uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Fedwa Laamarti
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Derivation of Component Designs from Global Specifications

TITRE DE LA THÈSE / TITLE OF THESIS

G. Bochmann
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

D. Amyot

D. Petrin

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

DERIVATION OF COMPONENT DESIGNS FROM GLOBAL SPECIFICATIONS

Fedwa Laamarti

A thesis submitted to the Faculty of Graduate Studies and Postdoctoral Studies in partial fulfilment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

November 2010

Copyright ©Fedwa Laamarti, Ottawa, Canada

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-74189-4
Our file Notre référence
ISBN: 978-0-494-74189-4

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

The use of distributed systems is growing more and more, given the advantages that they present, particularly the powerful computational capabilities that they provide. However, one challenge of distributed computing is ensuring that once the different system components are collaborating, their combined behavior will result in the behavior that was originally intended for the global system, without coordination conflicts.

This thesis deals with automating the process of deriving the components behaviors of distributed systems, starting from the description of that global system, defined as a UML Activity Diagram. The derived component specifications are produced as Activity Diagrams, and provided to the user in a graphical form. The generated components diagrams include the exchange of the necessary coordination messages. We also explain through many test cases the importance and purpose of these coordination messages. Besides, tests were conducted to verify that the obtained components designs correspond to the intended results.

Contents

Abstract **i**

Acknowledgements **vi**

List of Figures **vii**

Abbreviations **ix**

1 INTRODUCTION **1**
1.1 Background 1
1.2 Motivation 2
1.3 Objectives 3
1.4 Thesis Scope and Contributions 4
1.5 Thesis Outline 5

2 BACKGROUND LITERATURE **6**
2.1 Distributed Systems 6
2.1.1 Definition and uses 6
2.1.2 Distributed computing problems 7
2.2 Unified Modeling Language 9
2.3 UML Activity Diagrams 9
2.3.1 Action Nodes 11
2.3.2 Control Nodes 11
2.3.3 Initial Node 12
2.3.4 Decision Node 12
2.3.5 Merge Node 13
2.3.6 Fork Node 14
2.3.7 Join Node 14
2.3.8 Final Node 14
2.3.8.1 Flow Final Node 15
2.3.8.2 Activity Final Node 16
2.3.9 Object Nodes 16
2.3.9.1 Activity Parameter Node 17

2.3.9.2 Pin . . . 17
2.3.10 Interruptible Activity Region . . . 19
2.4 Extensible Markup Language . . . 19
2.4.1 XML Schema Definition (XSD) . . . 20
2.5 XML Metadata Interchange (XMI) . . . 21
2.6 Document Object Model (DOM) . . . 22

3 FROM GLOBAL ACTIVITY DIAGRAMS TO COMPONENT ACTIVITY DIAGRAMS 23
3.1 Derivation of Component Designs . . . 24
3.1.1 Related work . . . 24
3.1.2 Characteristics of the derivation algorithm . . . 25
3.1.3 The Use of UML collaborations . . . 25
3.1.4 The notion of weak sequencing . . . 26
3.1.5 Starting, Participating and Terminating Roles . . . 26
3.1.6 Roles and system components . . . 27
3.2 Language constructs for describing the system's behavior . . . 27
3.2.1 Primitive activity . . . 27
3.2.2 Invocation of a sub-collaboration . . . 28
3.2.3 Strong sequence . . . 28
3.2.4 Weak sequence . . . 28
3.2.5 Choice . . . 28
3.2.6 Strong while loop . . . 29
3.2.7 Weak while loop . . . 29
3.2.8 Concurrency . . . 29
3.2.9 Competition . . . 30
3.2.10 Interruption . . . 30
3.3 Overview of the derivation method . . . 30
3.3.1 Computing the Starting, Participating and Terminating roles . . . 31
3.3.2 Deriving the components behavior from a global specification . . . 32
3.3.2.1 Primitive action: <action>$^{(r)}$ . . . 32
3.3.2.2 Invocation of a sub-collaboration: <subcol>$^{(R)}$ . . . 32
3.3.2.3 Strong sequence: C1 $;_s$ C2 . . . 33
3.3.2.4 Weak sequence: C1 $;_w$ C2 . . . 33
3.3.2.5 Choice: C1[]C2 . . . 33
3.3.2.6 Strong while loop: C1 $*_s$ C2 . . . 34
3.3.2.7 Weak while loop: C1 $*_w$ C . . . 34
3.3.2.8 Concurrency: C1 || C2 . . . 35
3.3.2.9 Interruption: C1 | > C2 else C3 . . . 35
3.3.2.10 Competition: C1 <> C2 . . . 36
3.3.2.11 The algorithm . . . 36
3.4 Summary . . . 40

4 REQUIREMENTS OF THE DERIVATION SOFTWARE TOOL 41

4.1 Extending the notation of Activity Diagrams for describing collaborations 42
4.2 Definition of requirements 43
4.2.1 Input notation for specifying collaboration roles 43
4.2.2 The allocation of roles to system components 43
4.2.3 Transformation algorithm 44
4.3 Test Scenarios 44
4.3.1 Strong Sequence 45
4.3.2 Choice 46
4.3.3 Strong While Loop 55
4.3.4 Weak While Loop 62
4.4 Summary 68

5 DESIGN CHOICES FOR THE DERIVATION TOOL 69
5.1 Strong Sequencing v.s. Weak Sequencing 69
5.2 Tree Representation for Activity Diagrams 73
5.2.1 Binary Trees 73
5.2.2 Binary Tree Traversal 74
5.2.3 The Input XMI Format 75
5.2.4 Input Validation 76
5.2.5 Tree Construction 77
5.3 Role Computation 81
5.4 Behavior transformation 81

6 IMPLEMENTATION OF THE DERIVATION TOOL 83
6.1 Implementation Tools 84
6.1.1 Graphical Modeling Framework 84
6.1.2 XMLBeans 85
6.2 Overall Architecture 86
6.3 The Generation of Java Classes 87
6.3.1 UML Objects 87
6.3.2 Java Classes 89
6.4 Syntactic and Semantic Validation 89
6.5 Global Activity Diagram Tree Construction 91
6.5.1 Collaboration Followed by a Sequence 92
6.5.2 Choice 93
6.5.3 Concurrence 95
6.5.4 Loops 95
6.5.4.1 Strong Loop 97
6.5.4.2 Weak Loop 98
6.6 Calculation of Roles 98
6.7 Nested Activity Diagrams 100
6.8 Behavior Transformation 100
6.9 Summary 101

7 TESTS AND RESULTS **103**

7.1 Tests Performed on the Derivation Tool 103
7.2 Case Study 105
7.3 Summary 111

8 CONCLUSION AND FUTURE WORK **112**

8.1 Conclusions 112
8.2 Future Work 114
8.2.1 Interruption/Competition Construct 114
8.2.2 Weak/Strong Sequencing Priority 114
8.2.3 Automatic Layout 115

Bibliography **115**

A The Derivation Tool User Manual **119**

A.1 The Input Files 119
A.2 The Processing 120
A.3 The Output Files 120

Acknowledgements

It is a pleasure for me to thank the people who made this thesis possible.

First, I would like to express my sincere gratitude to my supervisor, Professor Gregor v. Bochmann, for his valuable guidance, his great suggestions and insightful conversations during the completion of this thesis. I thank him for always being available for consultations throughout my project. I also thank him for all the time he spent reading my writings and providing highly beneficial feedback.

Many thanks to the University of Ottawa for giving me the opportunity to pursue my research in such a great environment.

I deeply thank my parents for their continuous encouragement throughout my studies. I am obliged to them for being very supportive and for inculcating in me the love of seeking knowledge.

I am grateful to my husband for all his help and encouragement throughout this thesis. His support was an important factor in the success of my work.

I wish to express my special thanks to my sweet one year old daughter for her patience during the completion of my thesis.

I also thank all my family members, my friends and my colleagues for providing a loving environment for me.

List of Figures

2.1 UML Diagram Types 10
2.2 UML Action Notation 11
2.3 UML Control Nodes 12
2.4 Decision node example 13
2.5 Merge node example 13
2.6 Fork node example 15
2.7 ObjectNodes (taken from [16]) 16
2.8 Activity Parameter Nodes (taken from [16]) 17
2.9 Input Pins (taken from [16]) 18
2.10 Pins' Effect (modified from [16]) 18
2.11 Interruptible activity region example (taken from [19]) 19
2.12 DOM table (taken from [22]) 22
2.13 DOM Tree Example (taken from [22]) 22

4.1 Representation of starting, participating and terminating roles 43
4.2 Strong sequence test AD 47
4.3 Derivation of the strong sequence test AD for the component q 48
4.4 Derivation of the strong sequence test AD for the component p 49
4.5 Derivation of the strong sequence test AD for the component r 50
4.6 Choice test AD 51
4.7 Derivation of the diagram "Choice test case" for the component p 52
4.8 Derivation of the diagram "Choice test case" for the component q 53
4.9 Derivation of the diagram "Choice test case" for the component r 54
4.10 Strong loop Activity Diagram 57
4.11 Derivation of the strong loop AD for the component p 58
4.12 Derivation of the strong loop AD for the component q 59
4.13 Derivation of the strong loop AD for the component r 60
4.14 Derivation of the strong loop AD for the component st 61
4.15 Weak loop Activity Diagram 64
4.16 Derivation of the weak loop AD for the component p 65
4.17 Derivation of the weak loop AD for the component r 66
4.18 Derivation of the weak loop AD for the component q 67

5.1 Weak and strong sequence ambiguity 70
5.2 Weak and strong sequence : priority of strong sequencing 71

5.3 Weak and strong sequence : priority of weak sequencing 71
5.4 A Binary Tree 74
5.5 Depth First Search 75
5.6 An Activity Diagram example 79
5.7 Tree for the Activity Diagram above 80

6.1 Editing an Activity Diagram with UML2 Tools (GMF based editors) . . . 85
6.2 Overall Architecture of the Implementation of the Derivation Algorithm . 88
6.3 Tree Construction in the case of a Strong Sequence 94
6.4 Tree Construction in the case of a Choice 96
6.5 Tree Construction in the case of a Concurrence 97
6.6 Tree Construction in the case of a Strong Loop 99

7.1 Case study: order processing (modified from [10]) 106
7.2 Derivation of the order processing AD for Customer service 107
7.3 Derivation of the order processing AD for Finance 108
7.4 Derivation of the order processing AD for Order fulfilment 109
7.5 Derivation of the order processing AD for the Customer 110

Abbreviations

AD	Activity Diagram
CBA	Call Behaviour Action
CF	Control Flow
cim	choice indication message
DOM	Document Object Model
EMF	Eclipse Modeling Framework
fm	flow message
FN	Final Node
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
HTML	HyperText Markup Language
IN	Initial Node
MOF	Meta Object Facility
OMG	Object Management Group
PR	Participating Role
SR	Starting Role
TR	Terminating Role
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	EXtensible Markup Language
XSD	XML Schema Definition
W3C	World Wide Web Consortium

Chapter 1

INTRODUCTION

1.1 Background

Multiple types of applications developed for distributed systems are being used everyday. These applications are needed in real life, for example banking applications, or the ever-growing popularity of web-based applications. Indeed, the Internet is a distributed system immensely large, used by billions of people worldwide. Internet users receive information and services at will, without having any idea about the nature of the system providing them. In fact, an aspect of a distributed system is that it seems to provide its services by a unique component [1]. This joins the definition by Tanenbaum, who says "A distributed system is a collection of independent computers that appear to the users of the system as a single computer"[2].

In [3] a distributed system is defined by Coulouris as a system "in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages". Another definition provided by Lamport [4] is that "a distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages".

The usage of distributed systems is expanding more and more because of the numerous advantages that distributed computing presents.

1.2 Motivation

One of the important advantages of distributed computing is that the capabilities of the participating components put together, gives distributed systems the potential to be much more powerful than systems running on stand-alone computers.

Another benefit is parallelism. In distributed systems, components can be used concurrently to solve computational problems. Resources can also be shared between multiple processes and thus accessed simultaneously by users. For this, the coordination of actions assigned to different components may be necessary at times, and can be done when needed by message passing.

Distributed systems also allow scalability. They are suitable to serve a growing number of users, and to satisfy the requirements for increased performance [5]. This can be achieved by incrementally adding more resources, or faster processors to the system.

However, when modeling applications for distributed systems, the designer faces two major challenges. They are related to two aspects of distributed systems, namely: (1) the components participating in the system do not share any common clock, and (2) they do not share any common memory space [6]. The first aspect causes uncertainty about the order in which events located in different components happen [4]. So a specific component does not have visibility about when other components execute certain tasks. However, it could still be necessary to ensure coordination between components, as far as some specific actions are concerned, such as accessing a shared resource, or updating the same data.

As for the second aspect, the lack of shared memory, it has as consequence that components can only communicate through message passing. This leads us to the approach adopted by this thesis, which avoids discordances between the different processes, using message sending where appropriate. These messages can be of different kinds following the cases as we will discuss later.

1.3 Objectives

During the design of a distributed system, among the first stages is the modeling of the global system's behavior. Then comes the need to generate the separate designs of the different components that participate in the system. These have to be modeled in such a way that when each one of the components performs its assigned behavior, the global outcome would meet what was originally designed for the system as a whole.

However, once the global behavior of a distributed system is designed, deriving the components specifications for that system is not straightforward. In reality, if each component just executes its corresponding part as it was designed for the global system, there may be discordances between components' operations. The existence of such discordances has to do with the physical separation of the components, which means, as mentioned before, that there can be no global clock shared among them, and therefore no global ordering of their actions. So, to ensure this ordering of distributed events and eliminate possible conflicts between components, there needs to be more work done after the modeling of the system as a whole.

This thesis focuses on assisting the designer in the modeling of distributed computing systems, by automating a part of the design process, that is, the derivation of individual models for each component, based on the global specification of the system.

1.4 Thesis Scope and Contributions

The contributions of this thesis are the following:

- We present the ambiguity that may occur when we introduce the use of weak sequencing in the Activity Diagram of a distributed system in addition to the strong sequencing. We explain that this ambiguity does not exist for standard UML Activity Diagrams which only make use of strong sequencing. We show how this problem can be solved by prioritizing one type of sequencing over the other.

- We designed a way to automate the derivation process of the component designs of a distributed system from the global specification of that system. We built a software tool that takes as input an Activity Diagram representing the global system model, and derives the separate behaviors of all components that participate in the system, as well as generates the coordination messages that are necessary for the synchronization between the components of the system. This derivation tool then produces the respective Activity Diagrams representing the design for each of the system components. Particularly, this derivation tool allows for the global system's Activity Diagrams to be drawn graphically using an Eclipse plugin, and also produces graphical output Activity Diagrams for the system components.

- We show through multiple test cases, situations where the coordination messages exchanged between different components avoid design problems of the system, such as race conditions.

1.5 Thesis Outline

This thesis is organized as follows:

In Chapter 2, we present some background material that will be needed throughout this thesis. We give an overview of distributed systems and explain some of the design problems related to distributed computing. We also present some background information related to UML Activity Diagrams, since these will be the input as well as the output of our software tool, as we explain later.

In Chapter 3, we discuss the approach followed to generate specifications from the global design of a distributed system, for each component participating in that system.

Chapter 4 discusses the requirements of our software through multiple case studies, showing where the developed software can be of help to the designer, to avoid discordances between components. We included for every case, the global Activity Diagram, which was used as an input, as well as the derived Activity Diagrams which represent the intended output of the software tool.

In Chapter 5, we explain the conceptual choices that were done during the development of the software tool.

Chapters 6 is concerned with the implementation phase of our project. We first present the tools that we used for the development of our software tool. We then describe how the derivation algorithm is automated, as well as the assumptions under which the developed tool can be used. This tool takes as input an Activity Diagram that represents the global distributed system's design, and produces as output multiple Activity Diagrams, representing the specifications of the different components.

Chapter 7 presents a case study of using the tool, and explains the derived Activity Diagrams for the system components that are generated by our tool.

Finally, we use chapter 8 to conclude and discuss possibilities for future work.

Chapter 2

BACKGROUND LITERATURE

This chapter covers the background of the present thesis. First, it describes distributed systems briefly, as well as some problems related to distributed computing. And then, since Activity Diagrams present the main input to our program, this chapter introduces different UML notations used to build these diagrams. We also present some background material related to the Extensible Markup Language (XML) needed in our project.

2.1 Distributed Systems

2.1.1 Definition and uses

The term distributed systems is usually used when different processes communicate with each other through message passing. This could refer to processes that are located in physically separated computers that are part of the same network. It could also mean independent processes that are located on the same computer and that communicate with each other.

One of the uses of distributed systems is the achievement of the same shared goal, by several physically separated components. These components perform each their individual tasks in the global system, while interacting with each other to coordinate between these tasks. Distributed systems can also be used when there is no common goal; in this

case, the message passing between the components of the system allows, for example, the sharing of common resources such as system hardware, software, and data.

An important advantage of distributed computing over using a single supercomputer is reliability. If a computation is distributed over multiple computers, and one of them fails, then this will not affect the rest of the computers, so the problem concerning that computer can be solved individually, and taking less time to diagnose the source of the problem. Whereas, if a system uses a single powerful computer, and this computer fails, then the whole system will be down, unless of course, that computer is redundant, which will be costly because of the high performance of the machine.

But of course controlling multiple computers to achieve the system's goal comes at a cost and can be challenging. Indeed, if we want to take advantage of all these benefits of distributed computing, we need to be able to coordinate the tasks of these components in order to obtain the specification that was intended in the design of the system.

2.1.2 Distributed computing problems

Without the use of some specific mechanisms, there could be many problems that arise when we derive the specifications of each component participating in the system, from the global behavior of that system. Such problems include deadlocks, where a component waits indefinitely for the reception of a message. Another possible problem are race conditions. We say that there is a race condition [7] in a specification if there is a possibility that two events are executed in a different order than what was stated in the specification. For example, a message could be received by a component when this component was supposed to perform another task, and is not ready to receive that message. This is called an unspecified reception [8].

A mechanism that can contribute to resolving coordination conflicts between system components is message ordering [8]. A race condition can occur if messages, sent from a component to another, are delivered through the network in any order, possibly different from the order in which they were sent (out-of-order delivery). One cause for race could be that if two messages are sent by a component, then the receiving component is

designed to receive them in the specific order in which they were sent. So if it receives them in a different order then this can be the cause of an unspecified reception. This condition could be avoided by the use of a communication network with in-order delivery.

In the same context of message ordering, we can also dissociate between reception and consumption of messages at a given component. The component could impose a total ordering of messages for consumption, if it stores the messages at their reception, and then reorders them for consumption as needed. There is also the reordering of messages between sources [8], where a component stores messages received from each sending component in a different FIFO buffer. It may then choose from which buffer to consume the next message. Otherwise, there may be no ordering at all, in which case the components have no control over what message they will consume next, so components just consume messages in the same order in which they are received. In the latter case, the system is more prone to race conditions unless it makes use of other methods of coordination between components.

Another kind of coordination problem between system components could arise for example in the case of strong sequencing between two activities where multiple components collaborate. Strong sequencing between these two activities means that the first one should be totally completed before the second one could start. But the problem here is that the components starting activity 2 may not know whether the first activity is completed or not. The components that have this information are actually the terminating components of activity 1. Here, we need to introduce the sending of some messages, called coordination messages. These are messages that will contribute to the synchronization between the different components that play a role in the global system. The purpose of these coordination messages in this case, is to enforce the strong sequencing. These messages will be sent from all the terminating components of activity 1 to the starting components of activity2 [9]. Other types of coordination messages are used following the different cases where a conflict between system components may arise. This is what will be detailed in this work.

2.2 Unified Modeling Language

The Unified Modeling Language (UML), created by the Object Management Group (OMG), is a standardized modeling language for software systems, especially used for object oriented systems. It is a set of graphical notations, based on a single metamodel, that allows the description and design of software systems, by creating visual models and documenting them.

There are thirteen different diagrams in UML, categorized into three types as shown in Figure 2.1.

In what follows, we are going to focus our attention on one of the behavior diagrams which is the Activity Diagram, since this is the diagram that we use in our project.

2.3 UML Activity Diagrams

The activity diagram is a behavior diagram of UML, it can be used to model the dynamic behavior of a system. It describes the business process, the workflow or the procedural logic of a system.

Activity Diagrams provide a way to show the ordering of activities performed in a system, by using constructs such as sequences, choices and concurrency.

In a given activity diagram, every activity consists of a certain number of nodes, linked to each other by control flows or object flows [11]. These flows are represented by arrows indicating the direction of the flow in the activity diagram.

The nodes can be either object nodes, action nodes or control nodes.

- Action nodes: execute operations on control and data tokens that was sent to them, and then send control and data tokens to some other action nodes.
- Object nodes: they represent data passed between different action nodes, or tokens kept for a certain time before they move down the flow.

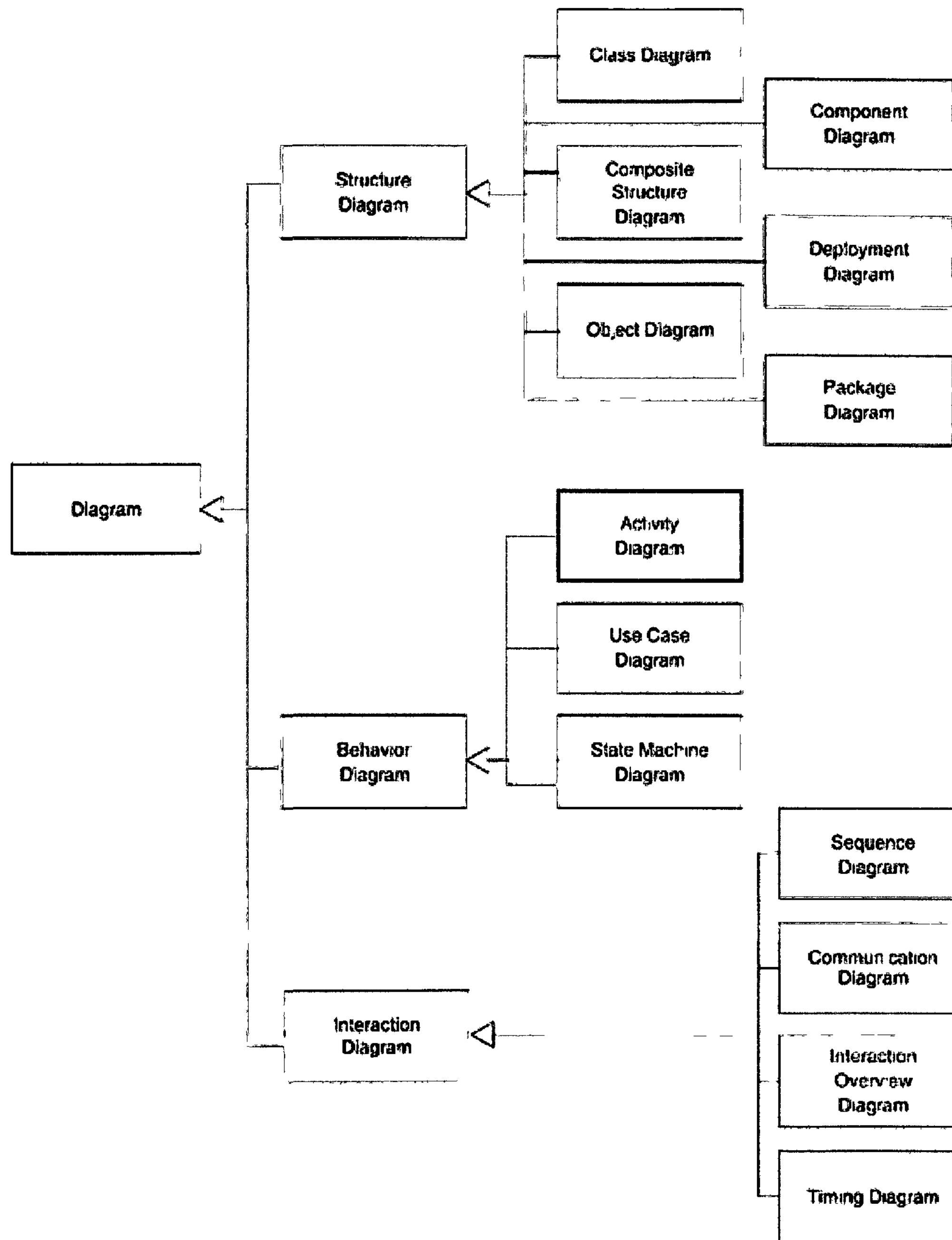

FIGURE 2.1: UML Diagram Types (corrected from [10])

- Control nodes: specify how the data or tokens will move forward through the diagram.

The nodes can be linked as we mentioned either by object flows, or by control flows.

- The object flows allow data to be sent from an action to an other action where it will serve as input.
- The control flows indicate the order of execution of the actions connected by them.

In the sections that follow, we will look in more detail at each of these elements that compose an activity diagram in UML.

2.3.1 Action Nodes

Actions are the basis for UML activity diagrams [12]. They are indeed the only elements in the activity diagram that have an executable functionality. This means that any behavior defined by the activity has to be divided into actions that will represent it. Actions contained in a certain activity can also call the behavior defined by another activity [13].

Actions are denoted by round-cornered rectangles. Their notation is as shown in Figure 2.2.

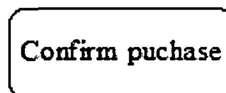

FIGURE 2.2: UML Action Notation

2.3.2 Control Nodes

The control nodes used in activities in UML 2, are as shown in Figure 2.3.

These different types of control nodes are described in the following sections.

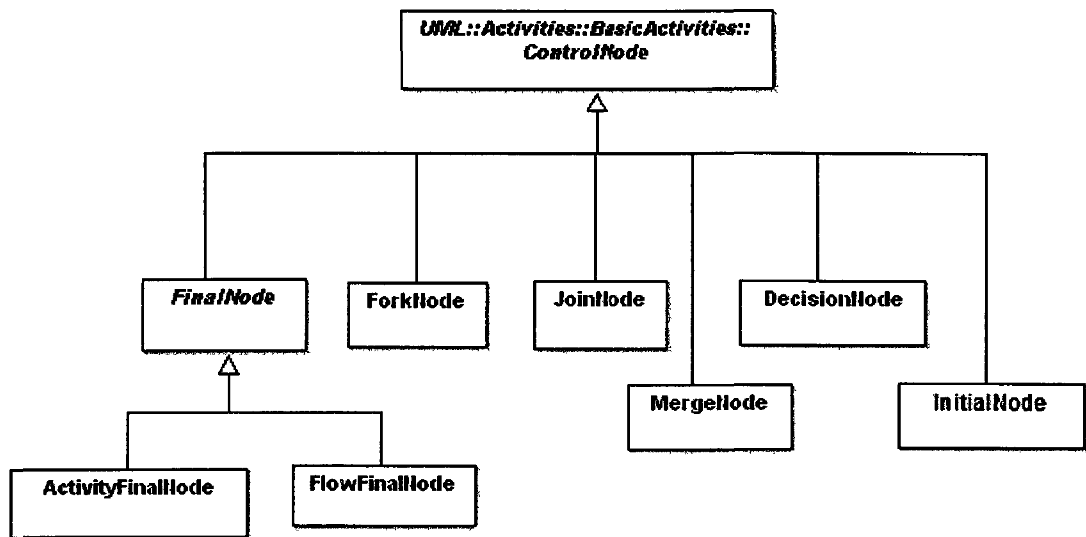

FIGURE 2.3: UML Control Nodes (completed from [14])

2.3.3 Initial Node

The initial node is the starting point of an Activity Diagram. An activity can contain one or more initial nodes. Because it is the starting point, an initial node can only have outgoing edges and cannot have any incoming edges.

The notation for an initial node is a filled circle. An example of the initial node can be found in Figure 2.6. The Receive Order, which is the action that comes after the initial node, is started just after the activity starts.

2.3.4 Decision Node

A decision node is a control node at which an incoming edge arrives and branches into more than one outgoing edge. It is a node in the activity where the choice is made to continue the flow in a specific direction, based on the evaluation of the conditions specified on the outgoing edges. These conditions are called guards.

Figure 2.4 is a simple example that shows the usage of a decision node.

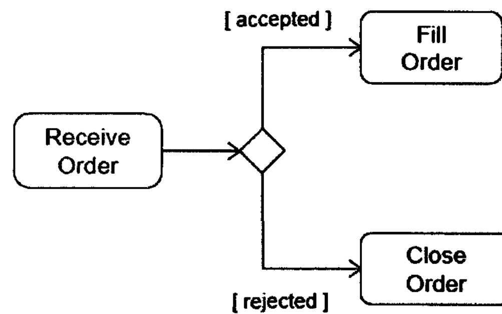

FIGURE 2.4: Decision node example (taken from [15])

2.3.5 Merge Node

This is a control node that is used to reunite alternate incoming flows. The Merge node's notation is the same as for the decision node, except that the merge node has multiple incoming edges, and only one outgoing edge, while it is the opposite for the decision node.

The example shown in Figure 2.5 shows a merge node used in combination with the decision node. The merge node allows the flow in the activity to reunite after it was split at the decision node following the action Receive Order.

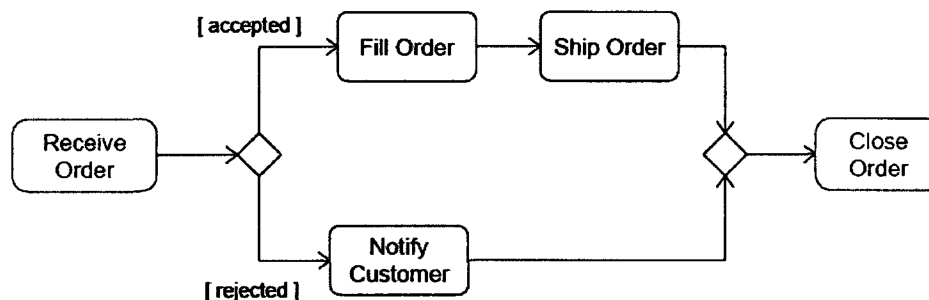

FIGURE 2.5: Merge node example (taken from [15])

2.3.6 Fork Node

The fork node, in an activity diagram, is the control node that allows activities to be executed concurrently. It can have only one incoming edge. It then splits it into multiple outgoing edges. The fork node is notated as a segmented line.

In the example in Figure 2.6, the actions that come after the fork node, that is Fill Order and Deliver Order on one side, and Send Invoice and Receive Payment on the other side, are actions that happen in parallel. It is acceptable by the semantics of the diagram that they happen simultaneously, or sequentially, or with some interleaving. So the occurrence of actions in one side does not affect or constrain the timing of the occurrence of actions in the other side.

This allows the modeler to specify that some actions have to occur in parallel, while still being open as to the ordering in which these actions are executed. This is a useful aspect of Activity Diagrams, that allows modeling concurrency in distributed systems.

2.3.7 Join Node

Join nodes are the control nodes that support concurrency in UML, along with the fork nodes. Join nodes combine multiple flows into a single one. They must have at least one incoming edge, and exactly one outgoing edge.

If we consider the example in the Figure 2.6, we see that after the two concurrent flows, we have a join node that takes both flows before letting the Close Order action happen. This is because the join node will cause the activity to wait until all the incoming flows have arrived at the join, and only then can the activity proceed with the outgoing flow.

2.3.8 Final Node

Final nodes are used to stop flows in an activity. A final node cannot have any outgoing edges, and can have any number of incoming edges. There are two types of final nodes in activity diagrams: flow final nodes and activity final nodes.

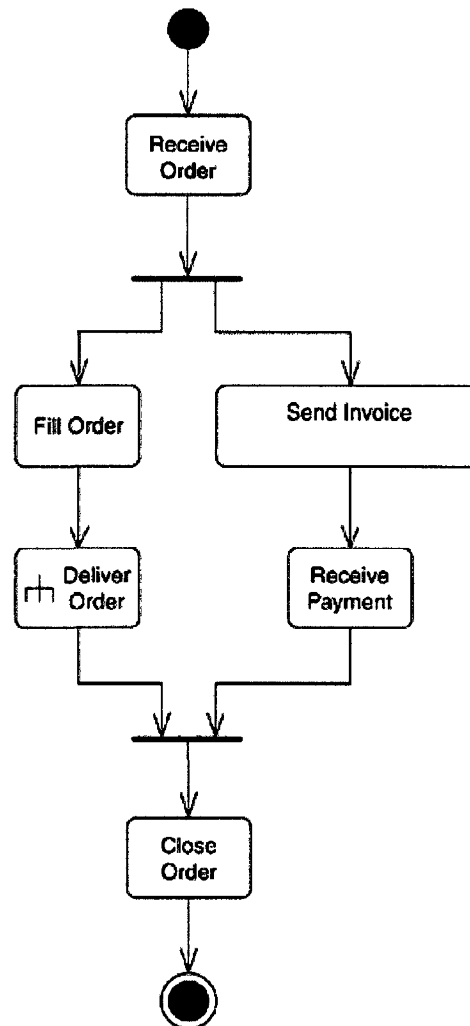

FIGURE 2.6: Fork node example (taken from [10])

2.3.8.1 Flow Final Node

It is used to stop any flows that come to it. The flow final does not terminate any other flow in the activity, that does not arrive at it.

2.3.8.2 Activity Final Node

This type of final node is used to terminate all flows in an activity. An activity may contain multiple activity final nodes. If any of them is reached, it stops all flows in this activity.

2.3.9 Object Nodes

Object Nodes are constituents of the UML activity diagram capable of holding and storing data temporarily. There are four types of Object Nodes as shown in Figure 2.7 in terms of functionality. The maximum number of values an object node can hold, called the upper bound, depends on its type. There are object nodes that can hold only a single token and there are others that can handle more than one.

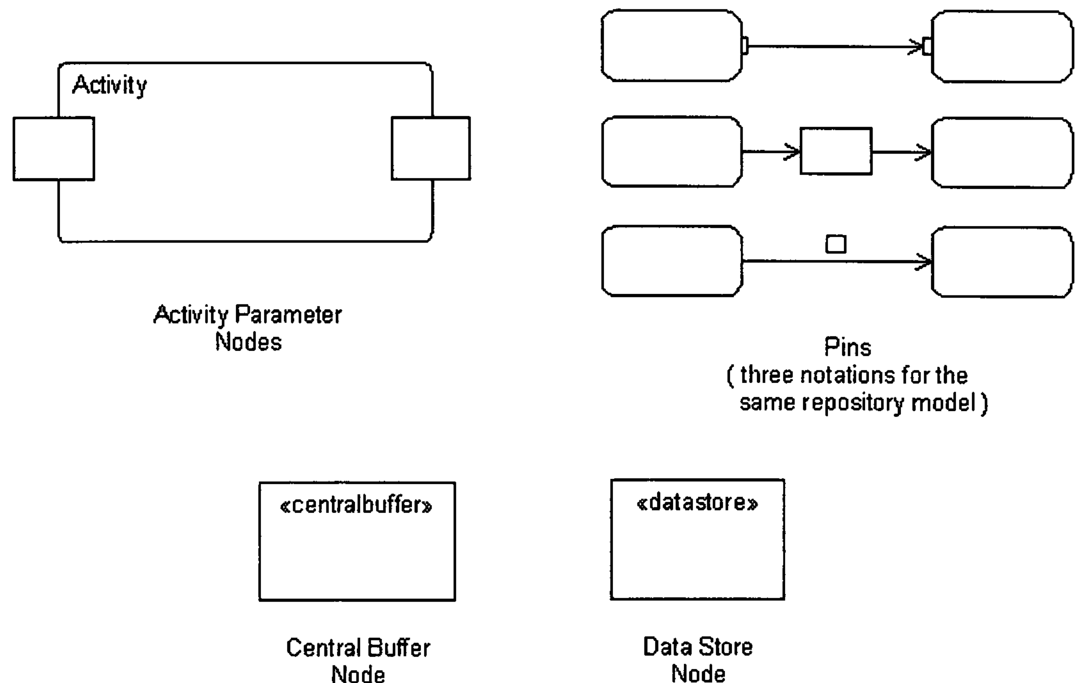

FIGURE 2.7: ObjectNodes (taken from [16])

2.3.9.1 Activity Parameter Node

Activity Parameter Nodes present parameters of the containing activity. These parameters specify the types of values that are input to the activity or output from the activity.

Figure 2.8 illustrates an example of parameter nodes in a partial activity. It has two input parameter nodes on the left and two output parameter nodes on the right. The input parameter nodes get their values simultaneously at the moment the activity starts while the first output parameter node that gets a value holds the received value until a value reaches the other parameter node.

Parameter nodes specify the type of values they can hold. The example in Figure 2.8 shows parameter nodes that can hold values of type 'Company' and 'Share'. If no type is specified, the corresponding parameter node can hold any type of values. In addition, parameter nodes may specify the state of the object that they can hold. In the example shown in Figure 2.8, the parameter node 'Merged Company' can only hold objects that are in the state 'New', and the parameter node 'New Shares' can only hold objects that are in the state 'Unsold'.

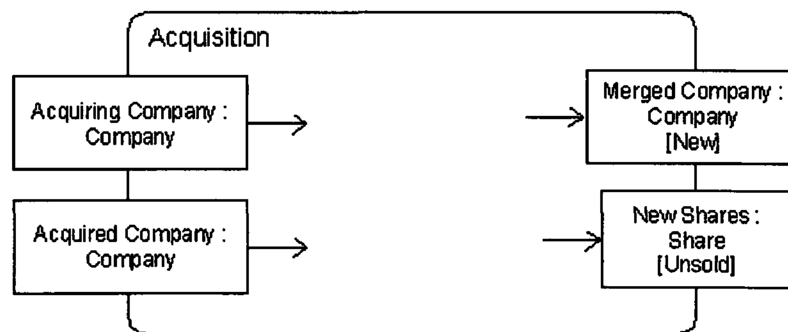

FIGURE 2.8: Activity Parameter Nodes (taken from [16])

2.3.9.2 Pin

Pins are object nodes where input values wait for each other in order to be passed all together to the action. Figure 2.9 illustrates an example with two input pins. In the

case one of the input pins receives a value, it holds it and waits for the other input pin to receive its value. Then the two values are passed to action which will make it start immediately.

A Pin can be described either only through the object moving through it, or also through the effect its actions have on that object. Such effect is one of the four values: CREATE, READ, UPDATE, DELETE. Figure 2.10 illustrates an example where 'Take Order' action creates an instance of order while the action 'Fill Order' updates that instance of order.

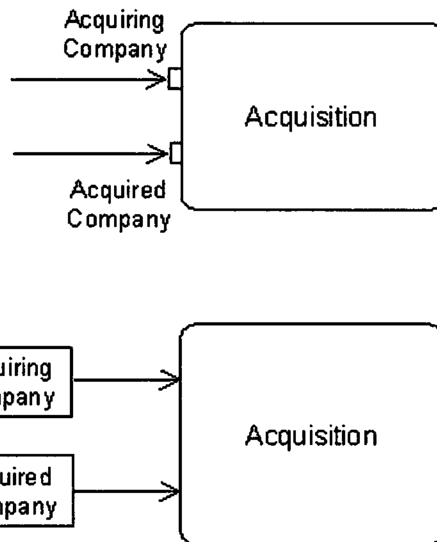

FIGURE 2.9: Input Pins (taken from [16])

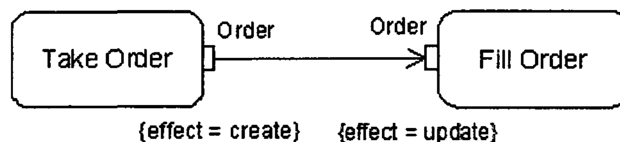

FIGURE 2.10: Pins' Effect (modified from [16])

2.3.10 Interruptible Activity Region

The Interruptible Activity Region is used when a group of action nodes may be interrupted during its execution. These action nodes are surrounded by a dashed rectangle, traversed by a zigzag line, called interrupting edge [17]. These actions are executed normally, unless the interruption happens, in which case the interrupting token traverses the interrupting edge, and all the actions and tokens in the interruptible activity region are terminated [18].

An example of an interruptible region is shown in Figure 2.11. The two actions "Retrieve Customer Number" and "Search Order Archives" are contained in the interruptible region. If the interruption "Cancel Search Request" occurs during the execution of these two actions, then their execution is aborted, and "Display Cancellation Message" is executed instead. Otherwise, the execution of these two actions continues normally and then the action "Display Results" takes place.

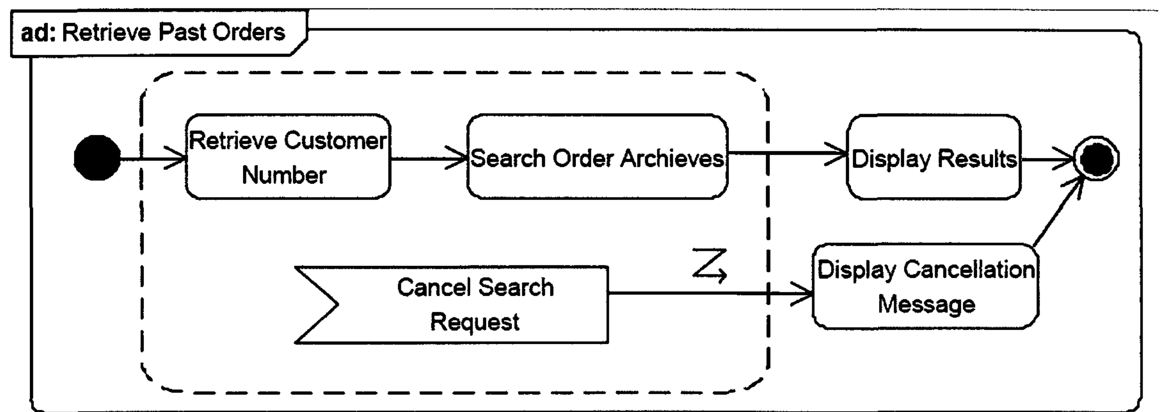

FIGURE 2.11: Interruptible activity region example (taken from [19])

2.4 Extensible Markup Language

The Extensible Markup Language (XML) was developed by a working group of the W3C [20]. It is a markup language designed to be self-descriptive in order to carry,

store and transport data. In the real world, computer systems and databases contain data in incompatible formats and XML provides a software- and hardware-independent way of storing and sharing data between different computer systems. XML's flexibility in defining tags is an important feature, facilitating data exchange between incompatible systems. It is amazing how quickly XML became a standard and how quickly this standard was adopted by a large number of software vendors.

W3C has recommended two notations for defining the structure of XML documents:

- Document Type Definition (DTD)
- XML Schema Definition (XSD)

It is the XSD that has been adopted to define the UML metamodel for the reasons explained in the next section.

2.4.1 XML Schema Definition (XSD)

An XML Schema Definition (XSD) is an XML encoded document that describes the structure of XML documents. It presents an XML-based alternative to the Document Type Definition (DTD). XML Schema became a W3C Recommendation in May 2001[21].

Both Document Type Definition (DTD) and XML Schema Definition (XSD) are industry standards, and both define the building blocks of an XML document. While DTDs are still used for the definition of the structure of many legacy XML documents, XSD is gaining support from the major XML software providers.

Although DTDs have the merit to use a very compact syntax, they still are limited when it comes to specifying the content inside the elements: everything is a string. XSDs, on the other hand, have the following merits over DTDs [21]:

- XSD supports primitive (built-in) data types (eg: xsd:integer, xsd:string, xsd:date, and so on) which makes it suitable to use in conjunction with other typed data, such as in relational data bases.

- XSD supports namespaces.
- XSD has ability to define custom data types, using object-oriented data modeling principles: encapsulation, inheritance, and substitution.

2.5 XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is an Object Management Group (OMG) standard for exchanging metadata information. It can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF). A standard way to exchange structured data between tools is the Extensible Markup Language (XML). The most common use of XMI is as an interchange format for UML models.

One important purpose of the XML Metadata Interchange (XMI) is to enable easy interchange of data between UML-based modeling tools and MOF-based metadata repositories in distributed heterogeneous environments. As part of model-driven engineering, XMI is also widely adopted as the medium by which models are passed from modeling tools to software generation tools.

Thus, XMI is based on three industry standards:

- XML: Extensible Markup Language, a W3C standard.
- UML: Unified Modeling Language, an OMG modeling standard.
- MOF: Meta Object Facility, an OMG language for specifying metamodels. MOF originated in the Unified Modeling Language (UML); the OMG was in need of a metamodeling architecture to define the UML. MOF is part of a four-layered architecture. It can be viewed as a standard to write metamodels.

The integration of these three standards into XMI allows tool developers of distributed systems to share object models and other metadata. XMI is an international standard since 2005.

2.6 Document Object Model (DOM)

The W3C Document Object Model (DOM) is a platform and language-neutral modeling paradigm for representing objects that are encoded as HTML, XHTML or XML documents. A DOM presents an XML document in the form of a tree of nodes allowing developers to navigate and search information efficiently, each node representing one of the building blocks of the XML document (element, attribute, etc).

Figure 2.13 illustrates a DOM tree built starting from the table in Figure 2.12:

```
<TABLE>
    <TBODY>
        <TR>
            <TD>Shady Grove</TD>
            <TD>Aeolian</TD>
        </TR>
        <TR>
            <TD>Over the River, Charlie</TD>
            <TD>Dorian</TD>
        </TR>
    </TBODY>
</TABLE>
```

FIGURE 2.12: DOM table (taken from [22])

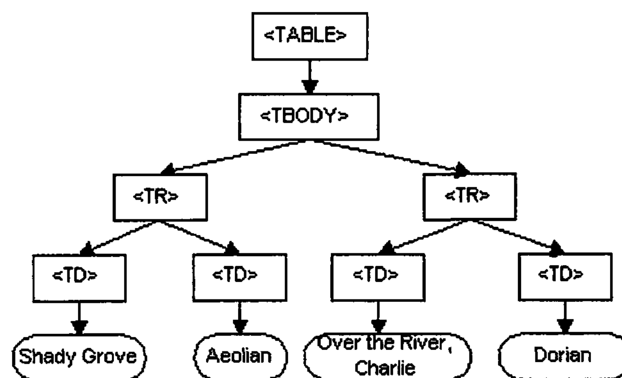

FIGURE 2.13: DOM Tree Example (taken from [22])

Chapter 3

FROM GLOBAL ACTIVITY DIAGRAMS TO COMPONENT ACTIVITY DIAGRAMS

During the design stage of distributed applications, the first concern is the definition of the system's description as a whole, to satisfy the specified requirements. The functions are thus defined globally, where different components will collaborate to achieve the system's specified behavior.

However, before the development stage of such a system can be started, it is necessary that the design of each component individually is specified, in such a way that it can perform its assigned functions while being physically separated from the other components of the system.

A derivation algorithm that generates the behaviors of different system components, from the global specification of the system, is proposed in [23]. We devote this chapter to a detailed description of this algorithm. The following chapters describe how this algorithm has been integrated into our software tool.

3.1 Derivation of Component Designs

3.1.1 Related work

The approach followed by the derivation algorithm in [23] to ensure synchronization between system components consists of introducing coordination messages exchanged between system components. This approach is similar to what was used in [9], where the sending and receiving of synchronization messages are introduced whenever there is a sequencing operator separating two subexpressions of a given service.

The derivation algorithm in [23] also adopts the concept of message buffering at reception. This means that whenever a message is received by some component in the system, this message is first added to a buffer and not necessarily consumed right away. Then, whenever one of the roles played by this component needs a specific message, that message can be consumed from the buffer, or it can be waited for if it was not received yet. A similar concept has been used in [24], where every SDL process has a single input buffer for the reception of messages. Then, in order to avoid deadlocks, an SDL save construct is introduced whenever a message can arrive at a component before it is ready to receive it. So in this approach, messages are put in the input buffer of a given component following their order of arrivals. Then, the "save" construct allows the reordering of these messages by the receiving component before they are consumed.

As far as the system's behavior representation is concerned, various notations for representing user requirements are detailed in [25]. In particular, this work describes UML Activity Diagrams as an "interesting way" to define the requirements of a system. We are particularly interested here in Activity Diagrams ; for a detailed comparison between different notations for representing system requirements, the reader is referred to [25]. Activity Diagrams show the dynamic behavior of a system, in terms of activities linked by constructs such as sequences, parallel constructs, choices and loops. In addition, Activity Diagrams allow the refinement of activities for a more detailed description of their behavior. This is done by defining sub-activities that decompose some activities previously defined as basic activities in the Diagram.

Activity Diagrams are also adapted to describe the global behavior of a system, where multiple components participate. In this context, the derivation algorithm adopts the UML Activity Diagrams for defining the dynamic behavior of the system, but replaces the activities by UML collaborations as explained later in section 3.1.3.

3.1.2 Characteristics of the derivation algorithm

The derivation algorithm [23] proposes a transformation from the global behavior of the system as a whole, to the behaviors of the different components that participate in that system. It also ensures, during this process, the coordination between the work of the different components of the system, by introducing the exchange of some messages, and keeping the number of these messages as small as possible.

No formal proof of correctness of the derivation algorithm is provided in [23], nor is it provided in this thesis. However, as it was discussed in [23], the sending and receiving of coordination messages related to a given situation are all defined as part of the same rule of the algorithm, as we will see in more detail in section 3.3.2. This gives one some assurance that the algorithm avoids deadlocks as well as unspecified receptions when the components collaborate to achieve the global behavior initially specified by the user.

In order to represent the specification of the global behavior of the system, the algorithm adopts the sequencing operators used in the notations of UML Activity Diagrams.

3.1.3 The Use of UML collaborations

In Activity Diagrams, the actions that compose an activity are performed by one specific component in the system. However, in the context of a distributed system, it is more interesting for the designer to be able to describe the system in terms of actions that are possibly performed by multiple components, rather than a single component.

In order to allow that, the actions in the Activity Diagrams are replaced here by UML Collaborations [26]. This way the system is modeled by an extended version of Activity Diagrams, in a sense that it makes use of collaborations instead of simple actions.

3.1.4 The notion of weak sequencing

An important aspect of this algorithm is that it takes into consideration the weak sequencing [26]. The weak sequencing between two collaborations means that each component that participates in these collaborations will start by performing its assigned actions related to the first collaboration. Then once it has completed its part there, it can immediately proceed with the second collaboration, without waiting for the completion of the first one by the other participating components. If this component did not have any actions to perform in the first collaboration, then it will just immediately start with the second one. This saves time by letting components proceed with their roles in the next collaborations as soon as they have completed the actions of the previous one. This way it allows some sort of parallelism in performing tasks by the components. It also saves exchanges of some coordination messages as we will see later in this chapter.

3.1.5 Starting, Participating and Terminating Roles

The first information needed by the algorithm are the starting, participating and terminating roles of each collaboration in the model. The definition of these roles is as follows:

- The **Starting roles** (SR) of a collaboration are defined as the roles that are responsible for the set of actions that initiate a collaboration, such that no other action in that collaboration is executed before any action of that set.
- The **Participating roles** (PR) of a collaboration are the roles that perform some action in that collaboration.
- The **Terminating roles** (TR) of a collaboration are the roles that are responsible for the set of "last" actions in a collaboration, such that no other action in that collaboration is executed after any action of that set.

3.1.6 Roles and system components

With the introduction of collaborations in the use of Activity Diagrams for modeling, and since many roles can participate in a given collaboration, it becomes necessary to specify which roles belong to which system components. In this context, we note that one system component can involve multiple roles, but each role belongs to one and only one system component. This allocation of roles to specific components is not specified by the derivation algorithm, but this information is important for its functioning and should be provided by the system architect.

An allocation function, Alloc(), is introduced here and defined in the algorithm as a function that returns, for a given role r, the system component to which r belongs.

3.2 Language constructs for describing the system's behavior

The different constructs that are considered by the algorithm are mostly similar to those that we find in Activity Diagrams. The Activity Diagrams here are assumed to be well structured similarly to the notion of structured programming. These diagrams have thus a single start point as well as a single end point. Another assumption made here is about the constructs involving decision making. These are the choice construct as well as the while loops. In these cases, local choice is assumed, which means that a single component is responsible for making the choice.

The different constructs, their notation as written in [23], as well as their equivalent in Activity Diagrams, are explained in the following sections.

3.2.1 Primitive activity

This refers to a simple action, assigned to a single role.

The notation adopted for a primitive activity is <action>$^{(r)}$, where r is the role in charge of this action.

3.2.2 Invocation of a sub-collaboration

This is the call of a collaboration *subcol*, written <subcol>$^{(R)}$, that is performed by the set of roles R.

3.2.3 Strong sequence

This is the sequencing operator that is used by Activity Diagrams. There is a sequence between two collaborations in an Activity Diagram, when they come successively in the diagram, and are separated only by a control or data flow. The first collaboration has to be totally completed before actions related to the second one can start. This means that any role that is involved in the second collaboration, but not in the first one, still has to wait for the first collaboration to finish before it can perform its actions in the second one.

If the first collaboration is called C1 and the second one C2, then this construct is written C1 $;_s$ C2, where the 's' refers to 'strong' as opposed to 'weak' sequence.

3.2.4 Weak sequence

This sequencing operator is an extension to Activity Diagrams. The weak sequencing, as described above, between two collaborations C1 and C2 is written C1$;_w$ C2. The 'w' refers to 'weak' as opposed to 'strong' sequence.

3.2.5 Choice

It is used when we have two alternative collaborations C1 and C2 and only one of them will be executed. This construct is represented in Activity Diagrams by a Decision node.

The choice between two collaborations C1 and C2 is written C1[]C2.

3.2.6 Strong while loop

The Strong while loop is used when we have a loop of some collaboration C1, followed by another collaboration C2, with a strong sequence separating them. The strong loop also means that all successive executions of C1 are strongly sequenced.

The collaboration C2 can be empty and in this case it is written as ϵ. The strong while loop is itself written C1 $*_s$ C2.

It is assumed here that the choice at each end of the loop, between either re-executing C1 or executing C2, is a local choice. This assumption is ensured by limiting the set of starting roles of C1 to just one role which will perform the choice. If C2 is not empty, then the starting role of C2 has to be that same role.

In the Activity Diagram, the while loop is constructed by a merge node followed by the collaboration to be looped over, then followed by a decision node, where the decision is made to either repeat the loop, or exit the loop and perform C2. In the extended notation adopted here for the Activity Diagrams, the while loop has to be followed by a strong sequence then followed by the next collaboration C2.

3.2.7 Weak while loop

What was said for the strong loop is also true for the weak loop. The difference is that here, we have a weak sequence at the end of the loop.

3.2.8 Concurrency

This construct is used when two collaborations C1 and C2 take place in parallel. It is written C1 || C2.

In Activity Diagrams, the control node used for parallelism is the fork node.

3.2.9 Competition

The two collaborations in competition[1] are started in parallel, then as soon as one of them is completed, the other one is interrupted. The activity continues then normally with the next actions in the flow, that will receive the information about which of two collaborations completed successfully.

Competition between two collaborations C1 and C2 is written as C1 <> C2.

3.2.10 Interruption

In this construct, noted as C1 | > C2 else C3, once a collaboration C1 is started, it may be interrupted by another collaboration C2. If it is not interrupted, then C3 takes place as soon as C1 completes. Otherwise, if the interruption starts while C1 in still in progress, then C3 will not take place, and the activity continues with the actions that follow the interrupt construct.

In Activity Diagrams, the interruption corresponds to the interruptible activity region.

3.3 Overview of the derivation method

The algorithm takes the global specification of a distributed system, as well as the definition of the different roles allocated to the components that participate in the system. The algorithm calculates the starting, participating and terminating roles of the different constructs used in the global specification. Using these calculated roles, as well as the type of the construct itself, the coordination messages that need to be exchanged between components are then determined, and a derivation of the specification for each component is obtained.

[1]The competition construct is from an unpublished version of [23]

3.3.1 Computing the Starting, Participating and Terminating roles

Determining the starting, participating and terminating roles for a given construct depends on the type of construct used between two collaborations, as well as on the starting, participating and terminating roles of each of these two collaborations. The method of calculation is shown in the following table, taken from [23].

TABLE 3.1: Rules for calculating the starting, terminating and participating roles of a collaboration

Construct	Starting roles (SR)	Terminating roles (TR)	Participating roles (PR)
primitive activity	{r}	{r}	{r}
invocation of a sub-collaboration where R is the set of participating roles	SR(<name>)	TR(<name>)	PR(<name>) = R
strong sequence	SR(C1)	TR(C2)	PR(C1) ∪ PR(C2)
weak sequence	SR(C1) ∪ (SR(C2) - PR(C1))	TR(C2) ∪ (TR(C1) - PR(C2))	PR(C1) ∪ PR(C2)
choice	SR(C1) ∪ SR(C2)	TR(C1) ∪ TR(C2)	PR(C1) ∪ PR(C2)
strong while loop	SR(C1) = SR(C2) = {r} (assumption: single starting role)	TR(C2)	PR(C1) ∪ PR(C2)
weak while loop	same as for strong while loop	TR(C2) ∪ (TR(C1) - PR(C2))	PR(C1) ∪ PR(C2)
concurrency	SR(C1) ∪ SR(C2)	TR(C1) ∪ TR(C2)	PR(C1) ∪ PR(C2)
interruption	SR(C1)	TR(C2) ∪ TR(C3)	PR(C1) ∪ PR(C2) ∪ PR(C3)

3.3.2 Deriving the components behavior from a global specification

This section describes the steps of the algorithm in detail. It is then presented in a more formal way in a table taken from [23] at the end of this section. For each of the constructs defined earlier, we will explain the derivation rule introduced by the algorithm.

3.3.2.1 Primitive action: <action>$^{(r)}$

This is a basic action performed locally in Alloc(r), therefore, there are no coordination messages needed here.

3.3.2.2 Invocation of a sub-collaboration: <subcol>$^{(R)}$

There are no coordination messages needed for this construct either.

3.3.2.3 Strong sequence: C1 $;_s$ C2

In order to ensure that all the actions in C1 are terminated before any action of C2 can start, there are flow messages that are introduced whenever this construct is encountered. Indeed, the terminating roles of C1 must inform the starting roles of C2 of the completion of their tasks in C1. This information is passed between the components by means of flow messages, fm(x), that include a parameter x [9] which indicates the strong sequence construct it refers to.

So for the strong sequence to be enforced, all the components that have roles in the set of terminating roles of C1, must send fm(x) to all components with roles in the set of starting roles of C2. The only exception to this rule, is that if a component has a role that belongs to both sets, then it does not send a message to itself, since it is clearly not needed.

3.3.2.4 Weak sequence: C1 $;_w$ C2

In the case of weak sequence, no additional messages are needed.

3.3.2.5 Choice: C1[]C2

The choice here is assumed to be local to a single component. When there is choice between two alternatives C1 and C2, the components that participate in C1 but not in C2 (or inversely), receive a message from one of the components involved in C2, when the alternative C2 has been chosen.

The message exchanged in this case is called choice indication message (cim). It ensures that the components participating in only one alternative of the choice, will receive the information that the choice has been made, in the case where the alternative in which they do not participate has been chosen. The message cim has a parameter called y, that will indicate to which choice construct this message belongs.

3.3.2.6 Strong while loop: C1 $*_s$ C2

As mentioned earlier, the strong while loop is assumed to have a single starting role. There are two types of messages needed in this case.

Firstly, the while loop can be seen as a kind of choice, in the sense that at each execution of the loop, a choice has to be made between performing the loop C1, or exiting the loop and executing the next collaboration C2. Therefore, when C2 is chosen, a choice indication message needs to be sent to components that take part in C1 but not in C2. This message will be generated by the role responsible for the choice, which is the starting role of C1.

The second message needed is due to the strong sequencing. Since any two successive executions of C1 should be strongly sequenced, then a flow message should be sent by the terminating roles of C1 to its starting role.

3.3.2.7 Weak while loop: C1 $*_w$ C

The first message described for the strong while loop is also needed here.

Another information needed in this case is a loop counter. Indeed, because of the weak sequencing, the components involved in both collaborations C1 and C2, may not have the information of how many times C1 has been executed before C2 was started. So, such components will not actually know whether they have received all the messages that have been sent to them during the executions of C1. However, this information is necessary for them to make the decision on when they can consider that their role in C1 is finished, and that they can proceed with C2.

This is solved by the sending of the loop counter parameter in the first message received by components playing a role in both C1 and C2. Furthermore, the loop counter should also be sent in all cim messages.

3.3.2.8 Concurrency: C1 || C2

No coordination messages are needed for concurrency.

3.3.2.9 Interruption: C1 | > C2 else C3

Assumptions related to the interruption:

- The set of starting roles of C2 contains only one element r.
- The collaboration C2 can be written as <action>$^{(r)}$ $;_s$ C2'.

There are three types of coordination messages needed for this construct.

- Interrupt enable message (iem): This is sent by the starting role of the collaboration C1 to r. And it is sent as soon as C1 has started, in order to inform C2 that it can now start any time.
- Interrupt message (im): This message is sent by the component to which r belongs, to all components involving roles that participate in C1. It is sent whenever C2 starts. The components that receive it become interrupted if they did not finish performing their roles in C1 yet.

  iem and im messages both have a parameter z that allows distinguishing between messages sent from different interruption constructs.
- Flow messages: These messages will contain the information whether or not the components that involve roles participating in C1, were interrupted. These components will send these flow messages to the starting components of C2' and C3. If the messages indicate that the interruption has indeed occurred, then C2' will start. If on the opposite, no component has been interrupted, then C3 will start.

3.3.2.10 Competition: C1 <> C2

This construct is not defined in UML, but was defined in [23][2] as a construct that can replace the UML Interruption. This is achieved by the use of a central synchronizer that will identify which collaboration has completed its execution first. The central synchronizer will then take charge of sending the necessary interruption messages to interrupt the execution of the other collaboration involved in the competition construct.

The derivation rules for the competition construct are defined as follows. We first choose a component sc that will play the role of the central synchronizer. We then distinguish between two cases.

The first case is where we are deriving the behavior of component sc. In this case, in addition to performing its role in C1, sc will also receive flow messages from all the terminating components of C1. If this is done without interruption from C2, then sc notes that C1 has terminated, and sends an interruption to all participants of C2. This is done in parallel to sc playing its role in C2 and receiving a flow message from every terminating component in C2, and if this is done without interruption, then sc sends an interruption to all participants of C1.

The second case is where we are deriving the behavior of another component c different from sc. In this case, this component will play its role in C1, and as mentioned above, if it is a terminating components of C1 then it will send a flow message to sc. This process could be interrupted at any time by an interruption message sent from sc. Again this is done in parallel to c playing its role in C2, and then sending a flow message to sc if it is among the terminating components of C2. This process too could be interrupted by an interruption message from sc.

3.3.2.11 The algorithm

The following table is taken from [23]. It presents the algorithm that has been detailed in this section, in the form of rules that can be applied to a general specification. This

[2]This construct is from an unpublished version of [23]

specification is seen as a main collaboration C, which is then broken up into constructs that define relationships between sub-collaborations. The process continues recursively until we reach basic collaborations that cannot be decomposed any further. In the table, the function Tc(C) refers to the transformation function, which defines the actions performed by the component c in the collaboration C.

Structure of expression	Definition of Tc
C = <action>$^{(r)}$	Tc (C) = if Alloc(r) = c then "<action>" else ϵ Note: ϵ is the empty string and means that no actions need to be performed.
C = invoke <subcol>$^{(R)}$	Tc (C) = if c in Alloc(R) then "invoke <subcol>" else ϵ
C = C1 $;_s$ C2	Tc (C) = Tc (C1) ";" SFM(C1 , C2) ";" RFM(C1 , C2) ";" Tc (C2) *where* SFM(C1 , C2) = *if c in Alloc(TR(C1)) then* "send fm(x) to all c' in (Alloc(SR(C2)) – {c}) " *and* RFM(C1 , C2) = *if c in Alloc(SR(C2)) then* "receive fm(x) from all c' in (Alloc(TR(C1)) – {c})" Note: The term "– {c}" avoids that flow messages are sent to the component itself.
C = C1 $;_w$ C2	Tc (C) = Tc (C1) ";" Tc (C2)
C = C1 [] C2	Tc(C) = DOcim$_c$(C1, C2) [] DOcim$_c$(C2, C1) *where* DOcim$_c$(C1, C2) = *if c in Alloc(PR(C1)) then* "(Tc (C1)" *if c is responsible for cim then* "|| send cim(y) to all c' in (Alloc(PR(C2)) - Alloc(PR(C1))))"; *else if c in (Alloc(PR(C2) - Alloc(PR(C1))) then* "receive cim(y)"

Structure of expression	Definition of Tc
	Note: The function $DOcim_c$(C1, C2) generates code for performing C1, and looks after the transfer of choice indication messages from some component participating in C1 to those components not participating in C1, but in C2.
C = C1 $*_s$ C2	We assume that Alloc(SR(C1))={r} and that Alloc(SR(C2))={r} or C2 = ϵ. Tc(C) = "(" Tc(C1) ";" SFM(C1 , C1) ";" RFM(C1 , C1) ")* ; (" Tc(C2) *if c = r then* "|| send cim(y) to all c'in (Alloc(PR(C1)) - Alloc(PR(C2)) - {r}) " *if c in (Alloc(PR(C1)) - Alloc(PR(C2)) - {r}) then* "|| receive cim(y) from r " ")"
C = C1 $*_w$ C2	As above, except that the SFM and RFM constructs are absent
C = C1 || C2	Tc (C) = Tc (C1) || Tc (C2)
C = C1 | > C2 else C3	We assume that C2 has the form " <action>$^{(r)}$ $;_s$ C2' ". Tc (C) = NormalBeh ||* InterruptBeh . (see note below) These two parts communicate within each component using the following boolean local variables which are initially false: Interr : an interrupt occurred (but it may have occurred too late) Interrupted : the normal behavior has been interrupted In addition, a local variable I-Enabled is used by the InterruptBeh part. The action "wait(x)" waits until the expression x becomes true. **NormalBeh** = *if c in Alloc(PR(C1)) then* "(Tc (C1) | > (wait(Interr); Interrupted := true;) else ϵ); " *if c in Alloc(TR(C1)) then* "send fim(x, Interrupted) to all c' in

<table>
<tr><th>Structure of expression</th><th>Definition of Tc</th></tr>
<tr><td></td><td>((Alloc(SR(c'2)) ∪ Alloc(SR(C3)))–{c});"(* this is similar to SFM(C1, C'2 || C3) *)
if c in (Alloc(SR(C'2)) ∪ Alloc(SR(C3))) then
"((* similar to RFM(C1, C'2 || C3) *)
(for all c' in (Alloc(TR(C1))–{c}) do (receive fim(x, i) from c';
if i then Interrupted := true;);
if not Interrupted then DOcimc (C3, C'2);)
||* (wait(Interrupted); DOcimc (C'2, C3))) "
else "(DOcimc (C'2, C3) [] DOcimc (C3, C'2)); "

InterruptBeh =
if c = r then (
if c in (Alloc(SR(C1)) then "I-Enabled := true; "
else "for all c' in (Alloc(SR(C1)) - {c}) do (receive iem(z) from c'; I-Enabled := true)
|| (wait(I-Enabled); <action> (* this may never happen *) ;
Interr := true; send im(z) to all c' in (Alloc(PR(C1)) - r) ;) "
else (c not equal r *)* (
if c in Alloc(SR(C1)) then "send iem(z) to r; "
if c in Alloc(PR(C1)) then
"(receive im(z) from r (*may not happen *);
Interr := true;) "</td></tr>
<tr><td>C[3] =
C1 <> C2</td><td>We choose a component sc which determines which of the two collaborations terminates first; it is the only terminating component of this composition. C_{sc} is a dummy action located at sc.</td></tr>
</table>

[3]This construct is from an unpublished version of [23]

<table>
<tr><th>Structure of expression</th><th>Definition of Tc</th></tr>
<tr><td></td><td>*if* $c = cs$ *then*
((Tc (C1) ; SFM** (C1 , Csc)) | > interr1) ;
if there was no interr1 then raise interr2 locally and send interrupt message to all participants of C1)
|| ((Tc (C1) ; SFM* (C1 , Csc) | > interr2) ;
if there was no interr2 then raise interr1 locally and send interrupt message to all participants of C2)
else
((Tc (C1) ; SFM* (C1 , Csc)) | > interr1)
|| ((Tc (C1) ; SFM* (C1 , Csc)) | > interr2)
Note: this may lead to some dangling messages that are not received; they should be eliminated automatically (we need appropriate identifiers)</td></tr>
</table>

3.4 Summary

In this chapter, we have described the derivation algorithm [23] that defines a method to derive components designs from a global specification. This derivation ensures that the coordination between the different components in the distributed system is done in a way that avoids conflicts between these system components. This is ensured by the introduction of several coordination messages exchanged between the system components and allowing their global coordination.

Chapter 4

REQUIREMENTS OF THE DERIVATION SOFTWARE TOOL

In the following, we will present the requirements of our derivation software tool for distributed systems. We will also describe case scenarios for the different constructs covered by the Derivation Tool, to help better understand the requirements for this tool. For each case, we will present the result that needs to be obtained as the derived behavior for the components that participate in the Activity Diagram. We will then explain the importance of the derivation rules that are applied and how they prevent coordination problems between the system components.

Note: The Activity Diagrams presented in this chapter are diagrams that were used as input or were obtained as output from our Derivation Tool after the development phase.

4.1 Extending the notation of Activity Diagrams for describing collaborations

The Derivation Tool takes as input an Activity Diagram describing the specification of a distributed system, and derives the designs of the different components participating in that system. The output is then the set of Activity Diagrams describing the behavior of these components individually. For each system component, the corresponding Activity Diagram generated by the Derivation Tool is to be executed locally by this component. However, for the global Activity Diagram of the distributed system, which is the input of the Derivation Tool, each activity in the diagram may involve several system components that play a role in its execution. For example, in the global Activity Diagram shown in Figure 4.2, the call behavior action CBA1 involves two different components, 'p' and 'q', which both play a role in it. For this reason, the Derivation Tool takes as input an extended version of Activity Diagrams, where activities are replaced by UML Collaborations involving multiple components, as was described in the derivation algorithm [23] presented in Chapter 3.

The collaborations are specified in the input Activity Diagram, using UML Call Behavior Actions. However, since the collaborations may involve several roles, which is not the case for UML Actions, we then need a way to allow the user to specify the roles for each collaboration in the diagram. For this purpose, we make use of UML stereotyping as we explain in the next section, in order to define the starting, participating and terminating roles for collaborations.

Another extension of Activity Diagrams that we note here is related to the number of starting actions of sub-activities involved in an Activity Diagram. In standard UML Activity Diagrams, each sub-activity starts with a single action, even if several input parameters are involved. However, in our case, collaborations can have multiple starting actions, that may be played by different roles. This means that a specific collaboration may have multiple starting roles specified for it.

4.2 Definition of requirements

4.2.1 Input notation for specifying collaboration roles

In order to derive the components designs, the Derivation Tool needs information about the starting, participating and terminating roles of each of the collaborations of the global system's Activity Diagram. Figure 4.1 shows how the starting, participating and terminating roles of collaborations are represented in the input Activity Diagrams. The set of the starting roles for a given collaboration is indicated by a pin whose label starts with "S:", followed by a list of comma-separated roles. For example, the label 'S:p,q' for some collaboration means that p and q are both starting roles of that collaboration. The same goes for the set of participating roles, except that the 'S:' is replaced by 'P:'. And similarly for the set of terminating roles, we have a 'T:' instead of 'S:'.

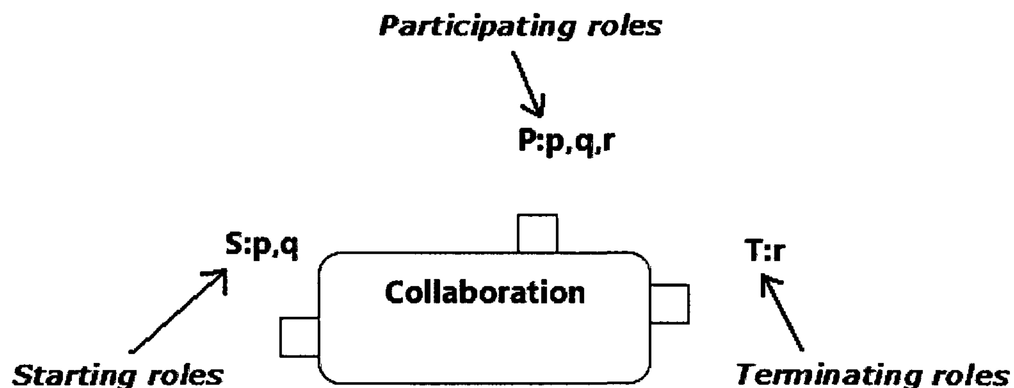

FIGURE 4.1: Representation of starting, participating and terminating roles

4.2.2 The allocation of roles to system components

Each system component participating in the distributed system may implement multiple roles that are responsible for actions performed locally at this component. It is up to the system's architect to determine how the system is physically distributed over different components, and which roles each component plays in the global system. The

Derivation Tool takes this information as an input (described later in Chapter 5), and internally defines an allocation function Alloc() [23], that returns for each role, the system component to which this role belongs.

This allocation function is important for the Derivation Tool for the following reason. On the one hand, the input global Activity Diagram does not contain any information about the system components themselves, but rather contains definitions of roles participating in the different collaborations. On the other hand, we do not want to generate the behavior for each role separately, instead, the output required from the Derivation Tool is the local design for each system component, involving all the roles it realizes. Here, the Alloc() function provides the translation from roles to components, allowing the Derivation Tool to include the right roles while generating a specific system component's Activity Diagram.

4.2.3 Transformation algorithm

The Derivation Tool automates the algorithm described in Chapter 3 for deriving the component designs from a global system's specification [23]. This algorithm defines a method of transformation of a global distributed system's specification into the separate designs of the different components of that system. This transformation takes into consideration the global synchronization of the components, that makes the system consistent with what was designed for it, and that avoids race conditions. This is achieved by the introduction of coordination messages exchanged between system components. The algorithm considers the system's specification to be defined in terms of constructs similar to those used by UML Activity Diagrams, and considers the activities of the system to be collaborations between multiple components.

4.3 Test Scenarios

In what follows, we will present test scenarios involving different constructs covered by the Derivation Tool. For each scenario, we will explain the results expected from the Derivation Tool for every component participating in the system.

4.3.1 Strong Sequence

The diagram in Figure 4.2 is a test case for strong sequencing. In this diagram, we have a strong sequence between the collaboration CBA1 and CBA2, and then between CBA2 and the parallel composition of CBA3 and CBA4, and finally between the latter and CBA5.

There are three components involved in this diagram, which are p, q and r. For simplicity, let us also call p, q and r the roles allocated to these components respectively.

The derived Activity Diagrams generated by our Derivation Tool for the components q, p and r are shown in Figures 4.3, 4.4 and 4.5, respectively.

The following are the coordination messages exchanged between these components.

- Since q is the terminating component of the collaboration CBA1, it sends a flow message to r which is the starting component of the collaboration CBA2 that follows it. This is message fm(1) in the derived Activity Diagrams of q and r, where the parameter of fm uniquely refers to the strong sequencing construct between CBA1 and CBA2. This message will enforce the strong sequencing here, since r has to wait for this message before it can start the execution of CBA2.
- Next, r is a starting component of the parallel construct CBA3 || CBA4, because r is the starting components of CBA3. On the other hand, q and p are also starting components of this parallel construct, since they are starting components of CBA4. Therefore, r, q and p all needs to receive a fm from the terminating components of CBA2, that is r. The flow message from r to itself is eliminated, and the messages needed are one from r to q, and another one from r to p. These will ensure that the parallel execution of CBA3 and CBA4 will not start before CBA2 is completely terminated.
- In the last strong sequencing construct, we have multiple sending components as well as multiple receiving components of flow messages. Indeed, p, q and r are all terminating components for CBA3 || CBA4. And p and q both start CBA5. After eliminating messages from p to itself, and from a to itself, The messages that

will remain are a fm from p to q, another fm from q to p, and finally two flow messages from r to both p and q. These coordination messages will keep CBA5 from starting until CBA3 and CBA4 are both finished.

We note that in figure 4.3, the coordination messages "receive fm(3) from <r>" and "receive fm(3) from <p>" are not necessarily received in this order. However, this is not a problem because, as explained in section 3.1.2 of the previous chapter, the derivation algorithm adopts the idea of buffering the messages at reception, and later consuming them as needed. The same is true for the following figures where two coordination messages are received consecutively.

4.3.2 Choice

Let us consider the Activity Diagram shown in Figure 4.6. In this diagram, the collaboration CBA1 is followed by a choice between two collaborations, CBA2 where the component p does not participate, and CBA3 where the component p plays a role.

In respect of the local choice assumption, the set of the starting roles of CBA2 and CBA3 contains only one role which is r. That is the role responsible for making the choice.

When we apply the derivation algorithm to obtain the corresponding behavior for the component p, we obtain the Activity Diagram shown in figure 4.7

The component p will alternatively either perform its role in CBA3, or it will receive a message from r. That message is called choice indication message. It will ensure that the component p receives an indication in case the alternative CBA2 was chosen.

The behavior of the component q follows in Figure 4.8. As the component q is present in both alternative collaborations CBA2 and CBA3, the choice indication message is not needed in this case. Therefore the derivation for component q does not involve the sending of any particular coordination messages in the corresponding Activity Diagram in Figure 4.8.

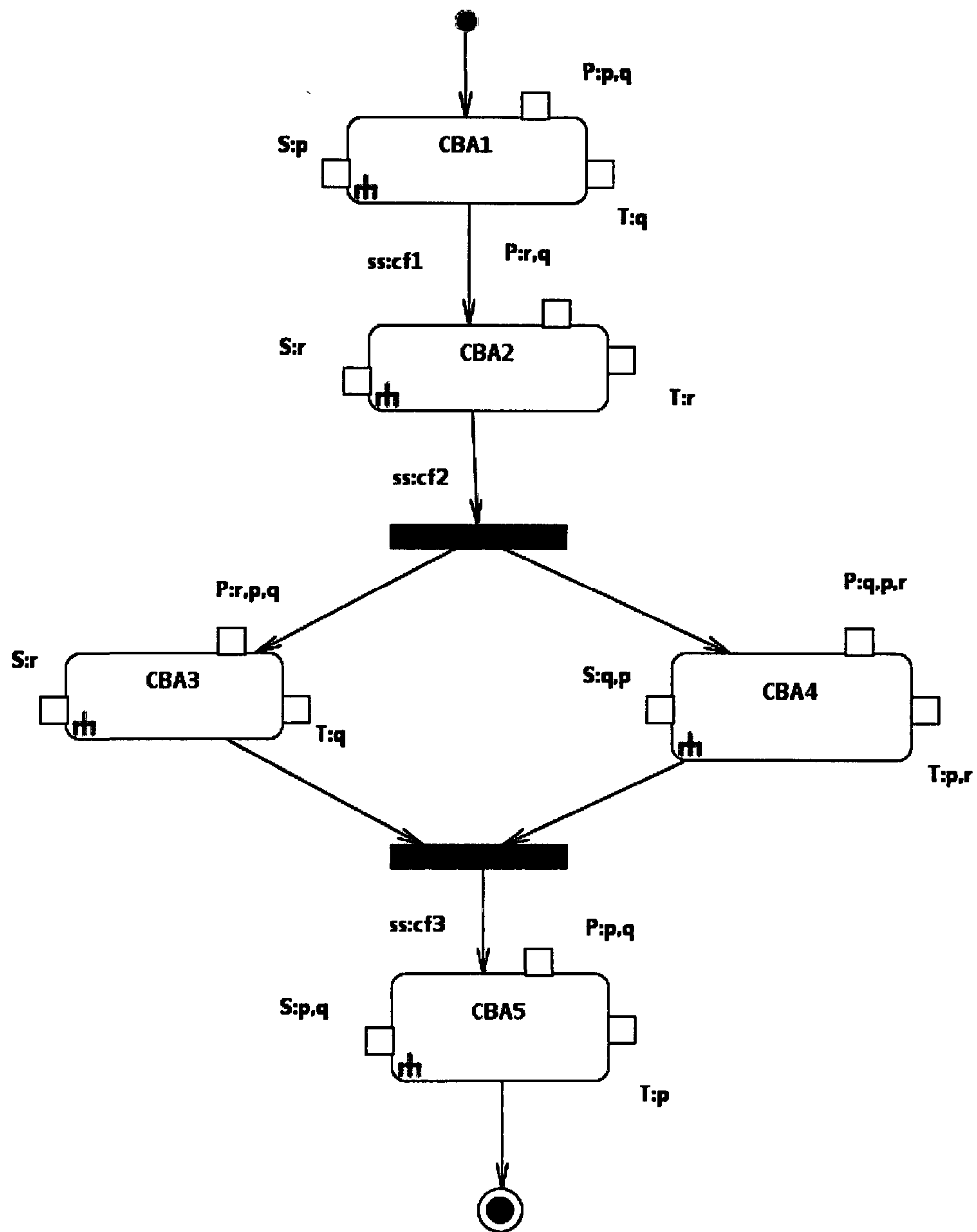

FIGURE 4.2: Strong sequence test AD

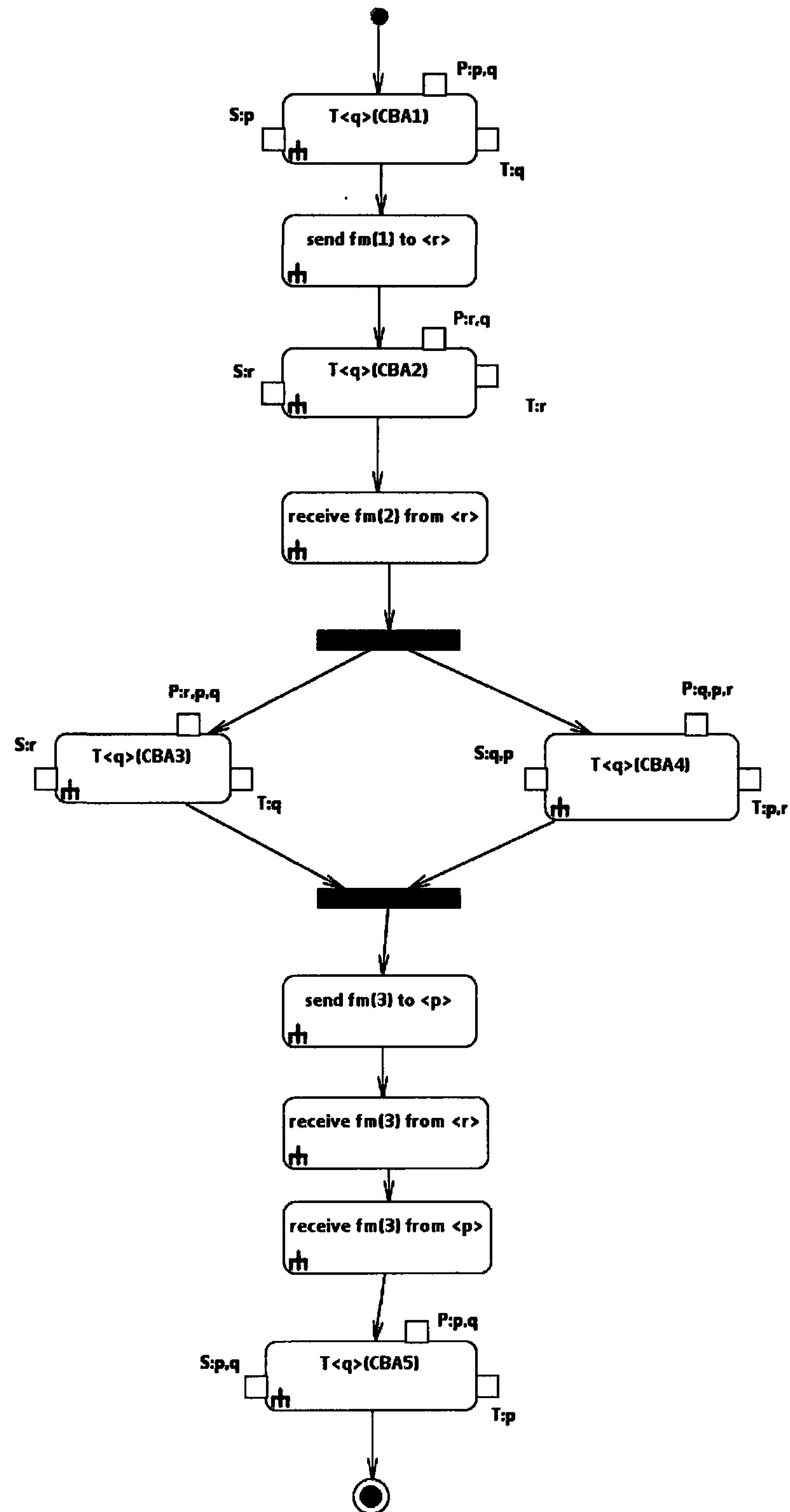

FIGURE 4.3: Derivation of the strong sequence test AD for the component q

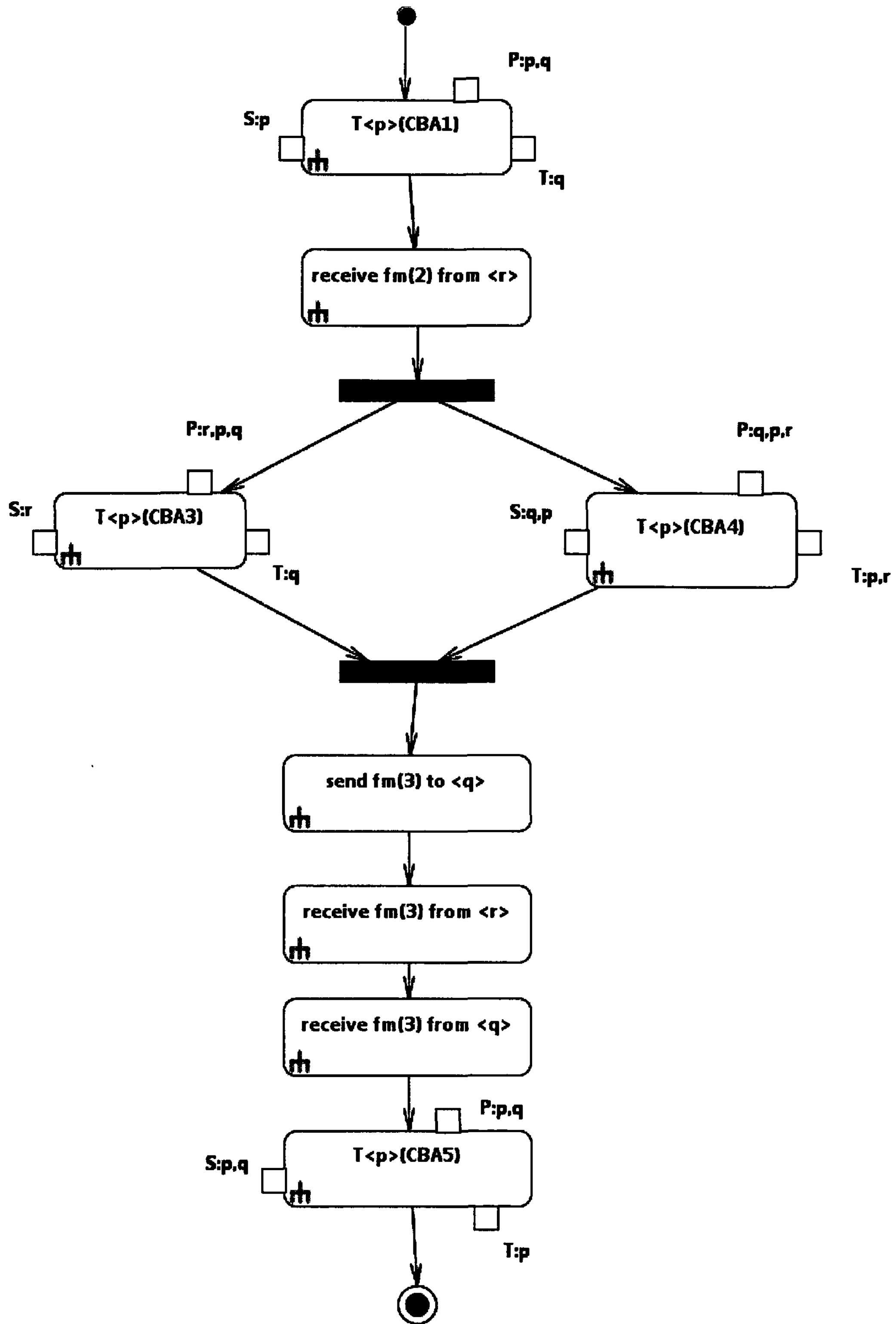

FIGURE 4.4: Derivation of the strong sequence test AD for the component p

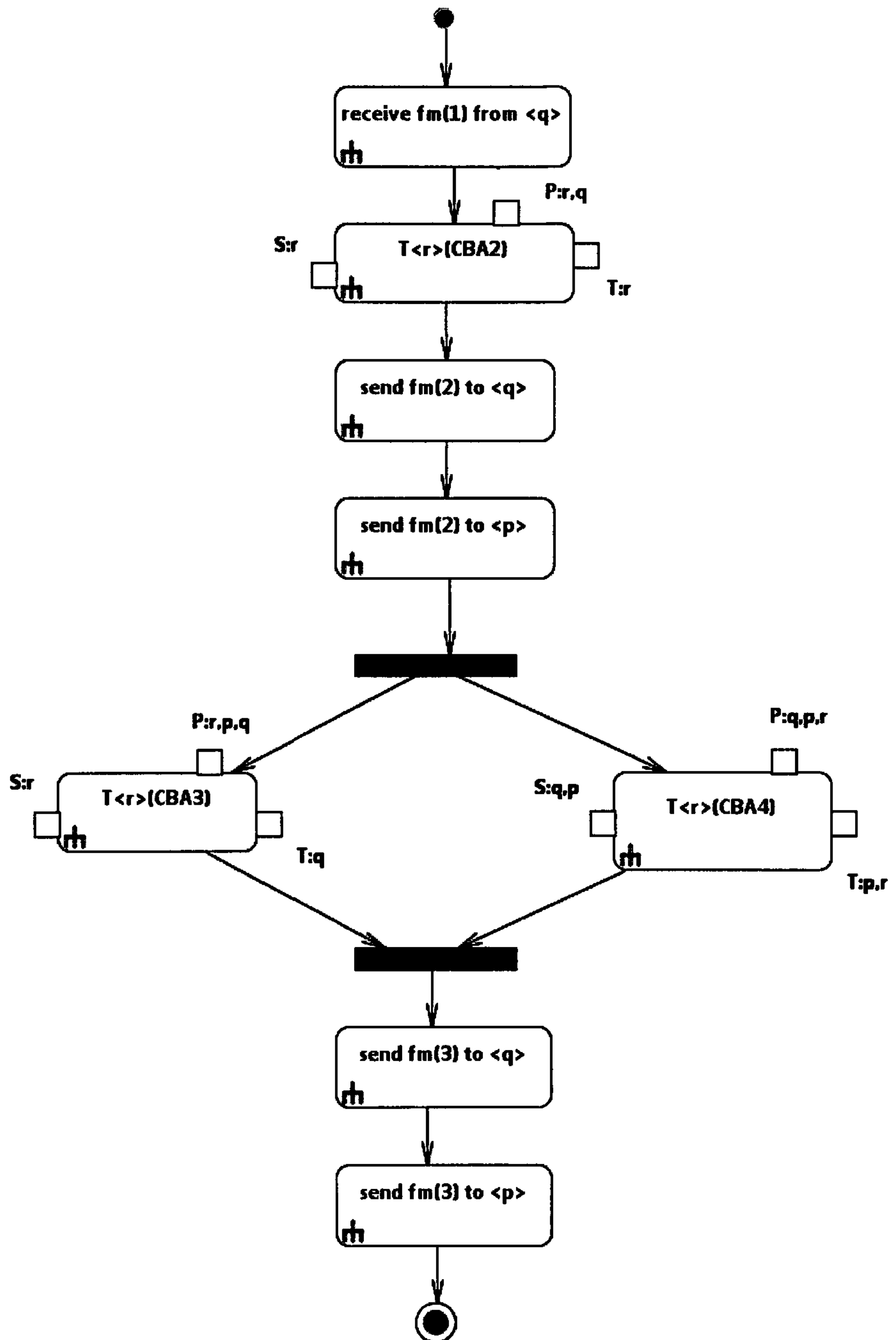

FIGURE 4.5: Derivation of the strong sequence test AD for the component r

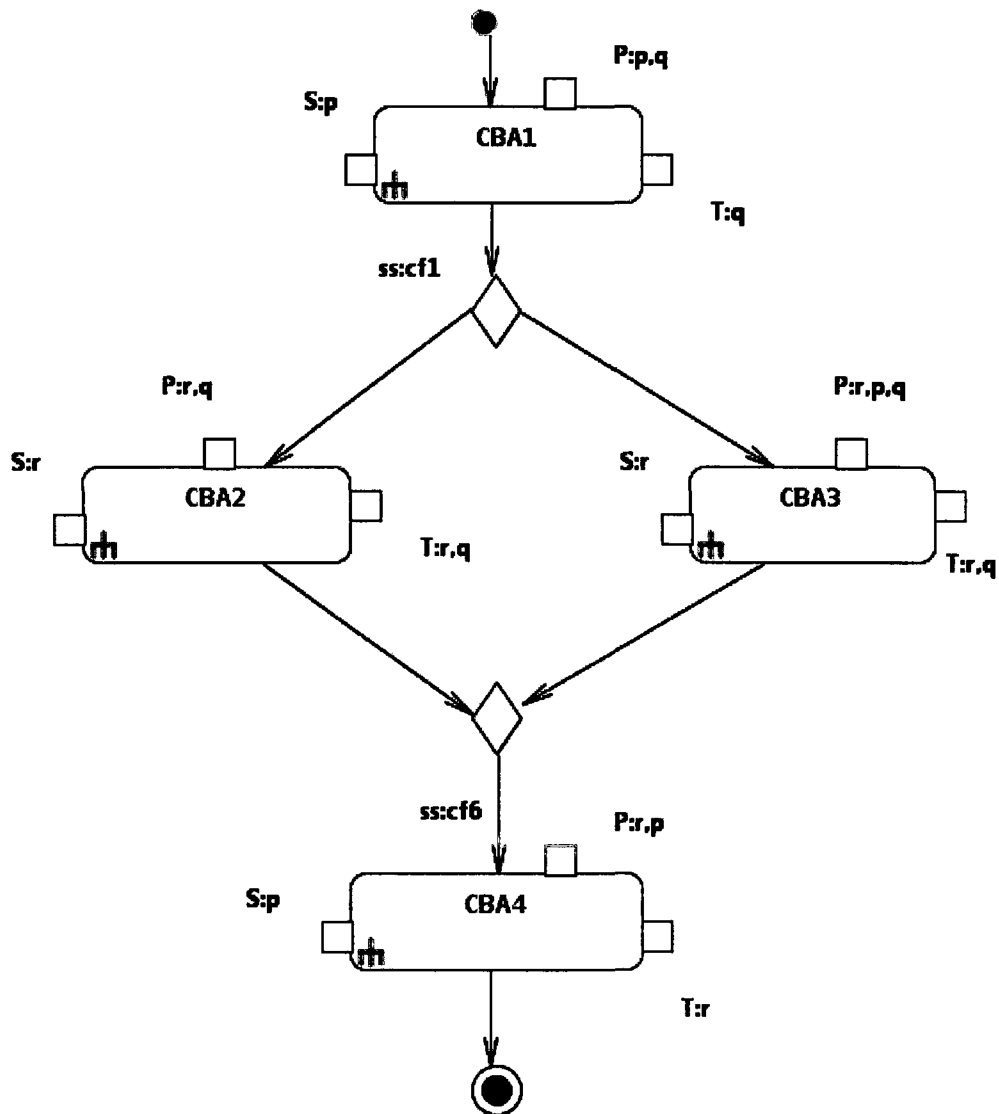

FIGURE 4.6: Choice test AD

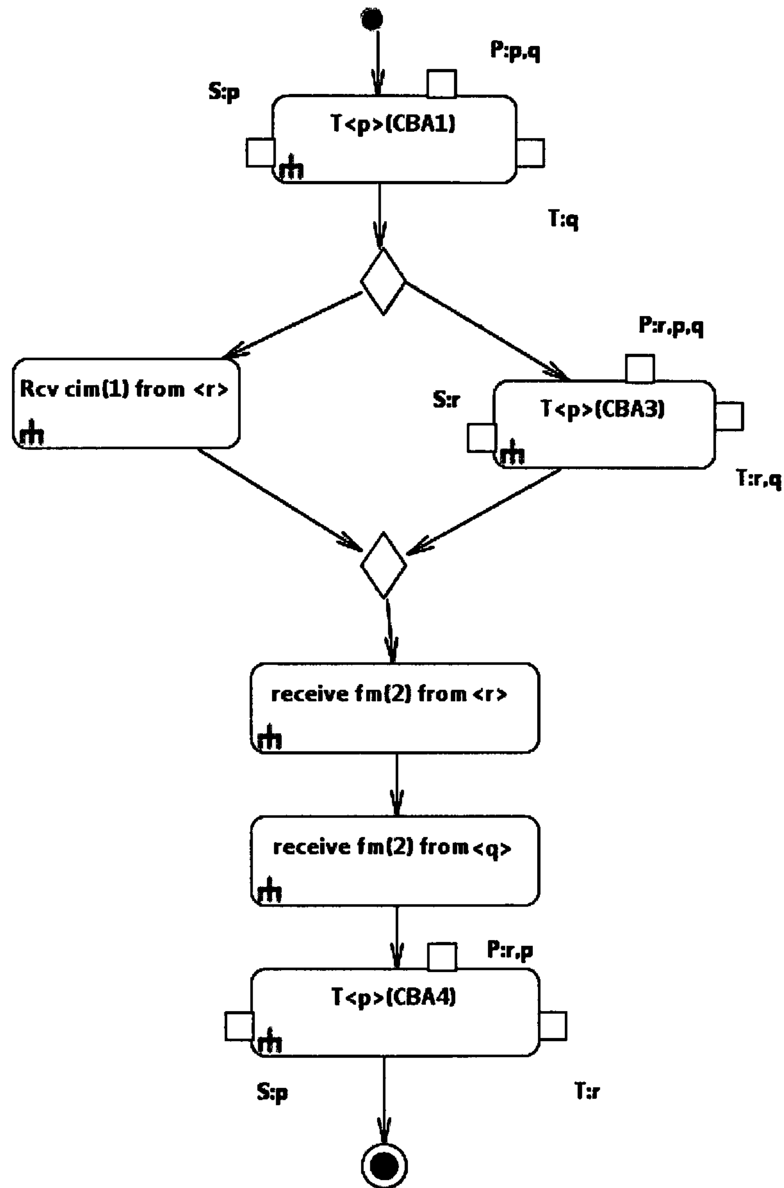

FIGURE 4.7: Derivation of the diagram "Choice test case" for the component p

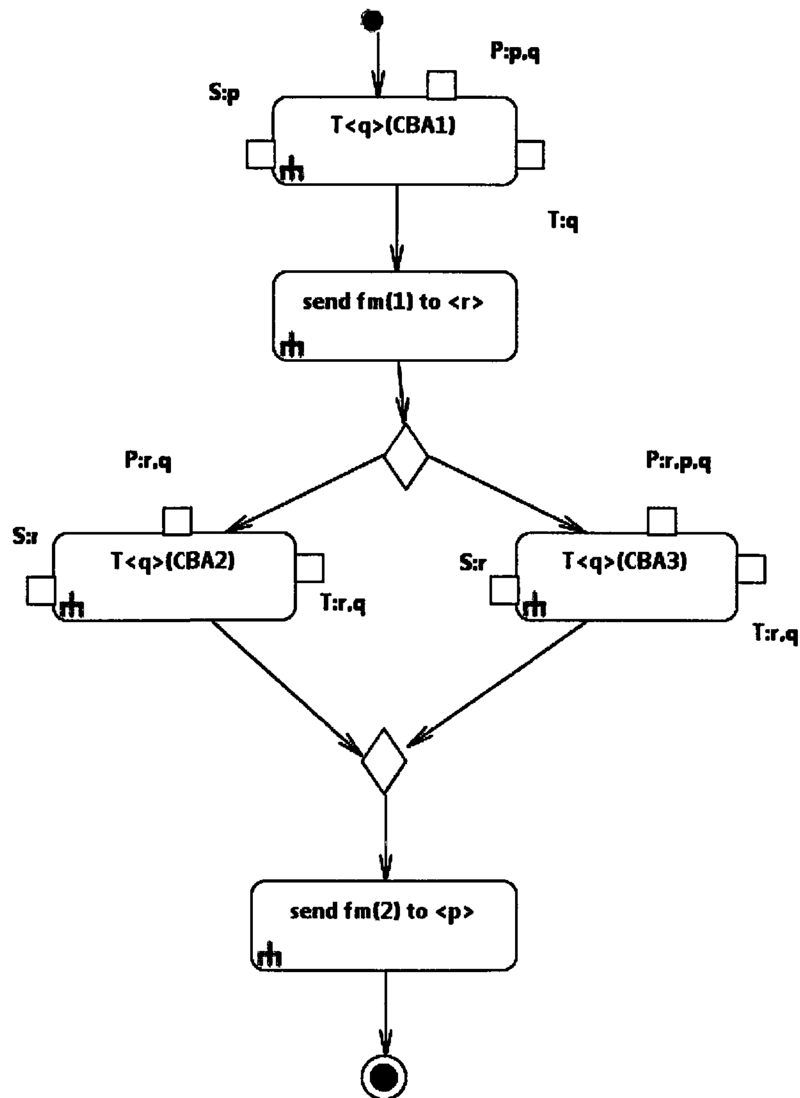

FIGURE 4.8: Derivation of the diagram "Choice test case" for the component q

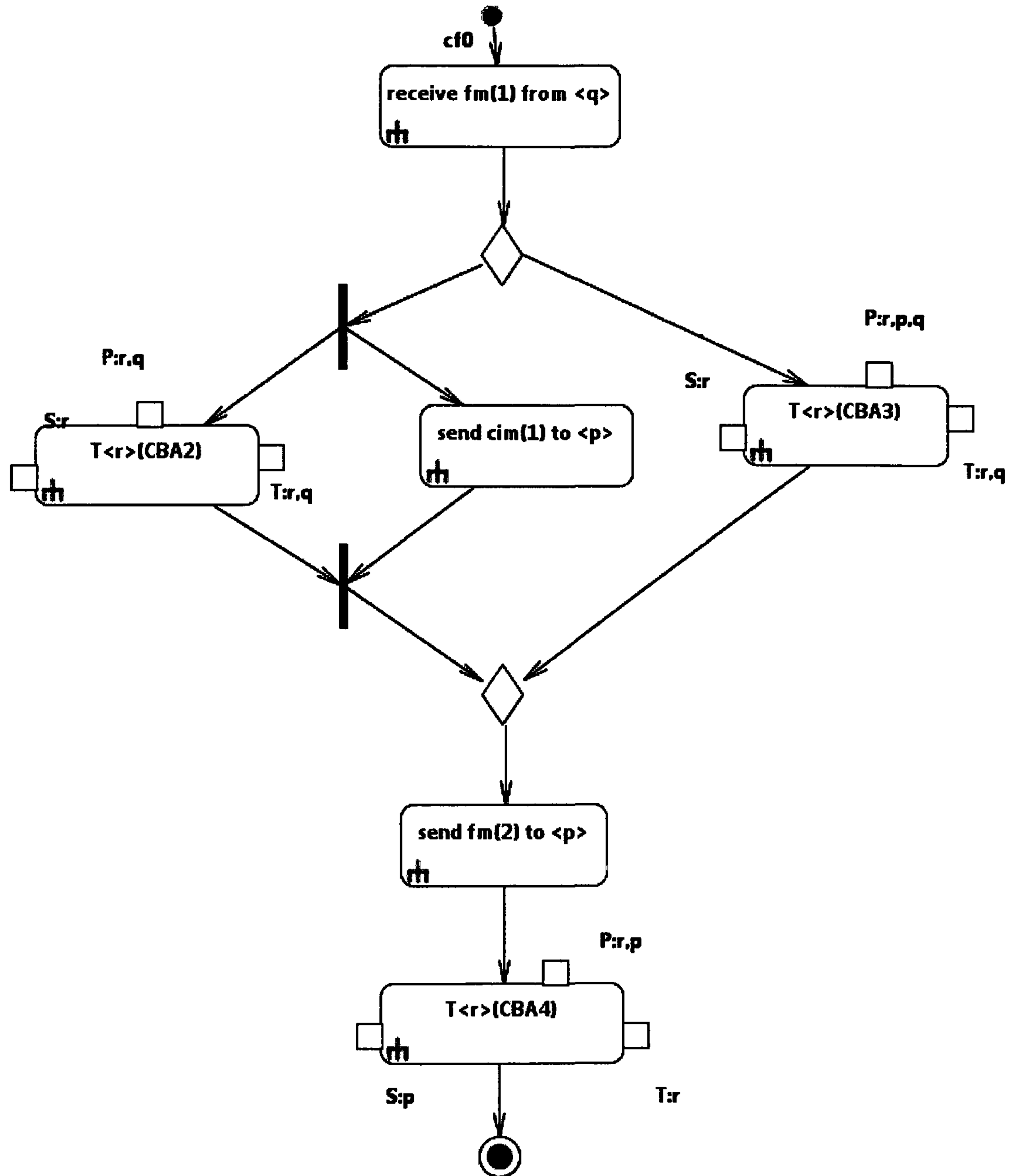

FIGURE 4.9: Derivation of the diagram "Choice test case" for the component r

For the component r, the derivation obtained by the transformation software is as in Figure 4.9. In parallel to performing its role in the collaboration CBA2, the component r sends a coordination message to p. This is the choice indication message explained above.

To further explain the importance of this derivation rule, let us assume that the choice indication message is not included in the behavior of the derived components. If the collaboration CBA3 is chosen, then there is no problem and the execution of the diagram is straightforward. But if the collaboration CBA2 is chosen, the component p does not know this information, and does not have any way to know when CBA2 will be completed. And since here it is the starting component for the following collaboration CBA4, it may be waiting indefinitely before starting the execution of CBA4.

4.3.3 Strong While Loop

In the test case of Figure 4.10, we have four collaborations. The first is CBA1, followed by a strong while loop containing CBA2 and CBA3 weakly sequenced, then followed by CBA4 which comes after a strong sequence.

There are five roles allocated to four components as follows: the roles p, r, and q belong respectively to the components p,r and q. And both roles s and t are allocated to a single component that we will call st.

The component p is involved in two kinds of coordination messages related to the loop.

1. The starting component of the loop is p, which has the behavior shown in Figure 4.11. Since this is a strong loop, its starting components need to receive a flow message from its terminating components. This is to ensure that the strong sequence is respected between every two successive executions of the loop.

   The loop involves CBA2 and CBA3, separated by a weak sequence. The terminating components of the loop include the terminating components of CBA2 that do not participate in CBA3, plus of course the terminating components of CBA3.

So the terminating components of the loop are r and st. Therefore, both r and st will send a flow message to p at each end of the loop.

2. Since p is the starting component of the loop, it has to send a choice indication message to all components that play a role in the loop but not in the collaboration CBA4 that follows the loop. This must be done when the loop terminates and CBA4 is performed. This means that p sends a cim to the component r. The component p does that in parallel to playing its role in the collaboration CBA4.

The next figure 4.13 represents the behavior generated for the component r. As already explained above, r will send a flow message to p at each end of the loop, as well as receive a cim message from p after the loop has ended. Note: The flow message "receive fm(2) from r" in Figure 4.11, which is part of the behavior of p, and the flow message "send fm(2) to p" in Figure 4.13, which is part of the behavior of r, both have the same parameter (2). This parameter is unique for each strong sequence construct, and avoids any possible confusion for the component sending the flow message and the one receiving it.

The generated behavior for component st is shown in Figure 4.14. One of the roles allocated to this component, which is t, participates in CBA4. This means that this component is involved in CBA4 which is the collaboration that comes after the loop. Therefore, this component does not need to receive any cim message after the choice has been made to end the loop.

So the only coordination message that component st is involved in, is the flow message that it has to send to component p at each end of the loop.

The generation of the behavior of component q, as shown in figure 4.12, is straightforward, since it does not have to send any coordination messages. Note: The global Activity Diagram (Figure 4.10) shows that the component q does not participate in CBA2. So after projecting this global behavior on the component q, this collaboration CBA2 is just eliminated which explains the fact that it is not included in Figure 4.12.

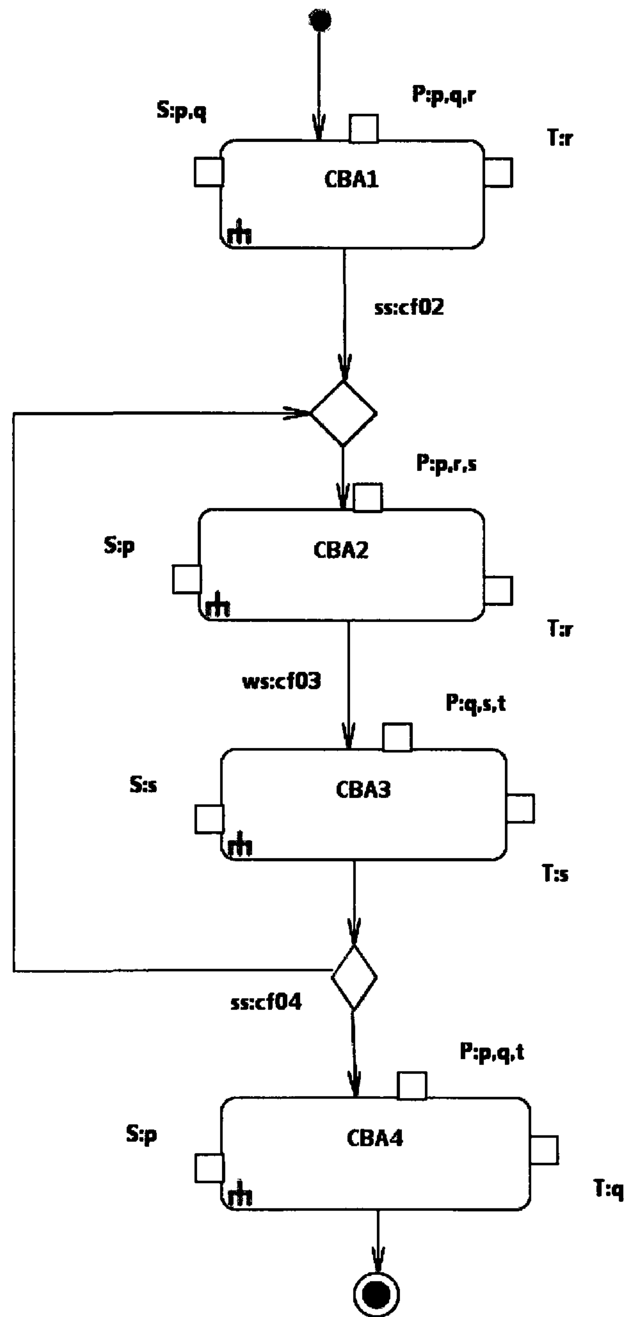

FIGURE 4.10: Strong loop Activity Diagram

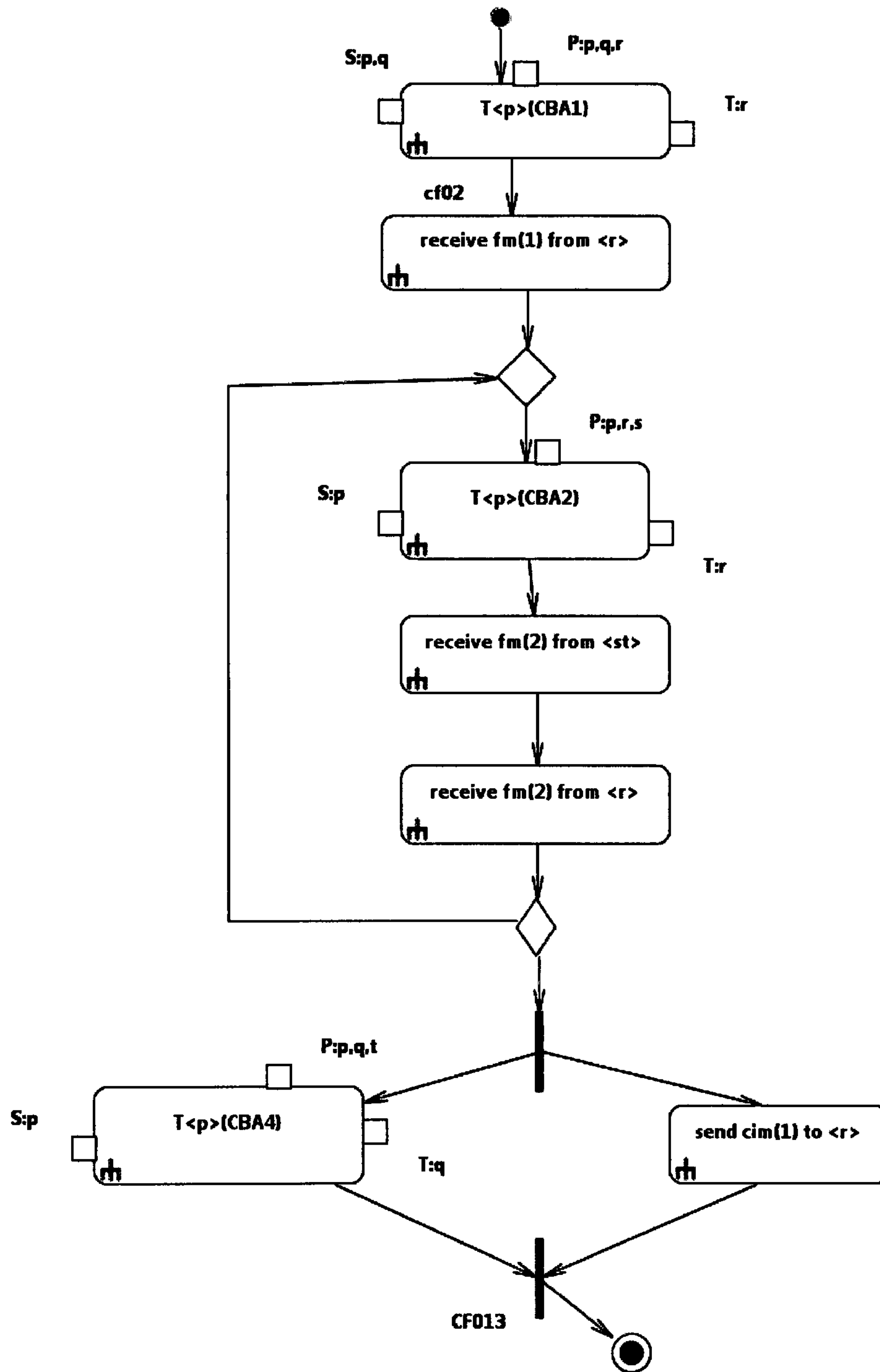

FIGURE 4.11: Derivation of the strong loop AD for the component p

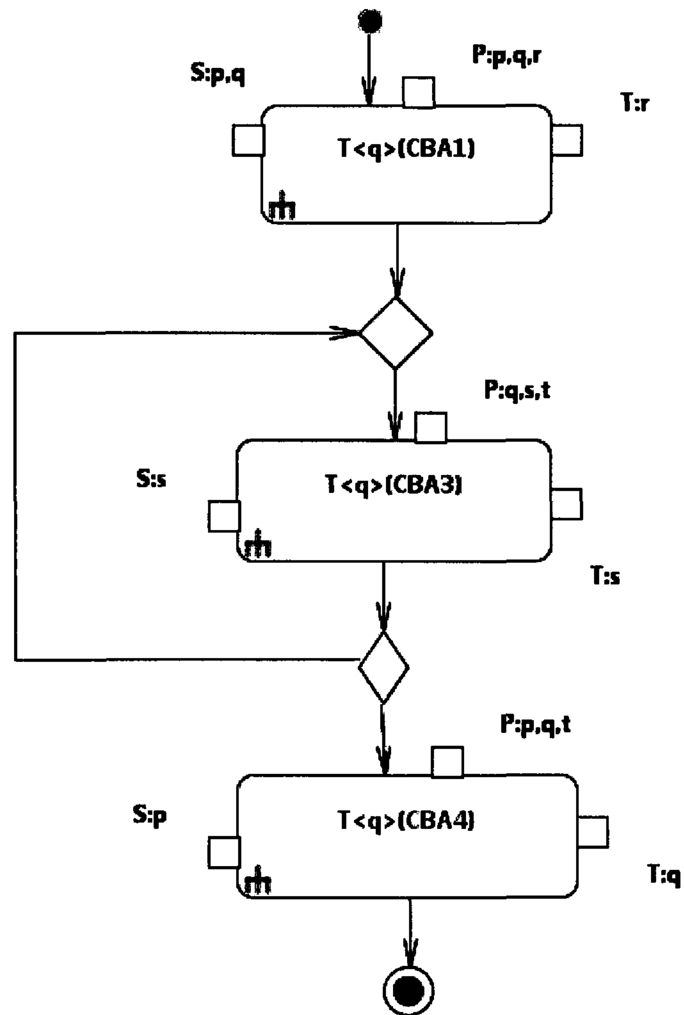

FIGURE 4.12: Derivation of the strong loop AD for the component q

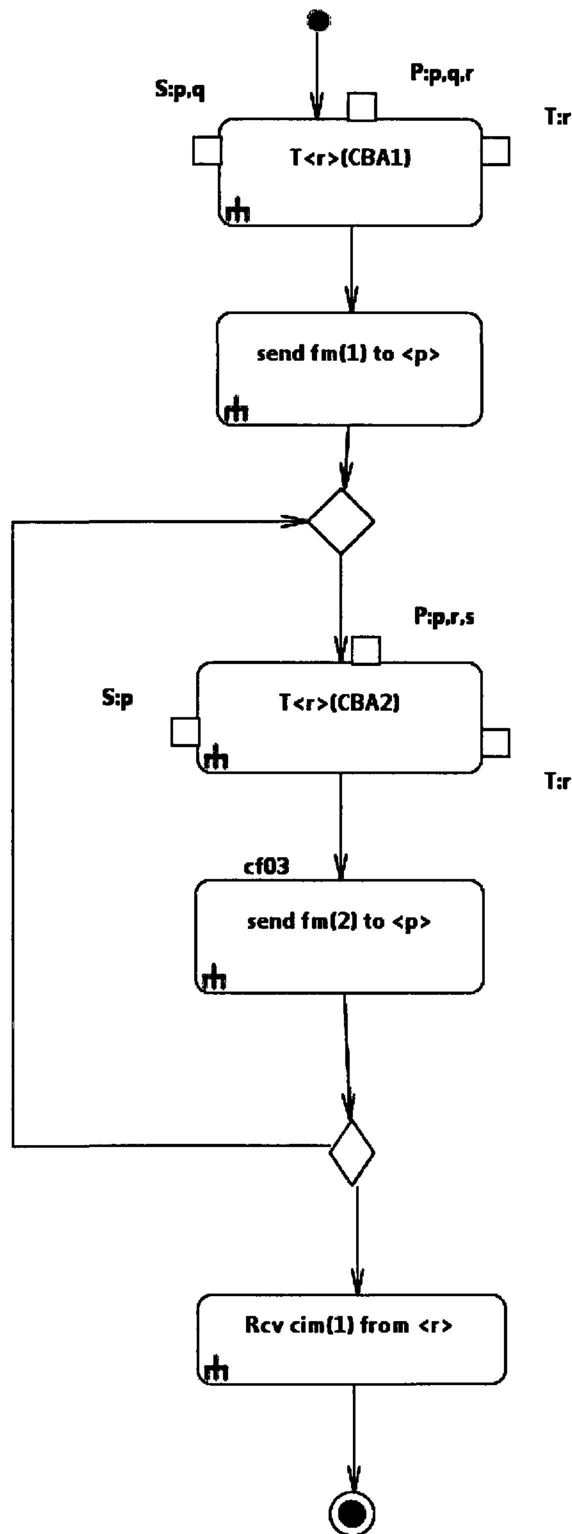

FIGURE 4.13: Derivation of the strong loop AD for the component r

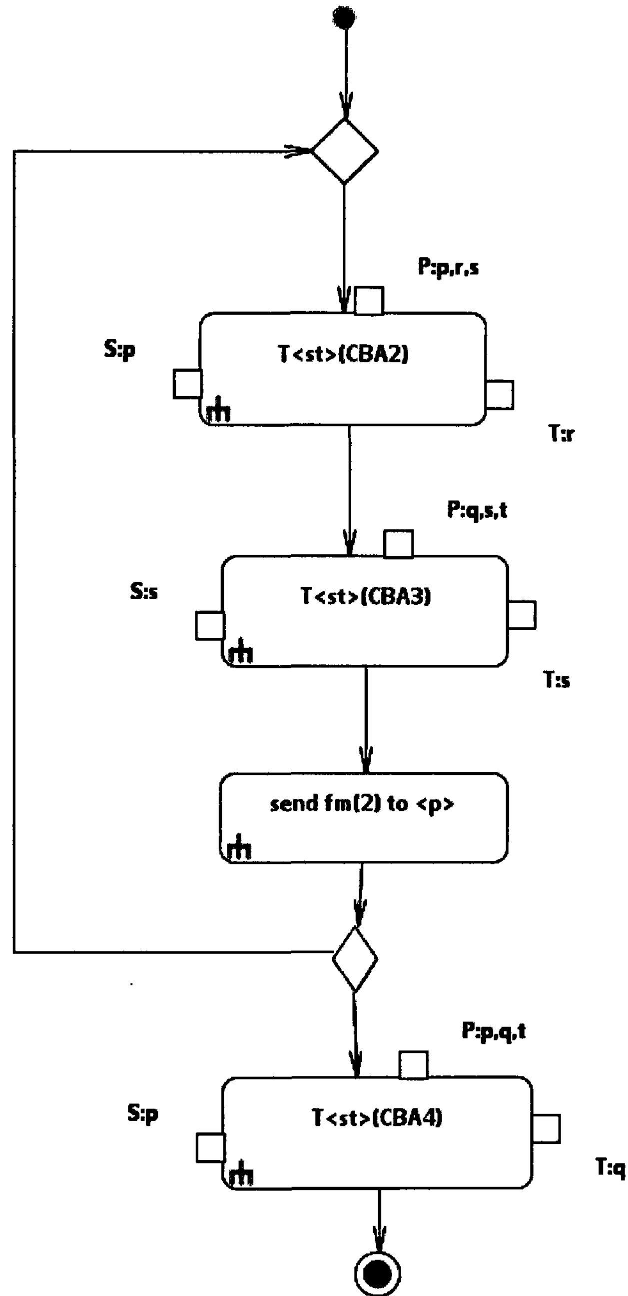

FIGURE 4.14: Derivation of the strong loop AD for the component st

4.3.4 Weak While Loop

The following scenario is an example for the weak while loop. As shown in Figure 4.15, the collaboration CBA1 is followed by a weak while loop including the collaborations CBA2 and CBA3, weakly sequenced. The loop is itself followed by the collaboration CBA4. CBA4 invokes another Activity Diagram that details its behavior, let us call it CBA4-Behavior. This diagram contains two sub-collaborations separated by a strong sequence. The first one is the sending of a message between two components, followed by a strong sequence, and then by a second sub-collaboration CBA4'.

The derivation of the behavior of components p, r and q are shown in Figures 4.16, 4.17 and 4.18, respectively. The first coordination message for component p is the flow message fm(1) that it receives from component r. This message is due to the strong sequencing between CBA1 and the loop, and since r is the terminating component of CBA1 and p is the starting component of the loop, then r has to send a flow message to p. This will prevent p from initiating the loop before CBA1 is terminated, because p is waiting for a message from r, which will not be sent until r is done with its role in CBA1.

Component p plays the starting role in the weak loop, and is the one responsible for the choice between either performing another execution of the loop or proceeding with the following collaboration CBA4. Since the component r does not participate in CBA4, and plays a non-starting role in the loop, then p is responsible for sending a choice indication message (cim) to r when p takes the decision to end the loop. p sends this message in parallel to performing its role in CBA4. This cim message will avoid that r waits indefinitely for another execution of the loop to be performed.

The cim message has two parameters. The first parameter refers uniquely to the choice construct between the loop and CBA4. The second parameter is a loop counter. This parameter indicates the number of times the loop has been performed. The component p as well as r keep track of an internal loop counter by executing the action "Increment LoopCounter" during each loop. Then when p sends the cim message to r after the last execution of the loop, the loop counter included in this cim message will allow r to know

exactly how many times the loop has been performed, and then compare it with its internal loop counter. If the latter is smaller, then r will not consume the cim message yet, and will still wait for additional messages belonging to the loop, while incrementing its internal loop counter at each loop instance. When they become equal, then r can consume the cim message and consider its role in the loop to be finished.

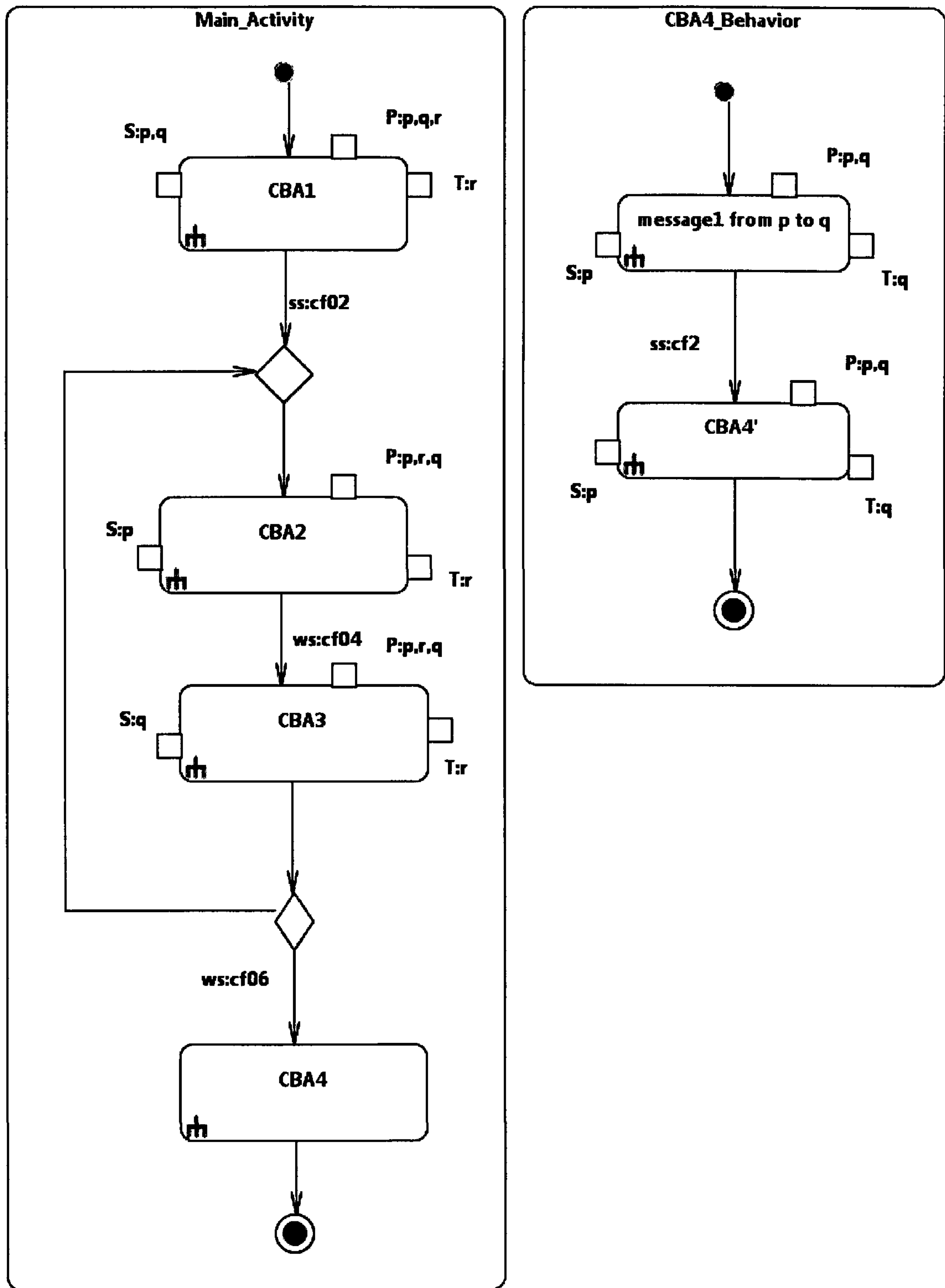

FIGURE 4.15: Weak loop Activity Diagram

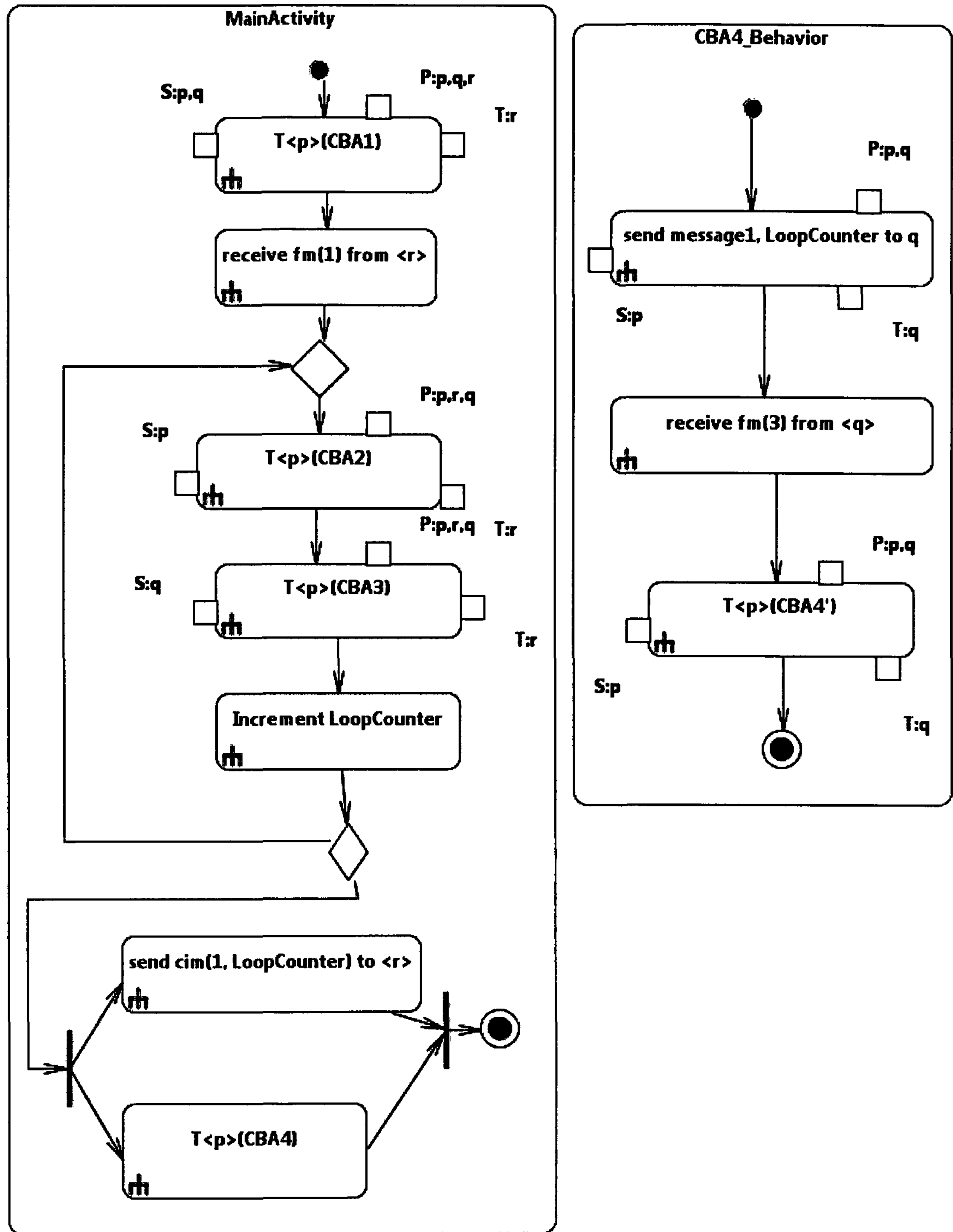

FIGURE 4.16: Derivation of the weak loop AD for the component p

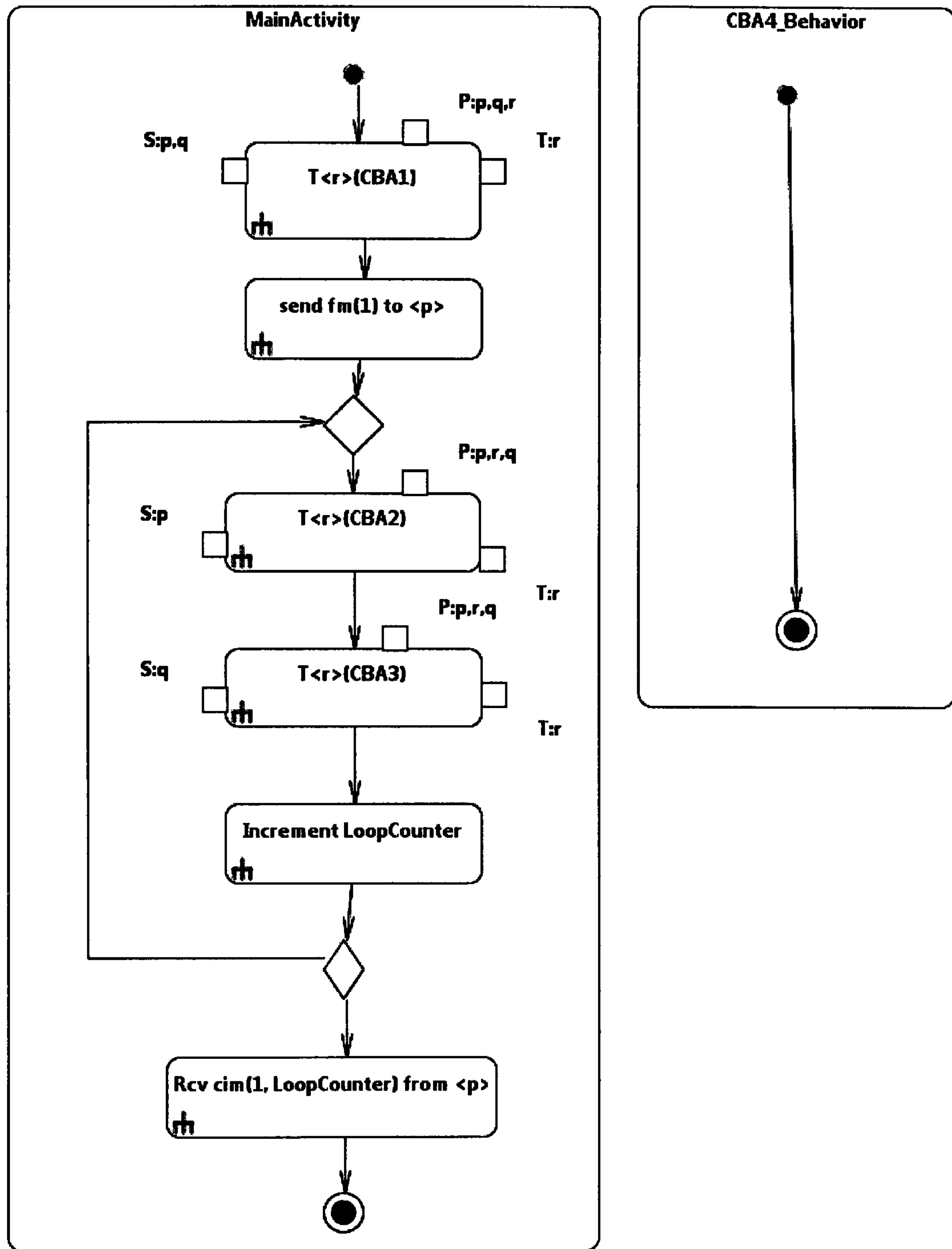

FIGURE 4.17: Derivation of the weak loop AD for the component r

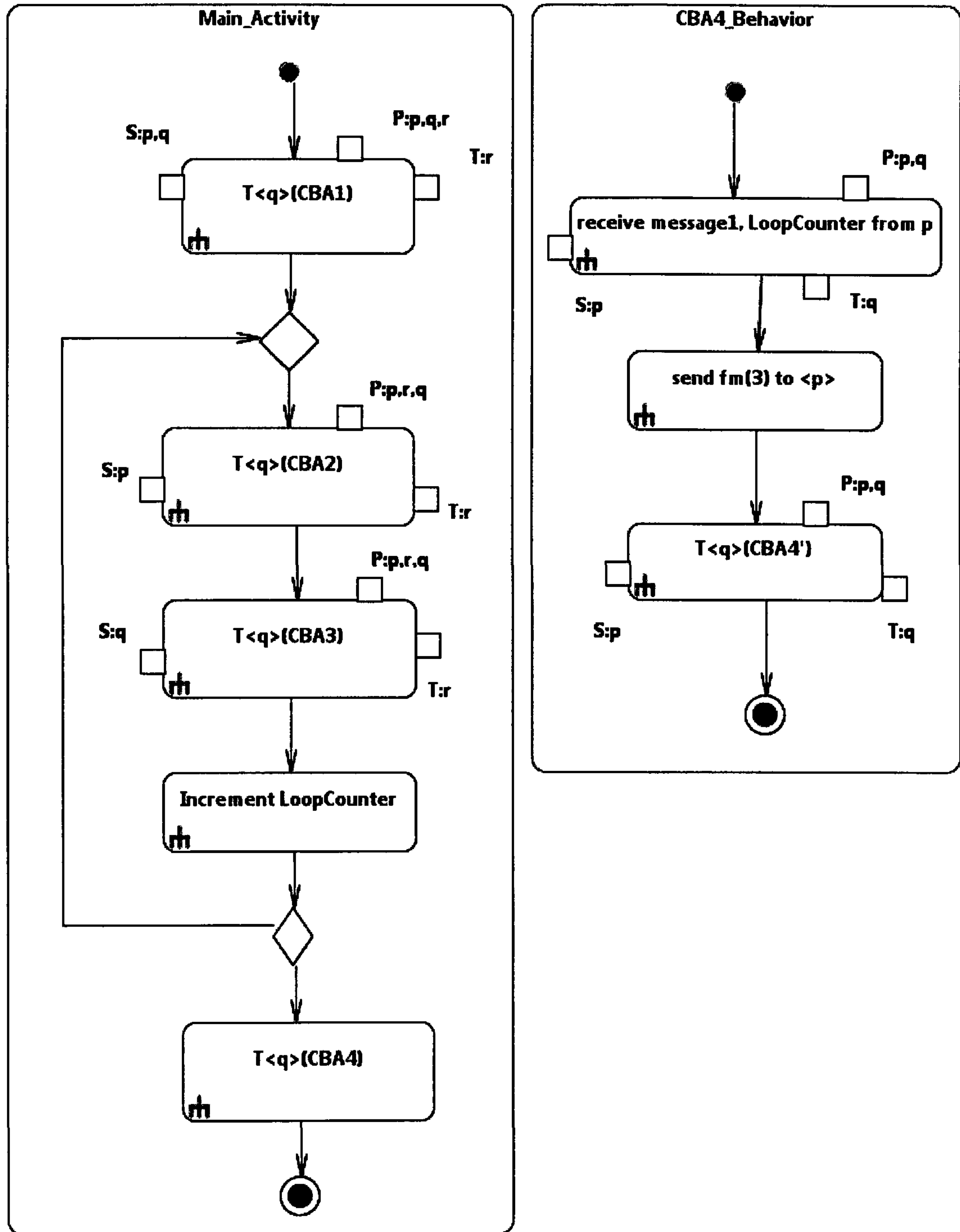

FIGURE 4.18: Derivation of the weak loop AD for the component q

Contrary to the other collaborations, the collaboration CBA4 that comes after the loop, does not have its starting, participating and terminating roles specified. This is permitted in this case because CBA4 makes a call to the sub-Activity Diagram CBA4-Behavior. The Derivation Tool calculates the roles of CBA4 from the sub-Activity Diagram, by using the rules of the table presented in Section 3.3.1.

The component q participates both in the collaboration CBA4 and in the loop. So q has to receive the loop counter as an additional parameter, as part of the first message that it receives in CBA4, that is message1 that belongs to the sub-Activity Diagram CBA4-Behavior. The role of the loop counter here is to allow component q to know whether the loop is actually finished, or if message1 has just been received earlier because of the weak sequencing, while the loop is still running. In the last case, the component q will not consume this message until it has received all messages belonging to the loop. q will be able to know this by comparing the internal loop counter it is keeping track of, with the loop counter received from p.

4.4 Summary

We have described in this chapter the different requirements for the Derivation Tool. We have shown for each construct the derivation rules that need to be applied, and we have illustrated that by examples of input Activity Diagrams for the global system, as well as output Activity Diagrams for the derived system components. We note that the diagrams used in this chapter are the actual ones that were obtained by our Derivation Tool after its implementation was finished.

We note that the GMF's UML2Tools plugin, which is the editor that we used in our project as we explain in Chapter 6, does not support yet the interruptible activity region, we are therefore not able to accomplish that specific construct. The extension of the Derivation Tool can be done when the GMF Editor is upgraded to support the interruption construct while creating Activity Diagrams.

Chapter 5

DESIGN CHOICES FOR THE DERIVATION TOOL

This chapter describes the design choices that were made during the implementation of the derivation algorithm. The first conceptual choice is related to the ambiguity that occurs when an Activity Diagram has different combinations of strong and weak sequencing constructs. This ambiguity is eliminated by prioritizing one type of sequencing over the other. The second conceptual choice concerns the tree representation for Activity Diagrams. The tree data structure provides efficient and fast data access through bottom-up traversals of the tree.

5.1 Strong Sequencing v.s. Weak Sequencing

In a regular UML Activity Diagram, the only sequencing used is strong sequencing. When we introduce the use of weak sequencing in the extended Activity Diagrams used here, there is an ambiguity that occurs when we have different combinations of strong and weak sequencing constructs. The cause of this ambiguity is that when we have a weak sequence followed by a strong sequence (or in the opposite order), the set of flow messages exchanged to ensure the strong sequencing, will be different, depending on whether priority is given to the strong or the weak sequence.

This ambiguity can be better understood in analogy with the mathematical operator precedence. Let us take for example 1 + 2 * 3. If we agree that priority is given to multiplication as it is usually the case, then there would be no ambiguity, and the result would simply be 7. But if we do not agree on any operators priority, then the expression 1 + 2 * 3 is ambiguous as the result could be 7 if priority is given to multiplication, or it could be equal to 9 if priority is given to addition. We could use parentheses and either write 1 + (2 * 3) or (1 + 2) * 3; this way the ambiguity is eliminated. Since in Activity Diagrams there are no parentheses, we could solve the problem by defining priority for either weak or strong sequencing.

To show the ambiguity in Activity Diagrams, let us consider the following simple example. Figure 5.1 shows the process through which some new computers are ordered. The following labels are used for the different roles that participate to the collaborations in the diagram: M: IT manager, S: Software developers, A: Accounting department, P: Hardware provider. To simplify the situation, let us also call M, S, A and P, the components that realize these roles.

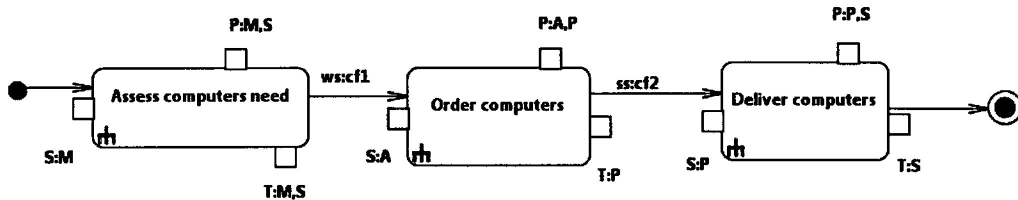

FIGURE 5.1: Weak and strong sequence ambiguity

There are actually two ways to deal with this Activity Diagram. The first one, illustrated by Figure 5.2, is to consider that the strong sequencing has priority over the weak sequencing, and therefore adopt the bracketing shown in that figure. This way, there is strong sequencing between the collaborations "Order computers" and "Deliver computers".

So the flow messages needed in this case, are messages sent from the terminating roles of "Order computers" to the starting roles of "Deliver computers". That is a message

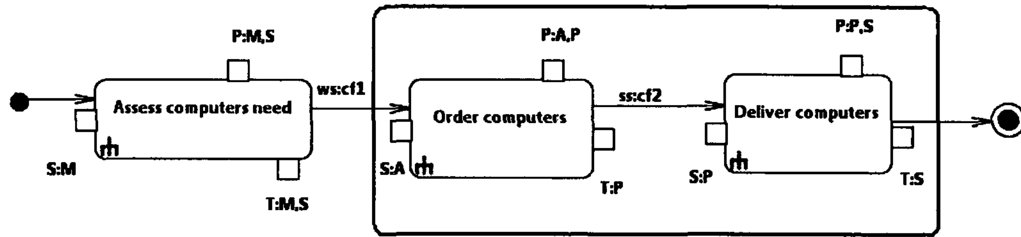

FIGURE 5.2: Weak and strong sequence : priority of strong sequencing

from P to itself which is actually eliminated by the algorithm. This means that in this case no coordination messages are needed at all.

The second way is to consider that weak sequencing has priority over strong sequencing. In this case the bracketing to be adopted is illustrated in Figure 5.3.

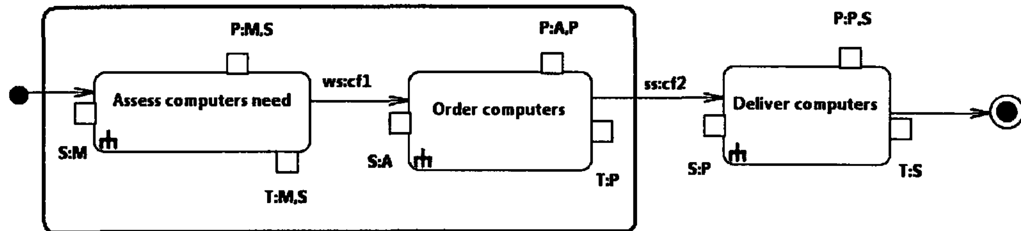

FIGURE 5.3: Weak and strong sequence : priority of weak sequencing

As the figure shows, there is strong sequencing between the combination of the collaborations "Assess computers need" and "Order computers" together, and "Deliver computers". Let us see what the required coordination messages are in this case.

The new set of terminating roles for the combination "Assess computers need" and "Order computers", following the table in Section 3.3.1, is (TR(Assess computers need) - PR(Order computers)) U TR(Order computers). This is equal to the set of roles { M, S, P }. So to ensure the right coordination between the components in the diagram, each role in this set above is required to send a flow message to each of the starting roles of the collaboration "Deliver computers". This means that each of { M, S, P } will send

a message to { P }. Since we eliminate the message sent from P to itself, then we have two messages, one from M to P and a second one from S to P.

So, in conclusion, we see that different sets of messages are exchanged, depending on the choice of priorities. The same problem arises if there are more than three collaborations involved, or if there are other constructs linking some of the collaborations, among the strong and weak sequences. We note that this problem does not occur in standard UML Activity Diagrams, since all sequences are strong sequences.

We believe that in most cases where strong sequence is used anywhere in a diagram, this strong sequencing needs to be enforced between all activities that come before this strong sequence and those that come after it. Therefore, we assume that weak sequencing has priority, as shown in Figure 5.3.

A feature that could be added later to the Derivation Tool would be to allow the user to choose which priority he would like to have in the diagram. Another feature that would be nice to have, would be to give the user a way to specify for each case of strong or weak sequencing, which priority he would like to adopt, by using some bracketing that would be allowed by the editing software when building the Activity Diagram. This way, the user does not necessarily have to use the same priority in all the cases.

This bracketing can currently be ensured by the use of sub-activities. Indeed, if the used wants to have strong sequencing between two collaborations, then he can use a sub-activity diagram where these two collaborations are separated from the rest of the activities in the main Activity Diagram. This way, even if this sub-activity diagram is preceded by weak sequencing, the strong sequencing between the two collaborations will still be ensured. This however may not be convenient for the user in case the Activity Diagram contains many such cases, and thus the user would need to use many sub-activities to separate strong sequenced collaborations from the rest of the diagram.

5.2 Tree Representation for Activity Diagrams

During the derivation process of an Activity Diagram, there are several operations that should be performed. The first operation consists of validating the Activity Diagram to make sure that all the conditions necessary for the derivation are satisfied. The second operation, which is launched only when the first one succeeds, takes care of computing the starting roles, terminating roles and participating roles of each collaboration. Once the sets of roles are determined, the third operation, which processes of the derivation transformations, is launched. These operations imply the need for several traversals of the Activity Diagram, and therefore a suitable data structure to facilitate the traversals and to make the data access more efficient.

A binary tree data structure would be an intuitive answer to this requirement. The binary aspect emanates from the fact that the transformations are based on rules involving exactly two sub-collaborations linked through a specific relationship which represents the sequencing construct. Concerning the interruption for which three collaborations 'C1', 'C2' and 'C3' are involved, we adopted the competition construct which complies with the binary tree structure.

The choice of the tree as a data structure to represent the Activity Diagram, allows us to go through the Activity Diagram in a suitable way, using tree traversals. Moreover, the parent/children concept in the tree makes the management of some specific relationships such as the strong sequence and the loop easier.

5.2.1 Binary Trees

A binary tree is made of nodes, where each node has a data element in addition to at most two pointers known as the "left" child and the "right" child. These left and right pointers recursively point to smaller "subtrees" on either side. The node that is the topmost node in the tree is called the "root".

A null pointer represents a binary tree with no elements (the empty tree). The formal recursive definition is: A binary tree is either empty (represented by a null pointer), or

is made of a single node, where both the left and right pointers point to a smaller binary tree (recursion) [27].

Figure 5.4 shows a binary tree with an illustration of the root node as well as the "left" and the "right" children nodes and their parent node.

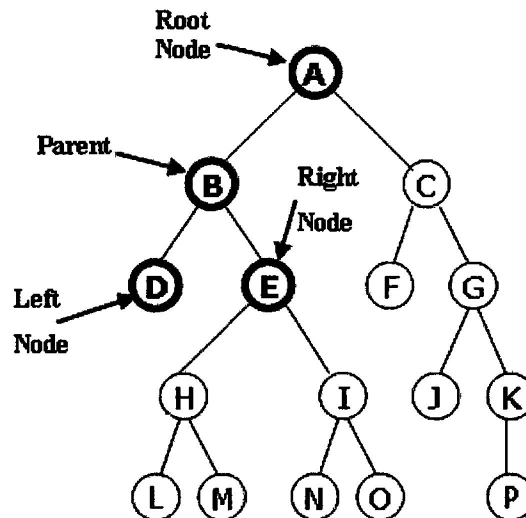

FIGURE 5.4: A Binary Tree

5.2.2 Binary Tree Traversal

There are many reasons why we might need to traverse a binary tree. For example, we might be looking for the number of nodes in that tree, we might as well wish to find the largest or the smallest value. The tree traversal algorithms that was used during our project is the Depth First Search (DFS).

A Depth First Search explores a path all the way to a leaf before backtracking and exploring another path. It allows to visit a node and then recursively visits the children of that node.

For example, in Figure 5.4, after searching A, then B, then D, the search backtracks and tries another path from B.

Figure 5.5 shows the order in which the nodes are explored, namely A B D B E H L H M H E I N I O I E B A C F C G J G K G C A. Each node is visited exactly three times. The line in the figure shows when the nodes are visited and not when they are processed. The processing of the tree nodes is usually carried out bottom-up, that is, a node is processed when all its children have been processed.

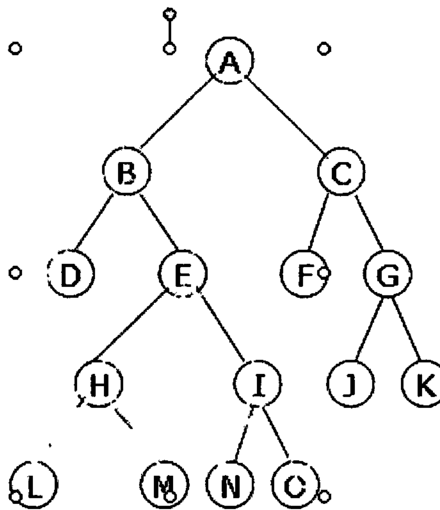

FIGURE 5.5: Depth First Search

Each intermediate node of the tree is visited three times when using a DFS traversal. Once on its way down the tree, a second time coming up from the left child, and a third time coming up from the right child. It is during the third visit that the data from the left and the right child is available, and then the processing (computation of roles in our case) is performed. Obviously, leaves are visited once since they have no children.

5.2.3 The Input XMI Format

The input of the Derivation Tool we developed is a global Activity Diagram in the form of an XMI file. The XMI file describes the Activity Diagram based on XML syntax. Each UML object has a name used to instantiate it in the XMI file. Each instance, no matter what type of object, has an identifier insuring its uniqueness. This identifier is

used as a reference to the corresponding instance. When two object instances are linked through a Control Flow Object instance, their unique identifiers are used to define that relationship using specific attributes.

Control flows define the attributes "source" and "target" holding the identifiers of the source and the target collaborations of the corresponding sequence. Any other UML object (Collaboration, Initial Node, Final Node, Fork Node, Merge Node, Decision Node or Merge Node) defines the attributes "incoming" and/or "outgoing". The "incoming" attribute defines the identifier of the control flow that ends at the UML element while the outgoing attribute determines the control flow that comes out of the UML element.

The definition of an Activity Diagram starts with an Initial Node, goes through a certain number of object instances (Collaboration, Fork Node, Join Node, Decision Node, Merge Node) linked through Control Flow Object instances and finishes with a Final Node.

5.2.4 Input Validation

The syntactical validation of the global Activity Diagram is performed automatically by XMLBeans [28] during the parsing of the XMI file. The parsing would simply fail if the XMI file contains an XML syntax error or its format does not conform to the schema of the UML Metamodel.

The semantic validation is performed by the Derivation Tool during the tree construction, and ensures that the following conditions are satisfied:

- A single Initial Node starts the Activity Diagram and a single Final Node ends it.
- There should be a non-empty set of starting roles, a non-empty set of participating roles and a non-empty set of terminating roles for each collaboration in the Activity Diagram. An exception to this rule is made for collaborations that make a call to a sub-Activity Diagram, in which case the roles of the calling collaboration are calculated from the called sub-Activity Diagram.
- All control flows are defined as either strong or weak. Obviously, this takes into account only control flows where this information is required.

- All the combinations of fork/join and decision/merge should be well-structured.
- All choices are local.
- The branches of the choice constructs have the same terminating roles.

5.2.5 Tree Construction

The tree construction starts by creating the root identified by the Activity Diagram identifier. This is the starting point of our binary tree. Each node in our tree will either be a leaf or will have two children nodes (left node and right node). A node will be a leaf only if it represents a call behavior action, otherwise, the two children nodes should be created inside the tree data structure.

The XMI file parsing allows pointing out the next object instance coming right after the current Node. The nature of the pointed object (Collaboration, Fork Node, Join Node, Decision Node, Merge Node) decides whether the child is a leaf or an intermediate node. The only case where a leaf is determined is when a call behavior action is encountered, otherwise an intermediate node is built and the tree construction function that was called previously, is called recursively. The recursion starts at the Initial Node, and ends when the Final Node is encountered.

The recursion paradigm is of much importance in the process of the tree construction, since we have a priori no idea about neither the nature of the Activity Diagram, nor its hierarchy.

Here are some rules that were implemented during the tree construction:

- If a call behavior action is encountered, the left child is created as a leaf, the right child is created as an intermediate node and the relationship (strong sequence or weak sequence defined by the control flow coming out of the encountered collaboration) is kept as an attribute in the data element of the parent node. The control flow coming out of the call behavior action in question is processed recursively.

- If a Fork Node is encountered, we go further in the Activity Diagram until we reach the corresponding Join Node. A First In Last Out (FILO) stack takes care of more than one embedded Fork/Join nodes. Two intermediate children nodes are created and the "parallel" relationship is specified in the data element of the parent node. The two control flows coming out of the fork node are processed recursively.

- If a Decision Node is encountered, we go further in the activity digram until we reach the corresponding Merge Node. A First In Last Out (FILO) stack takes care of more than one embedded Decision/Merge nodes. Two intermediate children nodes are created and the "choice" relationship is specified in the data element of the parent node. The two control flows coming out of the decision node are processed recursively.

- If a Merge Node is encountered we first make sure we are not in the previous case, then we go further in the AD until we reach the corresponding Decision Node. This will allow us to identify a loop. Whether the loop is strong or weak is determined through the nature of the control flow coming out of the Decision Node and not going back to the corresponding Merge Node. Two intermediate children nodes are created and an attribute holding the "loop" characteristic of the left child node is defined in its data element. The control flow coming out of the merge node and the one coming out of the decision node and not going back to the merge node are processed recursively.

In the case the global Activity Diagram calls other sub-Activity Diagrams, a tree is constructed for each called sub-Activity Diagram as described above. The constructed trees are then linked to the global tree in the following manner. Each node calling a sub-Activity Diagram in the global tree, points to the root of the tree of that sub-Activity Diagram.

Figure 5.6 shows an example of an Activity Diagram containing a fork/join combination and a decision/merge combination. Figure 5.7 illustrates the constructed tree corresponding to the Activity Diagram defined in Figure 5.6. The tree in the figure shows

the node attributes defining the relationship between the two children nodes. This relationship (strong or weak sequence, concurrence, choice, strong or weak loop) is retrieved from the XMI data during the tree construction process.

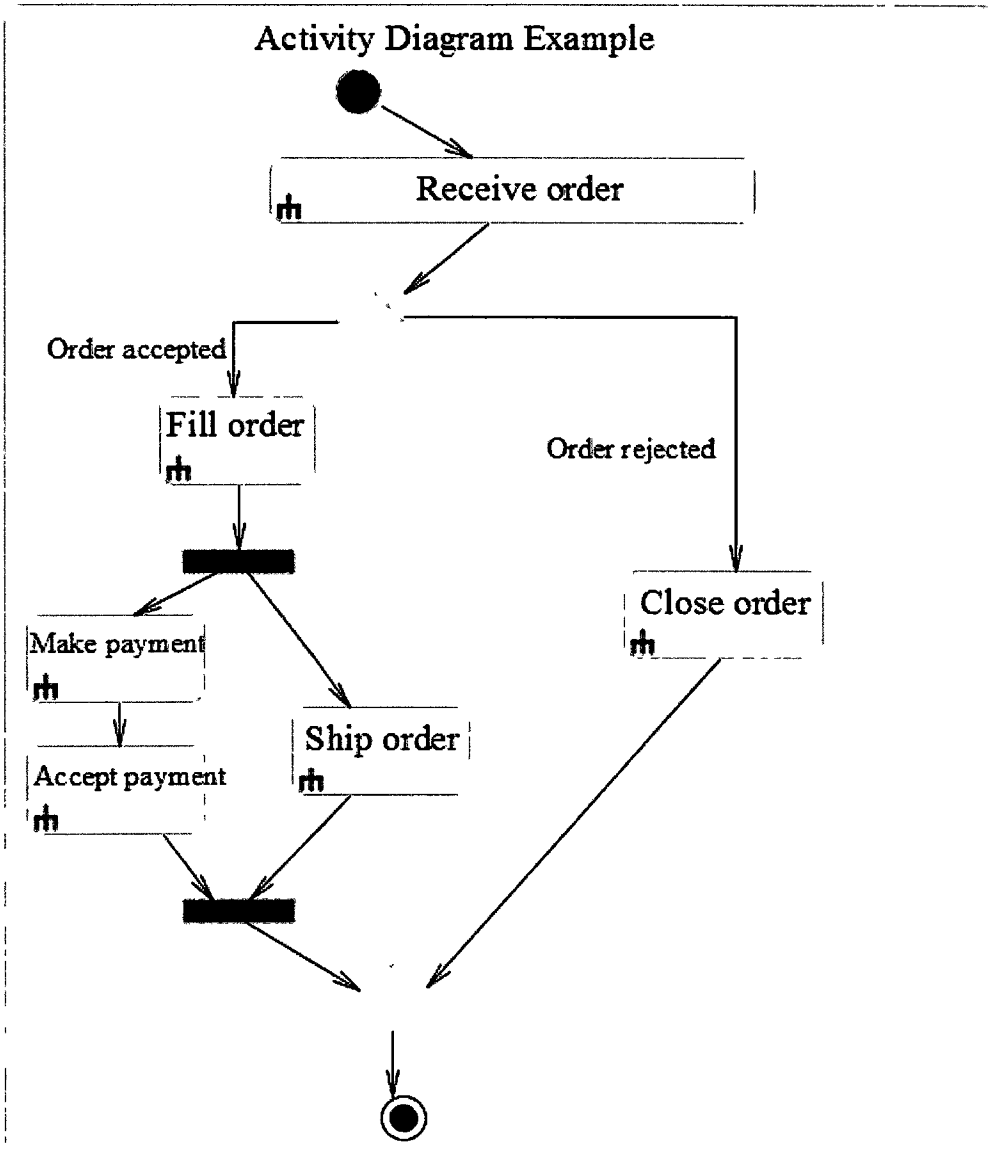

FIGURE 5.6: An Activity Diagram example

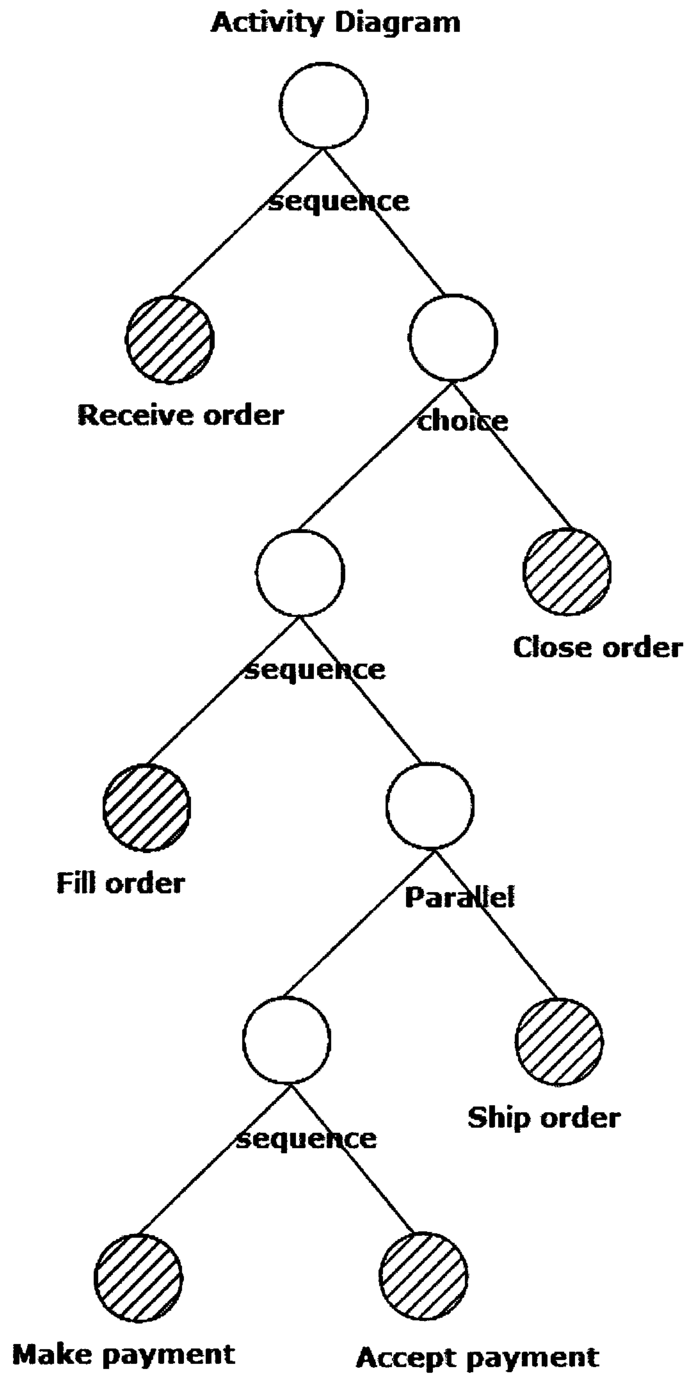

FIGURE 5.7: Tree for the Activity Diagram above

5.3 Role Computation

Role computation is achieved through a bottom-up traversal of the constructed tree. This ensures that the leaves are the first nodes to be processed and then their parents and so on up to the root. The roles (starting, participating and terminating roles) of the leaves are defined in the input Activity Diagram and therefore are available with the constructed tree, while the roles of the intermediate nodes up to the root are computed based on the rules defined in the role computation table of Section 3.3.1.

5.4 Behavior transformation

A bottom-up traversal is again used for the behavior transformation of every component in the system. For the leaves, which correspond to collaborations of the input Activity Diagram, we apply the transformation by changing the names of those collaborations. The naming change consists of adding a prefix 'T<c>' to the original names of those collaborations where 'T' stands for 'Transformation' and 'c' is the component name for which the transformation is performed. For example, if the name of a collaboration is "Order" in the input Activity Diagram, it becomes "T<compl>(Order)" in the generated Activity Diagrams describing the behavior of the component 'compl'.

For an intermediate node, we apply the transformation by running the rules of the transformation algorithm defined in the transformation table in Chapter 3. The applied rules in each case depend on the relationship between the children nodes. The relationship information, which is retrieved from the input XMI data during the tree construction process, is stored as an attribute of the intermediate node.

Since a transformation is related to a component, we repeat it for as many components as we have. This implies that we need a number of bottom-up traversals of the constructed tree equal to the number of the components in the Activity Diagram. The components are defined by the user through an 'INI' file (initialization file) where each component contains a non empty set of roles. No role can be part of more than one component. This 'INI' file has to respect the following format. Each component is defined in a separate

line, by the component name followed by the sign equal "=", followed by the list of roles that belong to this component separated from each other by a comma. A simple example of such an 'INI' file is as follows:

Component1 = Role11, Role12

Component2 = Role2

Component3 = Role31, Role32

This means that the components participating to the corresponding Activity Diagram are Component1 which implements Role11 and Role12, Component2 implementing Role2, and Component3 which implements Role31 and Role32.

Chapter 6

IMPLEMENTATION OF THE DERIVATION TOOL

This chapter presents the overall architecture of the derivation software tool, that does the transformation of a global Activity Diagram into component Activity Diagrams. The derivation process involves five phases described in detail in this chapter. Here is an overview of these phases:

- The generation of the Java instances corresponding to the input XMI elements: The XMI file describing the Activity Diagram to be transformed is parsed using XMLBeans in order to generate Java objects making it easier to handle the data. The parsing process of the input file also involves the syntactic validation, making sure that the input XMI file is well formatted.
- The semantic validation: This phase consists of verifying that the input XMI file respects the conditions detailed later in this Chapter in Section 6.4 before starting the derivation process.
- The global Activity Diagram tree construction: This phase consists of mapping the input Activity Diagram into a tree data structure. All leaves in the tree correspond to collaborations in the Activity Diagram, while the intermediate nodes specify the construct defining the relationship between its children. In this chapter, the global

Activity Diagram tree construction is described from an implementation point of view. The previous chapter described this tree construction thoroughly from a conceptual point of view.

- Calculation of roles: This phase is launched right after the tree is constructed. The fact that all the collaborations in the input Activity Diagram correspond to leaves of the constructed tree is a very important feature of our data structure. Indeed, the roles are specified for the collaborations by the user when defining the global Activity Diagram. Therefore, the roles are known for the leaves of the tree, since the leaves correspond to collaborations. As a result, a bottom-up traversal is enough to complete the calculation of the roles of the whole tree.

- The behavior transformation: This phase consists of applying the transformation on the tree mapping the global Activity Diagram. The transformation is performed through a bottom-up traversal of the tree for each component. At the end of each transformation, an XMI file representing the behavior of the processed component is generated.

First, let us start by presenting the different tools that were used during our project, namely the Graphical Modeling Framework of the Eclipse platform, as well as XML-Beans, which we used to generate Java classes from XML input documents as we will explain in detail later.

6.1 Implementation Tools

6.1.1 Graphical Modeling Framework

During the development of this project, the Eclipse platform was used. Eclipse is an open source community [29], that created the open Eclipse development platform and its related projects. The Graphical Modeling Framework (GMF) [30] is one of the Eclipse plug-ins. GMF is an open source tool that provides a set of generative components and a runtime infrastructure for developing graphical editors [31]. These editors can then

be used for modeling. A set of such GMF editors, UML2 Tools [32], was used in our project.

UML2 Tools provides the ability to view and edit UML diagrams. It offers a palette of graphical UML elements to drag and drop onto the editor. The palette's elements displayed at any given time, change according to the type of diagram being built. Figure 6.1 illustrates the editing of an Activity Diagram. For every diagram created, the editor generates two corresponding files. One of them is used to store the graphical information, and the other contains the data related to the diagram. Both files are stored in XML format.

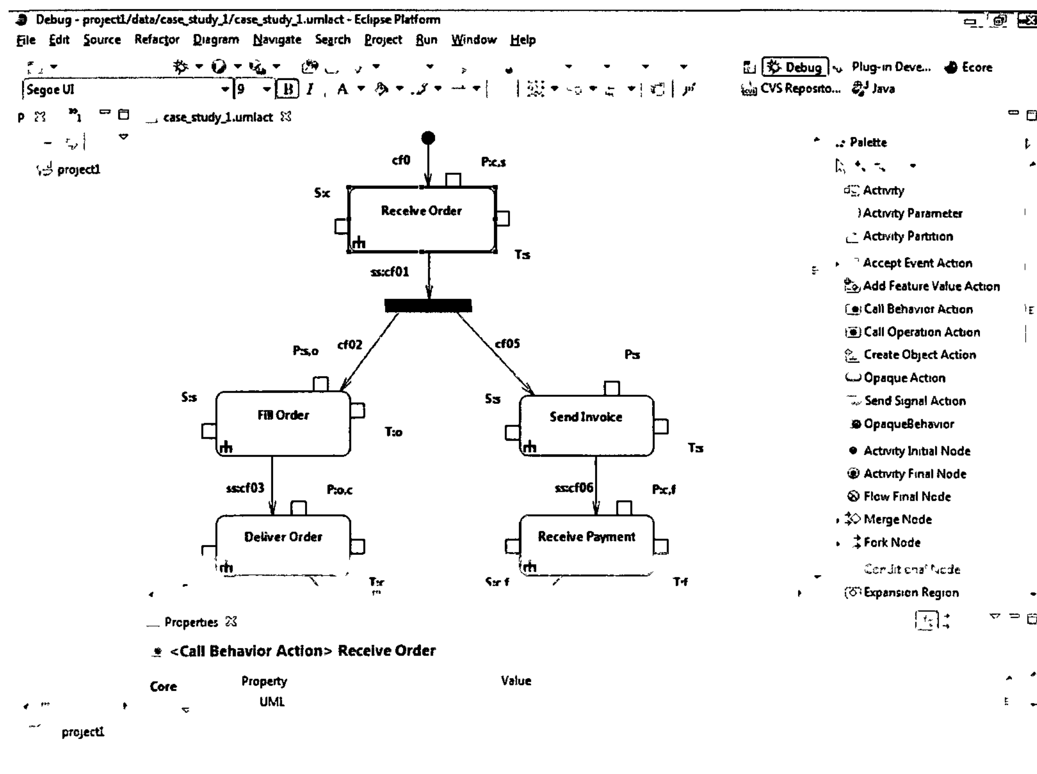

FIGURE 6.1: Editing an Activity Diagram with UML2 Tools (GMF based editors)

6.1.2 XMLBeans

XMLBeans is a technology for accessing XML data through translating it into Java objects. It was in September 2003 that BEA Systems XMLBeans submitted XMLBeans

to the Apache Incubator and Apache XML projects [33]. XMLBeans graduated then from incubation in June 2004 to become a top level project and part of the Apache XML federation. XMLBeans 1.0 has been successfully used as an underlying technology for several products as well as by a growing number of large users including some of the largest companies in the world [33].

The strength of XMLBeans comes from its ability to access the full power of XML in a Java-friendly way. The idea behind it is to take advantage of the richness and features of XML and XML Schema and have these features mapped as naturally as possible to the equivalent Java language constructs. The Java interfaces compiled based on XML documents can be used to read and modify XML instance data.

XMLBeans provides several ways to retrieve data from an XML document. One of these ways is to access the XML document data by parsing the input XML file into Java objects corresponding to the XML data elements. These Java objects are instances of the Java classes generated by XMLBeans while compiling the XSD file. The Java class generation is an operation that is carried out only once, while the instantiation of these generated Java classes following the parsing operation, is performed for each XML data file corresponding to the XSD Schema. The XML data, mapped into the Java instances, is managed through Java getters and setters.

We have opted for using XMLBeans to access information from XML files in our project, because this is powerful, and allows an easy manipulation of the data using the API provided by XMLBeans. In particular, it simplifies searches through the data as well as modifications of the data.

6.2 Overall Architecture

Our program expects three input files. The first file contains, in XMI format, the global Activity Diagram to be transformed. The second one, also generated by GMF's Editor, contains the graphical data of the global Activity Diagram. The last file is a configuration file in INI format, defining the components of the system and the roles assigned to these components.

As an output, our software produces an XMI file for each component of the system, describing an Activity Diagram for the component for which it was generated. In addition, the Derivation Tool duplicates the input graphical file of the global Activity Diagram, for each component of the system. Through a simple configuration of these graphical files, the GMF Editor can associate them to the generated components XMI files, and display the component Activity Diagrams graphically.

The adaptation of a component's graphical file consists of first changing its name to match the name of the corresponding component's XMI file. Then, all links inside the graphical file, that originally referred to the UML objects used in the global XMI file, are replaced by links to the UML objects in the component's XMI file.

This adaptation made by the Derivation Tool, allows the collaborations and the other UML elements that were originally in the global Activity Diagram as drawn by the user, to keep their original coordinates in the components' graphical files. The UML elements added by the Derivation Tool during the behavior transformation, are the only ones that need to be laid out manually within the GMF editor.

During the derivation process, the Derivation Tool performs two main operations. The first one is carried out only once. During this phase, XMLBeans is used to create Java classes corresponding to the elements of the UML Metamodel. XMLBeans takes the schema of the UML Metamodel as input. It then generates a jar file containing all the Java classes required to later parse XML files corresponding to this schema.

The second main operation is executed during each derivation process of a global Activity Diagram. This operation itself goes through five steps as illustrated in Figure 6.2. These steps are detailed in the coming sections.

6.3 The Generation of Java Classes

6.3.1 UML Objects

The XML schema of the UML metamodel defines all the UML objects that are used within the Graphical Modeling Framework (GMF) to construct Activity Diagrams. The

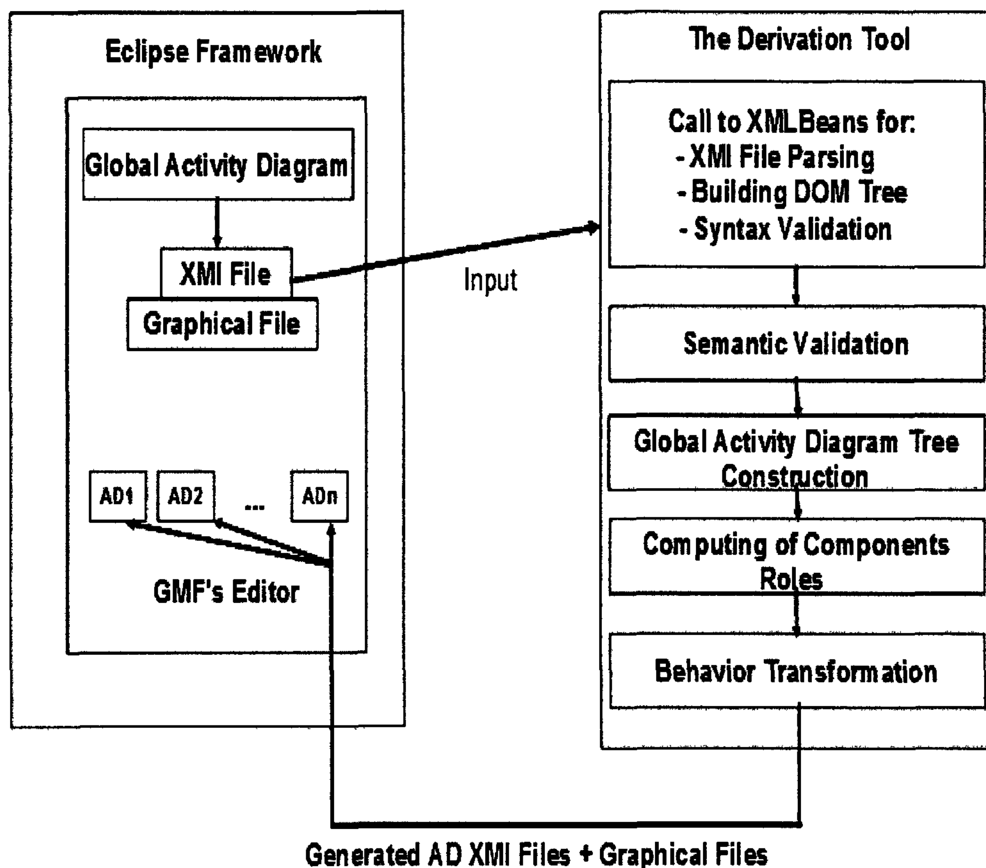

FIGURE 6.2: Overall Architecture of the Implementation of the Derivation Algorithm

data corresponding to these diagrams is saved as an XMI file. Each UML object within the Activity Diagram is defined as an XML element of the XMI file. The properties of that element determine its attributes such as its name, its incoming edges if there are any, its outgoing edges if there are any, the source and the target for the flow objects. Figure 6.1 illustrates the GMF Editor's palette containing the list of UML objects used to build Activity Diagrams. The tab 'Properties' holds the attributes of the currently selected object within the drawing area. Eclipse provides a unique identifier for each drawn object, which can be used for linking purposes.

All the flows in the Activity Diagram are represented by a 'Control Flow' object. Flow

objects are directed edges determining the direction of the flow within the Activity Diagram. A control flow object has a set of attributes among which the property 'source' and 'target' are very important. The source attribute holds the identifier of the collaboration or other object from which the flow starts while the target attribute holds the identifier of the object where this flow ends.

An invocation of a collaboration is represented in the GMF Editor using an object 'Call Behavior Action'. This will allow a call of a sub-activity diagram in case the collaboration requires a more detailed description. To link between the collaboration and its corresponding sub-activity diagram, the identifier of the latter is specified as a value for the property 'Behavior' of that collaboration. This operation allows the behavior decomposition of Activity Diagrams. There is no limit constraint regarding the depth of decomposition.

6.3.2 Java Classes

We have used XMLBeans to translate all the XML elements defined within the UML Schema file into Java classes. The attributes of each element are simply mapped into private attributes and their getters within the corresponding Java class. For example if an XML element holds an attribute 'name', its corresponding Java class will contain a private String attribute 'name' and a getter function which will give access to its value.

Although, the generated Java classes do not translate the relationship defined between the UML objects within the XMI file, they still provide an easier way to get access to and parse the XMI file elements.

6.4 Syntactic and Semantic Validation

The parsing of the input file provides a way to validate the XML syntax and its conformance to the schema corresponding to the UML Metamodel. The system is capable of catching any parsing problem and displays errors to the user accordingly. Obviously, if the XMI file is created through the GMF Editor, it should be parsed with no errors.

Once the syntactic validation is carried out successfully, the system starts the semantic validation to make sure the input Activity Diagram respects the following conditions:

- The Initial Node has no incoming edge.
- Each collaboration has exactly one incoming edge and one outgoing edge.
- In the case where the source of the incoming edge of a collaboration is not the initial node, its name has to specify whether the sequence is strong or weak.
- In the case where the target of the outgoing edge of a collaboration is neither the final node nor a join node nor a merge node, its name has to specify whether the sequence is strong or weak.
- Each Fork Node has exactly one incoming edge and two outgoing edges.
- Each Decision Node has exactly one incoming edge and two outgoing edges.
- Final Node has no outgoing edge.
- Each collaboration has its starting roles, participating roles and terminating roles defined correctly, following the syntax defined in Figure 4.1 of Chapter 4.
- Choices have only one starting role.
- Loops have only one starting role.
- The construct that immediately follows a loop, if it is non-empty, have only one starting role, and this role should be the same as the starting role of the loop that precedes this construct.
- Branches of the choice constructs have the same set of terminating roles.
- The naming of control flows has to respect the following convention, in order to allow the distinction between strong and weak sequencing:
  - In order to define a strong sequence, the property 'name' of the corresponding flow object should start by 'ss:', for 'strong sequence'.

– To define a weak sequence, the property 'name' of the corresponding flow object should start by 'ws:', for 'weak sequence'.

The nature of the sequencing (strong or weak) has to be defined on all the control flows in the global Activity Diagram, except if this information is not required. An example of a control flow where the nature of sequencing is not required is the flow coming out of the Initial Node or the one ending at the Final Node. Other control flows for which this information is not needed are the ones coming out of a Decision Node. Indeed, the nature of sequencing for those is obviously the same as for the control flow that targets that Decision Node. The same is true for Fork Nodes.

In the case that any one of the above conditions is not satisfied, an error message is displayed explaining the error in detail. If all the conditions are satisfied, the tree construction begins.

6.5 Global Activity Diagram Tree Construction

The tree data structure is implemented using linked lists. Each node of the tree is either a leaf representing a collaboration or an intermediate node pointing to a subtree. Each intermediate node has two children, left and right, corresponding to the collaborations C1 and C2, as described in the transformation table in Chapter 3.

The tree construction consists of mapping the Activity Diagram into a tree data structure for which the traversals are faster and more efficient. All leaves in the tree correspond to collaborations in the Activity Diagram, while the intermediate nodes (including the root) correspond to the construct that links the left child with right child of that intermediate node. This is clearly seen in Figure 5.7 of Chapter 5 that represents an example of the tree constructed for a global Activity Diagram.

There are five different cases that determine the relationship between the children of an intermediate node of the tree. These cases depend on the target node of the control flow

represented by that intermediate node. These cases are detailed in the coming sections. They are the following:

- Collaboration followed by Strong Sequence
- Collaboration followed by Weak Sequence
- Decision Node
- Fork Node
- Merge Node

In the cases of choice, concurrence and loop constructs, a stacking mechanism is required in order to detect the ending of each of these constructs. For this purpose, a single stack is defined.

In the case that a Decision Node is encountered during the tree construction, the program adds that Decision Node to the stack and then starts a search for the corresponding Merge Node in order to identify the choice block. In order to achieve this, the program goes through the Activity Diagram adding any encountered Decision Node to the stack. On the other hand, whenever a Merge Node is encountered, the corresponding Decision Node, which is the top element of the stack, is removed. The search ends when the stack gets empty signaling that the corresponding Merge Node is reached. Once the search for the corresponding Merge Node is completed and the choice block is identified, the tree construction process is resumed right after the Decision Node. The same mechanism is used to identify concurrence blocks (Fork Node/Join Node) and loop blocks (Merge Node/Decision Node).

6.5.1 Collaboration Followed by a Sequence

In the case where the target of the current control flow is a collaboration followed by a strong sequence, the tree construction is performed as follows. The Activity Diagram is divided into two sections representing the children of the current node:

- The first section (represented by C1 in Figure 6.3) represents the left child consisting of a leaf. The collaboration 'Collab. 1' in Figure 6.3 is mapped into the leaf node 'C1' in the tree. The roles of this leaf node are defined immediately since they are given by the user in the Activity Diagram.
- The second section (represented by C2 in Figure 6.3) represents the right child consisting of the rest of the tree. 'C2' is an intermediate node of the tree for which the roles are not known at this stage.

The relationship between 'C1' and 'C2' is a strong sequence. This relationship is specified in their parent node 'C' (see Figure 6.3). 'C' is identified using the id of the control flow pointing at 'Collab. 1' and specifies the relationship retrieved from the control flow whose source is 'Collab. 1'.

Finally, the intermediate node C2 is processed through a recursive call of the tree construction procedure, to continue the tree construction until the final node is reached.

The processing of the case where the target of the current flow object is a collaboration followed by a weak sequence is analogue to the case of a strong sequence.

6.5.2 Choice

Figure 6.4 shows the case where the current flow node is followed by a Decision Node. Encountering a Decision Node indicates the beginning of a choice relationship. Our program parses the XMI file following the flow of the Activity Diagram until it reaches the Merge Node corresponding to the first encountered Decision Node. In the case it encounters another Decision Node while looking for a Merge Node, it takes note that it should look for two Merge Nodes instead of one. This is implemented through a stack. Whenever the program finds a Decision Node it adds it to the top of the stack, and when it encounters a Merge Node it removes a Decision Node from the top of the stack. This operation continues until the stack becomes empty, which indicates that the Merge Node corresponding to the first encountered Decision Node is reached.

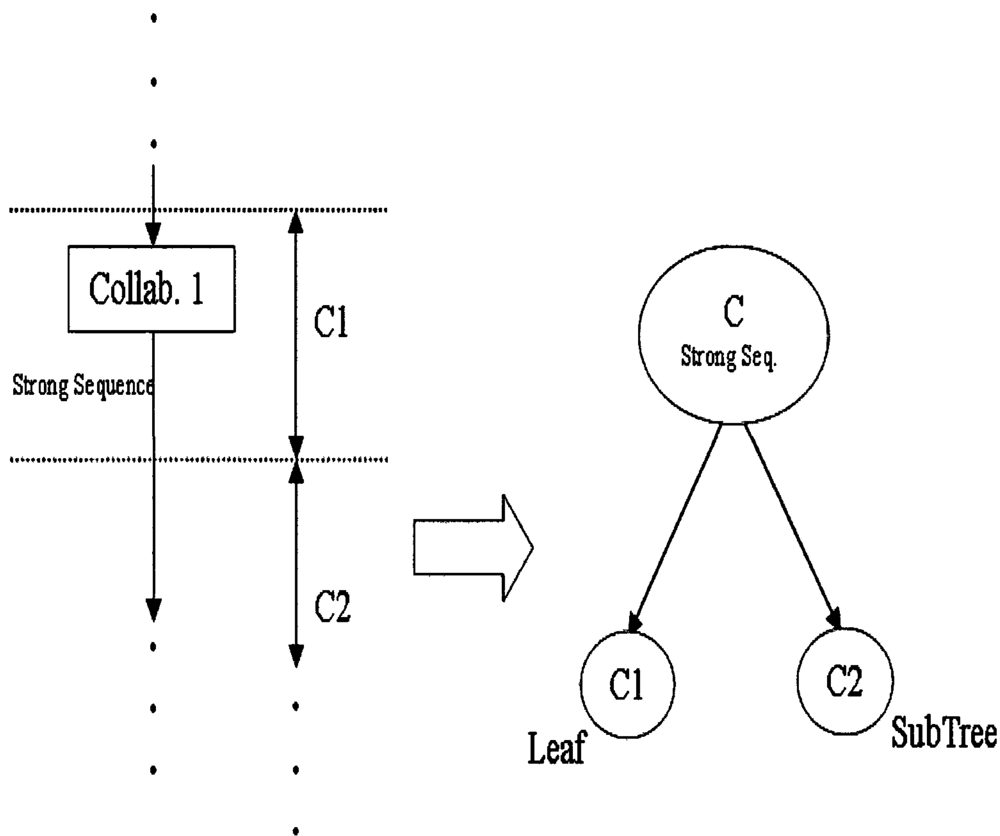

FIGURE 6.3: Tree Construction in the case of a Strong Sequence

In the case of a choice where the target node of the current flow node (represented by the parent Node 'C') is a Decision Node, four intermediate nodes are constructed:

- Subtree 'C1' (Blue Node in Figure 6.4): This is the left child of the parent node 'C'. It represents the choice block going from the Decision Node up to the corresponding Merge Node. Since a choice block consists itself of two paths corresponding to the two flow objects coming out of the Decision Node, it is split into two children nodes. Those two children nodes represent the subtrees mapping the two flow objects coming out of the Decision Node. The choice relationship between these two children nodes is specified at their parent node 'C1' (Blue Node in Figure 6.4). The two children nodes are constructed as follows:

  - Subtree 'C1' (Red Node in Figure 6.4): This subtree represents the left child of the choice block. It is mapped based on the first flow object coming out of the Decision Node. The procedure of tree construction is called recursively at this stage to process this flow object.
  - Subtree 'C2' (Red Node in Figure 6.4): This subtree represents the right child of the choice block. The other flow object coming out of the Decision Node identifies this node. The procedure of tree construction is called again recursively at this stage to process this flow object.

- Subtree 'C2' (Blue Node in Figure 6.4): This is the right child of the parent node 'C'. It represents the rest of the tree which is mapped using the flow object coming out of the Merge Node. It is the nature of this flow object that defines the relationship between the two nodes 'C1' and 'C2'. This relationship is defined at the parent node 'C'.

6.5.3 Concurrence

During the tree construction, if the program encounters a Fork Node, this is an indication that a Concurrence block is ahead. The program determines the Join Node that corresponds to the encountered Fork Node using the same stacking mechanism defined earlier for the choice construct.

Figure 6.5 illustrates the case where target of the current flow node is a Fork Node. The processing of the concurrence construct is similar to the processing of the choice construct described above.

6.5.4 Loops

A loop is recognized when a Merge Node is encountered first. In this case the program looks for the corresponding Decision Node using the same stacking mechanism described above.

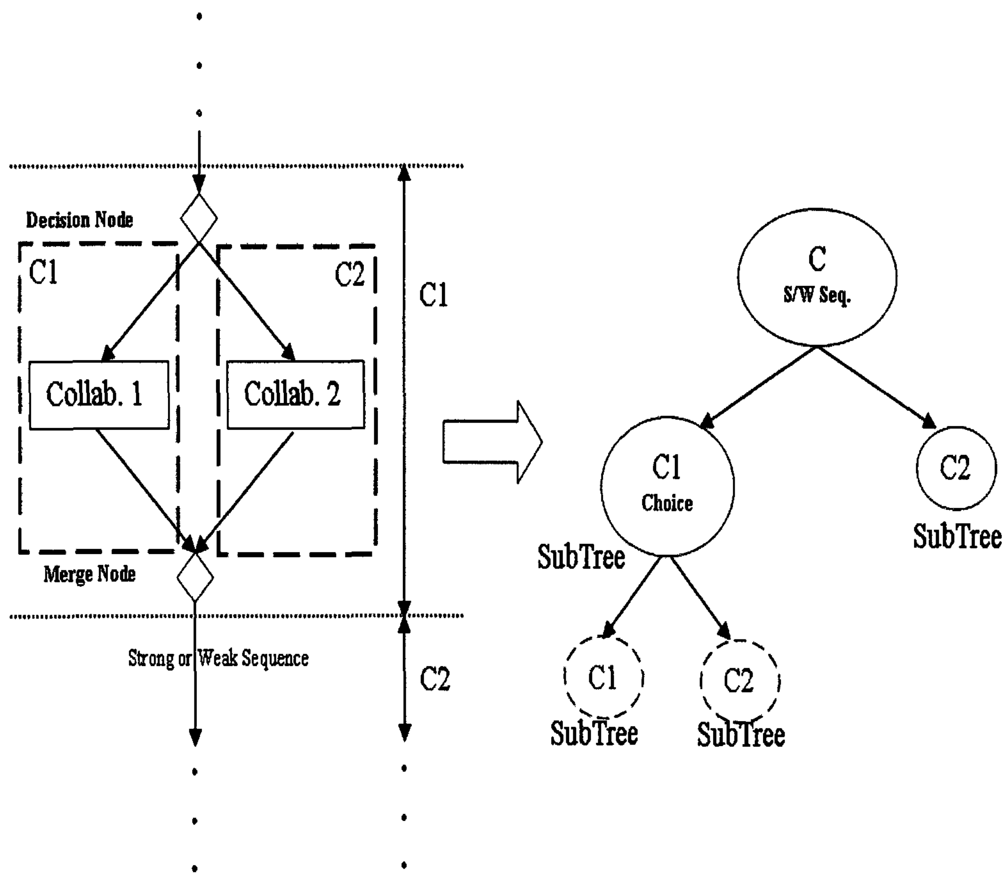

FIGURE 6.4: Tree Construction in the case of a Choice

A strong loop is a loop for which the flow object coming out of the decision node and pointing out the rest of the tree is a strong sequence. Figure 6.6 illustrates a strong loop. A weak loop corresponds to a loop for which the flow object coming out of the decision node and pointing out the rest of the tree is a weak sequence. Also refer to Figure 6.6 for a weak loop illustration, but strong sequence should be replaced by weak sequence.

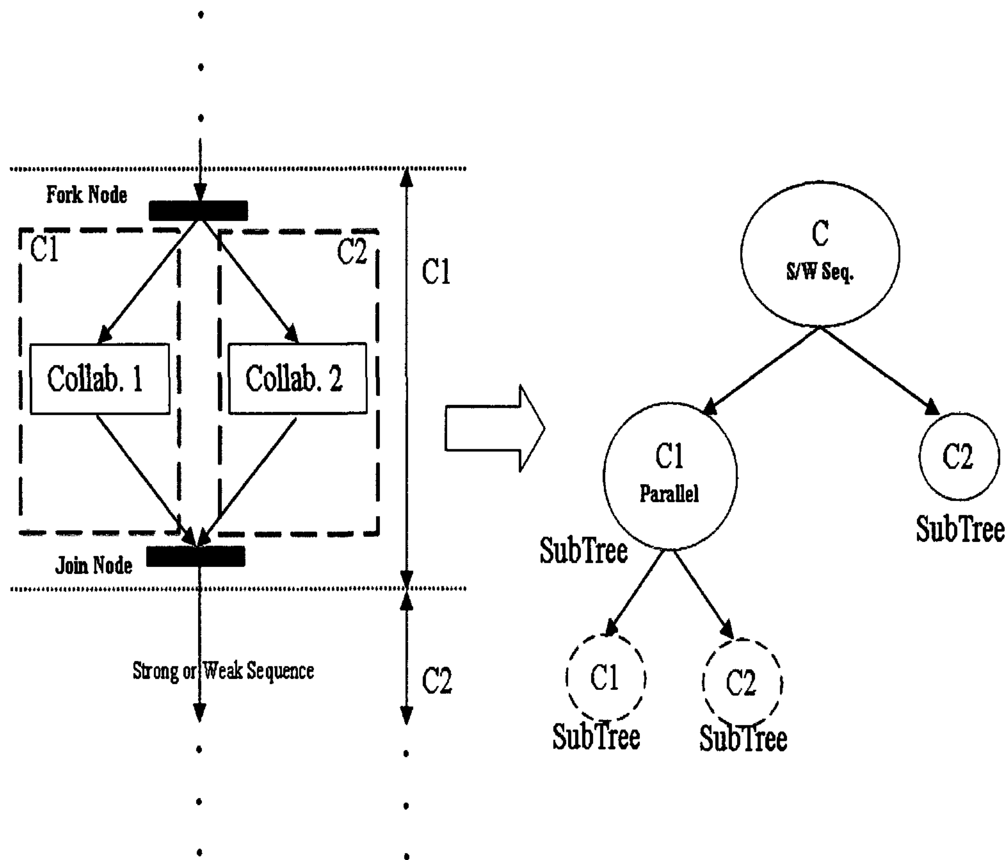

FIGURE 6.5: Tree Construction in the case of a Concurrence

6.5.4.1 Strong Loop

Figure 6.6 shows a strong loop for which the target of the current flow object is a Merge Node. The corresponding Decision Node is identified taking into consideration non-overlapping choices, concurrences and loops. In the case the target of the Merge Node is a strong sequence, the Activity Diagram is divided into two sections:

- Section 1 (represented by C1 in Figure 6.6) which represents the left child consisting of the first subtree mapping the loop block. This child node is identified by the flow object coming out of the Merge Node. A recursive call of the tree construction procedure is called to process this subtree.

- Section 2 (represented by C2 in Figure 6.6) which represents the right child consisting of the rest of the tree. The flow object following the Decision Node and pointing out the rest of the tree represents the intermediate node 'C2' in the tree. 'C2' holds the rest of the tree and corresponds to a subtree for which the roles are not known at this stage. Obviously, a recursive call of the procedure of tree construction handles the processing of the next flow object.

The relationship between 'C1' and 'C2' is a strong sequence relationship. This relationship is specified in their parent node. Figure 6.6 illustrates this aspect.

'C' is identified using the name of the flow object whose target is the Merge Node but specifies the relationship retrieved from the flow object whose source is the Decision Node and points out the rest of the tree.

The loop feature is also defined at the parent node level. This is to say that the left child 'C1' is a loop.

6.5.4.2 Weak Loop

Similar to the strong loop, except that strong sequencing is replaced by weak sequencing.

6.6 Calculation of Roles

Once the tree is constructed, the program starts the computation of the roles. A depth-first search traversal of the constructed tree is carried out for this purpose. The choice of the depth-first search traversal has been made since the leaf nodes have already their roles defined, so we need to start by these leaves and go up.

The path followed by each of these traversals is depth first, left to right, by recursive procedure calls. Each node of the tree is visited three times during each of the depth-first traversals, once on its way down the tree, a second time coming up from the left child, and a third time coming up from the right child.

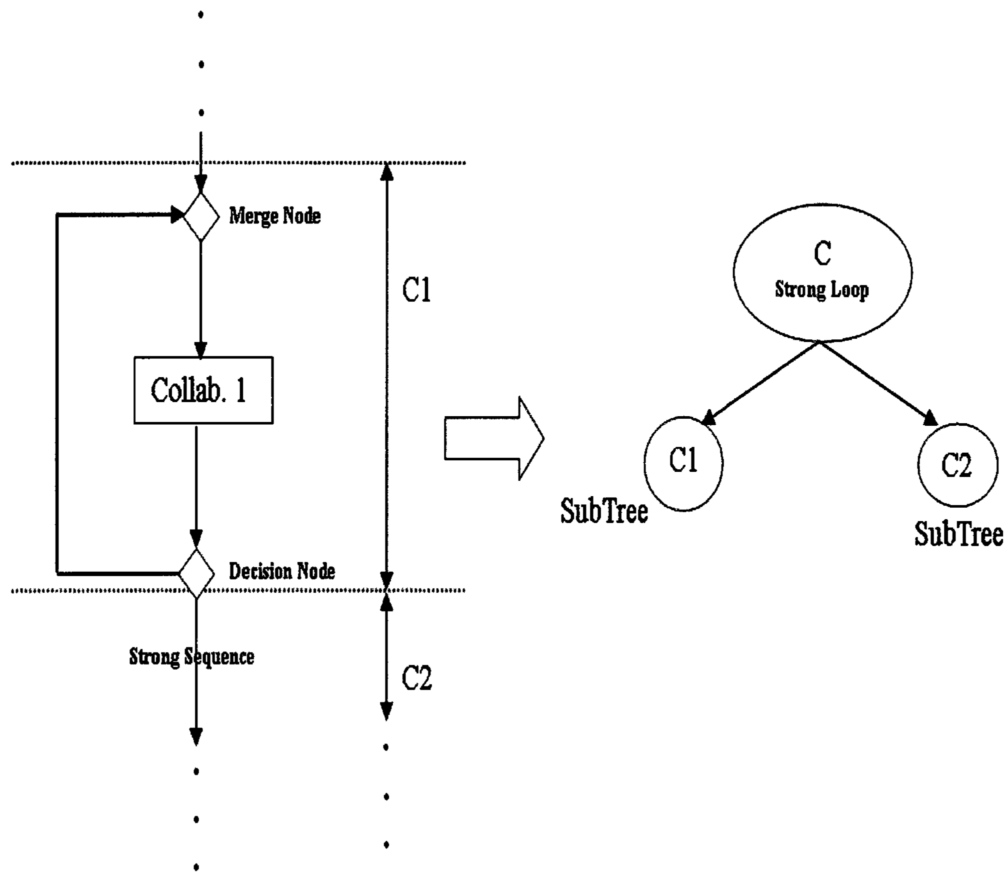

FIGURE 6.6: Tree Construction in the case of a Strong Loop

The implementation of the depth-first search traversal is done through a recursive call of a function based on the following:

- We start by calling the roles computation function on the root node by calling recursively the function on its left child and right child.
- Obviously, if a node is a leaf, its roles are already specified and as a result the recursion stops at this level.
- On the other hand, if a node is a two other recursive calls are performed on the left and right child.

The function of roles computation for intermediate nodes takes into consideration the relationship. See table of roles calculation in Chapter 3.

6.7 Nested Activity Diagrams

In the case where a collaboration in the main Activity Diagram calls another Activity Diagram, a tree is constructed recursively by processing the called Activity Diagram. The leaf node, representing the calling collaboration in the main tree, points at the root of the tree corresponding to the called Activity Diagram. This mechanism is applied whenever a collaboration calls another Activity Diagram. This is an important feature of the transformation software, since it does not limit the depth. So calls to other Activity Diagrams can be made as deep as required in order to develop the behavior of some collaborations. However, recursive activity calls are not allowed in the global behavior. This 'behavior call' relationship in the Activity Diagram is implemented as a parent/child relationship in the tree, and the role calculation of a collaboration that makes use of such calls is made by calculating the roles of the Activity Diagrams that are called, possibly through many depth levels.

6.8 Behavior Transformation

The last step of the derivation process consists of applying the behavior transformation rules. The transformation rules make use of the roles of each node in the tree. The algorithm implementing this step, typically recursive, iterates over the constructed tree using a Depth First Search (DFS) traversal. As a result of the transformation phase, an XMI file is generated for each component of the global system. The generated XMI files represent the behavior of the components in the global system.

In the case the encountered node is a leaf, the renaming of the collaboration is made in a way to indicate that the transformation is applied from a defined component '*c*' point of view. For example, if the encountered collaboration is named '*Reception*', and the transformation being applied is for the component '*c*', the name of this collaboration in

the transformed Activity Diagram corresponding to component 'c' is called '$T < c >$ (*Reception*)'.

The algorithm is called recursively whenever the current node being processed is an intermediate node pointing to a subtree, until the leaves are reached. The relationship defined in a parent node determines the type of the construct and whether there are any synchronization messages that should be implemented, following the rules defined in the transformation table of Chapter 3.

The transformation step is called for as many components as defined in the input file provided by the user and required by the program. Each component represents a list of roles. In a special case, we might have each component representing one and only one role from the set of roles of the main Activity Diagram.

As explained earlier in Chapter 4, the GMF's UML2Tools plugin does not support yet the interruptible activity region, we are consequently not able to implement that specific construct. When the GMF Editor is upgraded to support the interruption construct, the Derivation Tool will then be able to take into consideration the interruption construct, once its implementation in the binary tree has been added, as it is the case for the other constructs. The tree representation of the Activity Diagram will have to be updated to split the node corresponding to competition, into two children nodes 'C1' and 'C2' as it is described in Section 3.3.2.10 of Chapter 3. We then define the component cs that will play the role of the central synchronizer, which could be the terminating component of C1 or C2.

6.9 Summary

In this chapter, we first described the overall architecture of the transformation software. We then illustrated the five phases of the derivation process and we explained each phase in detail. The five phases are the following:

- The generation of the Java classes

- Syntactic and semantic validation
- Tree construction
- Calculation of roles
- The behavior transformation

Chapter 7

TESTS AND RESULTS

In this chapter we talk about the tests that were performed on the Derivation Tool and then present a case study that involves the exchange of several coordination messages. We then explain the main results obtained by the Derivation Tool. For a more detailed analysis of the derivation applied to the different sequencing constructs of Activity Diagrams, please refer to Chapter 4.

7.1 Tests Performed on the Derivation Tool

Tests have been conducted to make sure that the Derivation Tool produces accurate results. All the requirements in Chapter 4 have been satisfied, and the Activity Diagrams presented in that chapter are actual test cases used as input or obtained as output from our Derivation Tool.

Furthermore, the following tests have been successfully conducted on the Derivation Tool.

- First, The Derivation Tool was tested for different combinations of constructs:
  - Activity Diagrams with only strong sequences.
  - Activity Diagrams with only weak sequences.

- Activity Diagrams containing different combinations of weak sequences and strong sequences. This also allowed verifying that the priority adopted between strong and weak sequencing has been respected in all cases.
- Activity Diagrams with only a weak loop, or a strong loop.
- Activity Diagrams with only a choice construct or a parallel construct.
- Activity Diagrams containing different combinations of the above constructs, in different orders each time. This allowed testing that the Derivation Tool generates all necessary messages when combining different constructs, without mixing or omitting any of them.

- It has also been verified that the redundant messages from a component to itself are not generated, as per the rules of the transformation algorithm. For example, if there is strong sequencing between two collaborations and the component c plays a terminating role in the first collaboration and a starting role in the second collaboration, then we have checked that this component c will not send a message to itself.

- For each test conducted, we have verified that any coordination message sent by some component has a corresponding reception by a corresponding receiving component. These two messages, the sending and the reception, both have the same id and, of course, are of the same type of coordination message.

- Each test has been validated using the rules defined in the transformation algorithm. We have therefore made sure that all necessary flow messages were generated, by manually doing the transformation, and then verifying that the resulting messages correspond to the ones generated by the Derivation Tool.

- Some of the obvious tests that have been done consist of checking that all collaborations in which a given component participates are generated in the derived diagram of that component. On the other hand, we have also made sure that the collaborations in which a given component does not participate are removed, during the generation process, from the behavior of this component, and are therefore not shown in the Activity Diagram corresponding to this component.

- Furthermore, the assumptions made for the Derivation Tool have been cheked one by one. We have made sure that in the case that one of the assumptions is not satisfied, a clear error message indicating the reason of the error is displayed to the user.

7.2 Case Study

Figure 7.1 shows an Activity Diagram for order processing. The different roles involved in this diagram are Customer Service, Finance, Order Fulfillment and the Customer. Let us call these roles s, f, o and c, respectively.

The derivation produces the Activity Diagrams shown in Figures 7.2, 7.3, 7.4 and 7.5, which represent the behaviors for the components s, f, o and c, respectively. The first construct in the Activity Diagram of the order processing is the strong sequencing between "Receive Order" and the parallel construct that follows. This strong sequence does not require any coordination, since the terminating component of "Receive Order" is the same as the starting component of the parallel construct.

The first construct involving coordination messages is the strong sequencing between "Send Invoice" and "Receive Payment". This requires the sending of flow messages from s to c and f. These correspond to the activities "send fm(4) to f" and "send fm(4) to c" shown in Figure 7.2, as well as their corresponding receptions "receive fm(4) from s" performed by f as shown in Figure 7.3, and "receive fm(4) from s" performed by c as shown in Figure 7.5.

Another exchange of flow messages is necessary after the parallel construct. Since the terminating components of this parallel construct are c and f, and the starting components of the following collaboration "Close Order" are f and o, then c and f send flow messages to f and o because of the presence of strong sequencing. We note that f does not send any flow message to itself. So this results in the following sending and receiving of messages: "send fm(2) to f" and "send fm(2) to o" performed by c as shown in Figure 7.5; "send fm(2) to o" and "receive fm(2) from c" done by f as shown in Figure 7.3; and

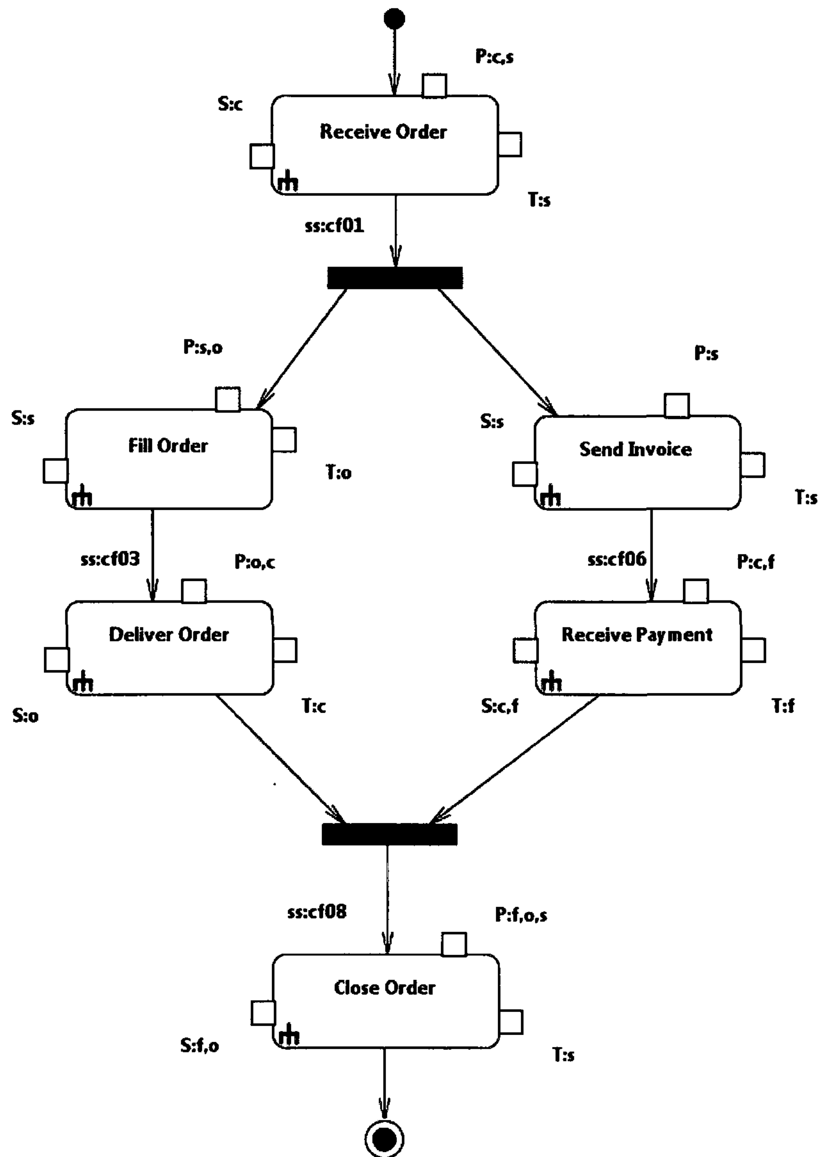

FIGURE 7.1: Case study: order processing (modified from [10])

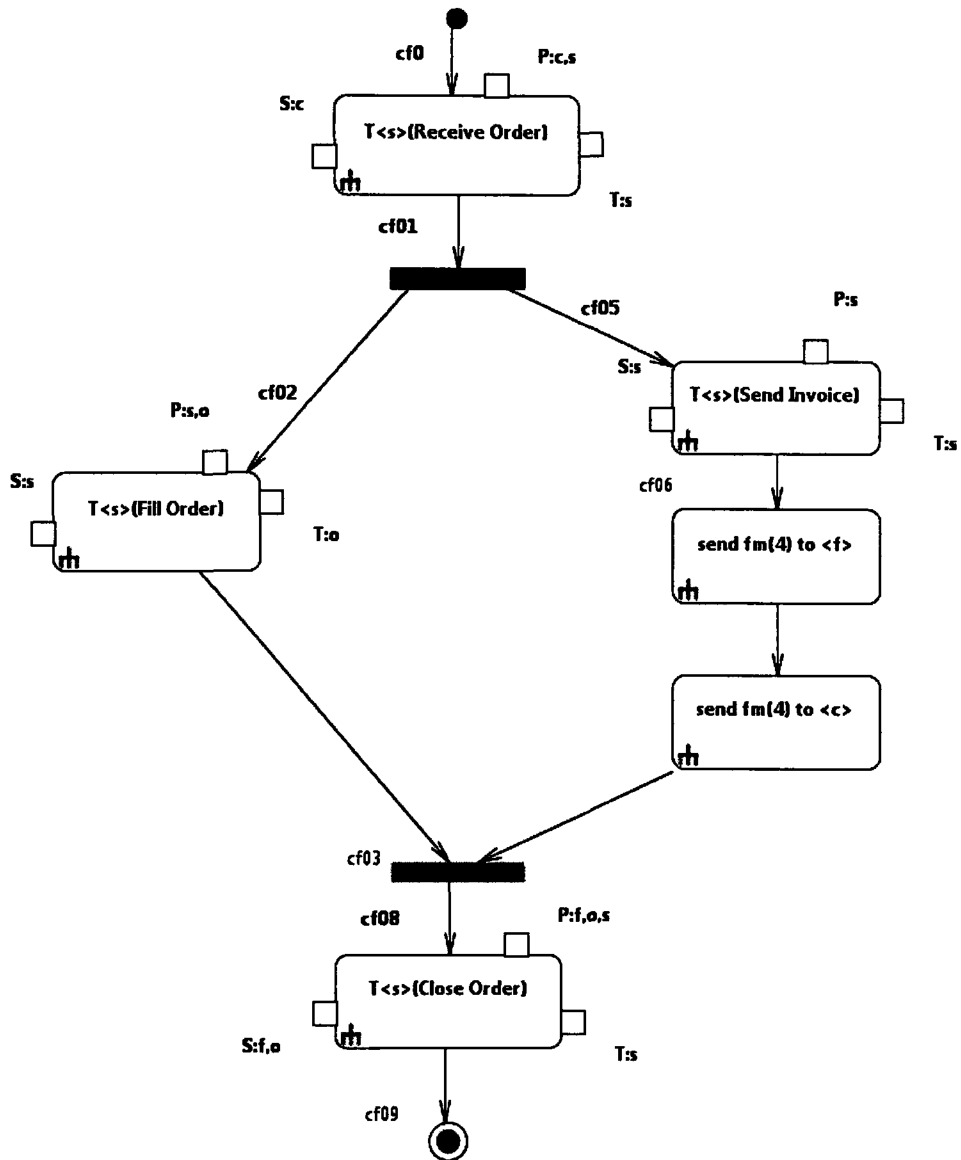

FIGURE 7.2: Derivation of the order processing AD for Customer service

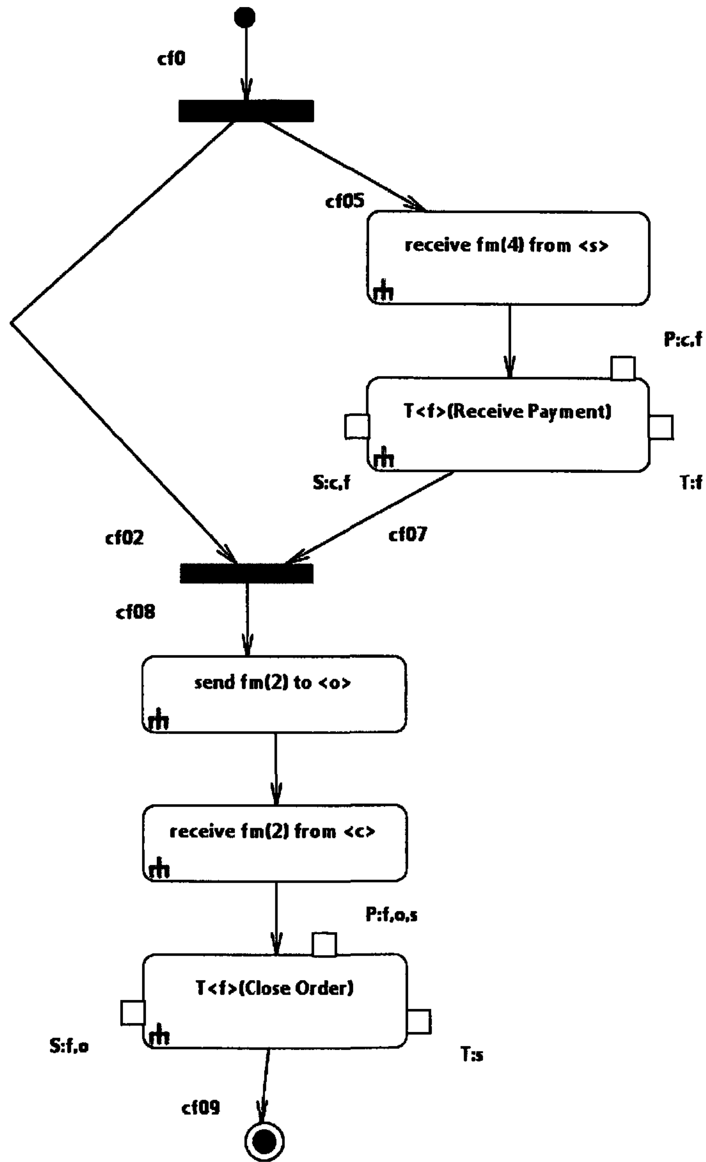

FIGURE 7.3: Derivation of the order processing AD for Finance

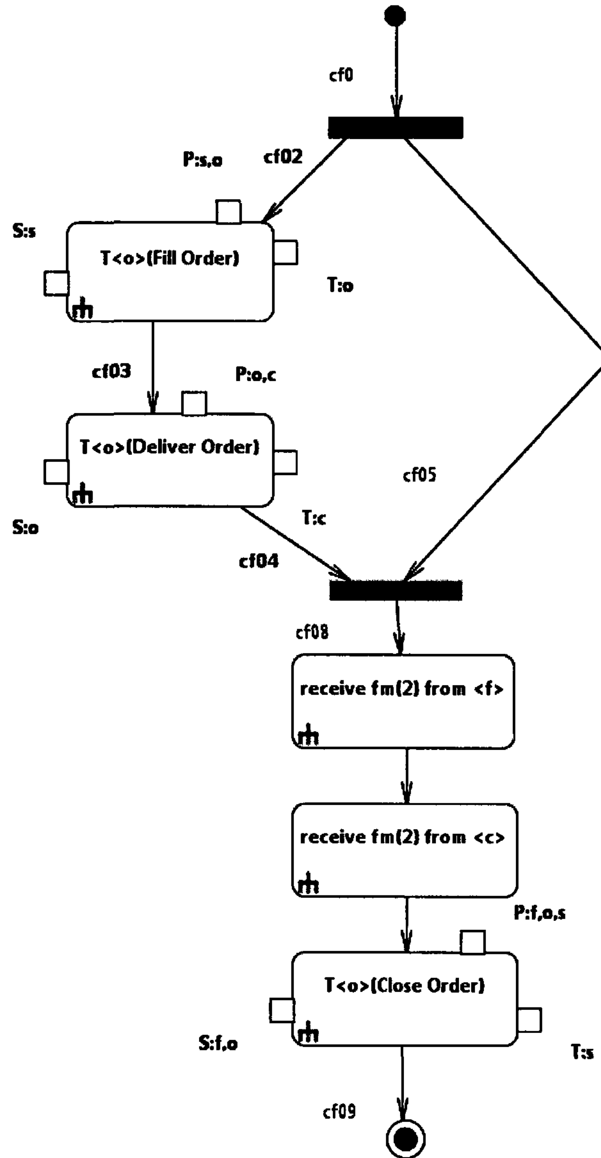

FIGURE 7.4: Derivation of the order processing AD for Order fulfilment

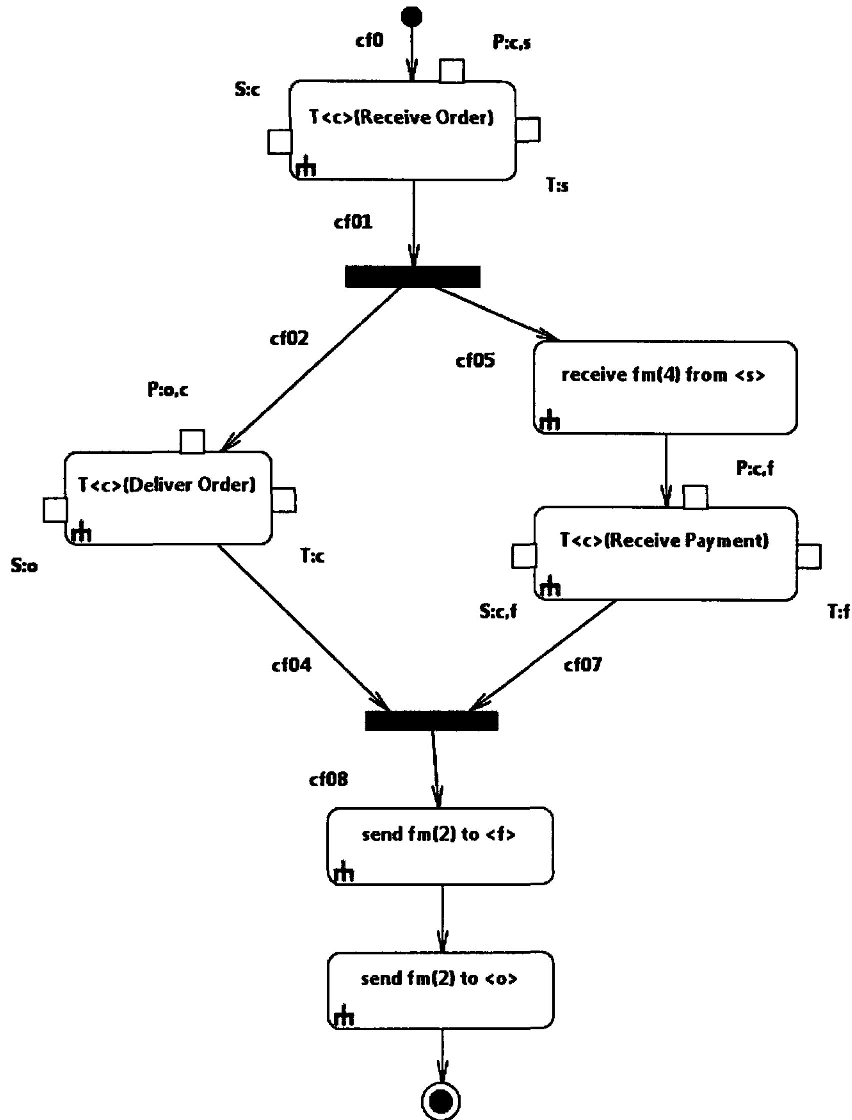

FIGURE 7.5: Derivation of the order processing AD for the Customer

finally “receive fm(2) from c” and “receive fm(2) from f” done by o as shown in Figure 7.4.

7.3 Summary

In this chapter we have presented a case study of order processing, where exchanges of flow messages are needed to ensure the coordination between the activities of components in the system. We have shown the derived Activity Diagrams generated by the Derivation Tool for the participating components and explained the main results obtained. We note that this involves only flow messages particular to this case study, and we refer the reader to Chapter 4 where we have covered and explained in detail coordination messages exchanged for all the constructs.

Chapter 8

CONCLUSION AND FUTURE WORK

8.1 Conclusions

Among the major merits of distributed systems is their ability to coordinate the actions of multiple processes on a network in an open and scalable way. This means that each component of the system is continually open to interact with the other components and the system can easily be altered in order to accommodate changes such as a change in the number of computing entities involved or in the shared resources available. Thus, given the combined capabilities of the distributed components, distributed systems are much more powerful than stand-alone applications. On the other hand, for a distributed system to be useful, it has to be reliable. Reliability is not an easy goal to achieve because of the complexity of the interactions between simultaneously running components. Synchronization between the different components of a distributed system is a key feature for its reliability. This synchronization can be performed through a message exchange in a way that avoids conflicts between the components of the system.

The main objective of this thesis is to build a derivation tool that is helpful to software analysts. The tool automates a derivation algorithm which, as described in [23], defines a method to derive the behavior of each component participating in a distributed system,

from the global specification of that system. The Derivation Tool takes as input an Activity Diagram representing the global system model and then derives the separate behaviors of each component that participates in the system. During this process, the Derivation Tool ensures the synchronization of the activities of the different components of the system, through the introduction of certain coordination messages exchanged between theses components. As an output of the Derivation Tool, the user gets the Activity Diagrams representing the behavior of each component of the system. The generated Activity Diagrams are drawn graphically using an Eclipse plugin.

The main use of the Derivation Tool is automating a part of the design process of distributed computing systems. This is done by the derivation of individual models for the components participating in the system, based on the global specification of that system. The global specification is defined by the user in a convenient way, since in can be drawn graphically using the GMF Editor within the Eclipse framework.

We have presented the different requirements for the Derivation Tool. For each construct used in the input behavior, the derivation rules required to be applied are described and then illustrated through examples of input Activity Diagrams for the global system, as well as output Activity Diagrams for the derived system components. The Derivation Tool has been used to generate the diagrams illustrating the behavior of the system components.

The conceptual choices that we have made during the design phase of the Derivation Tool have been described in detail, and the reasons behind these choices were explained. We have described the procedure that has been followed to represent an Activity Diagram as a tree data structure that allows efficient memory management and rapid data access. Also, we have presented the ambiguity occurring between strong sequencing and weak sequencing in the global Activity Diagrams used as input for our Derivation Tool, and we have shown that this ambiguity can be resolved by prioritizing one type of sequencing over the other.

We have described the overall architecture of the transformation software, which consists of five phases. Each phase has been described in detail. The five phases are the following:

- The generation of the Java instances corresponding to the input XMI elements
- The syntactic and semantic validation
- The tree construction
- The calculation of roles
- The behavior transformation

Finally, we have presented a case study of order processing, where exchanges of certain flow messages are needed to ensure the coordination between the activities of components in the system. We have generated the activity diagrams using the Derivation Tool for the participating components and we have explained the main results obtained.

8.2 Future Work

As future work perspectives for the Derivation Tool, we suggest the following:

8.2.1 Interruption/Competition Construct

Taking into account the interruption/competition construct during the transformation process. The GMF's UML2Tools plugin does not support yet the interruptible activity region, so neither the building of the global Activity Diagram, nor the display of the Activity Diagrams of the components that are involved in the interruption, are possible within the GMF's Editor. Therefore, we were unable to implement this specific construct. Once the GMF is upgraded to support the interruption for Activity Diagrams, the Derivation Tool could be extended to include the interruption/competition construct as was explained in Section 6.8 of Chapter 6.

8.2.2 Weak/Strong Sequencing Priority

Another option for future work on the Derivation Tool would be to allow the user to chose the priority between weak and strong sequencing that he would like to have in the

diagram. The current version of the transformation software prioritizes weak sequencing over strong sequencing in the whole Activity Diagram.

To go even further, the Derivation Tool may be upgraded in a manner that allows the user to adopt some kind of bracketing that will allow him to specify the scope of a weak or strong sequence that is defined in the Activity Diagram. For this purpose, an Activity Diagram object such as Regions can be used. Examples of such a bracketing system were shown in Figures 5.3 and 5.2 of Chapter 5. This way, the user does not necessarily have to use the same priority in all the cases where he uses strong or weak sequencing, rather he can adapt each situation to what is needed using the bracketing system.

8.2.3 Automatic Layout

The current version of the Derivation Tool requires manual graphical adjustments for the added UML elements in the generated XMI files. The collaborations and the UML elements that were originally in the global activity diagram keep their original coordinates in the generated activity diagrams representing the components' behaviors.

It would be good to have the transformation software laying out automatically the added UML elements in the output diagrams.

Bibliography

[1] K. Nadiminti, M. D. de Assuno, and R. Buyya, "Distributed systems and recent innovations: Challenges and benefits," *InfoNet Magazine*, vol. 16, September 2006.

[2] A. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[3] G. Couloris, J. Dollimore, and T. Kinberg, *Distributed Systems - Concepts and Design.* Addison-Wesley, 2005.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *ACM*, pp. 558–565, 1978.

[5] W. Emmerich, "Distributed system principles." Course notes, University College London, 1997.

[6] V. K. Garg, *Principles of Distributed Systems.* Springer, 1996.

[7] R. Alur, G. J. Holzmann, and D. Peled, *Tools and Algorithms for the Construction and Analysis of Systems*, ch. An analyzer for Message Sequence Charts, pp. 35–48. Springer Berlin / Heidelberg, 1996.

[8] H. N. Castejon, G. von Bochmann, and R. Bræk, "Realizability of collaboration-based service specifications," *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pp. 73–80, December 2007.

[9] G. v. Bochmann and R. Gotzhein, "Deriving protocol specifications from service specifications," *Proc. ACM SIGCOMM Symposium*, pp. 148–156, 1986.

[10] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison Wesley, September 2003.

[11] S. Maqbool, "Transformation of a core scenario model and activity diagrams into petri nets," Master's thesis, University of Ottawa, September 2005.

[12] C. Bock, "Uml 2 activity and action models part 2: Actions," *Journal of Object Technology*, vol. 2, pp. 41–56, September-October 2003.

[13] C. Bock, "Uml 2 activity and action models," *Journal of Object Technology*, vol. 2, pp. 43–53, July-August 2003.

[14] OMG, *Unified Modeling Language Superstructure, Version 2.2, formal/2009-02-02*, February 2009.

[15] C. Bock, "Uml 2 activity and action models part 3: Control nodes," *Journal of Object Technology*, vol. 2, pp. 7–23, November-December 2003.

[16] C. Bock, "Uml 2 activity and action models part 4: Object nodes," *Journal of Object Technology*, vol. 3, pp. 27–41, January 2004.

[17] C. Bock, "Uml 2 activity and action models part 6: Structured activities," *Journal of Object Technology*, vol. 4, pp. 43–66, May-June 2005.

[18] G. Engels, A. Forster, R. Heckel, and S. Thone, *Process Modeling using UML*, pp. 83–117. Wiley, 2005.

[19] D. Xu, W. Liu, Z. Liu, and N. Philbert, "Tool support to deriving test scenarios from UML Activity Diagrams," *2008 International Symposium on Information Science and Engieering*, vol. 02, pp. 73–76, December 2008.

[20] "World wide web consortium. see http://www.w3c.org,"

[21] "XML Schema Tutorial, http://www.w3schools.com/schema,"

[22] P. L. Hgaret, L. Wood, and J. Robie, "What is the Document Object Model?." `http://www.w3.org/TR/DOM-Level-2-Core/introduction.html`, November 2000.

[23] G. v. Bochmann, "Deriving component designs from global requirements," *Proc. Intern. Workshop on Model Based Architecting and Construction of Embedded Systems (ACES)*, September 2008.

[24] F. Khendek and X. J. Zhang, "From MSC to SDL: Overview and application to the autonomous shuttle transport system," *Scenarios: Models, Transformations and Tools*, pp. 228–254, 2003.

[25] D. Amyot and A. Aberlein, "An evaluation of scenario notations and construction approaches for telecommunication systems development," *Telecommunication Systems Journal*, vol. 24, pp. 61–94, September 2003.

[26] H. N. Castejon, G. von Bochmann, and R. Bræk, "Using collaborations in the development of distributed services," *submitted for publication*, 2008.

[27] N. Parlante, "Binary trees." `http://cslibrary.stanford.edu/110/BinaryTrees.html`, 2001.

[28] "http://xmlbeans.apache.org/docs/2.0.0/guide/congettingstartedwithxmlbeans.html," July 2008.

[29] T. E. Foundation. `http://www.eclipse.org/modeling/`.

[30] T. E. Foundation, "Graphical modeling framework." `http://wiki.eclipse.org/Graphical_Modeling_Framework`.

[31] I. Schieferdecker and A. Hartman, *Model Driven Architecture - Foundations and Applications.* Springer, 2008.

[32] T. E. Foundation, "Mdt-uml2tools." `http://wiki.eclipse.org/MDT-UML2Tools`.

[33] T. A. S. Foundation. `http://xmlbeans.apache.org`, December 2009.

Appendix A

The Derivation Tool User Manual

This appendix presents the Derivation Tool user manual. It describes the steps that the user should follow in order to derive the components behaviors from a given global Activity Diagram.

A.1 The Input Files

The Derivation Tool requires three input files. They are the following:

- Two files describing the global Activity Diagram. These two files are the XMI file and the graphical file. They are generated by the Eclipse framework when the user defines and saves the global activity diagram.

  1. The XMI file: This file contains the definition of the global activity diagram in XMI format. It holds all the Activity Diagram elements and their relationships.
  2. The graphical file: This file describes the coordinates of the Activity Diagram elements when displayed graphically. The user can change the location and the size of any element and the changes will be saved in this file.

- The component file: This file lists the different components of the global system. The user has to define this file before starting the derivation process. The Derivation Tool generates a behavior for each component defined in the component file, which is defined according to the format described in Section 6.8.

A.2 The Processing

This part of the derivation process is transparent to the user except when it fails. In this case, a message explaining the reason why the derivation process has failed, is displayed to the user. In order to avoid those errors, the user should respect the assumptions described in Section 5.2.4.

A.3 The Output Files

Once the input files are ready, the Derivation Tool is launched in order to derive the behavior of the global system components. For each component, the Derivation Tool generates two files holding its behavior, which are the XMI file and the graphical file as illustrated in Figure 6.2. They are distinguished from the input files by adding the component identifier at the end of the file name. The output files can be read through the Eclipse framework in the same way as the input files.