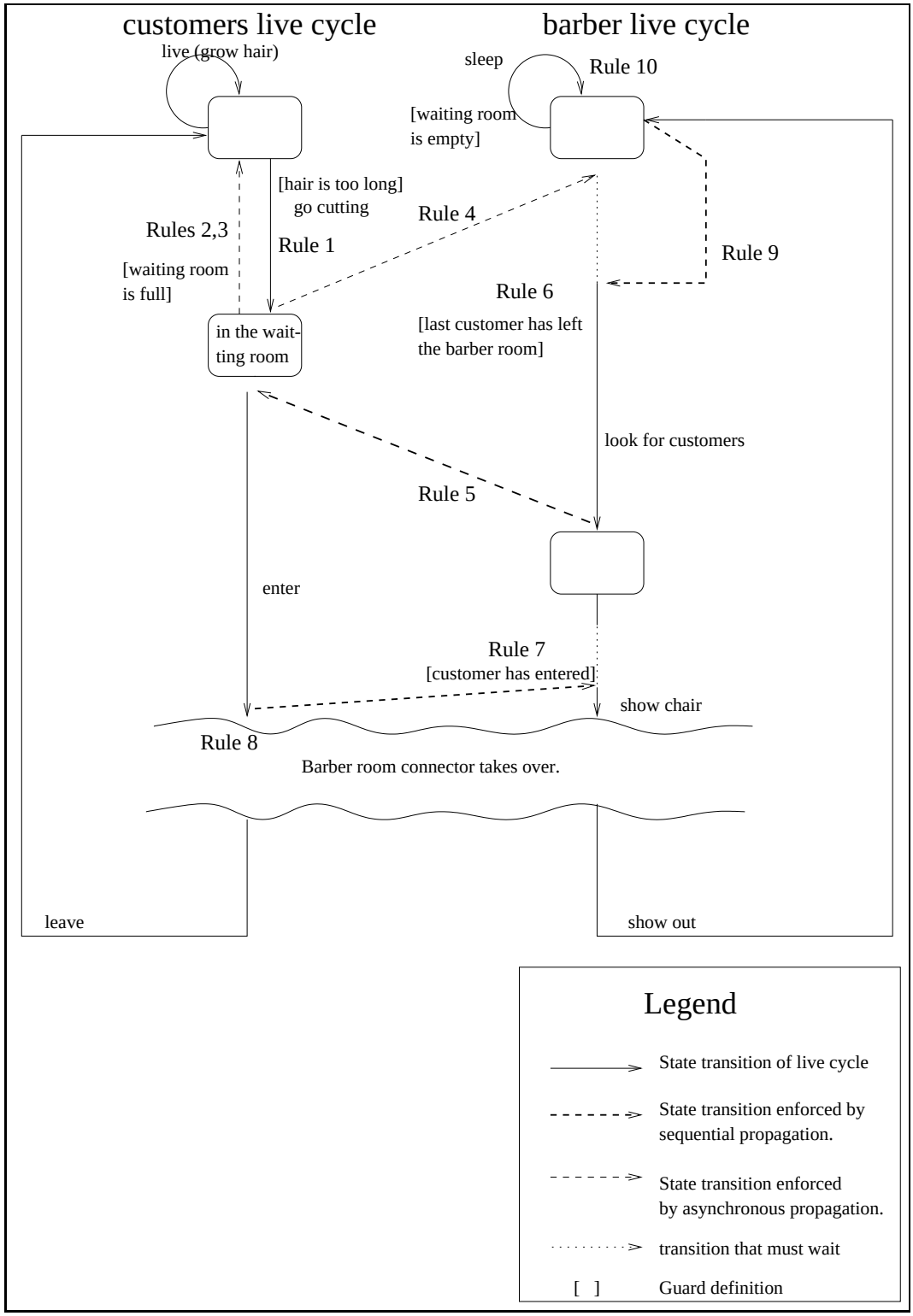

Figure 14.4: The different state transitions outside of the barber room.

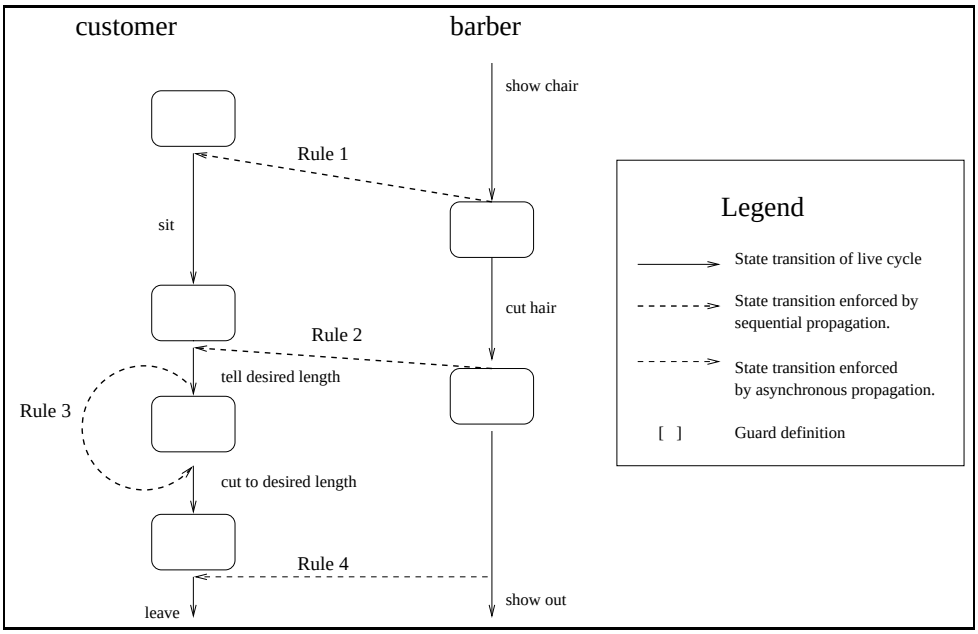

Figure 14.5: The different state transitions inside the barber room.

14.10.3 Evaluation

The example implementation uses exception handling, dynamic creation of connectors and connector programming. We tried to stick closely to the description of the example, in order to encounter limitations of FLO/C. This is the reason why the example uses a large connector defining 10 rules.

The example illustrates the trade-off between responsibilities of the connectors and the components (see section 7.4). The components have a simple self-call graph and the connectors connect these graphs to enforce complex system behavior.

After this final and most complex example, we will summarize our example implementation work in the following section.

14.11 Evaluation of the Examples

The coordination examples were programmed in FLO/C to evaluate the model's expressive power. We implemented eleven toy examples partially taken from traditional and recent coordination literature. FLO/C programming was able to solve all problems and even some extensions to the problems. The examples were implemented in parallel to the development of the model. This helped us to identify and address problems and limitations of the early FLO/C model. Special features like the exception mechanism or connector programming (see section 4.5) were introduced to FLO/C because they turned out to be handy when implementing examples. The examples' implementation revealed fundamental problems like how to identify active objects (see section 7.4) that implies further research. We claim that the implementation of the example proves that FLO/C enables a programmer to implement complex object oriented coordination without expert programming skills.

Example overview. There are a lot of recurring themes in the examples of the previous sections. Most examples use specificators which enhances the run-time flexibility. The solutions are indepen-

dent of the number of participants, participating active objects can be added and removed on the fly. The solutions use asynchronous and sequential message flow to implement different styles of client-server relations. Throughout the examples sequential operators are used to connect together critical actions and asynchronous propagation is used to request the actions. The following table presents the most important features of each example. It shows what coordination problem is particularly interesting in the given example and what special purpose lead to the implementation of the example.

Example	Coordination problem	Special purpose
1) Vending machine.	Multi-object constraints: The parts of the machine can be accessed concurrently.	Our solution demonstrates the use of stateless connectors.
2) Synchronized movements.	Multi-object joint actions. Pessimistic transaction.	A small and typical example.
3) Unstable server.	Components can refuse the collaboration.	Here we present an application of FLO/C's additional exception mechanism.
4) Decrementor.	Complex asynchronous message flow.	The solution can be used to implement workers with heavy internal communication.
5) Workers and tools.	Mutual exclusion on shared resources.	The solution demonstrates explicit locks and relative roles.
6) Binary adder.	Asynchronous push-flow with splitters and mergers.	The solution presents a nested object hierarchy with composite objects and demonstrates the inheritance of composite object classes.
7) Dining philosophers.	Mutual exclusion on several shared resources and fairness.	Providing a solution to this famous example is "a must".
8) Workers-administrator.	Dividing tasks to workers thereby supporting different strategies.	The solution shows the usefulness of user-defined specificators.
9) Electronic vote.	Constrained proactive behavior. Deadlines and anonymity.	Our solution demonstrates connector programming when registering time and voters.
10) Sleeping barber.	Producers and consumers connected by a customizable buffer of fixed size. Two valid protocols at the same time: notifying the consumer and polling of the consumer.	Our solution presents dynamic creation and destruction of a specialized connector. It also demonstrates the responsibility tradeoff between connectors and components.

Figures of the example implementation. To complete the presentation of the examples we show occurrences and ratios of the FLO/C connectors and component classes used in the eleven example implementations.

Connectors	
connector classes	31
using keyword connector	10
rules	125
roles	79
rules per connector	4.0
roles per connector	2.5
consequences per rule	1.2
selectors per role in precondition	1.7

Components	
total number of component classes	39
composite object classes	11
components per composite	2.3
connectors per composite	1.9
connections per composite	4.1

The tables show that almost every third connector contains rules that refer to the connector itself by using the keyword connector (see section 4.5). Mostly these connectors use methods to access their own state that represents the state of the interaction. The frequent use of this feature justifies its introduction to FLO/C.

Thanks to the group management of FLO/C, the average number of rules per connector is low and the average of roles even lower. Only few rules use more than one consequence message per rule.

The average number of selectors per role in the preconditions of a connector is a direct speed-up factor in the rule lookup optimization we implemented (see section 13.2).

The average composite object encapsulates two other active objects that are connected by an average of two connectors (an interface connector and one for the inner behavior). The connectors attach to the components through an average of four connections.

Part IV

Finale

Chapter 15

Related Work

The FLO/C model is inspired by different areas of computer science. Its main goal is *multi-object coordination of active objects*. Thus FLO/C uses a simple active object model to express concurrent activities and it coordinates the active objects with explicit connectors. On the other hand FLO/C postulates the separation of interaction and computation. This is well-known in the *architectural design* community. FLO/C *implements* architectural design decisions. By bringing together these different aspects FLO/C differs from all previous approaches. However, we will discuss individual differences between FLO/C and of some of the latest approaches in related areas.

15.1 Coordination

15.1.1 Explicit Entities for Multi-Object Coordination

Beside FLO/C, several languages propose rule based declarations to coordinate concurrent objects [FA93, MU97].

Synchronizers. The synchronizers proposed by Agha and Frølund [FA93] have many similarities to FLO/C. The synchronizers express coordination patterns in the form of *multi-object constraints*. They support the separation of concerns and support the reuse of coordination code. The constraints ensure atomic execution of methods. Constraints can use the state of the synchronizer and change it. However, our model supersedes the model of synchronizer. First the expression of the coordination is no longer limited to the state of the connector itself. A connector *enforces* new behavior to the objects by *invoking* participant object services. Second, a connector is a dynamic run-time entity that can be dynamically created and destroyed. Third, a connector can to refer to *groups* coordinated objects and supports dynamic addition or removal of participants.

The coordination language facility (CLF). CLF [AFP96] is a rule based language to coordinate distributed objects. It is built as an additional coordination layer on top of CORBA [Obj91]. CLF introduces *coordinators* that negotiate between participating objects about abstract token productions and consumptions. The negotiation uses three phases (inquiry, reservation and confirmation/cancelation) which finally leads to coordinated execution of different actions in the participants. First the coordinator *inquires* if a participant holds a token that occurs on the left hand side of a rule. Each participant proposes a set of actions to remove the token. Then the coordinator asks the participant to *reserve* some of these actions. Finally when the coordinator has achieved the necessary

reservations to fulfill the rules, it confirms the reservations (and cancels the unnecessary reservations). Confirmed actions are guaranteed to execute, deleting the corresponding tokens. This leads to the placing of compensating tokens as specified in the right hand side of the rules.

CLF differs from FLO/C in that its coordination support is process oriented. CLF eases the implementation of workflow and not of concurrent programming in general. CLF supports long-lived inquiries. Its coordinators do not impose new state on the world like FLO/C's connectors. Furthermore, CLF does not have the benevolent liveness properties of FLO/C (see section 9.4).

Coordination policies. The coordination policies proposed by Minsky and Ungureanu [MU97] are a recent approach and most similar to FLO/C. They use explicit entities to control the message passing between groups of agents. Rules describe constraints over the message passing and enforce state changes in agents proactively. Although coordination policies are intended for use in distributed systems they do not address low-level usability responsibilities but coordinate at a high level of abstraction (just like FLO/C).

However, the coordination policies lack of a simple *rule composition* mechanism as proposed in FLO/C (see section 4.3). Furthermore, they do not offer a way to compose nested *object hierarchies* like composite objects do (see section 6).

15.1.2 Factoring Out Per-Class Coordination

All approaches of this category have recognized the advantages of factoring out synchronization and coordination code. However their coordination abstractions affect only one class at a time.

Generic Synchronization Policies (GSP). GSPs [McH94] improve the reusability of interaction code (the policies) as well as the code of the objects. Policies are generic, they can be applied to any class. This is achieved by using parameters for message selector groups (like reading selectors and writing selectors). Nevertheless, a policy is always applied to one single class. It is not possible to connect objects of different classes through GSP. GSP controls *intra*-object concurrency. Note however that there is a recent approach [SL97] which reified GSPs as first class entities. But even there GSPs do not feature high level multi-object coordination constructs like synchronized multi-object joint actions.

The D language. The D language [LK97] is an aspect oriented approach [Kic97] to coordination. Kiczales and Lopes observe code tangling in concurrent programming with JAVA. D aggressively adheres to syntactic separation of concerns by introducing three aspect languages. The first language expresses the basic functionalities of the system, the second one expresses remote access strategies and the third one expresses coordination of threads. A special tool called Aspect Weaver takes the programs written in the aspect languages and merges them to one executable program (in plain JAVA). D's coordination aspect language allows one to declare sets of mutual exclusive methods and a set of self exclusive methods per class. It expresses synchronization guards by featuring atomic pre- and post conditions to methods.

While aspect oriented programming bears much expressive power it lacks the dynamics of the FLO/C model. For example the D language cannot express run-time change of coordination policies. The D language lacks the notion of composition and it does not allow formal analysis. However, one of its aspect languages addresses remote access strategies. Such strategies are not yet featured in FLO/C.

15.2 Architectural Design

15.2.1 Formal Approaches for Architectural Design

Formal connectors. Formal connectors were introduced to describe how architectural components interact [AG94]. Objects specifications have ports. The formal connector defines roles and glue for these roles. All descriptions are expressed in CSP [Hoa85]. It is possible to validate formal connector architectures and prove that they are e.g. deadlock free. We used the idea of separating interaction and computation from the FLO model [DR97] which in turn is inspired by the formal connectors. The separation of concerns bears several advantages. An advantage is the ability to reason about the interaction properties of a system like for example the liveness without knowing the exact semantics of the computational behavior. This is of importance for the architectural design. In section 8 we formally introduce how interaction between components is modeled in FLO/C. Furthermore, the specification allows *reasoning* about the interaction behavior.

In contrast to formal connectors FLO/C supports the *implementation* of collaboration design. Another advantage of the separation of concerns is the improvement of the *reusability* of code. Of course, this does not pay off in the area of architectural design, because there the research does not deal with code. However, in FLO/C the reuse of code is demonstrated for example in section 6.2 where we can inherit composite object declarations. Furthermore, FLO/C provides constructs that allow the composition of high-level coordination abstractions like mutual exclusion and pessimistic transaction (see section 4.3). Such constructs are not included in the formal connectors.

Temporal Specification Object Model. TSOM (Temporal Specification Object Model) uses the propositional temporal logic (PLT) system [Ara95]. The interaction behavior of objects is described in terms of logical formulas. Similar to the FLO/C model, TSOM's method executions are atomic. The concurrency assumptions are the same as in FLO/C. TSOM allows the composition of objects.

For some operators there is a direct mapping between the two models: $(p \rightarrow \diamond q)$ is equivalent to p impliesLater q. Whereas the formalism of PLT allows verification (detection of conflicts in interaction specifications), it is limited in different ways (e.g. constraints cannot evaluate messages and composite objects have one set of constraints, instead of several connectors). Software reuse is not an issue in TSOM. Nevertheless, it could be interesting to translate verified TSOM specifications into FLO/C.

15.2.2 Connectors at Run Time

Different approaches have suggested ways to *implement* architectural design using components and connectors.

The FLO language. Ducasse introduced FLO [Duc97b][DR97] to provide programming support to factor out design decisions concerning *object interaction*. The FLO model implements dependencies between objects. It introduces explicit connectors to implement dependencies. They hold rules over the message passing of the participants the connector controls. FLO/C evolved from the FLO model. Therefore, most of the terminology of FLO/C is taken over from FLO. In fact FLO and FLO/C use a similar meta-level architecture to implement connectors and to control message passing (see section 10.4). However, FLO controls interaction in a purely *sequential way*. Therefore, it cannot be used to coordinate active objects.

The Chiron-2 (C2) architectural style. The C2 style [TMA+94] supports larger grained reuse and flexible system composition. It distinguishes between components and connectors. In C2 a configuration of a system of components and connectors is an architecture. Components communicate with each other asynchronously through connectors. The connectors are similar to "event busses" that propagate requests up and notifications down through an architecture. Each component declares which notifications and requests it sends and which requests and notifications it receives. Components are unaware of components placed lower in a C2 architecture. The connectors route, broadcast and filter events. They can define priorities for its connected components. A C2 architecture is message based, multi-threaded and assumes no shared address space.

C2 differs from the FLO/C model in many ways but in particular in the responsibility and the expressive power of connectors. C2 connectors do not enforce state changes of the components. They cannot be dynamically exchanged. Connectors on the same architectural level cannot cooperate. Recursive application of the C2 style (like composite objects of FLO/C) is not featured. Multi-object coordination is no goal of C2 therefore C2 provides no high-level coordination support (e.g. transactions).

15.3 Active Object Models

Our FLO/C model coordinates active objects. It does not introduce a new active object model but uses an ACTALK like approach to express concurrent activities.

ACTALK. ACTALK is a minimal open testbed for active objects [Bri89]. FLO/C uses a similar active object model as discussed in sections 2.2.1 and 11.1.1.

The object model ATOM. The ATOM model combines concurrency and object-oriented feature [Pap96]. A main issue is the reuse of code. The rich active object model (intra-object concurrency, abstract state, state predicates, state notification) gives ATOM much expressive power. On the other hand, the model does not factor out the inter-object communication into separate stand-alone objects (like connectors). Therefore, objects cannot be composed to build an object that fits in the ATOM model again. The model has no explicit support for multi-object coordination.

Chapter 16

Conclusion

With FLO/C we introduced an object oriented model for coordinating active objects. In contemporary concurrent programming (e.g. using JAVA), coordination is implemented using low-level constructs and/or declared by statements in the objects' class definitions. As stated in section 2.1 the programming of high level coordination tasks is tedious and error-prone. Furthermore, the code is tangled by coordination statements that are spread within domain-specific code.

High-level coordination support. FLO/C offers explicit, rule based connectors for coordination (section 4). FLO/C factors out coordination code into explicit coordination objects that collaborate using the abstraction of *synchronized multi-object joint actions* (section 2). FLO/C's rules are specified using 5 operators. Two operators support multi-object constraints (blocking and balking guards). Two execution ordering operators (push- and pull style) enable protected multi-object state changes and one operator enables light weighted communication (section 4.2). By incrementally adding rules FLO/C allows the straightforward enforcement of *conditional synchronization, mutual exclusion* and *communication* between groups of different kinds of objects. Thus it covers the coordination categories stated by Carriero and Gelernter [CG90].

Support for implementing interaction design. FLO/C divides programming in computation (done in active objects) and coordination (done in connectors). Thus it directly maps *architectural design* principles, and it enforces the separation of concerns (section 3.1). FLO/C offers the possibility to *implement* design decisions concerning the interaction between objects. *Composite active objects* allow the mapping of hierarchical designs and improve the scalability of the model (section 6). Connectors as explicit rule-based coordinators profit of the incrementability of rules. They collaborate through a uniform role fusion protocol (section 4.3).

Analyzing interaction design and implementation. FLO/C's formal specification allows the analysis of coordination behavior of a given FLO/C program. It enables us to formally specify the five operators of FLO/C and to prove their properties. Furthermore, we proved the deadlock freedom of any FLO/C program, and livelock freedom under certain restrictions (section 8).

Open Implementation. FLO/C is fully implemented using SMALLTALK (section 10) and it can be implemented in any object oriented language with an open meta-object protocol. The code is freely available at the author's web pages. We also implemented eleven non-trivial coordination problems

(section 14). Examples were taken from well-known as well as from recent coordination literature in order to demonstrate the expressive power of FLO/C's coordination mechanisms.

Additional contributions. In comparison to other high-level rule based approaches like synchronizers, FLO/C introduces additional abstractions by dealing with arbitrary object groups. It does not only constrain the components but enforce state changes in oder to ensure consistent global states. Furthermore, FLO/C is completely dynamic (section 4.4). It can establish and cancel connections at run time, coordination policies can be exchanged at run time and FLO/C allows new connectors to be created on the fly.

16.1 Future Work

Distribution and heterogeneity of active objects. We implemented the FLO/C model on a single processor machine, using this fact to simplify the reservation phase of the rule fusion (see section 11.2). Therefore, a future work will address real distribution. We believe that the formal FLO/C model presented in part II and distributed systems infrastructure such as CORBA [Obj91] can form a base 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. However, the FLO/C model needs be extend for distributed application. We need to add 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 (see section 2.2.2). An implementation of distributed FLO/C must implement the full negotiation of the reservation phase with its requirements (see section 8.3.2) and cannot shortcut the negotiation with implicit reservations (see section 11.2).

The handling of communication failures and roll-backs of synchronized joint actions also need considerable further efforts.

Architectural analysis. The formal specification of FLO/C in part II allows execution order analysis of a given set of rules. Future work could automate such analysis.

Another interesting future work could be an automatic translation of architectural design with formal connectors [Ara95] to FLO/C code, as well as query languages to prove properties of FLO/C examples (like the query language presented for formal connectors [NACO97]).

Language extensions. Another direction of future research could be the improvement of expressive power: finding additional operators, to refine the dependency managing policies (e.g. including priorities) and additional specificators to refine group management.

Like in FLO [DR97] we would like to introduce generic connectors, which rules are not restricted by explicit selector names, and which preconditions could contain advanced matching features like regular expressions.

Design principles to identify active objects and connectors. As seen in section 3.1 a functionality can be encapsulated into a passive object, an active object or a composite active object. The designer must consider the complexity of inner activities in order to select the proper object representation. Section 7.4 showed that it is not easy to decide how much inner activity an active object can bear and when to break the functionality into components of a composite active object.

Another design decision is how sets of rules should be grouped into connectors. In some examples (see section 14) we tried to identify chains of actions and encapsulated the corresponding rules into single connectors. In other examples we separated connectors that encapsulate pure interaction rules from connectors that encapsulate only synchronizing rules.

We claim that future research on the identification of active objects and connectors can be fruitful not only for FLO/C programming but also for the area of architectural design and coordination.

Bibliography

[AFP96] J.-M. Andreoli, Steve Freeman, and Remo Pareschi. The coordination language facility: Coordination of distributed objects. *Theory and Practise of Object Systems*, 2(2), 1996.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of ICSE'94*, 1994.

[Agh86] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AHM96] J.-M. Andreoli, C. Hankin, and D. Le M'etayer. *Coordination Programming - Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[Ara95] Constantin Arapis. A temporal perspective of composite objects. In *Object-Oriented Software Composition*, pages 123 – 152. Prentice-Hall, 1995.

[BA82] Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997. ISBN: 0-13-476904-X.

[Blo79] Toby Bloom. Evaluating synchronisation mechanisms. In *Seventh International ACM Symposium on Operating System Principles*, 1979.

[Bri89] Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pages 109–129. Cambridge University Press, July 1989.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs*. MIT Press, 1990.

[CM84] K.M. Chandy and J. Misra. The drinking philosopher problem. In *Transactions on Programming Languages and Systems*, volume 6. ACM, 1984.

[CM93] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings ECOOP'93, LNCS 707*, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Coi87] P. Cointe. Metaclasses are first class: The ObjVlisp model. In *OOPSLA'87 Proceedings*, pages 156–165, October 1987.

[Coi90] Pierre Cointe. The classtalk system: A laboratory to study reflection in Smalltalk. In *OOPSLA/ECOOP'90 Workshop on Reflection and Metalevel Architectures*, 1990.

[CTN98] J. Cruz, S. Tichelaar, and O. Nierstrasz. A coordination component framework for open systems. Working Paper, IAM, University of Berne, 1998.

[DBFP95] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna. A reflective model for first class dependencies. In *Proceedings of OOPSLA'95*, pages 265–280, Austin, October 1995. ACM. RR–95–24.

[Dij72] E.W. Dijkstra. Hierarchical ordering of sequential process. In C.A.R Hoare and R.H. Perrot, editors, *Operating Systems Techniques*. Academic Press, New York, 1972.

[DR97] Stéphane Ducasse and Tamar Richner. Executable connectors: Towards reusable design elements. In *Proceedings of ESEC/FSE'97, LNCS 1301*, pages 483–500, 1997.

[Duc97a] Stéphane Ducasse. Des techniques de contrôle de l'envoi de message en Smalltalk. *L'Objet*, 3(4), 1997. Numero Special Smalltalk.

[Duc97b] Stéphane Ducasse. *Intégration réflexive de dépendances dans un modèle à classes.* PhD thesis, Université de Nice-Sophia Antipolis, 1997.

[FA93] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In *Proceeding of ECOOP'93, LNCS 707*, pages 346–360. Springer Verlag, July 1993.

[Fer89] Jacques Ferber. Computational reflection in class based object oriented languages. In Norman Meyrowitz, editor, *Proceedings of OOPSLA'89*, pages 317–326. ACM, October 1989.

[Frø96] Svend Frølund. *Coordinating Distributed Objects*. MIT Press, 1996.

[Gen81] Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. In *Software – Practice and Experience, vol.11*, pages 435 – 466, 1981.

[Gol97] Mickael Golm. Design and implementation of a meta architecture for Java. Master's thesis, Institut für Mathematische Maschinen und Datenverarbeitung der Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

[GR83] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HL85] D Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[How95] T. Howard. *The Smalltalk Developer's Guide to VisualWorks.* SIGS Books, 1995.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. In *Proc. of IMSA'92 Workshop on Reflection and Meta-Level Architecture*, 1992.

[Kic97] Gregor Kiczales. Aspect-oriented programming. ECOOP, 1997.

[Lea97] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[LH89] K. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, September 1989.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical report, Xerox Palo Alto Research Center, 1997.

[MA93] S. Matsuoka and A.Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*, chapter Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Language., pages 107–150. MIT Press, 1993.

[MC94] T. Malone and K. Crowston. The interdiciplinary study of coordination'. *ACM Computing Surveys*, 1994.

[McA95] Jeff McAffer. Meta-level programming with coda. In *Proceedings of ECOOP'95, LNCS 952*, pages 190–214. Springer-Verlag, August 1995.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[MU97] Naftaly H. Minsky and Victoria Ungureanu. Regulated coordination in open distributed systems. In *Proceedings Coordination '97*, pages 81–97, 1997.

[NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Applying static analysis to sofware architectures. In *ESEC '97*, pages 77 – 93, 1997.

[Obj91] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1991.

[Pap96] Michael Papathomas. Atom: An active object model for enhancing reuse in the development of concurrent software. *RR 963-I-LSR-2, IMAG-LSR*, 1996.

[Par95] ParcPlace-Digitalk. *VisualWorks User's Guide*, 1995.

[Riv96a] F. Rivard. *Smalltalk et Réflexivité*. PhD thesis, Ecole des Mines de Nantes, 1996.

[Riv96b] Fred Rivard. Smalltalk : a reflective language. In *Proceedings of REFLECTION'96*, pages 21–38, 1996.

[Riv97] Fred Rivard. *NeoClasstalk v1.2*. Ecole des Mines de Nantes and OTI, http://wfn.emn.fr/dept_info/perso/rivard/informatique/, 1997.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SKT96] S. Skublics, E. Klimas, and D. Thomas. *Smalltalk with Style*. Prentice-Hall, 1996. ISBN: 0-13-165549-3.

[SL97] Jean-Guy Schneider and Markus Lumpe. Synchronizing concurrent objects in the pi-calculus. In *Proceedings of Langages et Modèles à Objets '97*, 1997.

[TMA+94] R. Taylor, N. Medvidovic, K. Anderson, J. Whitehead Jr., J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A component- and message-based architectural style. *IEEE*, 1994.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP '88*, pages 55 – 77. Springer-Verlag, 1988.