UNIVERSITY OF NEWCASTLE UPON TYNE

DEPARTMENT OF COMPUTING SCIENCE

Multiparty Interactions in Dependable Distributed Systems

by

Avelino Francisco Zorzo

Ph.D Thesis

Newcastle upon Tyne, UK

1999

To my girls

Mári and Gabriela

Abstract

With the expansion of computer networks, activities involving computer communication are becoming more and more distributed. Such distribution can include processing, control, data, network management, and security. Although distribution can improve the reliability of a system by replicating components, sometimes an increase in distribution can introduce some undesirable faults. To reduce the risks of introducing, and to improve the chances of removing and tolerating faults when distributing applications, it is important that distributed systems are implemented in an organized way.

As in sequential programming, complexity in distributed, in particular parallel, program development can be managed by providing appropriate programming language constructs. Language constructs can help both by supporting encapsulation so as to prevent unwanted interactions between program components and by providing higher-level abstractions that reduce programmer effort by allowing compilers to handle mundane, error-prone aspects of parallel program implementation.

A language construct that supports encapsulation of interactions between multiple parties (objects or processes) is referred in the literature as *multiparty interaction.* In a multiparty interaction, several parties somehow "come together" to produce an intermediate and temporary combined state, use this state to execute some activity, and then leave the interaction and continue their normal execution.

There has been a lot of work in the past years on multiparty interaction, but most of it has been concerned with synchronisation, or handshaking, between parties rather than the encapsulation of several activities executed in parallel by the interaction participants. The programmer is therefore left responsible for ensuring that the processes involved in a cooperative activity do not interfere with, or suffer interference from, other processes not involved in the activity.

Furthermore, none of this work has discussed the provision of features that would facilitate the design of multiparty interactions that are expected to cope with faults - whether in the environment that the computer system has to deal with, in the operation of the underlying computer hardware or software, or in the design of the processes that are involved in the interaction.

In this thesis the concept of multiparty interaction is integrated with the concept of exception handling in concurrent activities. The final result is a language in which the concept of multiparty interaction is extended by providing it with a mechanism to handle concurrent exceptions. This extended concept is called *dependable multiparty interaction.*

The features and requirements for multiparty interaction and exception handling provided in a set of languages surveyed in this thesis, are integrated to describe the new dependable multiparty interaction construct. Additionally, object-oriented architectures for dependable multiparty interactions are described, and a full implementation of one of the architectures is provided. This implementation is then applied to a set of case studies. The case studies show how dependable multiparty interactions can be used to design and implement a safety-critical system, a multiparty programming abstraction, and a parallel computation model.

Acknowledgments

I would like to express my sincere gratitude to several people who have contributed in various ways to the completion of this thesis. First and foremost, I thank my supervisor, Professor Brian Randell, who has *significantly* contributed to my research. His expertise in fault tolerance and many other subjects was fundamental to my learning, as well as the development of my research work and thesis. Since my arrival to England he was more than a supervisor, and made every effort to make me feel at home.

I am also very much indebted to Jie Xu, Sascha Romanovsky and Robert Stroud, for the discussions we had while using Coordinated Atomic (CA) actions to implement the control software for the FZI Production Cell I. Those discussions led me to work with the subject I present in this thesis. I would also to thank them for their help in the following projects: Jie for our experiments with fault tolerance and reflection - this work introduced me to fault tolerance; Sascha for our numerous discussions of how to implement CA actions, and for our discussions on the Gamma paradigm; Robert for his help in designing the framework for the dependable multiparty interaction, which has been used to implement several applications. I am also indebted to the people that worked in the DeVa project, that taught me how a research project should be carried out. Thanks to Giovanna DiMarzo and Nicolas Guelfi for our discussions on the Gamma paradigm, which helped us to produce several research papers.

I would like to thank Martin Beet for his help in reading this thesis and for being an excellent friend. I am also grateful to Marinho Barcellos who shared the office with me, and helped me several times. Thanks also to Richard Achmatowicz, with whom I had several talks about TLA.

Thanks to Professor Santosh Shrivastava that has helped me in being very practical with regard to my thesis. His observations while discussing the subject of my thesis were very important for its completion.

I shall never forget the support and love shown by my wife, Mári, who has had to bring up our little daughter Gabriela almost by herself. I know that, if it was not for her help and support, this thesis would not be completed. She was also the one that has pushed me hard in the direction of research and teaching, and I am very grateful for that.

Throughout my time in Newcastle, I have met many friends and colleagues who have made my time here in Newcastle very joyful. Although omitting important names, I shall mention Ulrike Beet, Cláudia Barcellos, Sérgio/Patrícia Cavalcante, Eduardo Figueiredo, Carlos/Luciene Longo, Marcelo/Célia Rabello, Craig/Sandra Thomson, Barry Hodgson, Nick Cook, Gerry Tomlinson, Dave Hartland, Jason Steggles, Graham Morgan, Rob Allen, Savas Parastatidis, and Ian Welch.

Many thanks to my employer in Brazil, PUCRS — Pontifícia Universidade Católica do Rio Grande do Sul, which has invested in me and allowed me to be on leave from the Computing Science Department for the last four years.

Last but not least, my sincere thanks to the Brazilian Research Agency CNPq, which provided me with financial support (grant number 200.531/95.6) in the period 1995-1999.

Contents

List of Figures

Chapter 1

Introduction

Parallel programs are usually composed of diverse concurrent activities, and communication and synchronisation patterns between these activities are complex and not easily predictable. Thus, parallel programming is widely regarded as difficult: [Foster 1996], for example, says that parallel programming is "more difficult than sequential programming and perhaps more difficult than it needs to be". In addition to the normal programming concerns, the programmer has to deal with the added complexity brought about by multiple threads of controls: managing their creation and destruction and controlling their interactions via synchronisation and communication.

Furthermore, with the proliferation of distributed systems, computer communication activities are becoming more and more distributed. Such distribution can include processing, control, data, network management, and security [Neumann 1996]. Although distribution can improve the reliability of a system by replicating components, sometimes an increase in distribution can introduce some undesirable faults. To reduce the risks of introducing faults when distributing applications, and of coping with residual faults, it is important that this distribution is implemented in an organised way.

As in sequential programming, complexity in distributed, in particular parallel, program development can be managed by providing appropriate programming language constructs. Language constructs can help both by supporting encapsulation so as to prevent unwanted interactions between program components and by providing higher-level abstractions that reduce programmer effort by allowing compilers to handle mundane, error-prone aspects of parallel program implementation [Foster 1996].

A mechanism that encloses multiple processes executing a set of activities together is called a *multiparty interaction* [Joung & Smolka 1996] [Forman & Nissen 1996] [Attie *et al.* 1993] [Evangelist *et al.* 1989]. In a multiparty interaction, several executing processes somehow "come together" to produce an intermediate and temporary combined state, use this state to execute some joint activity, and then leave the interaction and continue their normal execution.

There has been a lot of work in the past years on multiparty interaction, but most of it has been concerned with synchronisation, or handshaking, between parties rather than the enclosure of several programmed activities executed in parallel by the interaction participants. For example, specification languages like CSP [Hoare 1985], LOTOS [Brinksa 1988], or programming languages like Ada95 [ISO 1995], only deal with synchronisation between processes. However, the programmer designing a set of processes that are taking part in a cooperating activity is left with full responsibility for ensuring that it is just these processes that are involved in the activity, and that they do not interfere with, or suffer interference from, other processes that are not supposed to be involved.

Interference from processes that are not involved in an interaction can lead to unexpected results, hence failure to deliver the requested service. Furthermore, to keep track of all processes that have interfered with the participants of an interaction is a very complex task. Usually, circumstances which may prevent an operation from providing its service are called exceptions. These exceptions have to be handled with care, since the state of a system can be inconsistent when their occurrence is detected [Cristian 1989]. The treatment of exceptions has been studied for many years and mechanisms for handling exceptions in sequential processes have been included in several programming languages [Gosling *et al.* 1996] [ISO 1995] [Meyer 1992] [Liskov & Snyder 1979] [Goldberg & Robson 1983] [Ullman 1994]. Although there has been a lot of research in treating exceptions in sequential processes, language mechanisms for handling concurrent exception in programming languages are as yet in their early stages.

This thesis will investigate the possibility of integrating concurrent exception handling to the multiparty interaction concept. The thesis will also show how this extended multiparty interaction concept, i.e. multiparty interaction with concurrent exception handling, can be embedded in programming languages and implemented in object-oriented languages. The extended multiparty interaction concept will be called *dependable multiparty interaction* and will be able to cope with several concurrent exceptions being raised during the multiparty interaction.

This thesis is organized as follows.

Chapter 2. In this chapter the description, properties, basic mechanisms and related work on the concept of multiparty interaction is presented. A case study is introduced to show how multiparty interactions can help in guaranteeing safety properties of a system.

Chapter 3. In this chapter exception handling is described. A description of terms related to exception handling used by the research community is presented. This chapter focuses on the way modern languages (using Java and Ada as examples) deal with exceptions during the execution of a program.

Chapter 4. The concept of *dependable multiparty interaction* is presented in this chapter. A language that includes dependable multiparty interaction is introduced. This new language is based on two existing languages that have a multiparty interaction mechanism as a basic construct. A formal semantics for the new multiparty interaction mechanism is presented using the Temporal Logic of Actions.

Chapter 5. This chapter first discusses the way a set of components that compose a dependable multiparty interaction can be distributed. Second, it presents a complete framework for implementing dependable multiparty interactions in an object-oriented language.

Chapter 6. Full designs and implementations of several case studies are presented in this chapter. All these case studies are implemented using the dependable multiparty interaction concept introduced in Chapter

4. The first case study is related to an industrial control system in which safety and fault tolerance properties must be guaranteed. This case study is a more complex version of the case study presented in Chapter 2. The second case study describes how the dependable multiparty interaction concept can be used to implement another multiparty interaction abstraction. The third case study shows how dependable multiparty interactions can help to introduce fault tolerance in a parallel computation model.

Appendices. A set of six appendices is provided at the end of this thesis. They provide, respectively: a survey of multiparty interaction mechanisms; a survey of exception handling in programming languages; the syntax of the language introduced in Chapter 4; the TLA$^+$ operators used in the formal specification of the language presented in Chapter 4; and the complete Java code for the dining philosophers problem using DMIs.

Chapter 2

Multiparty Interactions

This chapter surveys the state-of-the-art in multiparty interaction mechanisms. It focuses on the design choices to be made when creating a new mechanism for dealing with interactions between several processes. Based on these design choices several languages that use a multiparty interaction as a basic mechanism have been developed [Hoare 1985] [Francez *et al.* 1986] [Charlesworth 1987] [Brinksa 1988] [Back & Kurki-Suonio 1988] [ISO 1995] [Jårvinen & Kurki-Suonio 1991] [Forman & Nissen 1996] [Petitpierre 1998]. A taxonomy for these languages is presented in Section 2.2. In Section 2.3 a complete example of how multiparty interactions can help in designing a safety-critical system is presented.

2.1 Design Choices

In designing a language for multiparty interaction, one must make a trade-off between the implementation efficiency of the language versus its expressive power. There are several choices that can be made when designing a multiparty interaction construct. For example, [Evangelist *et al.* 1989] describes a set of properties for a multiparty interaction that serves as an interprocess communication primitive. These properties are:

- Pre-synchronisation: in synchronous multiparty interaction constructs the participants of an interaction must synchronise before the interaction commences, i.e., if one participant arrives, it has to wait until all participants of the interaction have arrived. The main effect of this property is to provide a consistent combined state before the interaction starts.
- Split bodies: each participant in the interaction has its own set of commands that is executed in parallel as part of the interaction. Multiparty interactions that do not have split bodies, usually have just one block of code that is executed by only one participant of the interaction - a special case that will not be considered further here.
- Frozen initial state: the participants of the interaction view the combined state as frozen in the beginning of the interaction, until the end of the interaction, when all changes that were made take effect. From the point of view of participants that are not involved in the interaction, this property can be seen as an atomic change of state in the system. Such a guarantee will avoid wrong information being accessed by processes outside of the interaction.

Other features are included in certain language constructs, and are related to the way the multiparty interaction is activated or terminated. (See

[Forman & Nissen 1996] [Back & Kurki-Suonio 1988].) These features are:

- Preconditions: some interaction mechanisms provide a guard to check the preconditions to execute the interaction, hence the need for having synchronisation upon entry. If the precondition is true, then the interaction can commence, otherwise the interaction is not executed.

- Post-conditions: an assertion after the interaction has finished can be used to check that a set of post-conditions has been satisfied by the execution of the interaction.

Additionally, [Joung & Smolka 1996] presents a list of choices that can be made in the design of language constructs for multiparty interaction. These choices include the following:

- Biparty vs. multiparty interactions: as the name indicates, biparty interactions involve only two participants, and multiparty interactions are not so limited, and instead typically involve several participants.

- Fixed vs. variable interactions: in the former, the set of participants of an interaction is fixed, i.e. they do not change every time the interaction is executed. In the latter the participants are variable, i.e. participants can be different each time the interaction is executed. (Fixed interactions are referred to as "zeroth-order" and variable ones as "first-order" interactions in [Joung & Smolka 1994].)

- Conjunctive vs. disjunctive parallelism: conjunctive parallelism allows a set of interactions to be executed simultaneously as an atomic unit, while disjunctive parallelism chooses, non-deterministically, one interaction to be executed from a set of possible interactions.

- Synchronous vs asynchronous execution in the underlying system: in synchronous systems every interaction has to execute one step of its computation at a time, while in asynchronous systems there is no such restriction.

2.2 Basic Mechanism Constructs

Based on the above set of choices, [Joung & Smolka 1996] presents a detailed taxonomy of languages that have a multiparty interaction mechanism as a basic construct. Of particular interest here is that they point out that the expressive power of a language that has a multiparty interaction as a basic mechanism is dependent on the way participants can *enrole*[1] in an interaction. They identify four basic interaction constructs based on their support of multipartiness and variable interactions: *channels, ports, gates,* and *teams.*

A **Channel** is a primitive for biparty communication. It is used as a communication link between two processes. The communication actually occurs only when both processes are ready to communicate. One example of channel usage is the input/output command in CSP [Hoare 1985]. The input command $P_i?y$ of processes P_j, which inputs a value from process P_i into variable y, is complementary to the output command $P_j!x$ of process P_i, which outputs the value of expression x to P_j. The

[1]In [Forman & Nissen 1996] *"enrole"* is used to denote the assumption of roles by processes.

joint execution of these commands is equivalent to the assignment of x to y ($y \leftarrow x$).

A **Port** is also a primitive for interaction between two processes. It is a mechanism for achieving variable interactions that define an activity involving two "roles". In a port-based interaction, a process does not know in advance with which process it is interacting. For example, in the readers-writers problem, a port can be defined with two roles, one for the readers and the other for the writers. Any reader process that needs a new value from a writer can enrole into the reader's role, while any writer ready to output the role can enrole into the writer's role. One language that uses this kind of primitive is Ada, whose *rendezvous* represents a port-based interaction involving two roles, one assumed by a fixed callee and the other by callers.

A **Gate** is a multi-channel primitive that defines an interaction among a fixed number of processes. LOTOS [Bolognesi & Brinksma 1987] is an example of language that uses a gate as the way processes enrole in an interaction.

A **Team** has the same properties as a port, with a fixed number of roles, although the number of roles is not limited to two. A set of processes can jointly establish an instance of the team by filling all the roles. Examples of team primitives can be found in Multiway Rendezvous [Charlesworth 1987] or in DisCo [Jårvinen & Kurki-Suonio 1991].

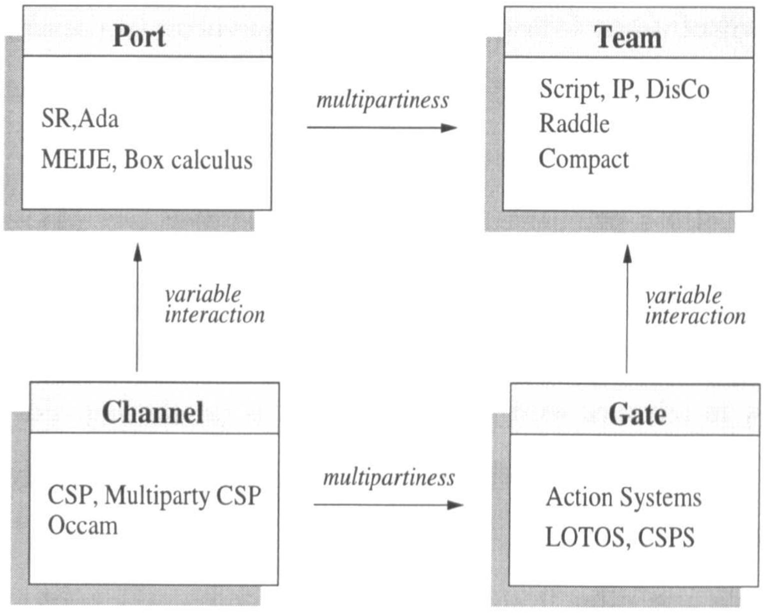

Figure 2.1: A Taxonomy of Languages for Multiparty Interactions

Figure 2.1 shows a possible taxonomy of a number of languages for multiparty interactions[2]. The figure represents the expressive power of the languages, where the team-based languages have the most expressive power and the channel-based languages have the least expressive power. This can be concluded from the following observations. Port-based languages can describe channel-based systems by assigning a port p_{ij} to each pair of processes P_i and P_j such that only P_i and P_j have access to p_{ij} and such that they always execute the same role in that port. Similarly, since ports are the biparty equivalent of teams, team-based languages can also describe channel-based systems. Similar reasoning regarding channels and ports shows that

[2]For further reading about the languages or algebraic models presented in Figure 2.1 refer to: CSP [Hoare 1985], Multiparty CSP [Joung & Smolka 1990], Occam [Hoare 1984], SR [Andrews *et al.* 1988], Ada, MEIJE [Simone 1985], Box calculus [Best *et al.* 1992], Action Systems [Back & Kurki-Suonio 1988], LOTOS [Bolognesi & Brinksma 1987], CSPS [Roman & Day 1984], Script [Francez *et al.* 1986], IP [Forman & Nissen 1996], Compact [Charlesworth 1987], Raddle [Forman 1986], and DisCo [Järvinen & Kurki-Suonio 1991].

team-based languages can describe gate-based systems. Given their greater expressive power, we have therefore chosen team-based interaction languages as the basis of the work we describe in this thesis.

2.3 An Example in DisCo

DisCo (Distributed Cooperation) is a specification language for reactive systems developed at Tampere University of Technology, Finland. DisCo is based on the Action Systems approach [Back & Kurki-Suonio 1988], in which a designer has to concentrate on the interactions between components rather than on the components themselves. An action system consists of a set of state variables and a set of actions. Each action is composed of a guard and a body. A guard is a boolean expression involving state variables, and the body is a set of commands to change the state of the variables. DisCo extends Action Systems into the object-oriented paradigm.

DisCo is called a specification language because it has features that do not allow direct implementation, or cannot be automatically implemented in a distributed fashion. However, DisCo specifications are executable in the sense that their simulation is possible with some interactive guidance from the user [Jårvinen & Kurki-Suonio 1990].

A program specification in DisCo is composed of a set of two basic components: *objects* and *actions*. Objects are instances of classes and are the means of representing the global state of a system. Objects are called participants in a DisCo action. Actions are the only units of execution in DisCo. They enclose a sequence of state transformations, and are the only means by which the state of an object can change. Actions are executed nondeterministically, and the execution of an action is atomic, meaning that once the execution of an action has started, it cannot be interrupted or interfered by other actions.

In this section we show how to apply DisCo to specify the basic elements of a production cell case study that was developed at the Forschungszentrum Informatik (FZI), Karlsruhe, Germany. This production cell case study involves the control of a set of machines that are used in combination in order to achieve a particular mechanical production process. The design of the control system for this production cell concentrates simply on the problems of how to ensure that the machines cooperate properly, i.e., do not interfere with each each other.

2.3.1 Case Study - FZI Production Cell I

The Production Cell model used in this section was developed in the Forschungszentrum Informatik (FZI), Karlsruhe, Germany, as a case study to present a realistic industry-oriented problem in which safety requirements play a significant role. It is not just a theoretical model; it is in fact based on an actual industrial installation in a metal-processing plant in Karlsruhe [Lewerentz & Lindner 1995].

The FZI Production Cell I model is composed of 6 devices (see Figure 2.2), 13 actuators, and 14 sensors. It processes metal plates in a press. Metal plates are conveyed to an elevating rotary table by a feed belt. A two-armed robot uses its first arm to take each plate as it arrives from the elevating

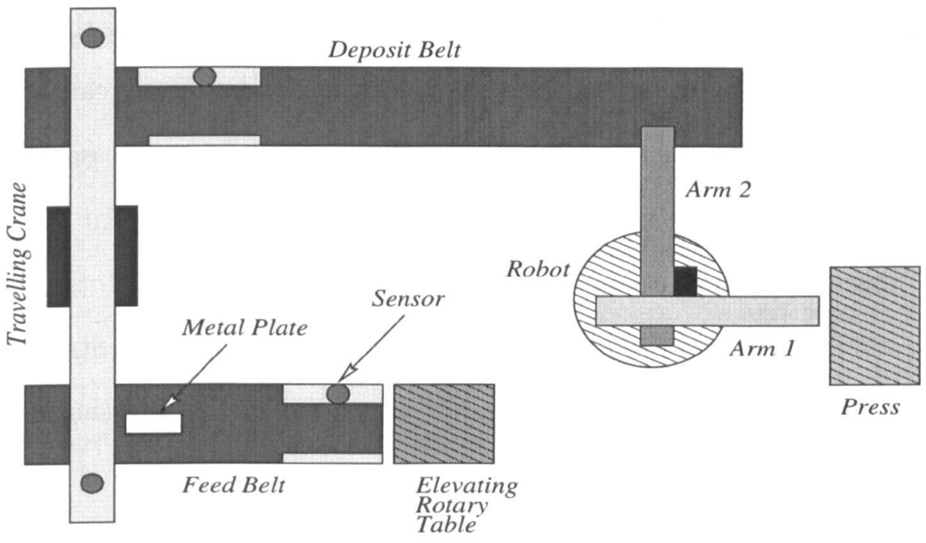

Figure 2.2: FZI Production Cell I

rotary table and place the plate in the press. The robot arm then withdraws from the press, and the press forges the metal plate. After the plate has been forged, the robot uses its second arm to take the forged metal plate out of the press and put it on a deposit belt. Finally, a traveling crane picks the metal plate up from the end of the deposit belt and takes it back to the feed belt again, thus allowing the model to operate in a continuous cycle without the need for an external operator.

Devices

- FEED BELT: its task consists of transporting metal plates to the elevating rotary table. The belt is powered by an electrical motor, which can be started up or stopped by a control program. A photo-electric sensor is installed at the end of the belt; it indicates whether a plate has entered or left the final part of the belt.

- ELEVATING ROTARY TABLE: its task consists of rotating the plates by about 45 degrees and lifting them to a level where they can be picked up by the robot's first arm. The vertical movement is necessary because the robot's arm and the feed belt are located at different levels, and because the robot cannot perform vertical translations. Rotation of the table is also required, because the arm's gripper cannot rotate and the robot is therefore unable to place the metal plates into the press in a straight position by itself.

- ROBOT: its task consists of picking up metal plates from the elevating rotary table; loading and unloading the press with metal plates; and placing the forged metal plates on the deposit belt. The robot possesses two independent orthogonal arms. For technical reasons, the arms are set at two different levels. Each arm can retract or extend horizontally. However, both arms rotate together. Mobility on the horizontal plane is necessary, since the elevating rotary table, press, and deposit belt are all placed at different distances from the robot's turning centre. The end of each of the robot's arms is fitted with an electromagnet that allows the arm to pick up metal plates.

- PRESS: its task consists of forging metal plates. The press consists of two platforms, with the lower platform being movable along a vertical axis. The press operates by pressing the lower platform against the upper one. Because the robot's arms are in different horizontal planes, the press has three positions: *i*) in the middle position it is loaded by the robot's first arm; *ii*) in the upper position the metal plate is forged;

and *iii*) in the lower position the press is unloaded by the robot's second arm.

- DEPOSIT BELT: its task consists of transporting the forged plates to the traveling crane. A photo-electric sensor is installed at the end of the belt; it reports when a metal plate reaches the end section of the belt. The control program then has to stop the belt. The belt can restart as soon as the traveling crane has picked up the metal plate.

- TRAVELING CRANE: its task consists of picking up metal plates from the deposit belt, moving them to the feed belt and unloading them there. It acts as a link between the two belts and thus makes it possible for the model to function continuously without the need for an external operator. The crane has an electromagnet as gripper which can perform horizontal and vertical translations. Horizontal mobility serves to cover the horizontal distance between the belts, while vertical mobility is necessary because the belts are placed at different levels.

Requirements

The controlling software must be implemented according to the rules laid down in the case study. In particular, it is important to guarantee that the following requirements are met:

- *safety*: in order to ensure that the system operates safely at all times, it is necessary to

  - restrict machine mobility;

  - avoid machine collisions;
  - avoid dropping plates outside safe areas, i.e. deposit belt, feed belt, and press;
  - keep plates sufficiently distant from each other;

- *liveness*: every metal plate introduced into the system via the feed belt must eventually be dropped by the crane on the feed belt again after having been forged;

Requirements such as *flexibility* or *efficiency* can be taken into account but must not violate the above requirements.

2.3.2 DisCo Objects in FZI Production Cell I

The state of a system in DisCo consists of a set of objects, which are instances of classes. Objects are composed of attributes that can represent simple values, such as integers or boolean values, or sets of simple values. Objects' attributes can also include finite state machines and references to other objects. Each object can contain as many groups of attributes as needed. An object can also be composed of other objects, e.g. the Robot object has two Arm objects (see Figure 2.3). An asterisk (*) in front of one element of a finite state machine denotes the initial value from that state group. DisCo objects do not have member functions (methods) as programming languages like C++, Eiffel, or Java. Instead, all changes on object's state are performed inside DisCo actions (see next section).

Figure 2.3 shows some of the DisCo class definitions for the FZI Production Cell I case study. In the examples, when an object of the Table class is

```
class Arm is
   extension: real := 0.0;
   state *magnet_off, magnet_on;
end;

class Robot is
   arm1, arm2: Arm;
   angle : integer := 0;
end;

class Table is
   angle : integer := 0;
   state *lower, upper;
   state *free, loaded;
end;
```

Figure 2.3: DisCo Objects for FZI Production Cell I

created, its initial state is: angle=0, lower, and free, meaning that the table is pointing towards the feed belt, it is in its lower position, and it is free (not loaded). In summary, the table is ready to be loaded. The robot, on the other hand, has two arms, each fully retracted and with their magnets turned off, and an angle that is not indicating any device in particular.

2.3.3 DisCo Actions in FZI Production Cell I

Actions are the execution entities in DisCo. (Such actions are, as shown in Figure 2.1, teams; more specifically they are multiparty interactions with a fixed number of variable participants.) Each action has a guard, which is a predicate, and a body. When the guard is true for a collection of potential participants, the action is said to be enabled. Objects are the participants in a DisCo action and assume a role in the action when the action is enabled. The body of an action consists of one sequential set of assignments and conditional statements which can refer only to the participants and parameters of the action.

In DisCo, there is no concept of process threads; actions are executed when their guard (**when** condition) is true and the objects involved in the action are not being used in another action. If two actions that use a same object can be activated at the same time, then only one of them will be executed (the choice is non-deterministic).

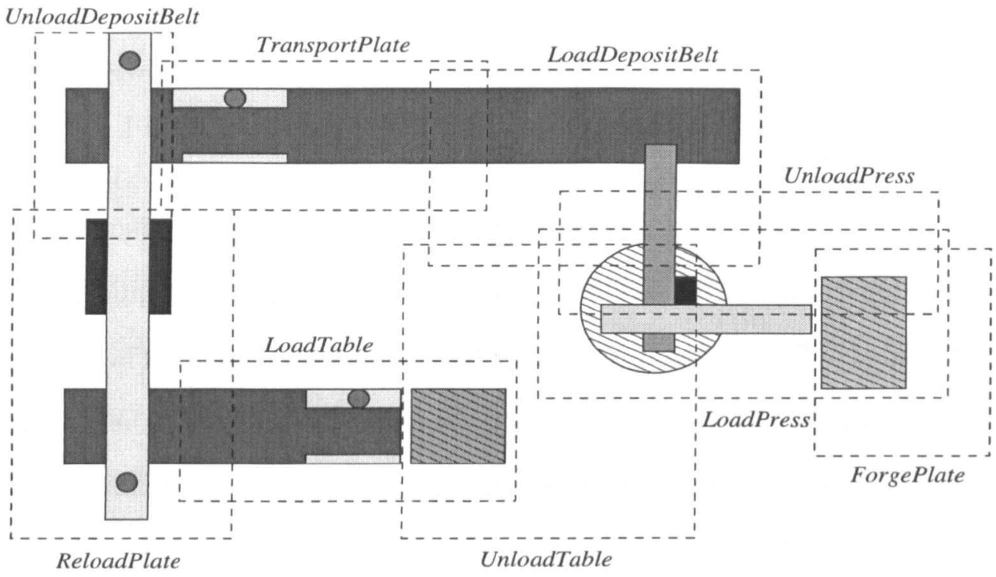

Figure 2.4: DisCo Actions in FZI Production Cell I

Before we give an example of actions in DisCo, let us consider how DisCo actions could be used for controlling the interactions between devices in the FZI Production Cell I case study. Figure 2.4 portrays the way in which we have chosen to use DisCo actions so as to structure the controlling of a sequence of operations between devices. Each DisCo action encloses a set of devices that must interact in a coordinated fashion to satisfy the requirements of the case study. If two such DisCo actions are shown as overlapping, this indicates that they must not be performed in parallel because they both involve the same device. The semantics of DisCo will guarantee this. For example, the LoadTable action cannot be executed in parallel with the UnloadTable action because both DisCo actions involve the table object, and the table can participate in only one of them. Our controlling software for FZI Production Cell I is thus composed of ten DisCo actions, as shown in Figure 2.4.

```
action LoadTable by fb:FeedBelt; t:Table is
when fb.loaded ∧ t.free do
   → t.lower;
   t.angle := 0;
   → t.loaded;
   → fb.free;
assert t.loaded ∧ fb.free
end;

action UnloadTable by t:Table; r:Robot is
when t.loaded ∧ r.arm1.magnet_off do
   r.angle := 50;
   t.angle := 50;
   → t.upper;
   r.arm1.extension := 0.5028;
   → r.arm1.magnet_on;
   → t.free;
   r.arm1.extension := 0.0;
assert t.free ∧ r.arm1.magnet_on
end;
```

Figure 2.5: LoadTable and UnloadTable Actions in DisCo

Figure 2.5 shows how the LoadTable and UnloadTable would be described in DisCo. It demonstrates how the table in FZI Production Cell I cyclicly activates the loading and unloading of the table with metal plates. The loading of the table is an example of a multiparty interaction that is executed by two parties: the feed belt and the table. The change of state value in an object is represented by $\rightarrow$ new-state-value, e.g. $\rightarrow$ t.loaded means that the table changes its state to the loaded state.

In DisCo, an action's precondition is specified via a guarded command, which is executed before the action commences (**when** clause). DisCo also allows a designer to specify assertions (**assert** clause) in the code of an action. Assertions are checked in the place they occur. If an assertion is not true, then the system is stopped. Action post-conditions can thus be specified by placing an assertion at the end of the action code (see Figure 2.5).

2.4 Discussion

Although there has been a lot of research on multiparty interaction mechanisms, to the best of our knowledge none has considered the provision of features that would facilitate the design of multiparty interactions that are expected to cope with faults — whether in the environment that the computer system has to deal with, in the operation of the underlying computer hardware or software, or in the design of the processes that are involved in the interaction.

In Appendix A, we present (the description of) several languages that have a multiparty interaction as a basic construct. An example is given to show how the multiparty interaction construct could be used in those languages. Even though the example is very simple, we can draw several conclusions from the use of the multiparty interaction mechanism in these languages. For example, while all the languages provide means of synchronisation upon entry, only a few provide some kind of precondition in the multiparty interaction, e.g. Action Systems and DisCo. In addition, languages that provide split bodies in the multiparty interaction, e.g. IP, provide a better structuring mechanism than those that do not, e.g. Action Systems, Ada, Multiway Rendezvous, and DisCo. In some languages, the multiparty interaction mechanism does not provide split bodies, and the mechanism is used only for synchronisation and communication, e.g. CSP, LOTOS and SR. Split bodies in those languages can be simulated by using several of their multiparty interaction mechanisms (see Appendix A).

The example that was presented in Section 2.3 and described using DisCo, could have been implemented using any of the languages listed in Section 2.2. DisCo was used because its **action** construct is a team, and as was mentioned in Section 2.2, teams provide the greatest expressive power among the basic multiparty interaction constructs. Furthermore, multiparty interactions in DisCo provide a way of testing pre and post-conditions. DisCo's major weakness is the fact that its action construct does not allow split bodies which could be used to describe the parallel activities that could be executed in the system. Because performance was not a major requirement in the case study, we decided to ignore this weakness of DisCo in the description of the control system for the case study.

It should be noted that the team concept has largely been investigated as a specification language construct, whereas the interest of this thesis is in utilising a team not just within specifications but also for structuring and designing actual programs. Moreover, to date programming languages usually provide only two-party synchronisation mechanisms, such as the rendezvous in Ada.

Finally, it was only possible to describe the case study as presented in this chapter because it does not include any kind of fault model. Note that the case study does not mention anything about a possible break in one of the devices or sensors. In an actual production cell these are situations that will happen, possibly quite frequently.

In the next chapters, exception handling is discussed in detail, and a new multiparty interaction mechanism that brings together several of the issues presented in this section is described. This new mechanism will deal with weaknesses such as the lack of a language that provides split bodies, preconditions, multipartiness, post-conditions, synchronisation and the most important, exception handling in the same mechanism.

Chapter 3

Exception Handling

It is very common for software developers to write programs under the optimistic assumption that nothing will go wrong when the program is executed. Unfortunately, there are many factors that can make this assumption invalid. For example, an arithmetic expression that may cause a division by zero; an array that is indexed with a value that exceeds the declared bounds; the square root of a negative number; a request for memory allocation during run-time that may exceed the amount of memory available; opening a file that does not exist; and many more.

When any of such event happens the system will often fail in an unexpected way. This is not acceptable in current programming standards. To improve reliability, it is important that such circumstances are detected and treated appropriately. Conventional control structures, such as the if-then-else command, are inadequate. For example, to check that an index of an array is always valid, a programmer could explicitly test the value of the index each time before using it, which is cumbersome and could often be forgotten or intentionally omitted. A better way would be to rely on the underlying system to trap the situation where array indexes are outside the array bounds. To cope with this kind of situation, several programming languages provide features for handling such circumstances, i.e. exception handling.

In this chapter, a review of definitions and the way exception handling is performed by several languages and systems is presented.

3.1 Definitions

Exceptions are usually defined in several different ways. Usually, the definition of exception is closely related to the definitions of *error*, *fault* and *failure*. Several researchers use these terms in a mixed way. In [Laprie 1995], an attempt to give precise definitions for the various attributes of computing systems dependability is made. The definition of error, fault and failure is given as follows.

> *A system **failure** occurs when the delivered service deviates from fulfilling the system function, the latter being what the system is intended for. An **error** is that part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a **fault*** [Laprie 1995].

The definition of exception varies considerably in the literature. Some of the definitions of exceptions found in the literature are given below.

1. *of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker* [Goodenough 1975];

2. *the term exception is chosen because, unlike the term error, it does not imply that anything is wrong* [Liskov & Snyder 1979];

3. *an exception denotes the occurrence of an error* [Young 1982];

4. *an exception does not represents an error condition, but some event whose possibility requires versatility of action* [Horowitz 1983];

5. *exceptions are rare situations detected by a run-time system or by a user program* [Szalas & Szcepanska 1985];

6. *an exception occurrence is a computational state that requires an extraordinary computation. Exceptions are associated with classes of exception occurrences. An exception is raised if the corresponding exception occurrence has been reached* [Knudsen 1987].

7. *an exception is a situation where a computation does not proceed as planned, perhaps because of an error* [Cox & Gehani 1989];

8. *exceptions are circumstances which might prevent a program from providing its specified service* [Cristian 1989];

9. *an exception is a situation leading to the impossibility of finishing a computation* [Dony 1990];

10. *the abnormal responses from a component are commonly referred to as exceptional responses or exceptions, particularly in software systems* [Lee & Anderson 1990];

As shown above, the term exception is used in many different ways, which makes it difficult to define exceptions accurately. We consider that some of the above definition do not express what an exception is or are very simplistic, e.g. the definition of exception in [Liskov & Snyder 1979], [Szalas & Szcepanska 1985], and [Dony 1990] can lead to misunderstandings. All the other definitions presented above are very similar. Throughout this thesis we will used the definition of exception we consider the most precise, i.e. *an exception is a circumstance which might prevent a program from providing its specified service*[1] [Cristian 1989].

In the next sections, we will present a set of features provided in programming languages to handle exceptions. The clear comprehension or exception handling mechanisms in programming languages depends on the understanding of the terminology of *exception occurrence*, *exception declaration*, *exception handler* and *signaling, raising, and catching exceptions.* Before going further and starting to describe exception handling mechanisms in programming languages, we present a brief definition for these terms based on [Goodenough 1975], [Cristian 1982], and [Campbell & Randell 1986].

Exception occurrence. The invocation of a service in which the objective of the service is not fulfilled. This implies that the determination of ex-

[1]Issues related to the time (when) this service is provided are not the concern of this thesis.

ception occurrences is dependent on what is considered as the objective of a service.

Signalling an exception. The notification of an exception occurrence to the invoker of the service.

Exception raising. The activation of an exception handler which is bound to the exception. The functionality of exception handling mechanisms in programming languages will be elaborated further in Section 3.4.2.

Exception handling. The handling of exceptions refers to the way a program attempts to recover from a situation in which some invoked services cannot achieve their objective, to a situation in which all invoked operations can, in principle, achieve their objective. There are several exception handling mechanisms in programming languages. In early programming languages, handling of exceptions was done by checking a returned error code signaled by a service (see Section 3.4.1). Recent programming languages offer a more sophisticated mechanism for exception handling (see Section 3.4.2).

Exception declaration. In some languages, exceptions can be declared in an exception declaration using a predefined exception type. In an exception declaration, a new exception is declared and an identifier is bound to this exception, so that the identifier denotes the exception (see Section 3.4.2.1).

Exceptional termination. When the invocation of a service results in an exception occurrence which is either handled in a handler bound to the service, or propagated to the service's invoker, the service is said to terminate with an exception. Such a termination of the service can also be referred to as an exceptional termination. If the handler bound to the service handles the exception and can provide its specified service, then the service terminates normally, i.e. as if the exception occurrence had not happened.

We could summarise the above definitions as follows. Exceptions are declared to denote a service that has not been achieved. If the service fails, then an exception is raised. The service could still achieve its objective by then executing some code designed to handle that exception. When the service cannot fulfill its obligations, even after handling the exception, then the service is said to terminate exceptionally and the invoker of the service is signalled with an exception.

3.2 Requirements and Decisions

When developing a new exception handling mechanism in a language, a designer has to take into consideration several sets of decisions and requirements. Some of the requirements for an exception handling mechanism are application independent, while others do depend on the kind of application. When developing and testing a program, for example, an appropriate response to an error could be to stop execution of the program and enter a debugger. In other situations this response would certainly be unacceptable. Several authors propose a set of requirements that should be met by a general-purpose mechanism for the handling of exceptions [Goodenough 1975] [Beek 1993] [Burns & Wellings 1996]. These requirements are presented below.

i) The mechanism should facilitate the creation of robust programs.

ii) All exception occurrences which are detected should allow exception handling to be executed.

iii) The mechanism should be easy to use and understand. It should consist of a small number of elements that can be used independently of other programming elements.

iv) There should be a clear separation of the code for normal service operation and the code for exception handling.

v) User programs should be able to handle exceptions detected in the user program itself and those detected in the routines of the supporting system in the same way.

vi) If an exception occurrence results in the termination of a call chain of several levels, each level should be allowed to fulfill its own finalization obligations (commands that are executed in order to bring the component in which the exception occurred to a safe and consistent state).

vii) The mechanism should not incur in run-time overhead to the normal execution of a program.

As stated by requirement (*ii*) above, common ways of handling exceptions, such as printing out an error message and then stopping the execution of the program, is unacceptable.

The structuring of complex programs is very important to improve the reliability of systems. Requirement (*vi*) states that a programmer should be able to use abstraction not only to facilitate the design of complex programs, but also to program the exception handling operations.

Although the set of requirements presented above is important when creating a new exception handling mechanism, none of the requirements mention anything about exception handling in concurrent processes. In a system where several processes are being executed concurrently, an exception occurrence in one of the processes can cause serious consequences in the other processes. Therefore, we believe that an important requirement for an exception handling mechanism is the provision of methods for handling concurrent exceptions or to help in the enclosure of concurrent processes that may be affected by an exception occurrence in a different process. (We return to these subjects in Section 3.4.3)

In addition to the requirements presented above, the following list of design issues for an exception handling mechanism will determine its semantics, usability, and ease of implementation ([Ghezzi & Jazayeri 1998] and [Sebesta 1989]):

a) Description of exceptions that can be handled;

b) The way those exceptions can be defined;

c) The units that can raise an exception;

d) The way the handlers are defined;

e) Definition of the binding between an exception and its handler;

f) The continuation point for the control flow after an exception has been handled;

The choices made with regard to these design issues differ from language to language and are discussed in the next sections.

3.3 Termination vs. Resumption Model

An important aspect of exception handling is that of where control flow should continue after an exception is handled. There are two possible solutions, which correspond to different styles of handling exceptions: *termination* and *resumption* models. The resumption model implies that the handler's code may cause control flow to return to the point where the exception was raised, whereas the termination model does not allow that. Instead the termination model implies that the unit in which the exception was raised is terminated and a command that does not belong to the unit is executed (in Section 3.4.2.4 this is discussed further). A few languages, e.g. PL/I, use the resumption model, while more languages support the termination model, e.g. Ada95, Java, C++. Some languages that use the termination model consider the situation in which the resumption model may be used once the handler is activated, for example the Eiffel **retry** instruction used inside a **rescue** statement.

Although for many years the debate on termination versus resumption gave no clear indication of which approach is superior, in recent years termination has gained wider acceptance. Indeed, even the main proponents of the resumption model have abandoned it in favour of the termination model [Cristian 1989]. Practical experience with languages providing the resumption model has shown that this model is more fault-prone [Horning 1979]. For example, its use can promote the unsafe programming practice of removing the symptom of an error without removing the cause; e.g. the exception raised for an unacceptable value of an operand could be handled by arbitrarily generating an acceptable value and then resuming the computation.

3.4 Approaches for Exception Handling

3.4.1 Using Return Values

The simplest way of exception handling is by passing special codes through arguments or returned values of procedures. For example, this is the way internal exceptions are handled in the Unix operating system. Although the advantage of this method is that it is very simple to understand and requires no special support, there are several drawbacks to this way of exception handling. Using return values as exception codes means that the results of all operations must be checked for the occurrence of exceptions. This may result in the following several severe consequences:

i) The creation of robust programs becomes very difficult. The amount of code needed just to check error return codes becomes enormous. A single test can easily be forgotten. This will not interfere with normal program execution and may thus remain unnoticed for a long time.

Then when there is an exception occurrence, the failure to test for the exception occurrence will cause the program to continue in an incorrect state with possibly disastrous consequences.

ii) Many system operations do not return any exception code. Therefore, no exception handling can be executed for those exception occurrences. Consider, for example, arithmetic operations such as division. A simple division command could result in a division by zero. It is unacceptable that such an exception occurrence simply prints a message on the terminal and stops the system, because this makes it impossible for the user program to do any damage confinement and exception handling.

iii) The checking code for an exception occurrence will be mixed with the code for normal program execution, leading to programs that are hard to read and maintain.

As shown above, checking return values does not provide a way of handling exceptions that satisfies the requirements presented in Section 3.2. In order to support the handling of exceptions such that it conforms to those requirements, some programming languages have therefore implemented advanced exception handling mechanisms in sequential processes.

3.4.2 Exception Handling in Sequential Processes

Advanced exception handling mechanisms in most programming languages differ only in minor aspects, the underlying concepts being the same. This section will focus on the common qualities of the mechanisms rather than on the differences. The examples given will use either the Ada95 [ISO 1995] or the Java [Gosling *et al.* 1996] programming languages.

3.4.2.1 Exception Declaration

Exceptions can be declared (and created) in exception declarations. Figures 3.1 and 3.2 show how exception declaration is done in Java and Ada95 respectively. For instance, in Java all user-defined exceptions are declared as classes that inherit from the **Exception** class. Exceptions are instances of these new classes.

```
public class SensorException extends Exception {
    public SensorException() {};
}

SensorException exception = new SensorException(); // Java exception
```

Figure 3.1: Exception Declaration in Java

```
SensorException : exception; -- Ada95 exception
```

Figure 3.2: Exception Declaration in Ada95

3.4.2.2 Exception Raising

Programming languages provide a statement to raise an exception. In Java and Ada95, this is done by means of the predefined **throw** and **raise** instruction respectively (see Figures 3.3 and 3.4).

When an exception is raised, it may be handled by an exception handling code. If there is no exception handling code for that exception, then the

```
throw exception; // Java exception declared previously
```

Figure 3.3: Raising Exception in Java

```
raise SensorException; -- Ada95 exception being raised
```

Figure 3.4: Raising Exception in Ada95

exception is signalled to the invoker of the service where the exception was raised.

3.4.2.3 Exception Handlers

An exception handler is bound to a syntactical unit and to an exception. In this way associations or bindings between the handler and the unit, and between the handler and the exception are created. In some programming languages, a handler can be bound to multiple exceptions, or to an exception that is a top-level exception to several exceptions. Also, several exception handlers can be bound to the same unit. The kind of unit that exception handlers can be bound to depends on the programming language used. Handlers can usually be bound to procedure and function bodies. Some languages also allow handlers to be bound to blocks of commands (other than procedure and function bodies) and to modules or packages. The binding of a handler to a unit and to an exception is usually static, which means that the association between the handler and the unit and between the handler and the exception is determined at compile-time. In Ada95, handlers can be attached to a block, a body of a subprogram, a package or a task. In Ada95, exception handlers are specified by means of the reserved word **exception** followed by the declaration of one or more handlers at the end of a unit (see Figure 3.5). In Java, exception handlers are attached to a block of statements by means of the **try/catch** command (see Figure 3.6).

```
begin
   -- ... statements
   exception when SensorException => -- handler for SensorException;
end;
```

Figure 3.5: Exception Handler in Ada95

```
// attaching an exception handler to a block of statements in Java
try {
   // ... statements
} catch (SensorException se) {
   // handler for SensorException
}
```

Figure 3.6: Exception Handler in Java

A unit is said to have a handler for a specific exception if a handler which can catch the exception is bound to the unit. A handler can catch only those exceptions that are bound to it. In Figure 3.6, the Java body has a handler for the SensorException exception (the Ada95 block, in Figure 3.5, has also a handler for the SensorException exception). Most programming languages allow special handlers to be defined that can catch any exception, or all exceptions for which no other handlers are defined. This is necessary for the handling of "out of scope exceptions" or to allow the unit in which the exception occurred to finish in a consistent state. Out of scope exceptions are exceptions which are propagated out of the scope of their declaration. If this occurs, the only way to catch them is with an "any handler". In Java, an any handler is declared by means of specifying Exception as the exception of a handler (see Figure 3.7) and in Ada95 **others** is used for this purpose (see Figure 3.8).

```
// attaching an exception handler to any exception in Java
try {
   // ... statements
} catch (Exception e) {
   // handler for any exception
}
```

Figure 3.7: Any Exception Handler in Java

```
begin
   -- ... statements
   exception when others => --handler for any exception;
end;
```

Figure 3.8: Any Exception Handler in Ada95

3.4.2.4 Handlers' Responses

When an exception is raised in a unit for which the unit has a handler, control will pass to the handler. There are six main different ways in which a handler can end. They are all based on the termination or resumption model presented in Section 3.3. Each has a different effect on the continuation of the control flow. Four of them are associated with the exception caught. These are referred to as the return, propagate, retry and resume response. The fifth response would be to raise another exception. In the literature a sixth response is often given, namely the transfer response which can continue with any statement in the unit associated with the handler. This functionality is similar to that of the goto statement. Since goto programming is generally acknowledged to be bad programming practice, the transfer response is considered as being undesirable. The functionality of the other five responses are explained below.

The return response has a similar effect as the return statement, which can be used to return from procedures or methods. The invocation of the unit to which the handler is bound is terminated. The value returned by the handler is used as the returned value of the terminated unit. In most systems, this response is the default way of returning from an exception handler if no explicit response is programmed in the handler.

The propagate response from a handler causes propagation of the exception to the invoker of the unit to which the handler is bound. For the invoker of the unit it makes no difference whether an exception is propagated to it directly by a unit which has no handlers for an exception, or whether the exception is first handled by a handler and then propagated by means of the propagate response from the handler.

The retry response first terminates the invocation of the unit associated with the handler. The unit is then reinitialized and execution continues at the start of the unit.

The resume response causes execution of the program to continue right after the point where the exception was raised. The resumption response assumes that the exception handler has corrected the error causing the exception in such a way that the program can be continued right after the point where the exception was raised. The resume response is not possible for all exceptions. Whether or not a handler should be able to issue a resume response is not only the responsibility of the handler, but also of the part of the program where the exception was raised. For many exception occurrences it is clear at the time of raising the exception that continuation after the exception occurrence should never take place. Therefore, most programming languages that allow the resume response have two ways of raising an exception. When an exception is raised in the normal way, the resume response from a handler is not allowed. It is only allowed when an exception is raised using a special raise primitive.

The fifth way to continue after execution of a handler is to raise another exception in the handler. This exception is then propagated to the invoker of the unit associated with the handler, just as if the exception was propagated directly from the unit.

Ada95 supports the return, raise of a new exception and propagate response. In Ada95, the propagate is done by calling the predefined raise operation with no arguments. Java supports the return, propagate, and the raise of a new exception. The propagate response is invoked in Java by rethrowing the exception that was caught. Examples of the resume response can be found in PL/I and Eiffel.

3.4.3 Exception Handling in Concurrent Processes

Exception handling in sequential programs is a well-known subject with several languages providing mechanisms for handling exceptions. Exception handling in parallel programs is much more complex than in sequential programs. Exceptional termination in a process can have a strong impact on other processes. For example, consider a set of processes that communicate with each other via a rendezvous mechanism. A process may terminate abruptly due to the presence of an exception. Processes that need to communicate with the process that was terminated, may be suspended for ever because the terminated process will not be ready for communication anymore.

The problem of dealing with concurrent exceptions has been addressed in different systems [Digital 1986] and models [Campbell & Randell 1986] [Issarny 1993]. For example, the VAXELN programming environment from Digital [Digital 1986] provides means for a process to raise exceptions in other processes. The raising of an exception in a different process is done in an unstructured manner. A process can enable or disable this kind of exceptions. Raising an exception in a process that has disabled this kind of exception has no effect. The approach of allowing an exception in a process to be raised in a different process outside a structured framework can have a devastating effect on program modularity. This is especially the case when the raising of exceptions in other processes cannot be restricted.

In [Campbell & Randell 1986], a model for dealing with concurrent exceptions explores the use of an exception tree. Exceptions that can be signalled by a component of a parallel block C are organized in a tree structure. The root of this tree contains the *universal exception*, i.e. the exception that represents the whole exceptional domain of C. In their model, when more than one exception is raised concurrently, a handler for an exception that is ancestor to all the ones that were raised is executed. In the worst case scenario, a handler for the *universal exception* is executed. In [Issarny 1993], a different model that relies on the definition of resolution functions within classes is presented. In this model, a resolution function takes a sequence of exceptions as input parameter and returns an exception.

Despite all this work, mechanisms for dealing with concurrent exceptions in programming languages are still in their early stages. For example, in the Ada95 [ISO 1995] rendezvous mechanism, if an exception is raised during a rendezvous and not handled in the **accept** statement, that exception is propagated to both tasks and must be handled in two places. However, Ada95 does not provide any mechanism to handle concurrent exceptions.

3.5 Comparative Evaluation

In Appendix B, a survey is given of the different approaches used in a number of programming languages to provide exception handling. Although a lot of research has been done in recent years and terminology is becoming uniform, there are still differences and there is no consensus on a standard mechanism that languages should adopt. Furthermore, the way concurrent exceptions have to be handled is not a major concern in the languages surveyed.

One of the major issues in defining an exception handling mechanism in a programming language is the way programs should be structured to handle the exceptional occurrences. In the languages surveyed, almost all languages have similar approaches for the type of exceptions that can be handled and how they can be defined. They all provide built-in and user-defined exceptions. One of the major differences is whether an exception can carry information or not. In PL/I, an exception is a named signal that does not allow any additional information to be carried with the exception to the handler. Languages like Ada95, CLU, Java, C++ or ML allow data to be carried along with the exception. C++, for example, allows any kind of object to represent an exception.

Exception handlers in Java and C++ can be attached to any block of commands. Eiffel allows exception handlers to be only attached to routines. In CLU, an exception handler cannot be attached to the routine in which the exception was raised. In ML exception raising and exception handlers are attached to expressions or functions. In PL/I the binding between an exception and an exception handler takes place when an **ON** condition is encountered during the execution. The handler is valid then during the execution of the block of commands in which the **ON** condition was declared, until the termination of the block.

In the languages surveyed, none provides means for handling concurrent exceptions. Some of the languages provide means for raising concurrent exceptions but it is presumed that they can be handled inside the processes (threads) in which they were raised. Ada95 provides a mechanism for raising the same exception in two processes. This situation may happen during an Ada95 rendezvous. If an exception is raised inside the rendezvous and is not handled, then the exception is propagated to both caller and callee tasks.

Exception handling is getting more and more common in programming languages, and it is currently accepted as an important mechanism to be provided in new languages. For example, Java takes the handling of exceptions very seriously and does not allow a program to be compiled unless exceptions that may be raised in a command are caught. Despite this wide acceptance of exception handling in "sequential" programs, the inclusion of exception handling mechanisms for concurrent exceptions in programming languages is still needed. The next chapter introduces the idea of handling concurrent exceptions in a multiparty interaction mechanism, and then shows how this could be embedded in a programming language.

Chapter 4

Dependable Multiparty Interactions

In the previous chapters *multiparty interactions* and *exception handling* were presented. As mentioned in Chapter 2, existing multiparty interaction mechanisms do not provide features for dealing with possible faults that may happen during the execution of the interaction. In some, the underlying system that is executing those multiparty interactions will simply stop the system in response to a fault. In DisCo, for instance, if an assertion inside an action is false, then the run-time system is assumed to stop the whole application. As explained in Chapter 3, this situation is unacceptable in many situations.

In this chapter, both mechanisms are brought together, i.e. exception handling is added to multiparty interactions. This new mechanism is called a *dependable multiparty interaction* (DMI). Specifically, a DMI is a multiparty interaction mechanism that provides facilities for:

- HANDLING CONCURRENT EXCEPTIONS: when an exception occurs in one of the bodies of a participant, and not dealt with by that participant, the exception must be propagated to all participants of the interaction [Campbell & Randell 1986] [Romanovsky *et al.* 1996]. A DMI must also provide a way of dealing with exceptions that can be raised by one or more participants. Finally, if several different exceptions are raised concurrently, the DMI mechanism has to decide which exception will be raised in all participants.

With respect to how the participants of a DMI will be involved in the exception resolution and exception handling, there are two possible schemes [Randell *et al.* 1997] [Romanovsky 2000]: synchronous or asynchronous. In synchronous schemes, each participant has to either come to the action end or to raise an exception; it is only afterwards that it is ready to participate in any kind of exception handling; this means that the participant's execution cannot be pre-empted if another participant raises an exception. In asynchronous schemes, participants do not wait until they finish their execution or raise an exception to participate in the exception handling; once an exception is raised in any participant of the DMI, all other participants are interrupted and handle the raised exceptions together. Although implementing synchronous schemes is easier than asynchronous, because all participants are ready to execute the exception handling, the synchronous scheme can bring the undesirable risk of deadlock. Therefore in this thesis the asynchronous scheme will be adopted.

- ASSURING CONSISTENCY UPON EXIT: a participant can only leave the interaction when all of them have finished their roles and the external objects are in a consistent state. This property guarantees that if something goes wrong in the activity executed by one of the participants, then all participants have an opportunity to recover from possible errors.

The key idea for handling exceptions is to build DMIs out of not necessarily reliable multiparty interactions by chaining them together, where each multiparty interaction in the chain is the exception handler for the previous multiparty interaction in the chain. Figure 4.1 shows how a basic multiparty interaction and exception handling multiparty interactions are chained together to form a composite multiparty interaction, in fact what we term a DMI, by handling possible exceptions that are raised during the execution of the DMI. As shown in the figure, the basic multiparty interaction can terminate normally, raise exceptions that are handled by exception handling multiparty interactions, or raise exceptions that are not handled in the DMI. If the basic multiparty interaction terminates normally, the control flow is passed to the callers of the DMI. If an exception is raised, then there are two possible execution paths to be followed: *i*) if there is an exception handling multiparty interaction to handle this exception, then it is activated by all roles in the DMI; *ii*) if there is no exception handling multiparty interaction to handle the raised exception, then this exception is signalled to the invokers of the DMI. The whole set of basic multiparty interaction and the exception handling multiparty interactions are represented as one entity, a composite multiparty interaction since they are isolated from the outside in order that, for example, the raising of an exception is not seen by the enclosing context of a DMI.

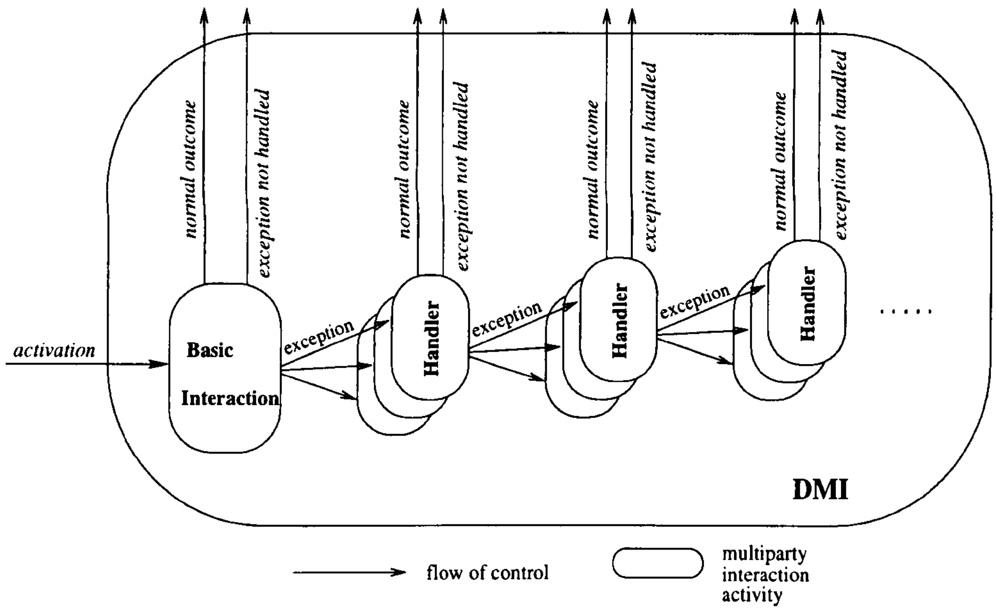

Figure 4.1: Dependable Multiparty Interaction

The exceptions that are raised by the basic multiparty interaction or by a handler, should be the same for all roles in the DMI. If several roles raise different concurrent exceptions, the DMI mechanism activates a exception resolution algorithm based on [Campbell & Randell 1986] to decide which common exception will be raised and handled.

In view of our interest in dependability, and in particular fault tolerance, we adopt the use of pre and post-conditions, which are checked at run-time. Regarding the remaining alternatives listed in Chapter 2, we have made the following design choices for DMIs:

i) Although the particular processes involved should be able to vary from one invocation of a DMI to the next, their number in a given DMI should be fixed.

ii) The processes should synchronise their entry to and exit from the DMI.

iii) The DMI mechanism should ensure that as viewed from outside the DMI, its system state should change atomically, though inside the DMI intermediate internal states will be visible.

iv) The way the underlying system executes a DMI can be synchronous or asynchronous.

The choice for allowing a varying set of processes to enrole into a DMI is related to the expressive power of the language construct we intend to provide. Recall from Section 2.2, that the basic construct that presents the higher degree of expressiveness is a team. A DMI is a team, hence choice (*i*) was made. Synchronisation upon entry and exit (choice (*ii*)) is crucial if we want to have some kind of guard to be tested before the DMI commences, or an assertion to be tested before the DMI terminates. For example, if participants in a DMI are allowed to terminate without synchronising upon exit, then the process of involving that participant in the handling of an exception raised by another participant of the DMI will be much more difficult. [Davies 1978] discusses several issues related to termination of processes that should not interfere with each other, e.g. issues related to error recovery before a process has terminated, or error recovery after a process has terminated the execution of an activity. Choice (*iii*) is related to the visibility of shared data inside the DMI and outside of the DMI. The related "frozen initial state" property discussed in [Evangelist *et al.* 1989] is used in relation to the participants that are outside the DMI, i.e. they see the change of shared data as being instantaneous when the DMI terminates. Our proposal differs from [Evangelist *et al.* 1989] in relation to the visibility of shared data inside the DMI. In our proposal, participants can exchange data inside the DMI, while in [Evangelist *et al.* 1989] participants of a multiparty interaction view shared data as "frozen" when the multiparty interaction commences.

In the next sections, a sub-set[1] of new language based on Interacting Processes (IP) [Forman & Nissen 1996] and Distributed Cooperation (DisCo) [Jårvinen & Kurki-Suonio 1991] is presented. This new language is a test bed for implementing DMIs. It provides DMIs as a basic language construct. Section 4.1 presents the sub-set of commands for the Dependable Interaction Processes (DIP) language. Section 4.2 presents the formal basis of some of the DIP commands, including the formal description of DMIs. The formal semantics of the DIP commands will be presented in Temporal Logic of Actions (TLA) [Lamport 1994].

4.1 Dependable Interacting Processes - DIP

Dependable Interacting Processes (DIP) is a language that allows a designer to specify exception handling in multiparty interactions. DIP extends languages like DisCo and IP (Interacting Processes), where exceptions are not considered in the specification of a system.

[1]The main idea behind the introduction of this new language is to show how the DMI concept can be embedded in a programming language, therefore simpler commands are prefered to more sofisticated ones, e.g. commands **to/from** instead of a **select** command for message passing.

A program in DIP is composed of a set of objects O (instances of classes), a set of teams T (instances of actions), and a set of players P (instances of processes).

- $program = \{O,\ T,\ P\}$

Each program in DIP has a global state that is represented by the set of objects O. The objects that represent the global state of a DIP program are called *global objects;* O_{global}. Changes to the global objects of a DIP program can only be made inside a team belonging to the set of teams T. Players in DIP are responsible for specifying the order in which the teams in T are executed, hence the order in which the state of a DIP program changes.

4.1.1 DIP Basics

This section presents the core commands and the basic data types of DIP. The commands can be used inside a team or inside a player's main body. In the following definitions we use [] to express optional items and *italics* to represent something that is yet to be defined.

4.1.1.1 DIP Commands

A command in DIP can be one of the forms: assignment command, state change, selection command, iteration command, role activation, message passing, block of commands, exception handler, exception raising.

- Assignment command:

  Syntax: *variable* := *expression*

In an assignment command, first the *expression* is evaluated and then the computed value is stored into *variable*.

- State-change command:

  Syntax: -> *object-state* [:: *exception-list*]

  A state-change command changes the state of the object to the new state represented by *object-state*. During the changing of state, the state machine can raise exceptions specified in *exception-list*.

- Selection command:

  Syntax: **if** (*expression*) **then** *command* [**else** *command*]

  A selection command executes the command after the word **then** only when the computation of *expression* results in the true value. If the **else** part is inserted in the selection command, then the command after **else** is executed only when the evaluation of *expression* is false.

- Iteration command:

  Syntax: **loop** [**while** (*expression*)] [**until** (*expression*)] **do** *command*

  The iteration command expresses several possibilities of executing the command it encloses. Firstly, if neither the **while** clause nor the **until** clause is used, then the command will be executed forever. If the **while** clause is used, first the expression is evaluated, and if the value of the expression is true, then the enclosed command is executed. If the result of the expression is false, then the enclosed command is not executed and the iteration is terminated. If the **until** clause is used, then after the enclosed command was executed the expression is evaluated and if its value is true, the iteration terminates, otherwise the enclosed command may be executed again.

- Role activation:

  Syntax: *role-name@team-name* **with** *list-of-parameters*

  This command activates the role *role-name* in the team *team-name*. The *list-of-parameters* is a list of objects separated by comma. Complete explanation of roles and teams follows in the next sections.

- Message sending:

  Syntax: *variable* **to** *role-name* [**blocks**]

  The value stored in *variable* is sent to a role (roles have to be in the same DMI). If the clause **blocks** is specified then the command blocks the role until the receiver (role) has read the message.

- Message receiving:

  Syntax: *variable* **from** *role-name* [**blocks**]

  The value sent by a role is stored in *variable*. If roles have not sent a message to the role that is trying to receive a message, then an empty value is stored in the variable (this empty value can be tested using a constant called **empty**). If the clause **blocks** is specified, then the command blocks until a message arrives. The role that is receiving a message can indicate which role it wants to receive a message from. Messages are treated in the order "first-in, first-out".

- Requeueing message:

  Syntax: **requeue** *variable*

  The value in *variable* is stored in the queue of messages of the corresponding role.

- Block of commands:

  Syntax: **begin** *list-of-commands* **end**

  Syntax: *list-of-commands* :: *command* [; *list-of-commands*]

- Exception handler:

  Syntax: **try** $command_n$ **on exception** ($exception\text{-}list_{e1}$) $command_{e1}$ ...

  The $command_n$ is executed. If an exception in $exception\text{-}list_{ei}$ is raised in $command_n$, then $command_{ei}$ is executed. If there is no handler for a raised exception, then the exception is signalled to the invoker of the **try** command. This command is similar to the one used in C++ and Java.

- Exception raising:

  Syntax: **raise** *exception*

  This command represents the raising of an exception and is similar to the one used in C++ or Java.

All the above commands are used inside roles in DIP teams, or inside the main code of a player. Apart from the **to/from** commands, all the other commands are based on commands that are used in several other languages, e.g. exception handler and raising are based on Java; role activation is based on IP; state-change command is based on DisCo; the assignment, iteration, and selection command are similar to the ones used in several languages, such as Eiffel, Java, DisCo, C++, ...

4.1.1.2 DIP Data Types

In DIP, basic data types can be declared inside objects, teams, and players. The data types used inside an object will compose part of the object's state. For teams and players, basic data types are used to declare local variables, which cannot be seen outside the scope they were declared. This prevents two players or roles from seeing the same variable.

DIP provides the following basic data types: **integer**, **float**, **char**, **string**, **boolean**, and **list** (elements of a list can be accessed via an index, e.g. l[0] gives the first element of the list l). Variables of these data types are declared inside a class; at the beginning of the declaration of a process; or, at the beginning of the declaration of a role.

4.1.2 DIP Objects

All operations that change the state of a program in DIP are realised inside multiparty interactions (i.e. in DIP teams as explained later). Therefore objects are only used for keeping the program state, or intermediate state of teams or players, i.e. objects do not provide methods as in programming languages like C++, Java or Eiffel. Objects in DIP are thus similar to objects in DisCo.

Each object in DIP has its own state, that is part of the global state of the program, and a set of assertions to check that the state of the object does not violate some requirements specified by the designer of the object. Assertions are checked at the moment an object is created and at the end of the execution of the multiparty interaction. Because the state of an object is modified only inside a multiparty interaction, during the execution of the multiparty interaction, the object's assertions do not need to remain valid. Each assertion is represented by a boolean expression. If the evaluation of this boolean expression results in false, then the exception attached to the assertion is raised into the multiparty interaction that used that object. If the assertion is false at the time the object is created, and this object belongs to the global state of the program, the program players do not start their execution and the program is terminated abnormally.

Classes

Objects are instances of classes. A class is a template description that specifies the state of a set of objects. The state is represented by a set of variables and a set of state machines. Class definition in DIP has the format shown in Figure 4.2.

```
class class_name is
    state_definition
    assertions_definition
end class
```

Figure 4.2: Class Declaration in DIP

Classes in DIP can be extended to form new classes. A special way of extending classes is using *inheritance*. In DIP, we use inheritance as follows. A new class N can inherit from any existing class. A class can inherit from more than one existing class (this is called *multiple inheritance* [Meyer 1997] [Ghezzi & Jazayeri 1998]). The state of an object from class N is composed of the new state defined in N and the state defined in the classes N is inheriting from. Multiple inheritance is used in languages like Eiffel [Meyer 1992] and C++ [Stroustrup 1994].

Multiple inheritance may bring a possibility of name clashes. Some of the variables of the extended classes can have the same name but different or unrelated semantics that are inherited from different classes. Any language that has multiple inheritance must deal with this problem of name clashes [Meyer 1997]. Three conventions are possible for solving this problem: *i*) require the designer of a new class to remove any ambiguity, by explicitly qualifying the variable with the inheriting class name (this convention is adopted by C++ [Stroustrup 1994]); *ii*) choose a default interpretation, i.e. the underlying language mechanisms select one of the variants (this approach has been implemented in several Lisp-based languages); and, *iii*) rename the conflicting names in the the inherited class (this is the approach adopted in Eiffel [Meyer 1992]). In DIP, the name clashes are solved by qualifying class variables, i.e. using approach (*i*). Whenever there is an ambiguity in the access of a class variable, it must be resolved through qualifying the class variable with the appropriate class name.

In the example in Figure 4.3, three classes are defined. Each class has its own internal state. In the example, class C inherits from classes A and B. Classes A and B have one variable that have the same name, variable s, lines 2 and 6 respectively. An instance of C, object obj in line 13, when using the variable s will have to specify which s it is using, from class A or class B, see line 15 and line 16.

Examples and thorough discussion about inheritance and multiple inheritance can be found in [Meyer 1997].

```
class A is
   integer s,a;
end class;

class B is
   string s,b;
end class;

class C inherit A, B is
   float b;
end class;

C obj;

obj.A:s = ... // s from A
obj.B:s = ... // s from B
obj.b   = ... // redefined b
obj.a   = ... // a from A
```

Figure 4.3: Multiple Inheritance in DIP

4.1.3 DIP Teams

Teams are used to describe dependable multiparty interactions in DIP. A team t in DIP is composed of a name, a main body b (called simply the body from now on) and a set of handler teams H. The body of a team and the handler teams are associated via the exceptions that can be raised inside the body of the team.

- $t = \{name_{team},\ b,\ H\}$
- $H = \{h_1, ..., h_n\},\ where\ n \geq 0$

An example of the team structure is in Figure 4.4.

```
action $name_{team}$ is
    body is
        body b
    exceptions $e(e_1, e_2)$
    end body
    handler for $e$ is
    body
        body of handler for exception $e$
    end body
    handler for $e_1$ is
    body
        body of handler for exception $e_1$
    end body
    handler for $e_2$ is
    body
        body of handler for exception $e_2$
    end body
end action
```

Figure 4.4: Team Structure in DIP

4.1.3.1 Team Body

Each body b is composed of: *i*) a set of roles R; *ii*) a set of objects O_{roles} that are manipulated by the roles, i.e. objects that are sent to the team as role parameters; *iii*) a set of local objects O_{local} that have the same semantics as global objects with respect to the roles in R, i.e. roles can modify these objects only inside another team; *iv*) a set of local teams LT (nested teams) used to modify the local objects in O_{local}[2]; *v*) a boolean expression, called *guard*, that checks the preconditions of the team and must be true in order for the roles of the team to start; *vi*) a boolean expression, called *assertion*, that checks the post-conditions of the team and must be true in order for the roles to finish normally; and a set of outcomes OUT it can produce, i.e.

[2]Objects in O_{roles} can also be modified by a team in LT

a normal outcome or one exception which is signalled to the callers of the team. The exceptions a team can signal are expressed as a list after the word **exceptions**, see Figure 4.4. The list is structured as a tree, e.g. $e(e_1,e_2)$ represents a tree in which e is the parent of e_1 and e_2. Local objects and local (nested) actions are created when all roles become active and destroyed when the roles become inactive again.

- $b = \{R, O_{roles}, O_{local}, LT, guard, assertion, OUT\}$
- $O_{team} = O_{local} \cup O_{roles}$
- $O_{roles} \subseteq O_{global}$

The syntax of a team body is based on the syntax of IP teams, while the semantics of the use of objects is similar to DisCo actions. Teams in DIP differ from the IP teams in the sense that IP allows the static definition of processes that belong to that team, while in DIP the only static computation code allowed is the one inside the roles of the DIP team. The semantics of objects that are sent to a DIP team are very similar to the way objects are treated in DisCo actions, i.e. they can only be used in one team at a time. For a better understanding, see the examples in DisCo and IP in Appendix A.

4.1.3.2 Team Guard and Assertion

Guard is a boolean expression, a precondition, over the objects that are carried to the team by the roles (O_{roles}). This boolean expression is tested only when all roles become active in an execution of a team. The *guard* states a necessary condition, not sufficient, for the team to start. If the *guard* does not hold, then an exceptional outcome is produced and it can be treated by a handler. The *guard* can be empty, having the same effect as if it is always true.

- *The body of a team* ***starts*** *when* <u>*all*</u> *roles are active and the guard of the body is true.*
- *If team t_1 and team t_2 are active, then $O_{t_1} \cap O_{t_2} = \{\}$.*

Assertion is a boolean expression, a post-condition, over the team's objects (O_{team}). This expression must be true in order for the team to finish normally. Similarly to the *guard*, if the *assertion* does not hold, then an exceptional outcome is produced and it can be treated by a handler. The *assertion* can be empty, having the same effect as if it is always true.

- *The body of a team* ***terminates*** *normally when* <u>*none*</u> *of the roles of the body fail and the assertion of the body is true.*

In Figure 4.5, we show how a *guard* and an *assertion* can be inserted in the body of a team. The ***guard-boolean-expression*** and ***assertion-boolean-expression*** are expressions over the objects of the team, i.e. O_{team}

Team guard and assertion have similar meaning to guard and assertions in DisCo. There are some major differences though. First, an action in DisCo is only activated if the guard is true, while in DIP the *body of the team* is only activated if the guard is true. If the guard is not true in DIP, then an exception can be raised and a handler for that exception will be tried. Second, assertions in DisCo can be inserted anywhere inside an action, while

```
action name_team is
    body is
     guard guard-boolean-expression
       ...
     assertion assertion-boolean-expression
    end body
end action
```

Figure 4.5: Guard and Assertion Declaration in DIP

in DIP, the assertion is used only for testing post-conditions, hence it is tested at the end of DIP team's execution. The third difference is that a DIP assertion can raise an exception if it is false, and an exception handler may be executed. In DisCo, if an assertion is false, then the system is assumed to stop.

4.1.3.3 Team Outcomes

A team can produce two different kinds of outcomes (results): *i*) normal, when all roles are activated, *guard* and *assertion* are satisfied; *ii*) exceptional outcome, when *guard* or *assertion* are false; when a role fails to perform its activity; or when an object being manipulated by a role has at least one of its assertions signalling an exception.

4.1.3.4 Team Roles

Roles are the means for describing computation inside a team. Each role r_i has a name, a set of objects, O_{role_i}, and a set of commands C_{role_i} (commands were explained in Section 4.1.1). The objects used by the role are a subset of the objects of the team. Roles are passive entities but become **active** when players, or the roles in a containing team, activate them.

- $R = \{r_1, ..., r_m\}$, *where* $m \geq 1$
- $r_i = \{name_{role_i}, O_{role_i}, C_{role_i}\}$, $for 1 \leq i \leq m$
- $O_{role_i} \subseteq O_{roles}$

An example of roles in a team is shown in Figure 4.6.

```
action name_team is
    body is
        role r_1 with object_1 is
            commands for role r_1
        end role
        ...
        role r_m with object_m is
            commands for role r_m
        end role
    end body
end action
```

Figure 4.6: Roles Declaration in DIP

Roles in DIP, have a similar syntax to roles in IP teams, but their semantics differ greatly. For example, in IP, roles do not have to start at the same time, they are more like methods in object-oriented languages. Synchronisation can be achieved, in IP, by means of its interaction basic construct.

4.1.3.5 Team Handlers

Each team t can have an associated set of handlers h_i. Each handler is composed of a set of roles, a set of objects that are manipulated by the roles, a *guard*, an *assertion*, a set of exceptional outcomes, and a set of exceptions handled by this handler. A handler is activated when one of the exceptions it handles is raised in the body of a team or in another handler. Handlers can be used for several purposes: to recover a team from an error situation; to relax the guard of a team[3]; to relax the assertion of a team; to execute a new diverse version of a team with different guard, roles, and assertion; and so on. A handler has basically the same structure as the body of the team, but is activated by an exception or set of exceptions H_i. OUT_i is the set of outcomes the handler h_i can produce.

- $h_i = \{b,\ H_i,\ OUT_i\}$

Connection. A team t is connected to a handler h_i if there is an exceptional outcome from t that is handled by h_i. We express a connection by $\dashrightarrow$. A handler h_i is connected to a handler h_j, $h_i \dashrightarrow h_j$, if an exceptional outcome of h_i is an exception handled by h_j. A team t is also connected to h_j if there is a sequence of connections that leads t to h_j, i.e. there is a sequence $t \dashrightarrow h_i \dashrightarrow \ldots \dashrightarrow h_k \dashrightarrow \ldots \dashrightarrow h_j$. If h_i is connected to h_j, then h_j cannot be connected to h_i, i.e. the connections produce a directed acyclic graph.

In Figure 4.7 we show an example that contains a team t and a set of handlers that t is connected to. This set is expressed as $c_t = \{h_1, h_2, h_3, h_4, h_5, h_6, h_7\}$. The connection set for each of the handlers in Figure 4.7 is as follows: $c_{h_1} = \{h_4, h_5, h_7\}$, $c_{h_2} = \{h_5, h_7\}$, $c_{h_3} = \{h_2, h_5, h_6, h_7\}$, $c_{h_4} = \{h_7\}$, $c_{h_5} = \{h_7\}$, $c_{h_6} = \{h_2, h_5, h_7\}$, and $c_{h_7} = \{\}$.

[3]E.g. imagine that a team needs two devices to execute an activity, but only one is available (the other may be broken), the guard will not be true, but a handler more complex than the body of the team can execute the same activity in a degraded mode.

$$\begin{tikzcd}
 & & h_4 \arrow[dr] & \\
 & h_1 \arrow[ur] \arrow[dr] & & h_7 \\
b \arrow[ur] \arrow[r] \arrow[dr] & h_2 \arrow[r] & h_5 \arrow[ur] & \\
 & h_3 \arrow[u] \arrow[r] & h_6 \arrow[ul] \arrow[u] &
\end{tikzcd}$$

Figure 4.7: Connection between a Team and its Handlers

The set of roles R and the set of objects O_{roles} in a handler h_i are the same as in the team t that is connected to h_i. Each handler can be connected to several other handlers. *Guard* and *assertion* in a handler can be different from those in the team t. The set of exceptions that is handled by a handler is a subset of the exceptional outcome of the body of the team or handlers that are connected to it. If the commands of a handler role are the same as in the team t, then these commands are assumed in the handler role and do not need to be rewritten. The clause **same** after the declaration of a handler role specifies this property.

- *if* $t \dashrightarrow h_i$, *then* $H_i \subseteq OUT$
- *if* $h_i \dashrightarrow h_j$, *then* $H_j \subseteq OUT_i$

The exceptional outcome of a team (OT) is composed of the union of the outcomes of the body of the team b and the outcomes of all handlers that b is connected to.

- $OT = OUT \cup (\cup_{i=1}^{n} OUT_i)$

Concurrent exceptions raised in different roles are dealt with using the model proposed by [Campbell & Randell 1986] and discussed in Section 3.4.3.

When defining the exceptions the body of the team or one of its handlers can signal, the designer has to provide a hierarchy of exceptions in the format of a tree, so in the case of more than one exception being raised at the same time, the highest of the two will be handled, if there is a handler to treat that exception. In the case of the exceptions being on the same level of hierarchy, an immediately higher exception than the two will be handled. Two handlers cannot handle the same exception. If there is no handler for the raised exception, then it is signalled to the invokers of the team.

Actions

Teams are instances of actions. An action is a template description which specifies properties and behaviour for a set of similar teams. Actions are the way of defining the structure of a team presented in the previous section.

4.1.4 DIP Players

Players are the sequentially executing entities in a DIP program. They are responsible for activating the roles in a team. Each player p_i is composed of a name, a set of local objects O_{p_i}, and a set of commands C_{p_i}.

- $P = \{p_1, ..., p_l\},\ where\ l \geq 1$
- $p_i = \{name_{p_i},\ O_{p_i},\ C_{p_i}\}$

All players, in DIP, start their execution immediately after all global objects and global teams are created. The players cannot change the state of global objects, but can inform a team about what global objects will be used by a team when they activate a role in a team. Players activate a role using the enrole command.

Processes

Players are instances of processes. A process is a template that contains the sequence of commands to be executed by a player after the player has been instantiated. A process has the format presented in Figure 4.8.

```
process name is
    player local objects
    body is
      commands
    end body
end process
```

Figure 4.8: Process Declaration in DIP

4.1.5 An Example in DIP

The dining philosophers problem was introduced [Dijkstra 1965] to show how to achieve synchronisation when a process needs more than one resource to execute an activity, e.g. one philosopher needs two forks to start an eating activity. Note that in the original specification of the problem only one process would start the operation. For our purpose, i.e. multiparty interaction, we will consider forks as being processes rather then resources, and an eating activity will only happen when the three-parties are ready to execute it.

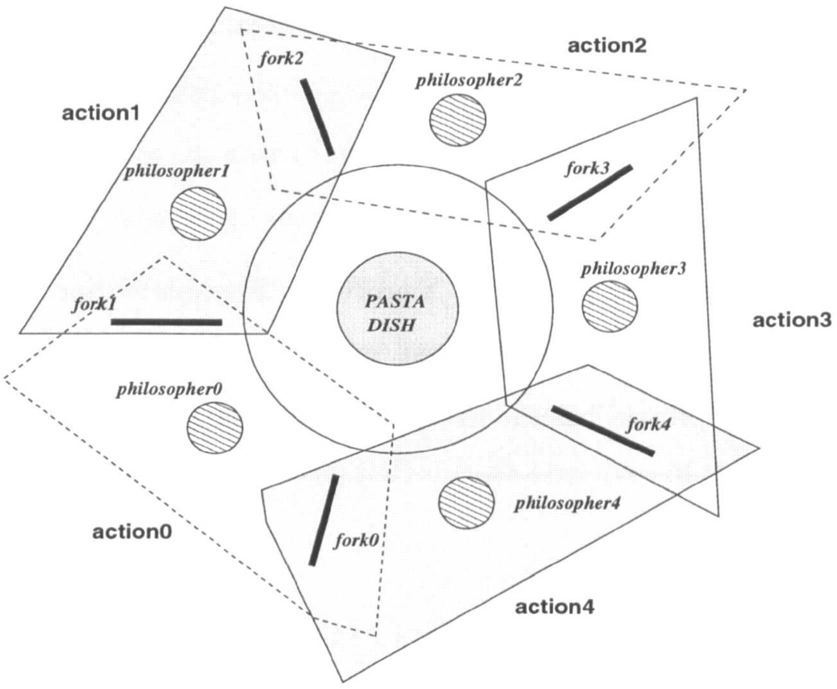

Figure 4.9: Dining Philosophers Problem

Definition: Let us consider a set of 10 processes divided in two groups of 5 processes: *forks* and *philosophers*, where *forks*= $\{fork_0, ..., fork_4\}$ and *philosophers*= $\{philosopher_0,..., philosopher_4\}$. These processes will, during the execution of the system, execute joint activities, called ***actions***= $\{action_0, ..., action_4\}$ that will be composed of 2 *fork* components and 1 *philosopher* component. Each component of *actions* is a tuple in the format $action_i = \{fork_i, philosopher_i, fork_{i+1}\}$, where $0 \leq i \leq 4$, and $i + 1$ is equivalent to $(i + 1) mod\, 5$. Let us also consider that each $action_i$ will need an external data called *pastaDish* in order to execute the joint activity, and that such external data can be accessed only by one action at a time. To better understanding of the problem see Figure 4.9.

Figure 4.12 shows how to instantiate forks, philosophers and actions in DIP. In the figure, philosophers are created as elements of a list that has indexes from 0 to 4. Upon instantiation the index of each element of the list is passed as parameter of the process.

```
Philosopher philosopher[i=0..4](i);
Fork fork[i=0..4](i);

EatingAction action[5];
```

Figure 4.10: Instantiating processes and teams in DIP

In Figures 4.10 and 4.11, we represent in DIP the processes that will execute the eating DMIs (named action). Each of these processes receives a number upon instantiation that specifies its identity, e.g. Philosopher p(1); creates a philosopher DIP object and the pn variable parameter is set to 1 to this philosopher object.

```
process Philosopher(pn:integer) is
  body
    loop
      action[pn].Philosopher with pasta
      // sleep ...
    end;
  end body
end process
```

Figure 4.11: Philosopher process in DIP

A fork process is also instatiated with a parameter that specifies which eating DMI it will act as right or left fork. If the parameter rn is an even number, then the fork process first executes the role on its right eating DMI, playing as the right fork role.

```
process Fork(rn:integer) is
 body
   loop
     if rn%2 == 0 then
       begin // action on the right first
         action[((rn+1)%5)+1].RightFork
         action[rn].LeftFork
       end
     else
       begin action on the left first
         action[rn].LeftFork
         action[((rn+1)%5)+1].RightFork
       end;
   end;
 end body
end process
```

Figure 4.12: Fork process in DIP

Every action[i] is an instance of the team EatingAction with the following roles: LeftFork, Philosopher and RightFork. This activity is executed in a coordinated way by the three participant processes. Firstly, the Philosopher and the LeftFork will synchronise their execution, and such synchronisation will represent the philosopher grabbing the left fork. A similar synchronisation will occur later representing the philosopher grabbing the right fork. After those, both forks will synchronise their execution in order to access the external data pastaDish. Such execution will consume part of the external data. When the LeftFork and the RightFork have got the "pasta", i.e. part of the external data, the three participants will synchronise their effort to "eat the pasta", i.e. consume the data. See Figure 4.13 and 4.14.

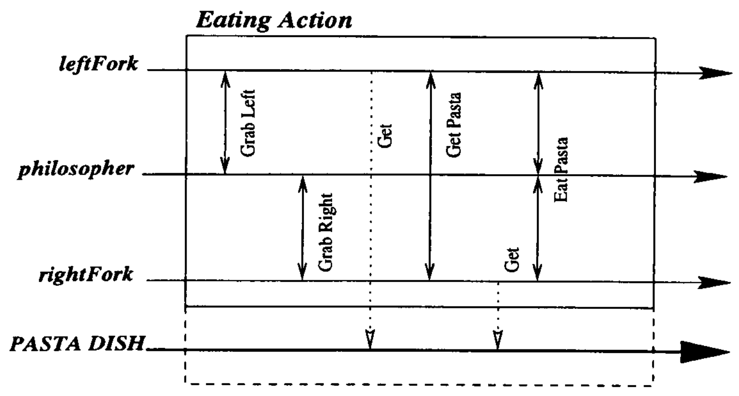

Figure 4.13: Eating Action

```
action EatingAction is
  body is
    role RightFork
      pasta from Philosopher // grabbed by philosopher
      ok from LeftFork
      y = pasta.Get
      y to Philosopher
    end role;
    role Philosopher with pastaDish:Pasta
      pasta to LeftFork
      pasta to RightFork
      y from RightFork
      x from LeftFork
      // eat x + y
    end role;
    role LeftFork
      pasta from Philosopher // grabbed by philosopher
      x = pasta.Get
      true to LeftFork
      x to Philosopher
    end role;
  end body;
end action;
```

Figure 4.14: Eating DMI in DIP

4.2 Formal Semantics

A well-defined syntactic and semantic description of a language is essential for helping good design and programming of a system. The *syntax* of a language describes the correct form in which programs can be written while the *semantics* expresses the meaning that is attached to the various syntactic constructs. While *syntax diagrams* and *Backus-Naur Form - BNF* have become standard tools for describing the syntax of a language, no such tools have become widely accepted and standard for describing the semantics of a language. Different formal approaches to semantics definition exist. For example, in *operational semantics* the behaviour of an abstract processor is used to describe the effects of each language construct; in *axiomatic semantics* rules are provided to show the relevant state changes immediately after the execution of a language construct; and *denotational semantics* programs are defined in terms of mathematical functions which are used to derive properties of the original program. This thesis does not describe these methods any further. Several authors report how to use these approaches for describing the semantics of programming languages [Tennent 1991] [Hennessy 1990] [Ghezzi & Jazayeri 1998].

Concurrent systems are usually described in terms of their behaviour - what they do in the course of an execution [Lamport 1999]. The Temporal Logic formal model [Pnueli 1977] was introduced to describe such behaviour of concurrent systems. A variation of Temporal Logic that makes it practical to write a specification as a single formula was presented in [Lamport 1994]. This variation is called Temporal Logic of Actions - TLA. TLA provides the mathematical basis for describing properties of concurrent systems.

In Section 4.2.1, an introduction to TLA is presented. In Section 4.2.2, the semantics of DIP is presented using TLA. The full syntax of DIP is presented in Appendix C.

4.2.1 Temporal Logic of Actions

The Temporal Logic of Actions - TLA [Lamport 1994] is a formalism suitable for describing state transition systems and properties of such systems using the same notation.

TLA is a linear-time logic in which expressions are evaluated for non-terminating sequences of states. Each sequence of states is called a behaviour. A state is an assignment of values to variables. Variables that are used to model properties are state functions, which have unique values in each state. A state function is a non-boolean expression built from variables, constants, and constant operators. Semantically, a state function assigns a value to each state. An individual state change is called a step. A step that allows variables to stay unchanged is called a stuttering step.

An action is a boolean expression containing primed and unprimed variables. Semantically, an action is true or false for a pair of states, with primed variables referring to the second state. An action is said to be enabled in a state s if and only if there exists some state t such that the pair of states $< s, t >$ satisfies that action.

Rather than presenting the full description of TLA, a simple program in DIP is presented with its corresponding TLA formula. The process, in

Figure 4.15, initialises a variable x with 0 and then keeps incrementing x by 1 forever.

```
process example is
 integer x := 0;
 body
  loop
     x := x + 1;
  end;
 end body
end process
```

Figure 4.15: Simple Program in DIP

The TLA formula for the above DIP process is defined as follows[4]:

$$\Pi \triangleq \begin{array}{l} \land \ (x = 0) \\ \land \ \Box[x' = x + 1]_x \\ \land \ WF_x(x' = x + 1) \end{array}$$

A TLA formula is true or false on a behaviour. Formula Π, presented above, is true on a behaviour in which the i^{th} state assigns the value $i-1$ to x, for $i = 1, 2, ...$. In the above TLA formula, the conjunct $(x = 0)$ specifies that initially, x is equal to 0; the conjunct $\Box[x' = x + 1]_x$ specifies that the value of x in the next state (x') is always ($\Box$) equal to its value in the current state (x) plus 1. The subscript x specifies that stuttering steps are allowed, i.e. steps where the value of x is left unchanged. The $WF_x(x' = x + 1)$ conjunct rules out behaviours in which x is incremented only a finite number of times. It asserts that, if the action $(x' = x + 1) \land (x' \neq x)$ ever becomes enabled and remains enabled forever, then infinitely many (x'

[4]The symbol $\triangleq$ means *equals by definition.*

`= x + 1)` $\wedge$ `(x'` $\neq$ `x)` steps occur. `WF` stands for Weak Fairness. In practice, the specification of `WF` above implies that any step changes `x`.

Because TLA formulae can be as long as several pages (see [Lamport 1996]), TLA$^+$ was introduced to help in writing such large formulae. TLA$^+$ is a language for describing TLA formulae. Operators in TLA$^+$ are classified as constant operators, action operators, and temporal operators. Appendix D shows the TLA$^+$ operators. (For further reading about TLA see [Lamport 1994] [Lamport 1999].)

In the next section the specification of some of the DIP commands is presented in TLA$^+$.

4.2.2 Properties of DIP

DIP basic commands do not differ from commands in "common" programming languages. The semantics of these commands have already been thoroughly described elsewhere. Therefore their semantics is not described in this section. Formal specification of commands such as assignment (:=), selection (**if then else**), iteration (**loop**) can be found in [Hennessy 1990] or [Tennent 1991].

In the following sections, we concentrate on formally describing the semantics of commands that differ from commands in other languages. Section 4.2.2.1 presents the formal specification of the command **to/from**. Section 4.2 presents the formal specification of a dependable multiparty interaction in DIP. TLA$^+$ is used as the specification language to express the semantics of these DIP commands.

4.2.2.1 Commands to/from in TLA$^+$

The communication between two processes, or roles, in DIP is done via the message sending **to** and the message receiving **from** commands. The behaviour of these two commands depends on the clause **blocks** at the end of the command. In order to specify these commands first we present the definition of a channel [Lamport 1999], which will be used in the specification of **to/from**.

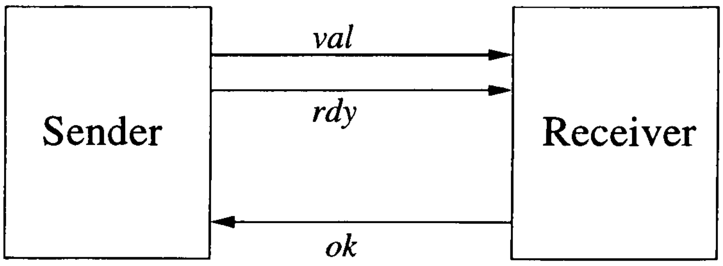

Figure 4.16: Channel between a *sender* and a *receiver*

Figure 4.16 shows how a *sender* and a *receiver* are connected via a *Channel* interface. Data is sent on *val*, and the *rdy* and *ack* lines are used for synchronisation. The sender must wait for an acknowledgement (an *ack*) for one data item before it can send the next data item. The interface uses the handshake protocol, described in the following example:

$$\begin{bmatrix} val = 0 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{Send(66)} \begin{bmatrix} val = 66 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{ack} \begin{bmatrix} val = 66 \\ rdy = 1 \\ ack = 1 \end{bmatrix} \xrightarrow{Send(23)} \begin{bmatrix} val = 23 \\ rdy = 0 \\ ack = 1 \end{bmatrix} \xrightarrow{ack\ldots}$$

Figure 4.17 shows the complete specification of `Channel` in TLA$^+$. The first step in defining a new module in TLA$^+$ is to give the module a name. The second step is to specify which modules are used in this module, e.g. `Channel`

```
------------------------------ MODULE Channel ------------------------------
EXTENDS Naturals
CONSTANT Data
VARIABLES c
TypeInvariant ≜ c ∈ [val : Data, rdy : {0,1}, ack : {0,1}]
----------------------------------------------------------------------------
Init ≜ ∧ TypeInvariant
       ∧ c.rdy = c.ack

Send(msg) ≜ ∧ c.ok = c.ack
            ∧ c' = [val ↦ msg, rdy ↦ 1 - c.rdy, ack ↦ c.ack]

Receive ≜ ∧ c.rdy ≠ c.ack
          ∧ c' = [val ↦ c.val, rdy ↦ rdy, ack ↦ 1 - ack]

Next ≜   (∃ msg ∈ Data : Send(msg)) ∨ Receive

Spec ≜ Init ∧ □[Next]_c ∧ WF_c(Receive) ∧ WF_c(Send(msg))
----------------------------------------------------------------------------
THEOREM   Spec ⇒ □TypeInvariant
============================================================================
```

Figure 4.17: The Specification of a *Channel*

module **EXTENDS** the **Naturals** module[5]. The constant **Data** is a parameter of the specification and can assume any value. The state of a channel is represented as a record with **val**, **rdy**, and **ack** fields. The type invariant asserts that the value of **c** is an element of the set of all such records **r** in which **r.val** is an element of the set **Data** and **r.rdy** and **r.ack** are elements of the set {0,1}. This is defined by the following TLA$^+$ line:

[5]Arithmetic operators like + are not built into TLA$^+$, but are themselves defined in modules. The usual operators on natural numbers are defined in the **Naturals** module. Their definitions are incorporated into the module **Channel** by the statement **EXTENDS Naturals**

TypeInvariant $\triangleq$ c $\in$ [val : Data, rdy : {0,1}, ack : {0,1}]

Initially, c can equal any element of the set of records r whose fields rdy and ack have the same value, so the initial predicate is expressed as:

```
Init ≜ ∧ TypeInvariant
       ∧ c.rdy = c.ack
```

In the next-state action Next, a step of the protocol either sends a value or receives a value. There are two separate actions: Send(msg) that describes the sending of a Data msg; and Receive that describes the receiving of a value. A Next step will either be a Send(msg) step or a Receive step, so it is expressed as:

Next $\triangleq$ ($\exists$msg $\in$ Data : Send(msg)) $\vee$ Receive

The Send(msg) action asserts that the value of the channel c in the end of the Send(msg) step, i.e. c$'$, has val equals to msg, rdy changes its value (to 0 or 1), and ack stays unchanged. The enabling condition of Send(msg) is that the ack and rdy fields are equal, i.e the receiver is ready to accept the next value from the sender, so the Send(msg) action can be expressed as:

```
Send(msg) ≜ ∧ c.ack = c.rdy
            ∧  c' = [val ↦ msg, rdy ↦ 1 - c.rdy, ack ↦ c.ack]
```

The Receive step only takes place when the sender has sent the data, i.e. rdy is different form ack. This step changes the value of ack leaving the values of rdy and val unchanged. The Receive operation is expressed as:

$$\texttt{Receive} \triangleq \begin{array}{l} \land\ \texttt{c.rdy} = \texttt{c.ack} \\ \land\ \texttt{c}' = \texttt{[val} \mapsto \texttt{c.val, rdy} \mapsto \texttt{rdy, ack} \mapsto \texttt{1 - c.ack]} \end{array}$$

The next step is to specify the whole behaviour of a channel as one single formula and not as a set of formulae. This formula is written with the temporal-logic operator $\Box$ (always). The formula $\Box$Next asserts that Next is true for every step in the behaviour. The formula Init, without a $\Box$, is an assertion about the beginning of the behaviour. So, Init $\land$ $\Box$ Next is true of a behaviour if and only if the initial step satisfies Init and every step satisfies $\Box$Next. To allow the channel to be part of a system the specification of channel has to allow stuttering steps. In the channel specification above, Spec has to allow steps that leave the channel c unchanged. The definition of Spec that allows stuttering steps is therefore:

$$\texttt{Spec} \triangleq \texttt{Init} \land \Box\texttt{[Next]}_{\texttt{c}}$$

The above formula specifies what a channel system must *not* do, e.g. the sender cannot change the state of the channel unless the receiver has signalled this by changing the value of ack to 1; or the receiver cannot read a value from the channel unless the sender has signalled it has changed the value of val. These properties are called ***safety properties***. The next step then is to specify that something *does* happen, e.g. once the receiver is ready to accept a value, a value will eventually be sent by the sender. This type of property is called ***liveness property***. In TLA$^+$ liveness properties can be expressed by a conjunct of the form $\texttt{WF}_\texttt{v}\texttt{(A)}$, which asserts that if A ever becomes forever enabled, then an A step must eventually occur. So, the Spec for the channel is expressed as:

$$\texttt{Spec} \triangleq \texttt{Init} \wedge \Box[\texttt{Next}]_{\texttt{c}} \wedge \texttt{WF}_{\texttt{c}}(\texttt{Receive}) \wedge \texttt{WF}_{\texttt{c}}(\texttt{Send(msg)})$$

The last item in the specification of the channel is a temporal formula that is satisfied by every behaviour. This kind of formula is called a THEOREM. In the channel specification, the theorem expresses that a channel c is a record of the format [val:Data, rdy:{0,1}, ack:{0,1}] for any behaviour satisfying Spec.

We will now use the channel specified above in order to describe the semantics of the commands **to/from**. Figure 4.18 shows how two processes, or roles, communicate with each other using the **to/from** command. The sender uses a channel, called to, to send data to the receiver. This data is stored until the receiver is ready to receive the data. The receiver then takes the data via the from channel. Let us recall the functionality of the **to/from** commands:

- **to** without the clause **blocks**: the data is read from channel *to* and is stored in a queue.
- **to** with the clause **blocks**: the data is read from channel *to*, stored in a queue, and the sender is blocked until the receiver is ready to read it from channel *from*.
- **from** without the clause **blocks**: if the queue that is storing data from channel *to* is empty, then empty data is read from channel *from*. If the

queue is not empty, then an element is taken from the queue, written on channel *from*, and read by the receiver.

- **from** with the clause **blocks**: the receiver only reads data from channel *from* when there is data to be read from channel *from*.

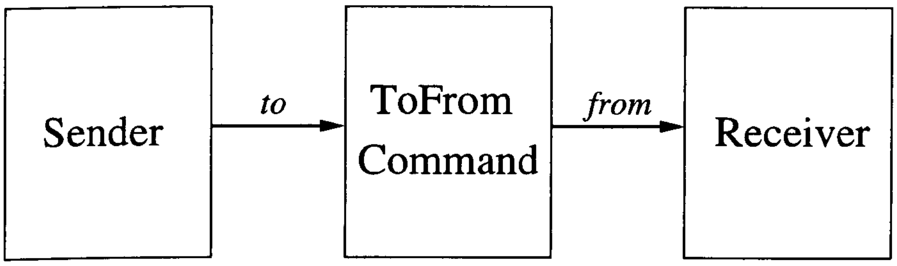

Figure 4.18: **to/from** Command

Figure 4.19 shows the complete specification of the **to/from** commands with and without the clause **blocks**.

The first step in specifying a new module in TLA$^+$ is to give the module a name: `ToFromCommand`. The next step is to declare which modules are going to be used: `EXTENDS Naturals, Sequences, Channel`. The specification of the module `ToFromCommand` continues by declaring the constant `Message`, which represents the set of all messages that can be sent. There are five variables: `to`, `from`, representing the channels, `q` which represents the queue of buffered messages, `msgsnd` which indicates the number of messages that were sent by the sender, `msgrcv` which indicates the number of messages received by the receiver. The value of `q` is the sequence of messages that were sent by the sender but were not yet received by the receiver.

The specification of the `ToFromCommand` module uses the definitions in the `Channel` module to specify operations on the channels `to` and `from`. This

```
──────────────── MODULE ToFromCommand ────────────────
EXTENDS Naturals, Sequences, Channel
CONSTANT Message
VARIABLES to, from, q, msgsnd, msgrcv
Chan(ch) ≜ INSTANCE Channel WITH Data ← Message, c ← ch
──────────────────────────────────────────────────────
Init ≜ ∧ Chan(to)!Init ∧ Chan(from)!Init ∧ q=⟨⟩ ∧ msgsnd=msgrcv
TypeInvariant ≜ ∧ Chan(to)!TypeInvariant
                ∧ Chan(from)!TypeInvariant
                ∧ q ∈ Seq(Message)
                ∧ msgsnd,msgrcv ∈ Naturals
Send(msg) ≜ ∧ Chan(to)!Send(msg)
            ∧ msgsnd' = msgsnd + 1
            ∧ UNCHANGED ⟨from, q, msgrcv⟩
BlockSend ≜ ∧ msgsnd = msgrcv
            ∧ UNCHANGED ⟨to, from, q, msgrcv, msgsnd⟩
TFCRcv ≜ ∧ Chan(to)!Receive
         ∧ q' = Append(q,to.val)
         ∧ UNCHANGED ⟨from, msgrcv, msgsnd⟩
TFCSend ≜ ∧ q ≠ ⟨⟩
          ∧ Chan(from)!Send(Head(q))
          ∧ q' = Tail(q)
          ∧ UNCHANGED ⟨to, msgrcv, msgsnd⟩
Receive ≜ ∧ IF (msgsnd > msgrcv) THEN
                    ∧ Chan(from)!Receive
                    ∧ msgrcv' = msgrcv + 1
            ELSE from' = [val ↦ ε,rdy ↦ from.rdy,ok ↦ from.ok]
          ∧ UNCHANGED ⟨to, q, msgsnd⟩
BlockReceive ≜ ∧ msgsnd > msgrcv
               ∧ UNCHANGED ⟨to, from, q, msgrcv, msgsnd⟩
Next ≜ ∨ ∃ msg ∈ Message : Send(msg) ∨ BlockSend
       ∨ TFCRcv ∨ TFCSend ∨ Receive ∨ BlockReceive

Spec ≜ Init ∧ □[Next]_⟨to,from,q,msgsnd,msgrcv⟩ ∧ WF_from(Receive)
──────────────────────────────────────────────────────
THEOREM Spec ⇒ □TypeInvariant
──────────────────────────────────────────────────────
```

Figure 4.19: TLA$^+$ Specification of **to/from**

requires two instances of that module. Instead of defining these two instances separately, the following parametrised statement is used:

```
Chan(ch) ≜ INSTANCE Channel WITH Data ← Message, c ← ch
```

The above definition allows that for any symbol σ defined in module `Channel` and any expression `ch`, `Chan(ch)!`σ is defined to be equal formula σ with `Message` replaced by `Data` and `ch` replaced by `c`. Then the `Send(msg)` action on channel `to` could be written as `Chan(to)!Send(msg)`.

The initial statement for `ToFromCommand` is a conjunction that initialises both channels using the channels' `Init` action, `q` as an empty sequence of messages, and the counters for messages sent (`msgsnd`) and received (`msgrcv`) as having the same value. The `Init` predicate is defined as:

```
Init ≜ ∧ Chan(to)!Init
       ∧ Chan(from)!Init
       ∧ q = ⟨⟩
       ∧ msgsnd = msgrcv
```

The definition of the type invariant for the `ToFromCommand` module is expressed using the type invariant from the `Channel` module (for either channels), the type of `q` that is a set of finite sequences of messages, and the type of `msgsnd` and `msgrcv` that are Natural numbers. The type invariant for the `ToFromCommand` specification is:

```
TypeInvariant ≜ ∧ Chan(to)!TypeInvariant
                ∧ Chan(from)!TypeInvariant
                ∧ q ∈ Seq(Message)
                ∧ msgsnd,msgrcv ∈ Naturals
```

The `Send(msg)` step writes the message on the channel `to` in the conjunct `Chan(to)!Send(msg)`, and increments the number of messages sent by 1. Subsequently, the message is read from the channel and stored in the sequence `q` by the action `TFCRec`. These two steps represent the command **to** without the clause **blocks**. The **blocks** clause in the command **to** is represented by the step `BlockSend`. This step is only taken when the number of messages sent is equal to the number of messages received, i.e. the receiver has read the last message sent by the sender. Therefore, the expression of the **to** command with the clause **blocks** is the step `Send(msg)` followed by the step `BlockSend`. The definition of the command **to** is defined as:

```
Send(msg) ≜  ∧ Chan(to)!Send(msg)
             ∧ msgsnd′ = msgsnd + 1
             ∧ UNCHANGED ⟨from, q, msgrcv⟩
BlockSend ≜  ∧ msgsnd = msgrcv
             ∧ UNCHANGED ⟨to, from, q, msgrcv, msgsnd⟩
TFCRcv ≜  ∧ Chan(to)!Receive
          ∧ q′ = Append(q,to.val)
          ∧ UNCHANGED ⟨from, msgrcv, msgsnd⟩
```

The definition of the command **from** without the clause **blocks** is represented by the `Receive` action. This action will put an empty (ε) value on the channel `from` if the number of messages sent is not greater than the number of messages received, i.e. there is no message to be read in queue or in the `from` channel. Otherwise, the receiver will read a message from the channel `from` and increment the number of messages received by 1. This increment may enable the step `BlockSend`, hence unblocking the sender. A message is written on the channel `from` by the action `TFCSend` when the sequence `q` is not empty. If the command **to** is used with the clause **blocks**, then first the step BlockReceive has to be taken followed by the step Receive. This sequence will guarantee that there is a value to be read, otherwise the receiver will be blocked until msgsnd > msgrcv. See the definition for the command **from** below:

```
TFCSend ≜ ∧ q ≠ ⟨⟩
          ∧ Chan(from)!Send(Head(q))
          ∧ q' = Tail(q)
          ∧ UNCHANGED ⟨to, msgrcv, msgsnd⟩
Receive ≜ ∧ IF (msgsnd > msgrcv) THEN
                     ∧ Chan(from)!Receive
                     ∧ msgrcv' = msgrcv + 1
            ELSE from' = [val ↦ ε,rdy ↦ from.rdy,ok ↦ from.ok]
          ∧ UNCHANGED ⟨to, q, msgsnd⟩
BlockReceive ≜ ∧ msgsnd > msgrcv
               ∧ UNCHANGED ⟨to, from, q, msgrcv, msgsnd⟩
```

A Next step of the ToFromCommand can: *i*) write a message to channel to in the Send(msg) action; or, *ii*) unblock the sender in the step BlockSend; or, *iii*) read the message from the channel to and store it in the sequence q in the TFCRcv action; or, *iv*) write a message that is stored in sequence q to channel from in TFCSend action; or, *v*) read a message from channel in the Receive action; or *vi*) unblock the receiver in the BlockReceive action. The Next step and the specification for the command **to/from** is defined as:

```
Next ≜ ∨ ∃ msg ∈ Message : Send(msg) ∨ BlockSend
       ∨ TFCRcv ∨ TFCSend ∨ Receive ∨ BlockReceive
```

The Spec for the **to/from** command is given by the following TLA$^+$ formula. The liveness property asserts that if there is a value in the channel from, then eventually this value will be read.

$$\texttt{Spec} \triangleq \texttt{Init} \land \Box[\texttt{Next}]_{\langle \texttt{to},\texttt{from},\texttt{q},\texttt{msgsnd},\texttt{msgrcv} \rangle} \land \texttt{WF}_{\texttt{from}}(\texttt{Receive})$$

4.2.2.2 DMI in TLA$^+$

In the previous section, the semantics of one the basic commands of DIP was formally specified using TLA$^+$. In this section we will present the semantics of a dependable multiparty interaction. A DIP **action** is the command structure that represents a DMI.

Before we start formally describing the semantics of DMI in TLA$^+$, consider the following:

- a DMI is represented by a set of roles that are executed by players. The players send objects to the roles;
- a player has to activate a role in DMI in order to execute the commands inside a role;
- a DMI only starts when all roles of the DMI have been activated, and the guard (boolean expression) at the beginning of the DMI is true;
- the DMI only finishes when all players have finished executing their roles, and the assertion at the end of the DMI (boolean expression) is true (if no exceptions were raised);
- roles can only access data that is sent to them when they are activated, or data that is sent to them by other roles belonging to the same DMI using the **to/from** command specified in Section 4.2.2.1;

- exceptions can be raised during the execution of a DMI. If exceptions are raised, then all roles that have not raised an exception are interrupted, and an exception resolution algorithm is executed when all roles have either raised an exception or have been interrupted.
- if there is a handler to deal with the exception that was decided upon by the exception resolution algorithm, then this handler is activated by all roles;
- if there is no handler to deal with the exception that was decided upon by the exception resolution algorithm, then the exception is raised in the callers of all roles;
- handlers have the same number of roles as the DMI to which they are connected.

In order to formally specify the semantics of a DMI in TLA$^+$, we will use the following sets and predicates:

- `Exceptions`: the set of exceptions handled by the DMI;
- `Commands`: a set of commands;
- `Objects`: a set of objects;
- `Roles`: contains the roles of a DMI. Each element of this set is a record with a field to represent the `state` of the role, a field to represent the `result` of the role after the commands of this role have been executed, a field to store those `commands`, and a field containing the set of `objects` manipulated by the role;

- `Handlers`: the set of handlers for the DMI;
- `GuardExpression(e)`: a predicate representing the execution of the pre-condition of the DMI. The parameter `e` contains the set of all tuples `<p,er,o>`, where `p` represents a player that is enroled to the role `er`, and `o` is the set of objects sent to the role by the player;
- `AssertionExpression(e)`: the same as `GuardExpression(e)` but for the post-condition of the DMI;
- `ExecuteCommands(e)`: execute the commands for the corresponding role;
- `Resolve(enroled)`: execute the exception resolution algorithm for all roles in the DMI. After this algorithm has been executed all roles will produce the same exceptional `result`.

The initial condition for the DMI is that all roles are in a waiting state, both `guard` and `assert` have the value `FALSE`, and the `enroled` and `elements` sets are empty. The `Init` predicate is defined in TLA$^+$ as:

```
Init ≜ ∧ ∀r ∈ Roles : r.state = "wait"
       ∧ guard = FALSE
       ∧ assert = FALSE
       ∧ enroled = {}
       ∧ elements = {}
```

The type invariant specifies that the `guard` and `assert` are `BOOLEAN` variables, the state of a role can can only have one of the values from the set {`"wait","ended","started"`}, and the result of a role can either have a value from the set {`"ok","interrupted"`} or from the set of possible exceptions in `Exceptions`. The type invariant is defined as:

$$
\begin{aligned}
\texttt{TypeInvariant} \triangleq\ & \land\ \texttt{guard, assert} \in \texttt{BOOLEAN} \\
& \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r.state} \in \{\texttt{"wait","ended","started"}\} \\
& \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r.result} \in \{\texttt{"ok","interrupted"}\} \cup \texttt{Exceptions}
\end{aligned}
$$

For a player p to enrole in a role er with a set of objects o, it has to execute the action `Enrole(p,er,o)`. This step is only enabled if role er belongs to the set of `Roles` in the DMI and no other player has enroled to such a role. This is expressed by the two first conjuncts of the following TLA$^+$ formula. If this step is enabled, then the role er is added to the `enroled` set and the tuple <p, r, o> is added to the `elements` set. The `Enrole(p,er,o)` action is defined in TLA$^+$ as:

$$
\begin{aligned}
\texttt{Enrole(p,er,o)} \triangleq\ & \land\ \exists \texttt{r}_1 \in \texttt{Roles} : \texttt{r}_1 = \texttt{er} \\
& \land\ \forall \texttt{r}_2 \in \texttt{Enroled} : \texttt{r}_2 \neq \texttt{er} \\
& \land\ \texttt{enroled}' = \texttt{enroled} \cup \{\texttt{er}\} \\
& \land\ \texttt{elements}' = \texttt{elements} \cup \{\langle \texttt{p,er,o} \rangle\} \\
& \land\ \texttt{UNCHANGED}\ \langle \texttt{guard, assert} \rangle
\end{aligned}
$$

The DMI only begins if all roles have a player enroled to and the precondition is true. The testing of the guard with all players enroled is defined by the following two TLA$^+$ conjunctions:

$$
\begin{aligned}
\texttt{Guard} \triangleq\ & \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r} \in \texttt{enroled} \\
& \land\ \texttt{guard}' = \texttt{GuardExpression(elements)} \\
& \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert} \rangle \\
\texttt{Begin} \triangleq\ & \land\ \texttt{guard} = \texttt{TRUE} \\
& \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r.state}' = \texttt{"started"} \\
& \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert, guard} \rangle
\end{aligned}
$$

The execution of all roles is defined in the action `ExecuteRoles`. This step is only enabled if all roles have `state = "started"`. If enabled, then the result of the execution of the set of commands of a role is stored in the field `result`. The `ExecuteRoles` is defined in TLA$^+$ as:

```
ExecuteRoles ≜ ∧ ∀r ∈ Roles : r.state = "started"
               ∧ ∀<p,r,o> ∈ elements : r.result′ = ExecuteCommands(<p,r,o>)
               ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
```

When all roles have executed their commands without raising an exception, i.e. their state is equal to "ok", the post-condition expression can be tested. The `assert` variable changes its value based on the execution of the `AssertionExpression(elements)` action. The post-condition of a DMI is defined as:

```
Assertion ≜ ∧ ∀r ∈ Roles : r.result = "ok"
            ∧ assert′ = AssertionExpression(elements)
            ∧ UNCHANGED ⟨enroled, elements, guard⟩
```

If no exceptions were raised, then the normal termination of a DMI is defined in the `NormalEnd` action. The condition that enables this step is `assert = TRUE`, i.e. the post-condition was passed. This step changes the state of all roles to "wait", meaning that the roles are ready to be executed again. The sets `enroled` and `elements` are emptied. The TLA$^+$ definition of `NormalEnd` is:

```
NormalEnd ≜ ∧ assert = TRUE
            ∧ ∀r ∈ Roles : r.state′ = "wait"
            ∧ enroled′  = ⟨ ⟩
            ∧ elements′ = ⟨ ⟩
            ∧ assert′ = FALSE
            ∧ guard′ = FALSE
```

```
─────────────────────── MODULE DMI ───────────────────────
EXTENDS Naturals, Sequences
VARIABLES enroled, elements, guard, assert
──────────────────────────────────────────────────────────
Init ≜ ∧ ∀r ∈ Roles : r.state = "wait"
       ∧ guard = FALSE
       ∧ assert = FALSE
       ∧ enroled = {}
       ∧ elements = {}
TypeInvariant ≜ ∧ guard, assert ∈ BOOLEAN
          ∧ ∀r ∈ Roles : r.state ∈ {"wait","ended","started"}
          ∧ ∀r ∈ Roles : r.result ∈ {"ok","interrupted"} ∪ Exceptions
Enrole(p,er,o) ≜ ∧ ∃r1 ∈ Roles : r1 = er
                 ∧ ∀r2 ∈ enroled : r2 ≠ er
                 ∧ enroled' = enroled ∪ {er}
                 ∧ elements' = elements ∪ {<p,er,o>}
                 ∧ UNCHANGED ⟨guard, assert⟩
Guard ≜ ∧ ∀r ∈ Roles : r ∈ enroled
        ∧ guard' = GuardExpression(elements)
        ∧ UNCHANGED ⟨enroled, elements, assert⟩
Begin ≜ ∧ guard = TRUE
        ∧ ∀r ∈ Roles : r.state' = "started"
        ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
ExecuteRoles ≜ ∧ ∀r ∈ Roles : r.state = "started"
               ∧ ∀<p,r,o> ∈ elements : r.result' = ExecuteCommands(<p,r,o>)
               ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
Assertion ≜ ∧ ∀r ∈ Roles : r.result = "ok"
            ∧ assert' = AssertionExpression(elements)
            ∧ UNCHANGED ⟨enroled, elements, guard⟩
NormalEnd ≜ ∧ assert = TRUE
            ∧ ∀r ∈ Roles : r.state = "ended"
            ∧ ∀r ∈ Roles : r.state' = "wait"
            ∧ enroled' = ⟨ ⟩
            ∧ elements' = ⟨ ⟩
            ∧ assert' = FALSE
            ∧ guard' = FALSE
──────────────────────────────────────────────────────────
```

Figure 4.20: TLA$^+$ Specification of a DMI (part 1)

Figure 4.20 shows the complete first part of the formal semantics of a DMI. In the figure all conjunctions are related to the normal execution of a DMI. In Figure 4.21, we define the formal semantics for the steps that are taken in case of one or more exceptions being raised. An exception can be raised during the execution of the set of commands of a role in the `ExecuteCommands` action.

The activation of a handler depends on the state of the roles. A handler is only activated when all roles have the same value for their `result`, and there exists a handler for the exception resolved by the resolution algorithm. The activation of a handler is defined as:

$$\begin{aligned}\texttt{ActivateHandler} \triangleq\ & \land\ \forall \texttt{r}_1,\texttt{r}_2 \in \texttt{Roles} : (\texttt{r}_1\texttt{.result} = \texttt{r}_2\texttt{.result})\\ & \land\ \exists \texttt{h} \in \texttt{Handlers} : (\exists \texttt{r} \in \texttt{Roles} : \texttt{r.result} \in \texttt{h.Exc})\\ & \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert, guard} \rangle\end{aligned}$$

The resolution algorithm on the other hand, is activated once all roles have raised an exception, i.e. their `result` belongs to the set `Exceptions`, or have been interrupted. The state of all roles has to be different from `"ok"`. This action is defined as:

$$\begin{aligned}\texttt{ExceptionResolution} \triangleq\ & \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r.result} \neq \texttt{"ok"}\\ & \land\ \texttt{Resolve(enroled)}\\ & \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert, guard} \rangle\end{aligned}$$

When a role terminates by raising an exception, then all other roles have to be interrupted, causing the exception resolution algorithm to be enabled. The step that represents the interruption of roles is `InterruptRoles`. This

```
InterruptRoles ≜  ∧ ∃r1 ∈ Roles : r1.result ∈ Exceptions
                  ∧ ∀r2 ∈ Roles : IF r2.result ∉ Exceptions
                                  THEN r2.result' = "interrupted"
                                  ELSE r2.result' = r2.result
                  ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩

ExceptionResolution ≜  ∧ ∀r ∈ Roles : r.result ≠ "ok"
                       ∧ Resolve(enroled)
                       ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩

ActivateHandler ≜  ∧ ∀r1,r2 ∈ Roles : (r1.result = r2.result)
                   ∧ ∃h ∈ Handlers : (∃r ∈ Roles : r.result ∈ h.Exc)
                   ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩

ExceptionalEnd ≜   ∧ ∀r1,r2 ∈ Roles : r1.result = r2.result
                   ∧ ¬∃h ∈ Handlers : (∃r ∈ Roles : h.exception ≠ r.result)
                   ∧ ∀r ∈ Roles : r.state = "ended"
                   ∧ ∀r ∈ Roles : r.state' = "wait"
                   ∧ enroled' = ⟨ ⟩
                   ∧ elements' = ⟨ ⟩
                   ∧ assert' = FALSE
                   ∧ guard' = FALSE

Next ≜  ∨ ∃p ∈ Players : (∃er ∈ Roles : (∃o ∈ Objects : Enrole(p,er,o)))
        ∨ Guard ∨ Begin ∨ ExecuteRoles ∨ Assertion ∨ End
        ∨ InterruptRoles ∨ ActivateHandler

Spec ≜ Init ∧ □[Next]⟨enroled, elements, assert, guard⟩

THEOREM Spec ⇒ □TypeInvariant
```

Figure 4.21: TLA$^+$ Specification of a DMI (part 2)

step is enabled when at least one of the roles has raised an exception. The raising of an exception is represented in the value that the role's `result` assumes. If the value belongs to the set of `Exceptions`, then the `InterruptRoles` action is enabled. The step will then set the state of all roles, which did not raise an exception, to "interrupted". Even if a role has terminated it will be interrupted when another role raises an exception. The InterruptRoles actions is defined in TLA$^+$ as:

$$
\begin{array}{lll}
\texttt{InterruptRoles} \triangleq & \land\ \exists r_1 \in \texttt{Roles} : & r_1\texttt{.result} \in \texttt{Exceptions} \\
 & \land\ \forall r_2 \in \texttt{Roles} : & \texttt{IF } r_2\texttt{.result} \notin \texttt{Exceptions} \\
 & & \texttt{THEN } r_2\texttt{.result}' = \texttt{"interrupted"} \\
 & & \texttt{ELSE } r_2\texttt{.result}' = r_2\texttt{.result} \\
 & \land\ \texttt{UNCHANGED } \langle \texttt{enroled, elements, assert, guard} \rangle &
\end{array}
$$

4.2.3 Discussion

The semantics described in Section 4.2.2.2 deal with the basic rules of a DMI, i.e. pre and post-synchronisation, roles activation, exception handling, and roles interruption. Furthermore, we showed, in Section 4.2.2.1, how the semantics of two of the DIP commands could be expressed in TLA$^+$. Although we did not provide formal semantics to all DIP commands, the specification for those two commands show how the semantics of other DIP commands could be specified.

We did not attempt to describe formally the semantics of the execution of the role's commands. The way external objects guarantee ACID properties is also not described (a formal description of the ACID properties can be found in [Lynch *et al.* 1994]). In [Schwier *et al.* 1997], for example, formal description of enclosure properties for a mechanism similar to the DMI (CA action [Xu *et al.* 1995]) is given in temporal logic.

Chapter 5

Implementation of DMIs

In the previous chapter, the DIP language, which supports dependable multiparty interaction as a basic construct, was presented. Although one can design, or implement, systems using just DIP, there are a lot of programming languages that are available for designing and implementing computing systems. Those systems may contain several multiparty interactions during their execution. Usually, those multiparty interactions are scattered throughout the programming code of the system. In this chapter, we present several different approaches for organizing multiparty interactions and implementing DMIs in object-oriented languages. Section 5.1 presents different ways of organizing objects in order to build DMIs. Section 5.2 presents a complete framework implemented in Java, with a set of programming techniques, that provides the necessary tools for constructing DMIs.

In Chapter 6, we show how the API that will be presented in this chapter can be used in the implementation of a control system for an industrial production cell. Appendix E shows a complete example of how to instantiate objects, and how to implement application code by extending the classes of the API.

5.1 Object-Oriented Architectures for DMIs

DMIs can be implemented in a number of ways. In this section we discuss alternative ways of providing a distributed implementation of DMIs in object-oriented languages, based on that given in [Zorzo 1999]. First of all, the various components that can be used to implement a DMI are described. Basically, a DMI is composed of:

- *Manager*: one component for controlling all protocols inside a DMI, such as: pre and post-synchronisation of processes that participate in the DMI; test of pre and post-condition of the DMI; exception handling between the processes; keeping control of internal and external data to the DMI; etc.;
- *Roles*: several segments of application code, each of them being executed by a process that participates in the DMI;
- *External data*: data that is external to the processes that are participating in the DMI. This kind of data might be accessed by other processes that are not participating in the DMI. Hence a special access control is required;
- *Local data*: data that is local to the DMI. Processes that are not participating in the DMI do not have access to this data.

In object-oriented languages, several possibilities for implementing roles and managers can be devised, e.g. roles as separate objects or as member functions of a manager object. Local and external data are represented as objects. The way they are structured and used in a DMI is described in Sections 5.1.1 and 5.1.2. Four possible architectures for implementing DMIs in object-oriented languages are described in Sections 5.1.3, 5.1.4, 5.1.5, and 5.1.6.

5.1.1 External Data as Atomic Objects

External data can be represented by external objects in all architectures that will be presented in the next sections. External objects are used in a DMI to get access to the state of an application while the DMI is in progress. Because these objects can be seen by other DMIs at the same time, they are provided with some type of transactional semantics in order to avoid wrong (temporary/incomplete) information being used before the DMI has finished. One way of providing these objects with transactional semantics is by using an existing object-oriented transactional system, for example, the Arjuna system [Shrivastava *et al.* 1991]. A simple mechanism that provides only mutual exclusion access to these objects could be provided, but this could restrict the types of applications that could use the DMIs.

5.1.2 Local Data as Objects

Local data can be represented by local shared objects. These objects are used by the roles in order to exchange information with each other. They are used only inside a DMI and their values are discarded after the DMI has finished. The roles in a DMI may also use private local objects. These objects are not used concurrently, so it is the responsibility of the role to take care of them.

5.1.3 A Manager-and-roles Object

The simplest way of organizing a DMI is to centralise everything in the same object, except the external objects. Figure 5.1 shows an example UML class diagram [Muller 1997] that represents this architecture. A DMI is represented by an abstract class that has to be extended by a new class (DMIExample in the figure). This new class has to implement at least the public manager method, and some private methods, which represent the roles of the DMI. The manager method is responsible for executing the protocols of a DMI, e.g. synchronization of participants, and giving access to the role methods (firstRole, secondRole, and thirdRole in the figure) of the extended class. Lists of input and output parameters are sent to the DMI via the manager method. The manager method is also informed about the role a participant wants to execute. The protocols that the manager method has to execute are implemented as protected methods in the DMI abstract class. Shared local objects are represented as private objects of the new extended class, e.g. SharedLocalObject object in the figure. This private object is used by all role methods for communication and cooperation. External objects have to extend some class that implements transactional semantics. Several external objects may be referenced (accessed) by a DMI. Implementations of similar approaches can be found in [Romanovsky & Zorzo 1999] and [Wellings & Burns 1996].

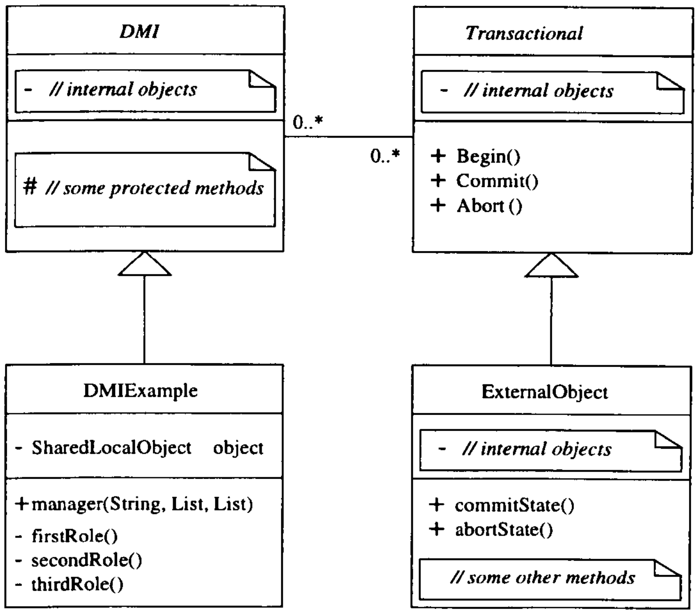

Figure 5.1: A Manager-and-roles Object (UML)

5.1.4 A Manager Object and Role Objects

The architecture with one manager and several role objects is a natural way of distributing DMIs. It allows information to be processed in the location where it is produced (in the roles) avoiding the overloading of a single host. Figure 5.2 shows an example UML class diagram that represents this architecture. The manager is represented by a class that contains the implementation of all protocols of a DMI, e.g. synchronisation of participants. The manager has references to all external and shared local objects in the DMI. The manager also controls the roles of the DMI, i.e. access to a role is through the manager using the execute method. When this method is called, the participant has to inform the manager what the input and out-

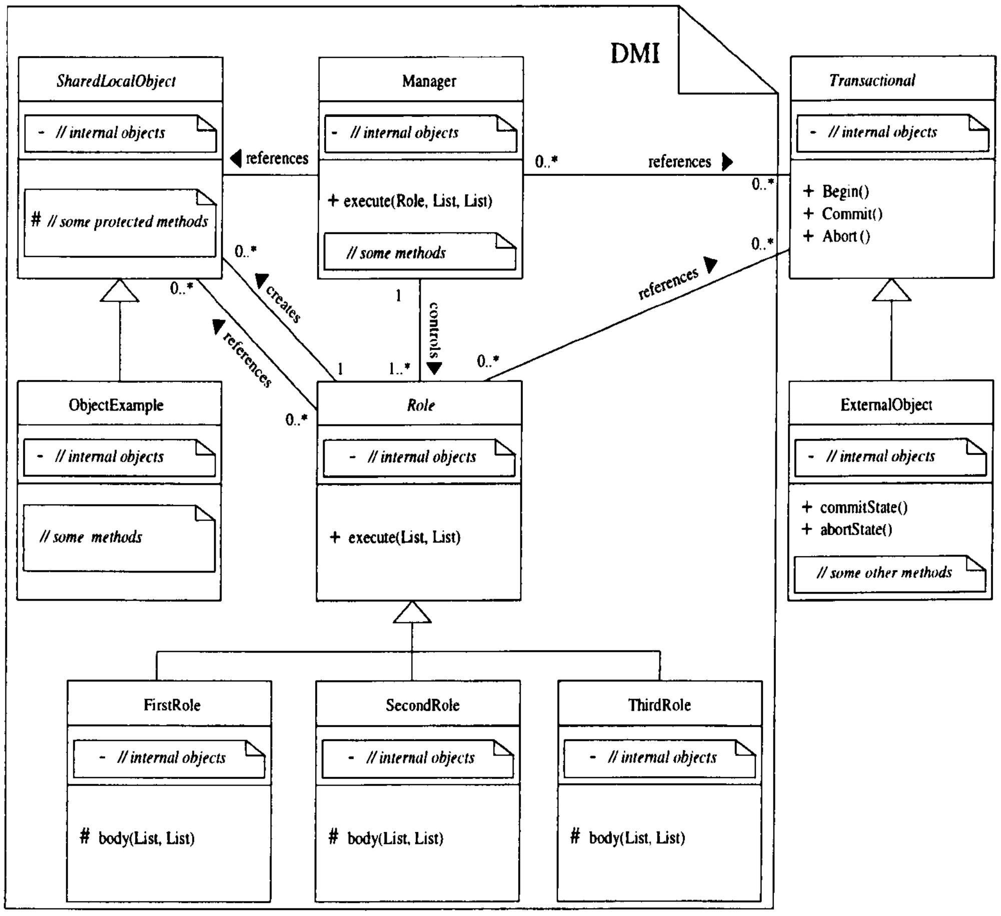

Figure 5.2: A Manager Object and Role Objects (UML)

put parameters are, and which role the participant is intending to execute. Roles are represented in this architecture as abstract classes that have to be extended by new classes that implement at least the protected method **body**. This method contains the application code for the role. Roles can create or reference shared local objects. External objects can also be referenced by roles. As shown in the figure, a DMI is represented by the manager, roles, and shared local objects classes. Implementations of similar approaches can be found in [Mitchell *et.al.* 1998] and [Zorzo *et al.* 1999].

5.1.5 Manager-and-role Objects

In this architecture, manager and role are implemented in the same object. The DMI is composed of several of these manager-and-role objects. The manager part in each object is responsible for maintaining contact with the other manager parts of the other roles in order to execute the main function of the DMIs, e.g. synchronisation, exception handling. Figure 5.3 shows an example UML class diagram that represents this architecture.

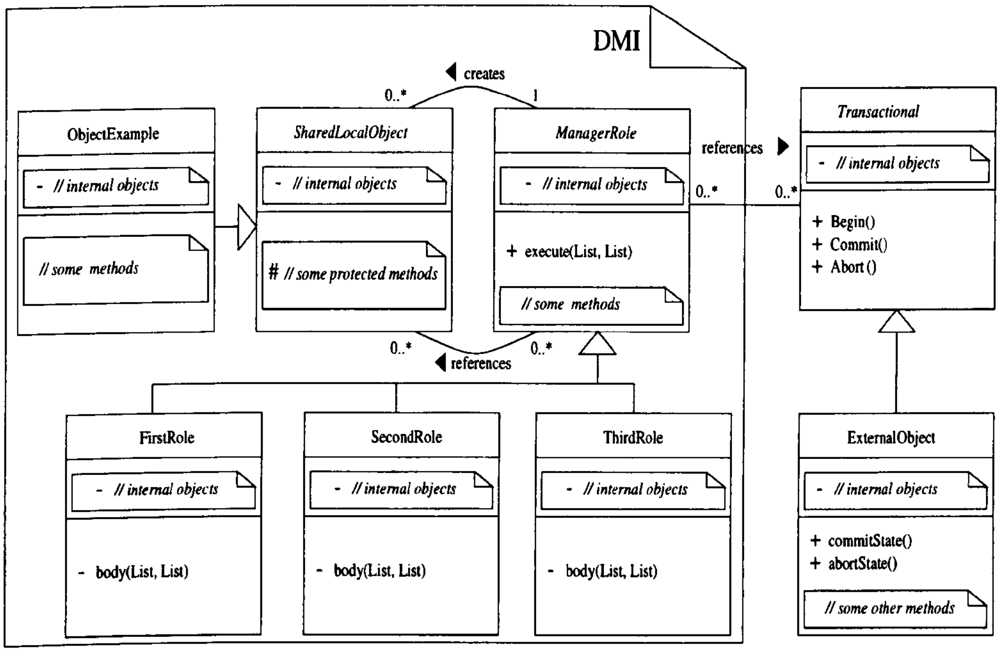

Figure 5.3: Manager-and-role Objects (UML)

5.1.6 Manager Objects and Role Objects

A fourth way of distributing parts of a DMI is shown in Figure 5.4. In this approach there are several managers and several roles. Each manager is responsible for controlling the access to one role, and for executing protocols together with all the others managers, e.g. the handling of possible exceptions that may be raised during the execution of one, or more, roles of the DMI. Each role is controlled by exactly one manager. The complete description and implementation of this architecture is presented in Section 5.2 (a short description can also be found in [Zorzo & Stroud 1999]).

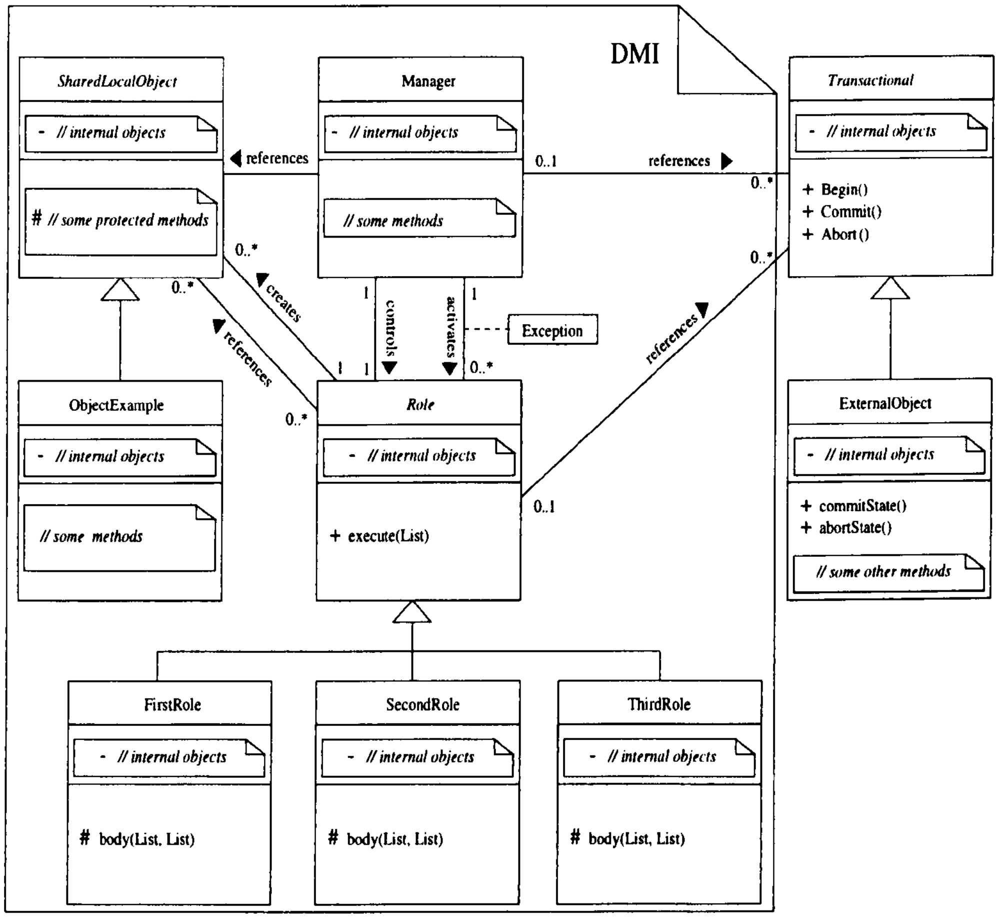

Figure 5.4: Manager Objects and Role Objects (UML)

5.1.7 Discussion

A brief comparison of the four architectures presented above shows that: the architecture in Section 5.1.3 is the simplest to implement, but may cause the overloading of one host; the architecture in Section 5.1.4 leaves the processing of application code to be executed on the node it is produced, but has the inconvenience of having a single point of failure (the manager); the architecture in Section 5.1.5 avoids the problem of the single point of failure and the overloading of a single host, but does not allow the separation of application code and the control of the interaction; the architecture in Section 5.1.6 may be the most difficult to implement but avoids all the problems the other architectures have, i.e. single point failure, overloading of a single host, and lack of separation of application code and interaction control. The next section describes in detail a framework that implements the architecture in Section 5.1.6.

5.2 A Complete Object-Oriented Framework for DMI

Chapter 4 showed how multiparty interactions could be linked together to implement DMIs. In this section a generic object-oriented framework for implementing DMIs is described. This framework is composed of four types of distributed objects (as described in Section 5.1): *roles*, *managers*, *shared local objects*, and *external objects*. Each of the these objects can potentially be distributed on a different host. Each DMI is represented by several sets of these remote objects: one set for the interaction when there is no failure, i.e. *basic interaction*, and several sets for dealing with exceptions that may be raised during the execution of the interaction (either during the basic interaction or during an *exception handling interaction*).

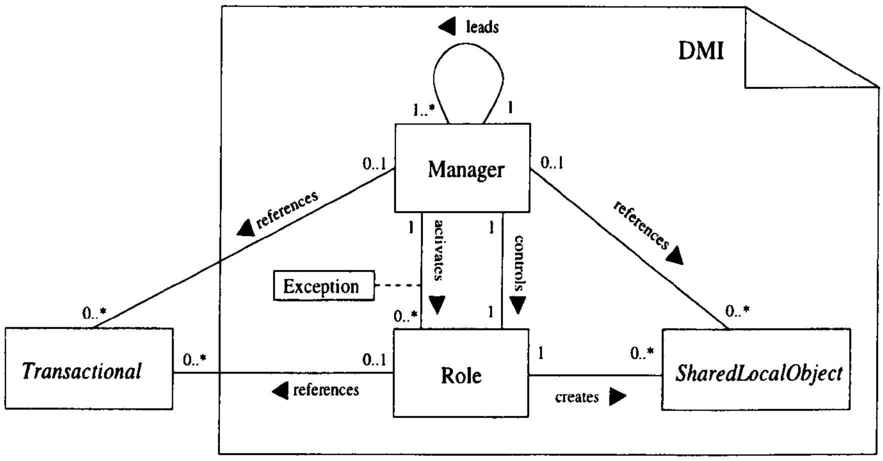

Figure 5.5: Class Diagram (UML)

Figure 5.5 shows a reduced UML class diagram version of Figure 5.4 that represents our API. The figure shows that each role is associated with only one manager, and each manager controls only one role. Managers are associated with a special manager, which is called the leader manager. The leader manager is the only manager that is associated with shared local objects. Shared local objects are created by roles, which (eventually) export them to the leader manager and thus make them accessible by the other roles. External objects are associated with both managers and roles. Managers will keep track of these objects for possible recovery, while the roles will use them. As shown in the figure, there is no class to represent either a basic multiparty interaction or a DMI in the framework. A DMI itself will be built using a sequence of programming steps to glue the components of the framework when they are being instantiated (created). A multiparty interaction consists of a set of managers linked together via a leader manager (represented by the **leads** association). Multiparty interactions are connected together in order to create a DMI via the **activates** association. A set of managers, in a multiparty interaction, will activate the appropriate set of roles for dealing with a raised exception. This new set of roles will be controlled by a different set of managers.

To program a new DMI using the framework, the first step is to define a new class that extends the Role class for each party in the interaction. The extended Role class should redefine at least one method: the body method. This method will contain the set of operations that will be executed by the participant that activates the role. Upon creation each Role has to be informed about the manager that will be managing this role. A manager that 'controls' a Role object is an instance of the Manager class. The Manager class provides a basis for coordinating the participants in a multiparty interaction. Separation of the manager from the roles allows the application code of the role to be distributed to the host where the application is created. This strategy will help in avoiding the overloading of one host with the control of the DMI and the application code. Furthermore, in the case of the host of the application role crashing, the manager can still run and recover, together with the other managers, from this crash.

The managers of all roles will compose the controlling body of the interaction. Each manager upon creation is informed of which manager will act as the *leader* in the interaction. The leader is responsible for controlling protocols for synchronisation between managers, for the exception resolution algorithm, and for keeping information about the shared local objects. Every manager is a potential leader in the framework, avoiding a possible single failure point, if the host of the leader crashes.

The framework described here is implemented using the Java language and its RMI ORB to distribute the objects of a DMI. (Other mechanisms, such as CORBA could have been used.)

5.2.1 Managers

The Manager class is the major class in our framework. Each role has to be managed by a different manager object. When instantiating a Manager object, the manager has to be informed of its name, the name of the interaction, the leader of the interaction (a manager without a leader is its own leader), and a list of exceptions that will be treated by the manager. Each exception in the list is associated with a role from an exception handling interaction. This new role, which is controlled by a different manager, will be called to treat that exception when it is raised in all roles of the interaction. The list of exceptions attached to the managers is the link between the multiparty interactions of a DMI. Exceptions that are raised in a multiparty interactions whose managers do not treat exceptions are propagated to the enclosing context. Figure 5.6 shows how to instantiate a set of managers for an interaction.

```
Manager mgr1 = new Manager("mgr1-name","DMI-name",eh1, null);
Manager mgr2 = new Manager("mgr2-name","DMI-name",eh2, mgr1);
```

Figure 5.6: Manager Objects Instantiation

There are four ways of creating a manager object: creating a leader manager object that handles exceptions; creating a leader manager object that does not handle exceptions; creating a manager object that handles exceptions and is led by a leader; or, creating a manager object that does not handle exceptions and is led by a leader. In Figure 5.6, two managers were created for the same DMI with mgrl acting as the leader of the DMI and mgr2 being led by mgrl). The ehl hashtable contains the list of exceptions that are treated by mgrl and the roles that are activated in the case of one of the exceptions that are in the list being raised. All hashtables of managers from the same DMI must contain the same list of exceptions to be handled. If the hashtables do not contain the same list of exceptions, then an exception is raised at creation time.

Each manager is activated when an *external thread* calls its starting method. This starting method creates a new *internal thread*, that will execute a the sequence of operations, shown in Figure 5.7, as the main execution body of the manager. The thread that activated the manager, i.e. the external thread, waits in the starting method until the internal thread has (been) terminated, or until the DMI that encloses the DMI that the external thread has just activated interrupts it due to an exception in one of the roles in the enclosing DMI. If the external thread is interrupted by the enclosing DMI, then it interrupts the internal thread and waits until the internal thread has terminated. The internal thread can terminate with one of the following results: an exception that was not dealt with inside the DMI; a normal termination, i.e. no exceptions were raised, or exceptions were raised inside the DMI but were handled properly; or, interruption by the external thread. An internal thread can only be interrupted by the external thread if the DMI has not started, i.e. the internal thread is still waiting for the other participants of the DMI to synchronise at the entry point.

```
try {
    synchroniseBegin(); // synchronise upon entry
    if (!roleManaged.preCondition(listOfParameters))
      throw new PreConditionException();
    roleManaged.bodyExecute(this,listOfParameters); // execute the role
    synchroniseEnd(); // wait everyone to finish before checking post-conditions
    if (!roleManaged.postCondition(listOfParameters))
      throw new PostConditionException();
    synchroniseEnd(); // really exit synchronously
} catch (Exception e) {
  // exception resolution part, call exception resolution algorithm
  // and activate role associated with the exception (if there is one)
}
```

Figure 5.7: Main Execution Code for a Manager Object

In Figure 5.7, the first activity the manager, i.e. the internal thread of the manager, executes is to synchronise itself with all the other managers in the interaction by calling the **synchroniseBegin** method. Once the leader has informed this manager that all the managers have been synchronised, the manager checks whether the role precondition is valid. The **preCondition** method receives all the objects that will be passed to the role managed by this manager as parameters. If the precondition is not satisfied, then a **PreConditionException** is thrown. The role is executed by the manager by calling the **bodyExecute** method of the role. After the role has finished its execution, the manager synchronises with all the other managers before testing its post-condition. If the post-condition is passed (accepted), then the manager synchronises with all the other managers and the interaction is finished. This sequence of steps is executed if all the activities of all the participants of an interaction finish without raising any exception. If an exception is raised during the execution of a role, the manager will catch this exception in the catch(Exception e) block. In this situation, the manager calls an exception resolution algorithm, and after receiving the exception it has to handle, the manager activates the role in the exception handling interaction that deals with this exception. If there is no exception handling interaction for the exception it has to handle, then this same exception is thrown by all managers in the DMI to the callers of the roles.

5.2.2 Roles

After a new Manager object has been created, the programmer of the multiparty interaction has to create a role object that will be controlled by that manager. This role object has to be an instance of the new role class derived from the Role class provided by the framework. Each new class derived from Role contains the main code for one of the roles that compose the multiparty interaction. Only objects whose type is derived from Role can belong to a multiparty interaction. When deriving a new class from the Role class, the programmer should implement at least one method: the body method, which will contain the main application code of that role. This method does not return any value. It receives a list of external objects as parameter (see Figure 5.8). If an exception is raised inside that role, then that exception can be handled locally by the role if the exception does not affect other roles. If the raised exception has any effect on the other threads, it must be thrown to the manager of that role which will notify the leader and interrupt all roles in the DMI. After all roles were interrupted the leader resolves which exception handling interaction will deal with that exception (or some common exception if more than one role raises an exception simultaneously, as it will be explained in Section 5.2.4).

```
public class RoleName extends Role {
  SharedObject so; // shared object
  public RoleName(Manager manager, String roleName) {
    super (manager, roleName);          // set role with name and manager
    so = new SharedObject();            // creates a shared object
    manager.sharedObject("soName", so); // export shared object
  }
  protected void body(Transactional list[]) throws Exception {
    // code for the body of the RoleName
  }
}
```

Figure 5.8: Extending Role Class

Extensions of the Role class are also responsible for declaring the shared local objects used for coordinating the roles within a particular interaction, and for checking part of the pre and post-conditions of the interaction. After the shared local objects have been created, the role must inform its manager about those objects using the sharedObject method. This will export the shared local objects, making it possible for other roles in this interaction to use them later. The pre and post-conditions of an interaction can be checked in a distributed way; each role checks part of the conditions, or one role could be delegated to check the whole pre and post-condition of the interaction. The delegation could be achieved by using shared local objects between the roles. The methods in which the test of pre and post-conditions are programmed are called: preCondition and postCondition. These methods may be redefined in the new role class. Figure 5.8 shows the extension of the Role class, and how shared local objects are created and exported by this new role.

Roles are distributed objects in the framework and provide a user with the following public methods: execute and bodyExecute. When the execute method is called, the role passes control to its manager which will execute the bodyExecute method of this role. The bodyExecute method takes the manager descriptor as its parameter. This descriptor is checked against the manager descriptor that the role was created with. This guarantees that only the manager of this role can execute its main body (body method of the new role class).

5.2.3 External, Local and Shared Objects

Two other classes are provided by the framework. The ExternalObject class which implements an interface called Transactional. The Manager class expects objects from such a class to implement the Transactional interface. The Transactional interface defines the following public methods: begin, commit, and abort. The ExternalObject class provides a basic implementation for the Transactional interface. Any new class extended from ExternalObject class is provided with this basic implementation of the Transactional interface but must provide its own definitions of commitState and abort-

State methods. The framework provides its own simple transactional system, but this could easily be replaced by an existing one like the Arjuna system [Shrivastava *et al.* 1991]. External objects are passed to the multiparty interaction via input parameters when activating a role.

The SharedLocalObject class represents the objects used by roles (in order) to exchange information with each other. These objects are used only inside the interaction and their values are discarded after the interaction has finished, either when the interaction terminates normally, or when an exception is raised by one of the roles. Figure 5.9 shows how a role can get a reference to a shared local object that was exported by another role (see Figure 5.8 for the exporting of a shared local object). In the body method of the role, the manager of that role is asked about an object called "soName" using the getSharedObject method. If the object exists then a reference to that object is returned, otherwise a null is returned.

```
SharedObject so = (SharedObject) manager.getSharedObject("soName");
```

Figure 5.9: Getting Reference to Shared Objects

Shared local objects are remote objects in the framework, so there is a chance that these objects can be accessed by adjacent interactions (e.g. parent or sibling interactions). However, even though shared local objects can be seen by other interactions, only threads that are executing the interaction (or roles belonging to that interaction) should be able to access them. To ensure these semantics, every time a thread starts to execute a role of a DMI, the managers inform the shared local objects about the threads that are authorized to access them. It is possible to perform an internal check because shared local objects are bound to particular "instances" of interactions at object creation time.

The roles in a multiparty interaction may also use private local objects. These objects are not used concurrently, so it is the responsibility of the role to take care of them. If any kind of recovery is necessary they have to be recovered by their owner. If the role that created local objects cannot recover them, then an exception should be raised.

5.2.4 Exception Handling

By default, the Manager class provides a built-in exception resolution mechanism based on [Romanovsky *et al.* 1996]. This mechanism works as follows. When a role raises an exception, its corresponding manager is notified of that exception. The manager then informs the leader which interrupts all roles that have not raised an exception. After all roles have been interrupted or have notified the leader manager of an exception (exceptions can be raised concurrently), an exception resolution algorithm is executed by the leader. This algorithm tries to find a common ancestor[1] exception from all raised exceptions. When such an exception is found, the leader informs all managers about that exception and an exception handling interaction is activated (in the same way a complete new set of managers and roles would be activated) using the exception handlers list which the manager was initialized with. If there is no interaction handler for that exception, a handler for a higher level

[1] A common exception of which all raised exceptions are subtypes. In the worst case scenario, the common exception is Exception.

exception is tried until the top level exception is reached, i.e. Exception class. If there is no handler even for Exception, then the exception is signalled to the enclosing context.

In the event of one of the managers or one of the roles crashing, the managers communicate with each other and decide to raise a CrashedManagerException or a CrashedRoleException exception. If the manager that has crashed was the leader, then a new leader may be chosen by the managers that are still running. If a CrashedManagerException or a CrashedRoleException are raised, then these exceptions are signalled to the callers of the DMI.

If users of the framework want to provide their own algorithm for deciding which exception is to be handled by all the roles, then the Manager class can be extended and a method called exceptionResolution must be provided. This method must return an exception that is derived from the Exception class. A list containing the exceptions that were raised by the roles is passed to the new exception resolution method.

Managers can also be interrupted by an enclosing DMI. In such a situation, the manager informs the leader it has been interrupted by the enclosing DMI and interrupts the thread that is executing the role. If none of the roles of the nested DMI raises an exception, i.e. all of them are interrupted, then an InterruptException is signalled and raised in the enclosing DMI.

5.3 Framework Performance

Figure 5.10 shows the costs of using the dependable multiparty interactions in the implementation of a system. The figure shows the cost of an empty

DMI and of a DMI where exceptions are raised by all roles of that interaction. In the graph, these two executions of the framework are compared with a simple multiway rendezvous[2] mechanism. Even though the framework adds an overhead to the application interaction, it benefits from the inclusion of features that help the programmer to enclose failures and abstract the interactions from the objects. Notice that an increase in the number of participants does not cause the implosion of the framework, which scales in the same way as a simple multiway rendezvous mechanism.

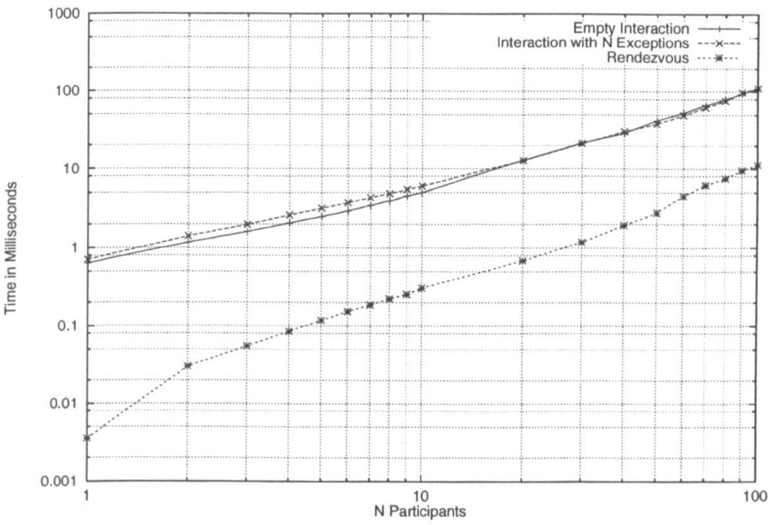

Figure 5.10: DMIs vs. Rendezvous

Furthermore, the overheads associated with using our framework are measured in microseconds whereas network and device overheads are measured in milliseconds, so the cost of using our framework is negligible for distributed applications such as the production cell described in Section 2.3.1. For example, the overhead of our API for an interaction involving two parties is

[2]We created a new class that provides only one Java synchronized method which blocks the first callers until it is called by the last caller (second caller for a 2-party rendezvous, third caller for a 3-party rendezvous, ...).

of around 1,100 microseconds (see Figure 5.11), while an interaction involving the robot's operations to unload the table, as described in Section 2.3.1, would take around 6,700 milliseconds, therefore the overhead of our API over the total time of the unloading of the table by the robot is less than 0.02 percent.

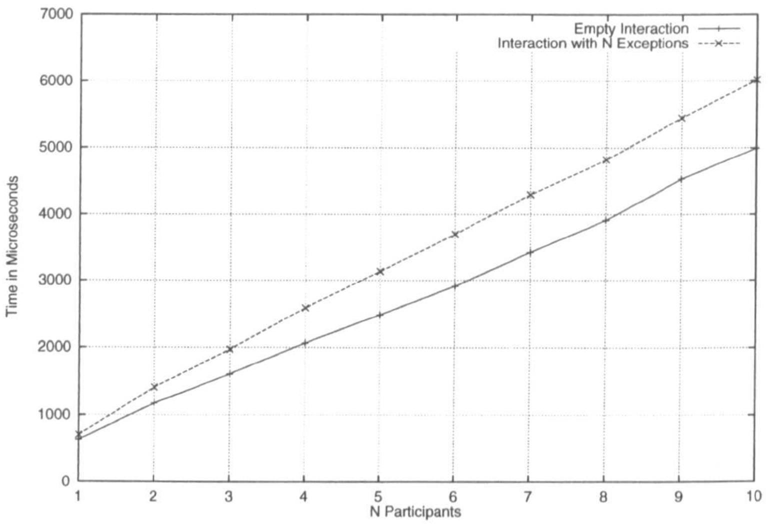

Figure 5.11: Cost of DMIs up to Ten Participants

The times in Figures 5.10 and 5.11 were measured on a 200Mhz Pentium PC running Linux and Java version 1.1.6. All participants were running in the same Java Virtual Machine.

Chapter 6

Case Studies

In this chapter the DMI concept, language, and framework are used to design and implement different systems and abstractions.

In Section 6.1, DMIs are used to design the control software for a production cell case study. The design approach used is heavily influenced by [Zorzo *et al.* 1999] (in which a control software for a production cell without fault tolerance requirements was described). The description of the design is given in the DIP language presented in Section 4.1. Full implementation of the control software is presented using the framework described in Section 5.2. (A short description of this implementation is also presented in [Zorzo 1999].)

In Section 6.2, DMIs are used to implement a structuring abstraction for cooperative and competitive concurrency. The way exception handling in DMI is organized for implementing this structuring abstraction is also presented in [Zorzo & Stroud 1999].

In Section 6.3, DMIs are used to implement a system that uses the

GAMMA paradigm to describe parallel computation. The approach used for implementing the GAMMA system is based on [Romanovsky & Zorzo 1999].

6.1 Case Study: FZI Production Cell II

The FZI Production Cell II case study [Lötzbeyer & Muhlfeld 1996] that is used for the work discussed in this section is a major extension of the production cell case study described in [Lewerentz & Lindner 1995] and presented in Section 2.3.1. The FZI Production Cell II case study describes a production cell composed of eight devices: two conveyor belts – a feed belt and a deposit belt, an elevating rotary table, two presses, a rotary robot that has two orthogonal arms, and two traffic lights. The state of devices is reflected by sensors. Each device has a set of actuators that are used by a control program to change their state. This new case study extends the previous one by adding an extra press in the production cell and extra sensors used for detecting faults in the production cell. The crane, which was used for making FZI Production Cell I cyclic, is replaced in FZI Production Cell II by two traffic lights. These traffic lights are used for permitting plates to be introduced to or removed from the production cell.

Unlike the FZI Production Cell I, failures of devices and sensors in FZI Production Cell II are of major concern. In particular, the cell is intended to be used to provide continuous service even if one of the two presses is out of order. The original, rather simplistic, production cell model assumes no device or sensor failures occur. FZI Production Cell II exposes more and richer issues related to failures and fault tolerance, and it is therefore a valuable case study for investigating and developing concurrent fault-tolerant software. Because devices, sensors and actuators can fail, the required control program is necessarily much more complex, hence more realistic, than one for the original, non-fault-tolerant production cell.

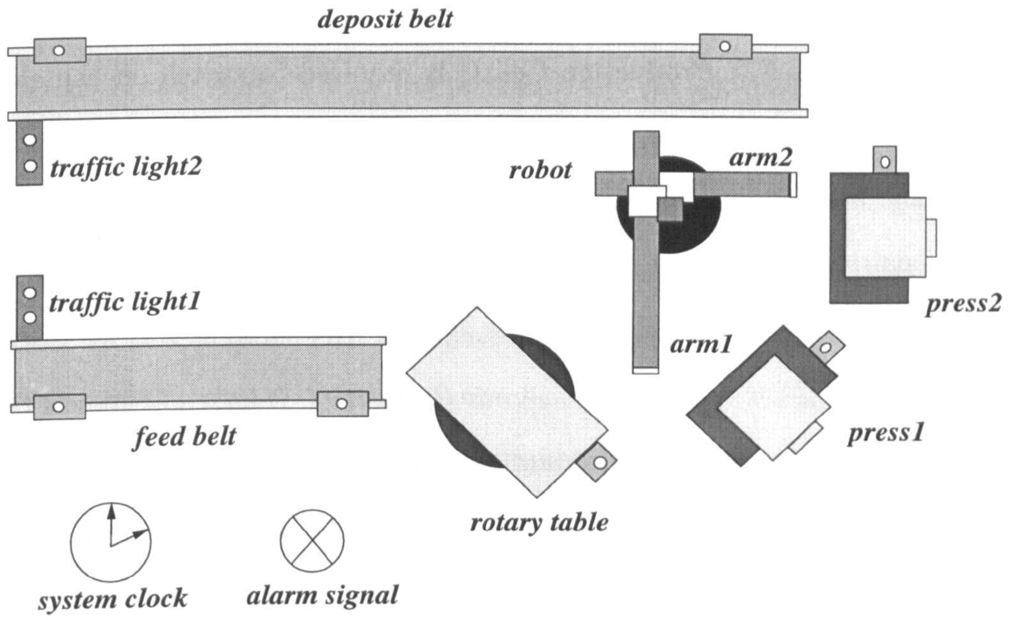

Figure 6.1: FZI Production Cell II

In FZI Production Cell II a complete production cycle of metal plates is as follows: *i*) if the traffic light at the beginning of the feed belt is green, then a metal plate can be added on the feed belt; *ii*) the feed belt conveys the metal plate to the elevating rotary table; *iii*) the table rotates and elevates to the position where the robot can grab the plate; *iv*) the first arm of the robot grabs the plate and places it into a free press (press1 or press2); *v*) the chosen press forges the plate; *vi*) the second arm of the robot removes the forged plate from the press and places it on the deposit belt; *vii*) if the traffic light at the end of the deposit belt is green, then the plate is carried out of the production cell by the deposit belt. Figure 6.1 shows all devices cited in this paragraph.

Furthermore, the FZI Production Cell II provides a global system clock that gives the current time at any instant, and an alarm signal mechanism for reporting component failures to the user of the production cell. The control program for the FZI Production Cell II is required to switch on the alarm signal whenever a failure is detected. The alarm signal can only be switched off by the user, indicating that all faulty devices, sensors or actuators have been repaired.

Requirements

The full specification of FZI Production Cell II calls for a controlling system which satisfies the following requirements:

- *Safety.* Device collisions and the dropping of plates must be avoided. Plates must keep a safe distance from each other. Device mobility must be restricted appropriately.

- *Liveness.* Any metal plate added into the cell via the feed belt must eventually leave the cell via the deposit belt and have been forged.

- *Fault Tolerance.* When a failure occurs, it should be detected and the system should be stopped in a safe state if possible. After recovery from the failure, the system should resume operation from this safe state.

- Other requirements, such as *flexibility* and *efficiency*, may be taken into account as long as they do not conflict with the ones above.

Section 2.3 presented a set of DisCo actions that would enclose the interactions between devices every time a metal plate was being passed from one device to another. The way DisCo actions were specified in Section 2.3 suffices to guarantee the safety and liveness requirements of the FZI Production Cell II case study. (A formal verification of the approach used in Section 2.3 is described in [Canver 1997].) However, using DisCo actions alone does not suffice to describe this extended case study, because DisCo actions do not provide any mechanism to deal with failures during the execution of an action. In this chapter, DIP is applied to implement the control software for FZI Production Cell II. DIP allows the specification of exceptional behaviour in FZI Production Cell II, hence it is possible to specify how fault tolerance requirements are achieved.

Failure Assumptions

Before describing the design and implementation of the control software for the FZI Production Cell II, we list the failure assumptions as defined by the creators of the case study:

- The system clock, two traffic lights, and the alarm signal mechanism are fault-free and do not fail;
- Values of sensors, actuators and clocks are always transmitted correctly without any loss or error;
- No failure can cause devices to exceed certain limiting positions, in the worst case devices are stopped automatically;

- All sensor failures are indicated by sensor values. Boolean sensors return a zero value, and enumeration type sensors return a specified value that indicates a failure;
- All actuator failures will cause devices to stop;
- When the system starts all devices are off.

Failure Detection

In order to detect various failures of sensors and actuators as well as lost plates, appropriate detection measures must be incorporated into the design of the control program. Assertion statements can be used as a form of failure detection measure. For example, after the control program has sent a control command to the robot and asked the robot to drop a plate into the press, the value of the sensor that reports a plate in the press must be checked by an assertion statement. If the sensor returns 0, indicating that no plate is in the press, then an appropriate exception must be raised.

There are several possibilities that could have caused this exception: *i*) the plate may have been lost, *ii*) arm 1 of the robot cannot drop the plate, and *iii*) the sensor of the press fails to report the blank that has been dropped into the press. If an on-line diagnosis algorithm could identify this failure as the sensor failure, exception handling and error recovery would be quite straightforward, e.g. notify the user by switching on the alarm signal and continue normal operations of the cell. However, most of the time distinguishing failures from each other at run-time is extremely difficult, if not impossible. In most cases, if a failure occurs and thus an exception is raised, the cell will have to be stopped in a safe state.

Failures of sensors that report press (or belts) positions and failures of the press actuator can be detected by assertion statements and identified unambiguously with the aid of stopwatches (these stopwatches are implemented using the global clock provided by the FZI Production Cell II). Such failures must be reported to the user through the alarm. Because the FZI Production Cell II has two presses, normal operations can be maintained by using the other press only.

A fault-tolerant program should have the ability of confining damages and failures. For the production cycle of the cell, a device or sensor failure should not affect normal operations of other devices. For example, when a failure of the robot occurs and is handled by the control program, the deposit belt should still deliver a forged plate, if there is one, to the plate consumer. In the following, we will demonstrate how DMIs can confine damages and failures effectively, and minimize the impact of component failures on the entire cell.

Design Decisions

The design for FZI Production Cell II addresses the safety, functionality, and efficiency requirements by separating the design into two levels. The first level deals with efficiency and flexibility issues, while the second level deals with safety and fault tolerance issues. The first level is designed as an *outermost DMI* formed by a set of device controller roles that determine the order in which devices interactions are executed. The second level is designed as a set of *nested DMIs* that controls the interactions between devices. These nested

DMIs are enclosed by the outermost DMI (see Figure 6.2[1]). As the safety and fault tolerance requirements are the most important in the system, they are satisfied, as far as possible, at the level of nested DMIs, while efficiency requirements are met by the device controller roles in the outermost DMI. However, some responsibility for exception handling has to be exercised at the outermost DMI. The device controller roles communicate with each other in order to schedule the execution of the nested DMIs. (The scheduling of nested DMIs can be programmed in several ways.) The outermost DMI encloses the entire FZI Production Cell II control program.

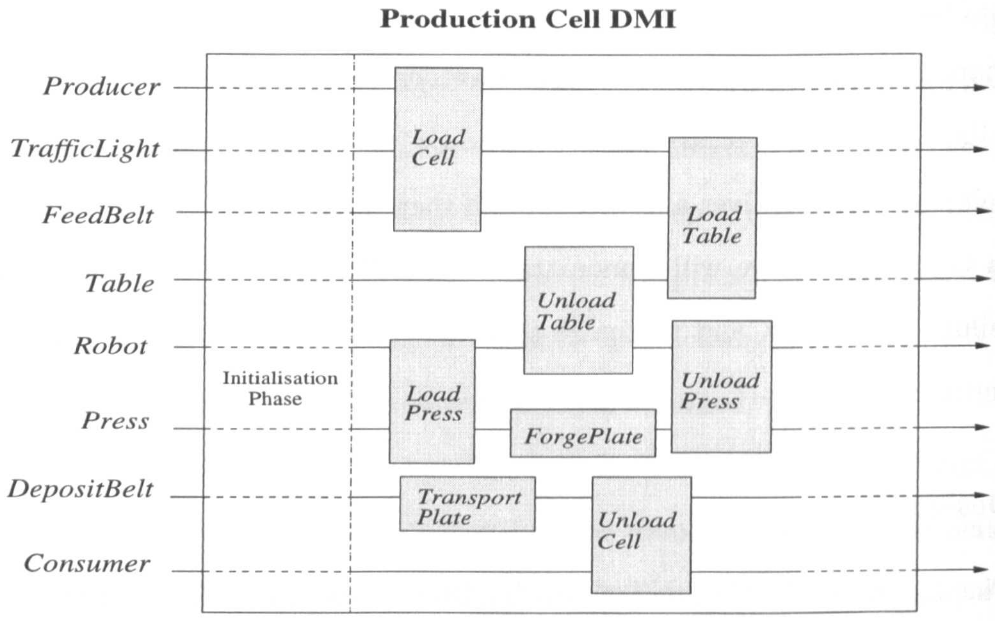

Figure 6.2: Outermost DMI

The priority is to provide a clear system design that is simple in structure and meets the case study safety and fault tolerance requirements. The intention is to structure all critical system activities out of nested DMIs. Nine

[1]Only one press controller is shown in the figure.

nested DMIs were designed to control the interactions between devices for the system. Each nested DMI controls one step of the plate processing and typically involves passing a plate between two devices.

The main design decisions are as follows. To meet the safety and fault tolerance requirements, the roles and boundaries of nested DMIs were chosen in such a way that neither plates nor devices could collide. Only one plate can be in a nested DMI, a plate cannot be involved in more than one nested DMI, and a device can participate in only one nested DMI at a given time. All device movements are performed within nested DMIs and the devices involved in a nested DMI are switched off before leaving that nested DMI. Thus, all devices are stationary when not under the control of a nested DMI.

An informal analysis of the safety requirements showed that using nested DMIs to design the system in this way would allow us to guarantee all requirements concerned with concurrent activities within FZI Production Cell II. The remaining requirements (limitations on the movements of individual devices) are not connected with system concurrency and are provided in our design by "defensive" and self-checking programming inside nested DMIs (e.g. checking each nested DMI's pre and post-conditions).

For each of the nested DMIs that encloses an interaction between devices, there is a set of post-conditions that hold if the nested DMI has executed without failure. The nested DMIs for FZI Production Cell II can in fact produce three different outcomes: *normal*, *abort* or *exceptional*.

A *normal* outcome is achieved if a nested DMI is able to satisfy its post-conditions and thereby fulfill its obligations. An *abort* outcome occurs if a nested DMI is unable to fulfill its obligations for some reason but is able to roll back the system state and undo any effects it might have had. The roll back is done without compromising the safety requirements set by the case study. All nested DMIs can produce an *abort* outcome, but some of them can only have this outcome if they have not started to execute some parts of the nested DMI, e.g. after the ForgePlate DMI has forged the plate, roll back is not possible and an *abort* cannot be achieved. The last possible outcome is a *exceptional* outcome. If an *exceptional* outcome is produced by a nested DMI, the system state is represented by the exceptional post-condition associated with the *exceptional* outcome produced. The execution of the outermost DMI will depend on the outcome of the nested DMIs, e.g. the outermost DMI will execute in a degraded mode when a press failure exception is signalled during the execution of the nested DMI that involves one of the presses (see Section 6.1.4 for exception handling in a nested DMI, and Section 6.1.5 for exception handling in the outermost DMI).

Using DMIs not only guarantees the safety requirements but simplifies the system design because DMIs are atomic and all internal DMI state is hidden from the outside. In addition, using DMIs guarantees such conventional features of concurrent programming as mutual exclusion: DMIs have synchronous entries, so a DMI cannot start until all of its roles are ready, and synchronous exit, which means that participants can leave the DMI only when all roles have completed their execution (see Chapter 2 and 4).

6.1.1 Device Controllers

Each device in FZI Production Cell II is controlled by a corresponding role in the outermost DMI. This role is responsible for specifying the sequence of nested DMIs in which the device participates. For cyclic execution, a device controller role has a straightforward structure with an endless loop, which means that the controller role goes through a set of nested DMIs. See the DIP example in Figure 6.3.

```
role TableController with table:Table is
  Plate plate;
  loop while (true)
      table@LoadTable with table, plate;
      table@UnloadTable with table, plate;
  end loop
end role
```

Figure 6.3: The TableController Role

Figure 6.4 shows the set of nested DMIs, dotted boxes, in our design superimposed on a drawing of FZI Production Cell II to indicate which devices are involved in which nested DMIs. Two nested DMIs whose dotted boxes overlap cannot be executed concurrently because a device cannot be involved in more than one nested DMI at once. For example, the LoadTable DMI has the feed belt, the rotary table and the traffic light at the beginning of the feed belt, as its roles, and the UnloadTable DMI has the table and the robot as its roles. These two nested DMIs therefore cannot run concurrently. Moreover, the design guarantees that the LoadTable DMI cannot start with the next plate until the table is ready, which means that the previous plate must have been picked from the table by the robot arm.

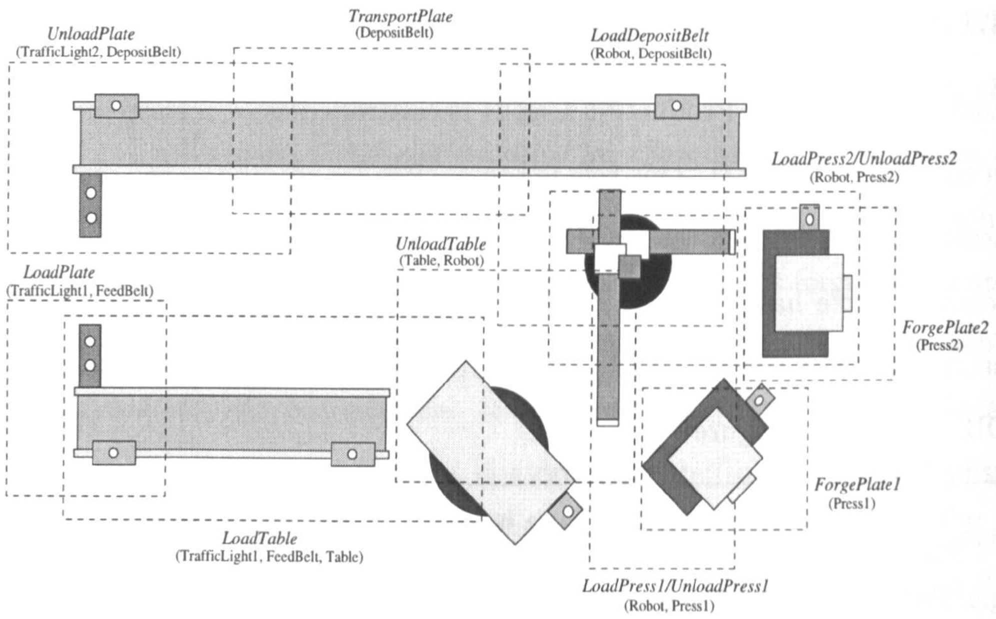

Figure 6.4: The Set of Nested DMIs in FZI Production Cell II

The DMIs themselves are the building blocks that guarantee system safety: their use prevents the building of a system in which plates and devices can collide. For example, if one of the device controllers incorrectly orders the sequence of nested DMIs in which a given device takes part, then the system will deadlock in a safe state (all nested DMIs will have finished their execution and no further nested DMIs will be activated, so the devices will not move, hence the safe state). Moreover, if any nested DMI fails and the controller roles do not provide proper recovery then the system stops. This is because its participants will not be ready for the execution of that or of the following nested DMIs, unless the controller performs some form of error recovery.

6.1.2 Sensors, Actuators and Plates

The approach used here for dealing with sensors and actuators basically relies on the general ideas from [Wirth 1977], which proposes two alternative methods of dealing with peripheral devices: synchronous and asynchronous. When a device is served synchronously, each input/output operation is synchronous and active waiting is hidden inside the operation. External software operates with this device by using an interface provided by the synchronous wrapper. With asynchronous wrapping, each device is represented as an active role which operates with the device, executes active waiting and can inform other roles about changes of the device state (this method was used in [Zorzo *et al.* 1999]). These two methods are obviously complementary and, generally speaking, each device may be manipulated synchronously at one system level and asynchronously at another. In this thesis the synchronous method is used.

After analysing FZI Production Cell II and the control system it was decided to treat all actuators and sensors as external objects in the system. The plates are also represented by external objects, which are shared by the nested DMIs. They are passed as input/output parameters among the DMIs (at the outermost DMI level).

There is a special role in the outermost DMI to monitor the state of all the plates in FZI Production Cell II. This monitor role competes with the nested DMIs to check the state of the plate objects using transactional semantics. Thus, when a plate object is not being used by a nested DMI, the monitor can access it by starting a transaction on that plate object (recall from that our framework provide a class called ExternalObject that provides transactional semantics). If the plate is being used by a nested DMI, then the monitor has to wait until the nested DMI is finished. The monitor cannot see any intermediate state of the plate. For example, when an exception is raised in a nested DMI, the monitor will not be able to see any change of the plate state caused by forward or backward error recovery. On the other hand, each execution of a nested DMI should also, in some sense, behave like a transaction with respect to the plate object it accesses.

6.1.3 Nested DMIS in FZI Production Cell II

The nested DMIs that were designed for the FZI Production Cell II case study typically have two basic roles, one that takes a plate as an input argument, and another that takes a plate as an output argument. The device associated with the role that has the plate as an input argument passes the plate to the role that has the plate as an output argument. All nested DMIs will only be activated if a set of preconditions is valid. The post-conditions are checked dynamically using acceptance tests [Randell 1975].

In the design, every nested DMI is composed of a set of roles, and manipulates a set of internal and external objects. Note that synchronous entry and exit, exception handling, recovery, consistency and atomicity of external and local objects, and so on, are guaranteed by the DMI mechanism.

There are nine nested DMIs for managing the interactions between devices: {LoadCell, LoadTable, UnloadTable, LoadPress, ForgePlate, UnloadPress[2],

[2]Actually, there are two instances of LoadPress, ForgePlate, and UnloadPress, one for each of the presses.

LoadDepositBelt, TransportPlate, UnloadCell}. A brief description of each of these nested DMIs is presented below.

- **LoadCell DMI:**

  roles – {producer, feedBelt, trafficLight}

  functionality – in this DMI a plate is sent to the DMI through the producer role as an input parameter; that plate is returned as an output parameter of the feedBelt role, meaning that the producer has dropped a plate onto the feed belt. (The producer role could of course be undertaken by a human being or by any other subsystem, e.g. a previous cell in the production line.) There is also a role for the traffic light at the beginning of the feed belt. This role closes the traffic light after the producer has dropped a plate onto the feed belt.

- **LoadTable DMI:**

  roles – {table, feedBelt, trafficLight}

  functionality – in this DMI a plate is sent to the DMI through the feedBelt role and that plate is returned by the table role. This DMI controls the movement of a plate along the feed belt until the plate drops onto the table. After the plate has been dropped onto the table, the trafficLight role opens the traffic light at the beginning of the feed belt. (Full description of this DMI is shown in Section 6.1.4.)

- **UnloadTable DMI:**

  roles – {table, robot}

  functionality – in this DMI a plate is sent to the DMI through the table role and that plate is returned by the robot role. This DMI controls all the device movements that are required in order for the first arm of the robot to grab a plate from the table.

- **LoadPress DMI:**

  roles – {press, robot}

  functionality – in this DMI a plate is sent to the DMI through the robot role and that plate is returned by the press role. This DMI controls the first arm of the robot dropping a plate into a press.

- **ForgePlate DMI:**

  roles – {press}

  functionality – in this DMI a plate is sent to the DMI through the press role and that plate is returned by the same press role. During this DMI, a plate is forged by the press, and external information about the plate is updated, e.g. the plate has been forged.

- **UnloadPress DMI:**

  roles – {press, robot}

  functionality – in this DMI a plate is sent to the DMI through the press role and that plate is returned by the robot role. The UnloadPress DMI controls the second arm of the robot grabbing a forged plate from the press.

- **LoadDepositBelt DMI:**

  roles – {robot, depositBelt}

  funcionality – in this DMI a plate is sent to the DMI through the robot role and that plate is returned by the depositBelt role. This DMI controls the second arm of the robot dropping a plate onto the deposit belt.

- **TransportPlate** DMI:

  roles – {depositBelt}

  functionality – in this DMI a plate is sent to the DMI through the depositBelt role and that plate is returned by the same depositBelt role. This DMI controls the transporting of plates from the beginning of the deposit belt to the end of the deposit belt, i.e. after the photo-electric sensor.

- **UnloadCell** DMI:

  roles – {depositBelt, trafficLight, consumer}

  functionality – in this DMI a plate is sent to the DMI through the depositBelt role and that plate is returned by the consumer role. The UnloadCell DMI controls the consumer taking a plate from the deposit belt. This consuming of a plate is only possible if the traffic light at the end of the deposit belt is open.

6.1.4 The **LoadTable** DMI

In this section a complete description of one of the DMIs described in Section 6.1.3 is provided. The LoadTable DMI was chosen because it is an example of 3-party DMI. We first present the DIP objects that are used in this DMI (see Figure 6.15).

The traffic light in our design is represented as a state machine, which can change from green to red and vice-versa. The initial state for the traffic

```
class TrafficLight is
   state *green, red;
end;

class Belt is
   state *off, on; // belt actuator
   state *begin_off, begin_on; // sensor at the beginning
   state *end_off, end_on; // sensor at the end
end;

class Table is
   state pos_feedbelt, pos_belt2robot, pos_robot, pos_out, pos_err ;
   state *lower, upper, none;
   state *free, loaded;
end;
 // "global" objects declaration
Belt feedBelt;
Table table;
TrafficLight trafficLight1;
```

Figure 6.5: DIP Objects for the LoadTable DMI

light is green, i.e. new plates can be inserted in the cell. The feed belt is a Belt. A belt is represented by three state machines: one for the belt actuator; one for the sensor at the beginning of the belt; and, one for the sensor at the end of the belt. The initial state for the feed belt is off; no plate at the beginning (begin_off); and, no plate at the end of the feed belt (end_off). The table is also represented by three state machines: one to represent the state of the angle of the table; one to indicate if the table is at its lower or upper position, or neither; and, one to indicate a plate on the table.

The main body of the LoadTable DMI is shown in Figure 6.6. In the figure, the preconditions for this DMI are specified in the **guard** statement, i.e. the table must be at its upper position (t.upper), the angle of the table must

```
action LoadTable is
  body is
   guard t.upper ∧ t.pos_robot ∧ t.free ∧ fb.off ∧ fb.begin_on ∧ fb.end_off ∧ tl.red
   role table with t:Table, plate:Plate
     →t.lower :: TableVerticalStuck
     →t.pos_feedbelt :: TableRotationStuck
     true to feedbelt
     plate from feedbelt blocks
     →t.loaded :: TablePlateSensor
     →plate.ONTABLE
   end role;
   role feedbelt with fb:FeedBelt, plate:Plate
     boolean tableready;
     tableready from table blocks
     →fb.on :: FeedBeltStuck
     →fb.begin_off
     →fb.end_on :: FeedBeltEndSensor
     plate to table blocks
     →fb.end_off
     →fb.off
     true to trafficLight
   end role;
   role trafficLight with tl:TrafficLight
     boolean feedbeltready;
     feedbeltready from feedbelt blocks
     →tl.green
   end role;
   assertion t.lower ∧ t.pos_feedbelt ∧ t.loaded ∧ fb.off
           ∧ fb.begin_off ∧ fb.end_off ∧ tl.green
  end body;
end action;
```

Figure 6.6: The LoadTable DMI

be indicating the position of unloading the table by the robot (t.pos_robot), no plates must be on the table (t.free), the feed belt must be switched off (fb.off), there must be a plate at the beginning of the feed belt (fb.begin_on), and the traffic light must be red (tf.red). The execution of the LoadTable

DMI is as follow. First the table is moved down, and then rotated to the position in which it can be loaded. Once the table is in position, a message is sent to the **feedbelt** role, so the feed belt can be switched on. When the plate has passed the sensor at the end of the belt, the **table** role receives the plate object from the **feedbelt** role. After that, the belt can be stopped, the **table** role can update the state of the plate object, and the **trafficLight** role can switch the traffic light to **green**. The post-conditions for this DMI are specified in the **assertion** statement.

In order to update the state of the system to reflect the movement of a plate from one device to another in the real world, it is necessary to simulate this movement in the memory of the computer by passing a plate identifier from one role to another. The command **to/from** is used to simulate this movement in Figure 6.6.

In Figure 6.6, we also indicate the exceptions that may be raised while the **LoadTable** DMI is being executed. As defined in Section 6.1, the traffic light does not fail, so no exception is raised during the change of the traffic light's state. The feed belt and table, on the other hand, can raise several exceptions. For example, when the table is rotating, the actuator can fail, hence an exception **TableRotationStuck** being raised. When an exception is raised in a role, all roles must be interrupted and exception resolution algorithm is started (see Chapter 4). After a common exception is decided upon, a handler for that exception is started (if there is one).

Figure 6.7, for example, shows the exception handler for the situation in which the sensor at the end of the feed belt fails. If the sensor cannot be changed to the state on, then an exception is raised. The **LoadTable** DMI can

```
handler for FeedBeltEndSensor is
  role table with t:Table, plate:Plate
     →plate from feedbelt blocks
     →t.loaded :: TablePlateSensor
     →plate.ONTABLE
  end role;
  role feedbelt with fb:FeedBelt, plate:Plate
     plate to table blocks
     →fb.off
     true to trafficLight
     raise FeedBeltSensor; // signal to outermost DMI
  end role;
  role trafficLight with tl: TrafficLight
     boolean feedbeltready;
     feedbeltready from feedbelt blocks
     →tl.green
  end role;
end handler;
```

Figure 6.7: Handler for FeedBeltEndSensor Exception

still finish its operation, i.e. the plate can still be loaded on the table, the plate state can be updated, and the traffic light can be switched to green. Notice that this handler will signal an exception to the outermost DMI, which can then stop the system in a safe mode. Recall that an exception that has been dealt with inside a team cannot be handled in the same team again, hence the **raise** command with, for convenience, the same exception name (see Chapter 4).

6.1.5 Controller Roles

The set of nested DMIs presented previously executes various critical operations, e.g. passing a plate between devices. The activation of these nested DMIs and the order in which they execute is the responsibility of a set of device/sensor controller roles, described in this section, which belong to an outermost DMI. The roles in this outer level DMI are activated by external executing threads (DIP players) created immediately after the system begins its execution.

The loading of plates in FZI Production Cell II, is performed by a special controller, ProducerController. Its responsibility is to interact with the user to put plates onto the feed belt one at a time. When the plate is ready for loading into the feed belt, the producer controller enters the LoadCell DMI, thereby blocking until the FeedBeltController indicates its readiness to accept the plate by also entering the LoadCell DMI (see Figure 6.8).

```
role ProducerController is
  Plate plate;
  loop while (true)
     plate from input blocks
     producer@LoadCell with plate
  end loop
end role
```

Figure 6.8: The ProducerController Role

After a plate has been deposited on the feed belt, FeedBeltController enters the LoadTable DMI. At the end of the LoadTable DMI, the plate will have been passed to the TableController and the FeedBeltController can wait for a new plate to arrive, and so on (see Figure 6.9).

The order in which nested DMIs are executed by the RobotController role depends on the messages it receives from the other controller roles. The RobotController role waits for a message to arrive and then decides which nested DMI to execute depending on the state of the arms of the robot. Some-

```
role FeedBeltController with feedbelt : FeedBelt is
  Plate plate;
  loop while(true)
     feedbelt@LoadCell with feedbelt, plate
     feedbelt@LoadTable with feedbelt, plate
  end loop
end role
```

Figure 6.9: The FeedBeltController Role

```
role RobotController with robot: Robot is
  Plate plate1, plate2; // plates on arms of the robot
  loop while(true)
    begin
     loop until message != empty
       begin
         message from TableController ;
         if (message = empty) then message from DepositBeltController
         if (message = empty) then message from Press1Controller
         if (message = empty) then message from Press2Controller;
       end
     if ((message=PLATE_ON_TABLE) and robot.arm1.mag_off)
         robot@UnloadTable with robot, plate1;
     else if ((message=PLATE_FORGED_IN_PRESS1) and robot.arm2.mag_off)
         robot@UnloadPress1 with robot, plate2;
     else if ((message=PLATE_FORGED_IN_PRESS2) and robot.arm2.mag_off)
         robot@UnloadPress2 with robot, plate2;
     else if ((message=DEPOSIT_BELT_FREE) and robot.arm2.mag_on)
         robot@LoadDepositBelt with robot, plate2;
     else if ((message=PRESS1_FREE) and robot.arm1.mag_on)
         robot@LoadPress1 with robot, plate1;
     else if ((message=PRESS2_FREE) and robot.arm1.mag_on)
         robot@LoadPress2 with robot, plate1;
     else requeue message;
    end
end role
```

Figure 6.10: The RobotController Role

times the RobotController role is not able to execute a nested DMI, e.g. if the TableController role delivers messages to the RobotController role faster than the PressController role, it is highly probable that two PLATE_ON_TABLE messages arrive before a FORGED_PLATE_IN_PRESS (1 or 2) message, so the first arm of the robot will still be holding a plate and therefore the robot will be unable to pick another plate up from the table. When such a situation occurs, the RobotController stores the received message back into its **to/from** command queue (see Figure 6.10).

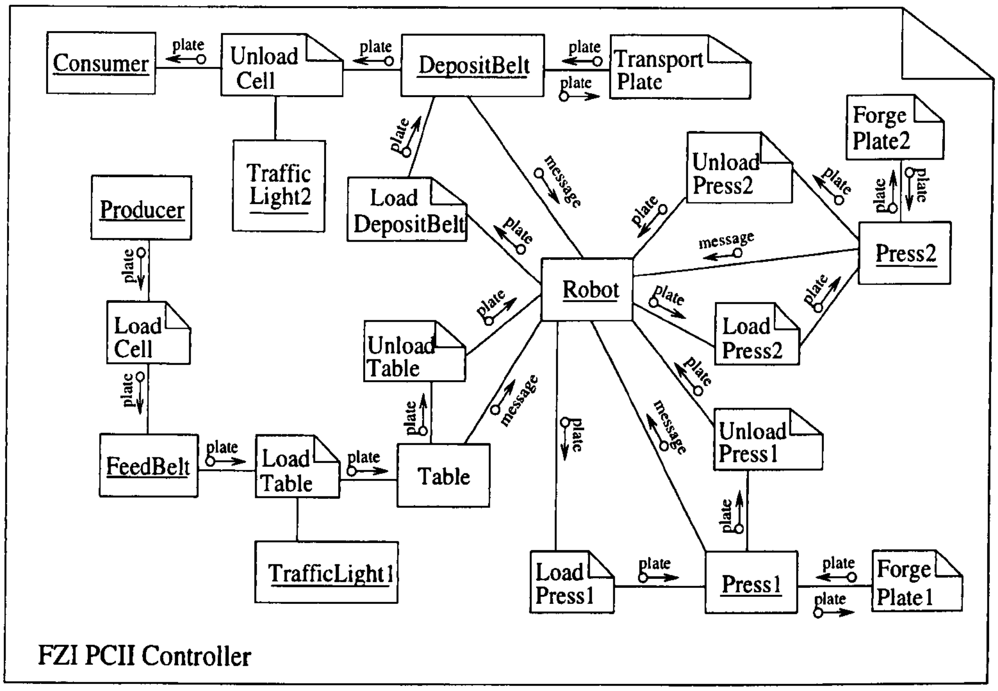

Figure 6.11: Relationship between DMIs and Device Controllers Roles

Figure 6.11 shows a UML diagram that represents the relationship between nested DMIs and device controller roles. In the figure, the device controller roles are represented as objects. The nested DMIs are represented as UML notes. These notes are much more complex structures (Figure 6.14, in the next section, shows the complete structure for the LoadTable DMI). Plates are passed between devices via the nested DMIs. One device sends a plate to the nested DMI, and the other device receives the plate in the nested DMI. The messages the robot controller receives from the other devices is also represented in the figure.

The outermost DMI, which encloses the controller roles presented in this section, is also responsible for either stopping the system or if possible keeping the system executing in a degraded mode, if one of the nested DMIs signals an exception. For example, when one of the presses fails. Figure 6.12 shows how the Press1Controller role should be specified in a handler of the outermost DMI if press1 fails. All the other roles would not change.

```
role Press1Controller is
  boolean ok := false;
  loop while (true)
    if not ok then ok from input blocks
    press@LoadPress1 with plate
    press@ForgePlate1 with plate
    press@UnloadPress1 with plate
  end loop
end role
```

Figure 6.12: The Press1Controller Role when Press1 Fails

6.1.6 Java Implementation

In this section, the API presented in Section 5.2 is used to implement the controlling software for FZI Production Cell II. The LoadTable DMI is used as an example of a 3-party DMI. This section shows how to create the managers and roles for the LoadTable DMI; how a role can be activated by a Java thread; and, how to extend the basic `Role` class to create a new role class.

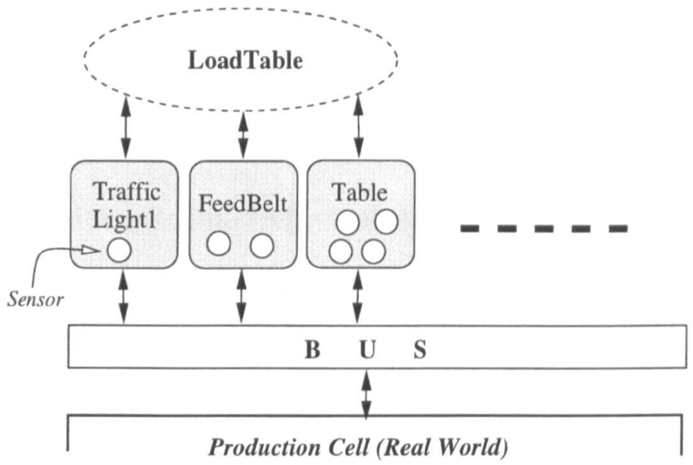

Figure 6.13: Structure of the FZI Production Cell II Implementation

First of all, Figure 6.13 shows how the system that controls FZI Production Cell II is structured. As mentioned in Section 6.1.2, there is one object for each of the devices in the production cell. These device objects are composed of other devices and sensors, e.g. the table device is composed of four sensors. The device objects are implemented as ExternalObjects. DMIs will access these external objects in order to change the state of FZI Production Cell II. For example, all interactions that are needed for the process of loading the table by the feed belt is enclosed by LoadTable DMI showed in Figure 6.13 (dashed ellipse). The LoadTable DMI uses three external objects and is composed of three role objects: TLight1Role, FeedBeltRole and TableRole. These three roles act upon the devices in a coordinated way to load a metal plate over the table. The execution of these activities will use a SharedLocalObject in order for the `TableRole` to inform the FeedBeltRole when the table is ready to be loaded. The FeedBeltRole will only turn the feed belt on, for moving the plate from its beginning to the table, when the table is in position. Figure 6.14 shows the UML object diagram for the LoadTable DMI.

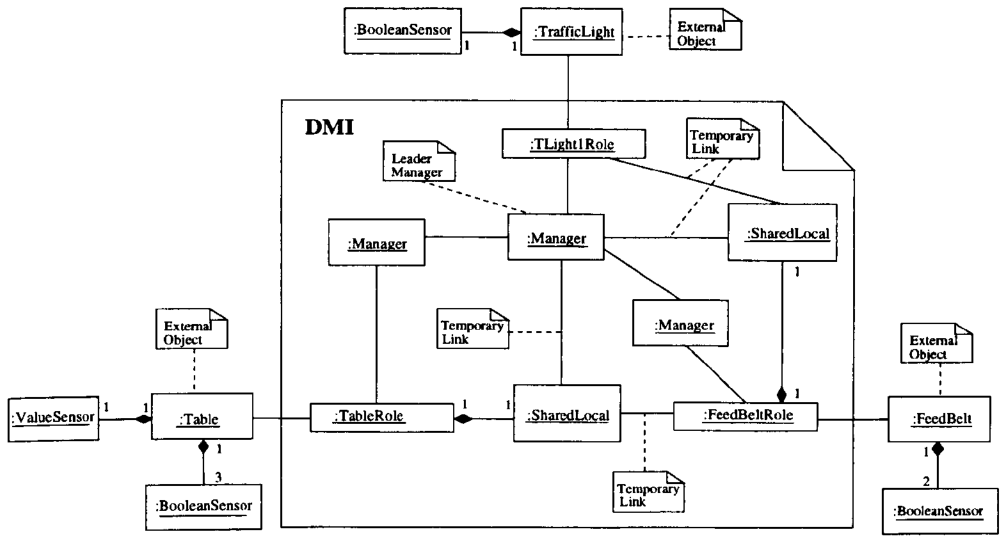

Figure 6.14: UML Object Diagram for the LoadTable DMI

Figure 6.15 shows how the manager and the role objects for the LoadTable DMI are created. In the figure, in line (3), mTable is a manager called "Table" for the "LoadTable" DMI, that has mTLight1 as the leader (when a leader is not specified, then that manager leads itself - line (1)). The eh_i is a hashtable object that contains the exceptions that are treated by the corresponding manager. Each entry of the hashtable has an exception and a role that is activated in the case of this exception being raised. After the managers are created, the role objects have to be created. The creation of a role object for the table is shown in line (6) in Figure 6.15. Each role object is created with a name and a manager that controls this role.

Figure 6.16 shows how a new role class extends our basic Role class.

```
(1)Manager mTLight1 = new Manager("TLight1","LoadTable",eh1);
(2)Manager mFeedBelt= new Manager("FeedBelt","LoadTable",eh2,mTLight1);
(3)Manager mTable  = new Manager("Table","LoadTable",eh3,mTLight1);

(4)Role tLight1    = new TLight1Role(mTLight1,"TLight1Role");
(5)Role feedBelt   = new FeedBeltRole(mFeedbelt,"FeedBeltRole");
(6)Role table      = new TableRole(mTable,"TableRole");
```

Figure 6.15: Objects Instantiation for the LoadTable DMI

```
public class TableRole extends Role {
 Synchronous waitTable; // shared object

 public TableRole(Manager manager, String roleName) {
  super (manager, roleName);              // set role with name and manager
  waitTable = new Synchronous();          // creates a shared object
  manager.sharedObject("waitTable", waitTable);// export shared object
 }

 protected void body(Transactional list[]) throws Exception {
  Table table = (Table) list[0]; // External Object
  Channel channel;
  try {
   channel = (Channel) manager.getSharedObject("channel");
   table.rotate(POS_FEEDBELT); // Rotate to loading angle.
   table.moveDown(); // Move table down to the loading high.
   waitTable.synchronize(); // Inform feedbelt that table is ready.
   list[1] = channel.get(); // Receive plate from feedbelt.
   list[1].changeState(ONTABLE);
  } catch (Exception e) { // OOPS! Problems. Stop everything!
   table.stop_h();
   table.stop_v();
   throw e;              // Pass the exception to manager.
  }
 }
}
```

Figure 6.16: Full Implementation of the Table Role in LoadTable

The new role class is responsible by redefining the body method and for creating and exporting local shared objects that are used by other roles. When an object of a role class is created, it has to be informed of the manager object that will be responsible for controlling this role (see lines (4), (5) and (6) in Figure 6.15). The code, in Figure 6.16, also shows the complete implementation of the body of TableRole for the LoadTable DMI. The code for the other roles, i.e. FeedBeltRole and TLight1Role, in the LoadTable DMI are implemented in a similar way. The Java code shown in Figure 6.16 represents the DIP table role shown in Figure 6.6.

In Figure 6.17, we show how to activate the table role, declared in Figure 6.15, in the "LoadTable" DMI. Each role has a special method called execute that passes the list of parameters (list) to the manager of that role. This is the only public method provided by a role (see Section 5.2.2).

```
table.execute(list); // execute table role sending list of objects
```

Figure 6.17: Execution of the Table Role in LoadTable DMI

6.1.6.1 Exception Handling

Every time an exception that may affect the whole DMI is raised in a role, that exception has to be thrown by that role (see **catch** block in Figure 6.16). The manager of that role will catch this exception and will start the exception handling process as explained in Section 5.2.4. Figure 6.18 shows a possible scenario where two exceptions are raised during LoadTable DMI. Two roles, FeedBeltRole and TableRole concurrently raise exceptions FeedBeltStuckEx-

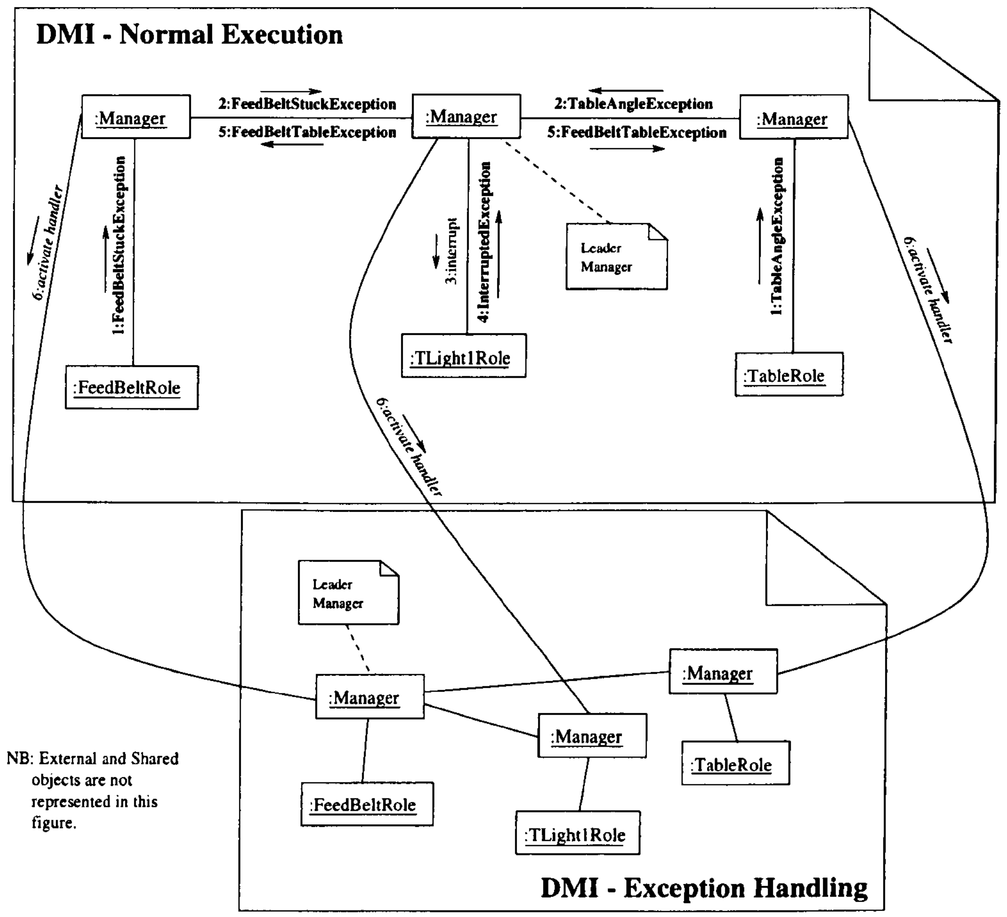

Figure 6.18: Dealing with Concurrent Exceptions

ception and TableAngleException respectively (step **1** in the figure). These exceptions are caught by the role managers which inform the leader about these exceptions (step **2**). The leader then detects that TLight1Role is still executing and interrupts the thread executing that role (step **3**). An InterruptedException is therefore raised by the manager of TLight1Role informing the leader that the role has been interrupted successfully (in this case the manager of TLight1Role and the leader are the same) (step **4**). The leader then decides upon which exception has to be handled: exception FeedBelt-

TableException in our example. Exception FeedBeltTableException is sent to all managers of the DMI (step **5**) which will activate the roles in an exception handling interaction to deal with exception FeedBeltTableException (step **6**). A new set of managers and roles in the handler will execute in the same way as if they belonged to a normal interaction.

We have based our implementation of exception handling for FZI Production Cell II on the failure analysis and definition presented in [Xu *et al.* 1998] and [Xu *et al.* 1999].

6.1.6.2 Controlling the FZI Production Cell II Simulator

The Java implementation of the FZI Production Cell II control software was used to drive a simulator provided by the FZI. Figure 6.19 shows a screen dump of the FZI Production Cell II simulator controlled by our implementation. Outlines of the nested DMIs are displayed over the simulator diagram. During system execution these outlines are coloured in gradually to show the progress of a DMI execution. In the figure, there are three DMIs that are active and being executed: LoadTable, LoadPress2, and ForgePlate1. If an exception, or concurrent exceptions, are raised in a DMI, the colour within the outline will change to indicate the dynamic process of exception handling.

Figure 6.20 shows a slightly modified version of the device and sensor failure injection panel provided by the FZI. Using this panel, failures can be easily injected into the FZI Production Cell II simulator. For example, a rotary motor failure or a rotary sensor failure of the robot can be injected by pressing the corresponding buttons in the panel. The original panel was modified in order to permit the injection of concurrent failures: a pair of

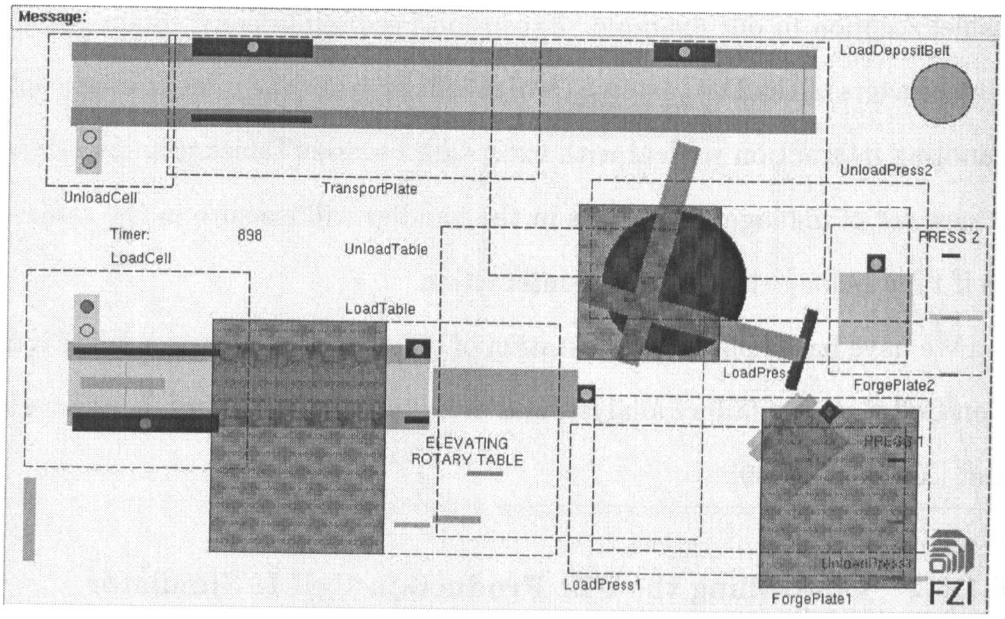

Figure 6.19: Driving the FZI Simulator

failures can be injected into the simulator if the failure mode selection is set to "Double". In this mode, two different failure buttons have to be pressed sequentially, but only the second failure will stimulate the actual injection of the two concurrent failures. After one or more failures are injected into the simulator, the error detection measures embedded in our control program detect them promptly. One or more corresponding exceptions will thus be raised.

During the testing phase and the demonstration of our implementation, all injected device or sensor failures were caught successfully and handled immediately by our control program. Even a previously unknown and indeterministic software bug in the FZI simulator was also detected by the testing of the post-conditions of the DMIs, and recovered from by the system.

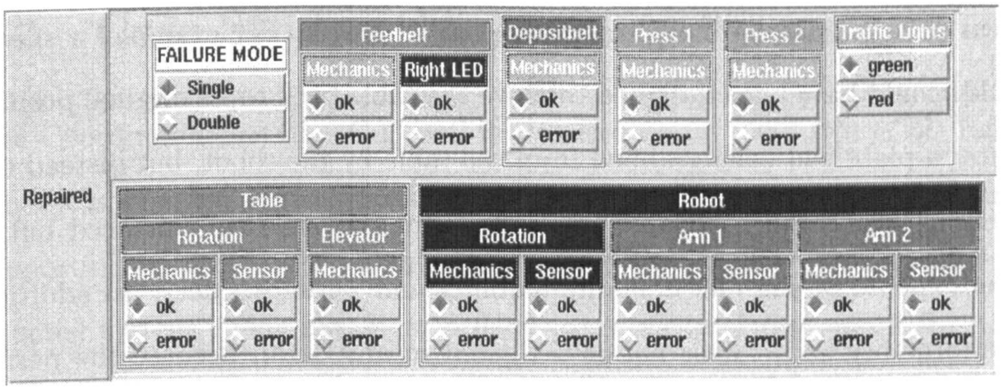

Figure 6.20: Failure Injection Panel

6.1.7 Discussion

The use of DMIs has helped us in extracting interactions between objects from the objects. This facilitated the design and implementation of objects for the FZI Production Cell II case study, in the sense that objects that represent devices are only concerned with the basic operation of the devices. For example, in designing and implementing the robot object we needed to consider only the operations that the robot can perform, e.g. operations to rotate the robot: *left*, *right*, and *stop*. Operations that are related to the environment the robot is inserted in, are not designed/implemented in the robot object, e.g. the unloading of the table, or the loading of the press by the robot. Because these operations can vary depending on where the robot is installed, they are left to be implemented in a separate place, making the reuse of the robot object possible without modifications. DMIs suit this sort of strategy very well, with the additional benefit of enclosing and recovering possible failures that may happen during this kind of interaction.

In the design, clear system structuring is preferred to maximising the system performance by including more parallel DMIs. For example, a special DMI could have been designed to move the table back to its original position after a plate had been grabbed from the table by the robot, but instead this is done inside the LoadTable DMI. Performance could be enhanced but at the expense of simplicity, maintainability and perhaps safety. In addition, performance issues were not investigated in any detail because the performance essentially depends on data about the time required to execute each operation, which is not provided as part of the case study. Without this data it is not possible to choose the solution with the best performance. However, in the design, as much parallelism was allowed as possible without sacrificing safety. Hence, the design of our controller roles is rather more complex than might otherwise be the case to allow flexible scheduling of DMIs. It is interesting to note that if too many parallel DMIs were designed for the purpose of enhancing performance, they would often end up waiting for each other, and the performance would degrade as a result.

The main lessons learnt from designing and implementing the FZI Production Cell II using DMIs are as follows. The use of DMIs has enabled us to design a two-level system, one level for the scheduling of DMIs, and another level for the enclosing of interactions inside DMIs. Using this approach, a clear separation of functionality between levels was achieved. Complex concurrent behaviour was hidden inside DMIs and levels. Furthermore, implementing fault tolerance for this case study, shows that using DMIs imposes a very disciplined and unified way of providing fault tolerance through the entire system.

6.2 Case Study: CA Action Abstraction

The Coordinated Atomic (CA) action [Randell *et al.* 1997] [Xu *et al.* 1995] concept is a unified approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object-oriented system. This paradigm provides a conceptual framework for supporting both cooperative and competitive concurrency and for achieving fault tolerance. It does this by extending and integrating two complementary concepts — conversations [Randell 1975] and transactions [Gray & Reuter 1993]. CA actions have properties of both conversations and transactions. Conversations are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain the consistency of shared resources in the presence of failures and of competitive concurrency.

6.2.1 Structuring CA actions with DMIs

CA actions are very similar to DMIs, but differ regarding how exceptions are dealt with. Remember that any of the roles of a DMI may raise an exception. If that exception cannot be dealt with locally by the role, then it must be propagated to the other roles in the DMI. The CA action abstraction, on the other hand, does not allow an exception to be dealt with locally; if an exception is raised, the handling of that exception has to involve all roles in the CA action. Since it is possible for several roles to raise an exception at more or less the same time, a process of exception resolution [Romanovsky *et al.* 1996] is necessary in order to agree on the exception to be handled within the DMI. In the CA action abstraction, an agreed exception is propagated to all roles of the CA action, in which some form of exception handling is invoked. In DMIs, a new multiparty interaction takes place to handle that exception. Another major difference between DMIs and CA actions, is the way that further exceptions are dealt with. In the CA action abstraction, if it is not possible to achieve either a normal outcome, or an agreed signal, using the exception handling, then the CA action should be aborted and its effects should be undone. Otherwise, a failure exception will be signalled to the external environment.

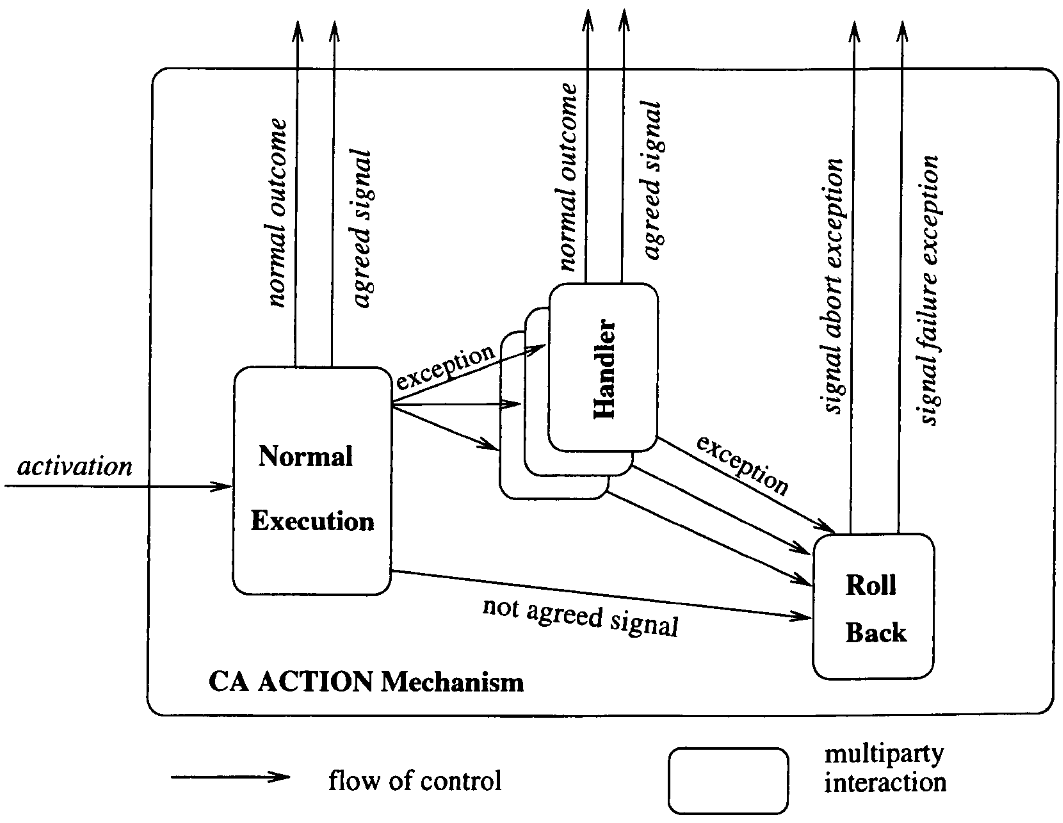

Figure 6.21: CA Action as a DMI

Figure 6.21 shows how we can structure the CA action mechanism using the DMIs proposed in this thesis. This figure is a more restrictive version of Figure 4.1 shown in Chapter 4. As mentioned in Chapter 4, the key idea for handling concurrent exceptions is to build a DMI out of basic multiparty interactions by chaining them together appropriately, so that each basic multiparty interaction in the chain is the exception handler for the previous basic multiparty interaction in the chain. As we can see in Figure 6.21, CA actions are implemented as a DMI (or DIP team) with the following structure: the main body of the DMI performs the normal execution of the CA action; a set of DMI handlers deal with the exceptions that may be raised during the normal execution of the CA action; and a special handler that deals with the rolling back process of the CA action.

The order in which the main body and the handlers shown in Figure 6.21 are executed is driven by the following CA action rules:

- *i*) Each role in a CA action may terminate normally or exceptionally by signalling an exception to the enclosing CA action. The roles should agree about the outcome of the action. However, a role may also signal an abort exception or a failure exception to indicate that the action should abort or fail (see (*vii*)). In the figure this rule is represented by the *normal outcome* and *agreed signal* lines.

- *ii*) If the roles do not agree about the outcome, then the underlying CA action support mechanism attempts to abort the action by undoing its effects on external objects. This rule is represented in the figure by the *not agreed signal* line from the normal execution interaction to the roll back interaction.

- *iii*) If the abort is successful, then an abort exception is signalled to the enclosing CA action; otherwise, a failure exception is signalled. See *signal abort exception* and *signal failure exception* lines in Figure 6.21.

- *iv*) If an exception is raised during the normal execution of a CA action, then control is passed to the corresponding exception handler for each role. If two or more exceptions are raised concurrently, then a process of exception resolution must take place first.

- *v*) If an exception is raised during the execution of an exception handler, then the underlying CA action support mechanism will attempt to abort the action (see (*iii*)) or else signal a failure exception to the enclosing action.

- *vi*) Once exception handling begins within a CA action, it is not possible to resume normal execution of the CA action but it is possible for the exception handlers to terminate normally or exceptionally, depending on the extent to which error recovery is successful. However, all roles must still agree about the outcome (see (*ii*)).

- *vii*) A role may signal abort or failure at any time to indicate that error recovery is not possible and the action must abort or fail. For the purposes of determining the outcome, failure takes precedence over abort which takes precedence over every other exception that can be raised internally.

- *viii*) For a given action, exception handlers are provided only for exceptions that are raised internally within that action. Exceptions that are signalled by an action are handled at the level of the enclosing action.

Thus, an action cannot provide an exception handler for its own abort or failure exceptions.

- *ix*) If an action terminates by signalling an exception to its enclosing action, then this triggers the process of exception handling in that action (see (*v*)).

6.2.2 Java Implementation

The semantics for handling exceptions presented in the previous section has been applied to our API in order to build CA actions (no change to the API has been made). In Figure 6.22, we show how three sets of managers would be created and linked together to respect the CA action semantics. The rollback roles controlled by the three first managers (mgr1-RB, mgr2-RB and mgr3-RB) are responsible for bringing the external objects to the state they had before the CA action had started. Note that there is no exception handling list for roles of these managers. Any exception that is raised during the rolling back interaction will be mapped to a *failure* exception and passed to the enclosing context. The second set of managers (mgr1-E1, mgr2-E1 and mgr3-E1) control the roles that will deal with exception E1 that may be raised in the *normal execution* interaction of a CA action. If any exception is raised during the interaction that deals with exception E1, then the roll-back interaction will be activated (rb1, rb2 and rb3 lists are passed to the managers of the handling interaction for E1, and these lists contain links to the roles of the roll-back interaction). The third set of managers (mgr1, mgr2 and mgr3) control the set of roles responsible for the normal execution of the CA action. If exception E1 is raised during the normal execution interaction, then the exception handling interaction for E1 is activated. If any other exception is raised during the normal execution interaction, then the roll-back interaction is activated (the e11, e12 and e13 lists contain links to the roles of the handling interaction for E1, and the roles of the roll-back interaction).

```
// set of managers for the roll-back interaction
Manager mgrRB1 = new Manager("mgr1-RB","CA", null, null);
Manager mgrRB2 = new Manager("mgr2-RB","CA",null,mgrRB1);
Manager mgrRB3 = new Manager("mgr3-RB","CA",null,mgrRB1);
// declaration of roles for the roll-back interaction
// and the rb1, rb2, rb3 hashtables
:
// set of managers to deal with E1 exception
Manager mgrE11 = new Manager("mgr1-E1","CA",rb1, null);
Manager mgrE12 = new Manager("mgr2-E1","CA",rb2,mgrE11);
Manager mgrE13 = new Manager("mgr3-E1","CA",rb3,mgrE11);
// declaration of roles for the handler interaction
// for E1 and the e11, e12, e13 hashtables
:
// set of managers for the normal execution
// of the DMI
Manager mgr1 = new Manager("mgr1","CA",e11, null);
Manager mgr2 = new Manager("mgr2","CA",e12,mgr1);
Manager mgr3 = new Manager("mgr3","CA",e13,mgr1);
```

Figure 6.22: Manager Objects Instantiation for a CA Action

6.2.3 Discussion

The main idea of describing CA actions in terms of DMI, and implementing them using the framework presented in Chapter 5, was to show how DMI can be used to implement different multiparty interaction abstractions. We have chosen to implement the CA action mechanism using our approach because it provides strong exception handling semantics for dealing with possible failures that happen during the execution of an interaction. Multiparty interaction mechanisms like *actions* in DisCo [Jårvinen & Kurki-Suonio 1991] or *teams* in IP [Forman & Nissen 1996] do not consider exceptions in the execution of an interaction, let alone any description of the semantics for dealing with exception that can happen during the execution of an interaction. Therefore, using our approach to implement such mechanisms would have been a trivial task.

6.3 Case Study: GAMMA Computation

GAMMA is a model of parallel computation based on the idea of multiset transformations. It behaves in a similar way to a chemical reaction upon a collection of individual pieces of data [Banatre & Métayer 1993]. Each step of a GAMMA computation involves selecting a set of values from the multiset and then combining them in some way to produce a new set of values. A distributed GAMMA model has also been proposed [DiMarzo & Guelfi 1998]. Its main novelties are distribution of multisets (each of them is presented as a set of local multisets), and distribution of chemical reactions. For example, the following GAMMA program performs the sum of a set of integers, i.e. it is always true that for every pair x,y in the multiset, the pair is replaced by x+y:

```
add:  x,y → x+y ⇐ true
```

6.3.1 Design of a GAMMA System

To demonstrate how distributed GAMMA computations can be implemented using DMIs, a simple example is used in this section. In this example, numbers from distributed multisets are summed and the result is stored in a multiset. The distributed GAMMA system is composed of a set of participants (located on different hosts), a scheduler (located on a separate computer) and a set of DMIs, each of them called Gamma. (This approach has also been used to demonstrate how CA actions could be used for implementing GAMMA computation [Romanovsky & Zorzo 1999] [DiMarzo *et al.* 1999].) The design has two levels. The first level is concerned with information exchange between computers (participants and a scheduler). This is the level on which the execution of the Gamma DMIs is scheduled (or the DMIs are glued together). At the second level of the design, each Gamma DMI performs a single step of the GAMMA computation by coordinating the interactions between roles and their access to external objects (multisets). On this level numbers are passed between different local multisets, summed, and the result is inserted into the multisets. As shown in Figure 6.23, each Gamma DMI has three roles: two producers (each of which supplies a number from its local multiset) and a consumer (which computes the sum of these numbers and stores the result in its local multiset).

The participants in the Gamma DMIs correspond to the computing resources available to perform the GAMMA computation. A participant starts when it is loaded into a client computer and establishes a connection with the scheduler. Each participant has a local multiset, i.e. a queue in which some

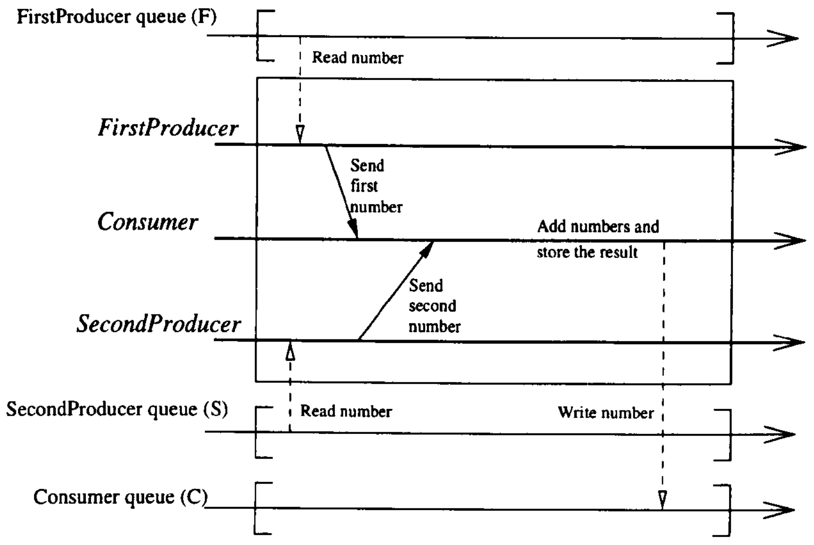

Figure 6.23: The Gamma DMI

part of the global multiset is kept. Each participant informs the scheduler when it receives a new number in its local multiset. The scheduler starts a new Gamma DMI whenever there are at least two new numbers available in local multisets. There can be as many Gamma DMIs active concurrently as there are pairs in all local multisets at a given time. Each participant creates a new thread to execute a role in a Gamma DMI and in this way it is possible for a participant to be involved in several Gamma DMIs at the same time without violating the atomicity property (for example, if there are several numbers available in its local multiset). This allows a better parallelisation of the GAMMA computation.

The table below shows the preconditions that must hold before the Gamma DMI starts its execution. The same table also shows the post-conditions for the normal outcome of the Gamma DMI (F, S, and C represent the local multisets of the participants; and x and y numbers from the F and S local multisets, respectively).

preconditions	*post-conditions*
F in FirstProducer	F - $\{x\}$ in FirstProducer
S in SecondProducer	S - $\{y\}$ in SecondProducer
C in Consumer	C + $\{x+y\}$ in Consumer

Figure 6.24: The Gamma DMI Pre and Post-conditions

6.3.2 Fault Tolerance in the GAMMA System

The original GAMMA paradigm [Banatre & Métayer 1993] assumed that there are no faults in the system. Fault tolerance requirements have been included in the system in order to demonstrate how DMIs can be used increase dependability and to make the system more realistic. The fault assumptions for this case study are: *i*) the addition operation can fail; *ii*) reading numbers from the multisets can fail; and, *iii*) writing numbers to the multisets can fail. Because these operations are executed only inside DMIs, the entire system fault tolerance is provided within the DMI framework. In particular, we assume that nodes and channels are reliable (i.e. their fault tolerance, if required, is implemented transparently for our system by the underlying support). For example, in our system, when the addition operation fails, a predefined exception ReactionException is raised in the thread executing a role in the DMI (see Figure 6.25).

After an exception ReactionException has been raised, the DMI support mechanism interrupts all the roles in the DMI and calls the handler for this exception (see Figure 6.25). Our design decision is to use forward error

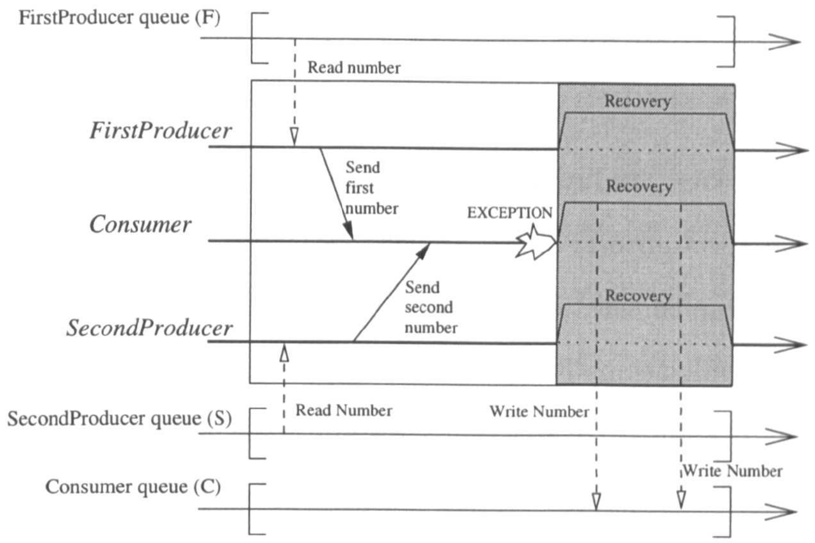

Figure 6.25: Forward Error Recovery in the Gamma DMI

recovery in the DMI in the following way: when the reaction fails, the consumer keeps both numbers by inserting them into its local multiset whilst the producers complete the DMI as if nothing has happened. Thus, if a fault happens during the action execution, the consumer recovers the system, but in this case there are two new numbers in the consumer's local multiset. We use a special outcome of Gamma to inform the scheduler about these new numbers. The table below shows the exceptional outcome post-conditions.

exceptional post-conditions
F - $\{x\}$ in FirstProducer
S - $\{y\}$ in SecondProducer
$C + \{x\} + \{y\}$ in Consumer

Figure 6.26: The Gamma DMI Exceptional Post-conditions

Gamma DMIs are atomic with respect to faults in the chemical reaction: the exception handlers guarantee "nothing" semantics for the global multiset (although the local multisets are modified during this recovery).

6.3.3 Discussion

The previous case studies dealt with designing a complex distributed control system (Section 6.1) and a multiparty interaction abstraction (Section 6.2). Their implementation showed that DMIs could also be used for implementing applications in such a way that maximum concurrency (parallelism) could be provided together with guaranteeing consistency of system state. Gamma computation is an example of such parallel computational model. Similar models are used in several languages and formalisms. In [Banatre & Métayer 1996], a number of languages and formalisms bearing similarities with the Gamma computation are surveyed.

Based on the example presented in this section, one could implement any of the models shown in [Banatre & Métayer 1996] using DMIs, e.g. Unity, Linda, Linear Objects. The same is valid for several extensions of the basic version of Gamma (such as schedules and local linear logic - further discussion about this topic can be found in [Banatre & Métayer 1996]). The common requirements for all of those models are: achieving maximum parallelism of basic operations, guaranteeing consistency of data accessed by these operations and atomicity of these operations. These are the type of properties DMIs provide.

In general, we would claim, based on these various case studies, that DMIs is a concept that can be employed in many different applications in which parts of systems can either compete for resources or cooperate to achieve some joint purposes.

6.4 Experience using DMIs in the Case Studies

The experiments presented in this chapter demonstrate that DMIs can be used as safe atomic building blocks for programming safety-critical systems and other programming abstractions. The use of DMIs greatly facilitated our ability to guarantee that our design and implementation satisfied the safety and fault tolerance requirements of the FZI Production Cell II case study. The resulting system has a clean design that was easy to understand and validate. In fact, formal validation [Canver *et al.* 1998] of the design approach used in this thesis has been performed for a virtually complete implementation of the FZI Production Cell II using CA actions. This formal validation showed that the design is correct and guarantees the liveness and the safety requirements of the case study.

DMIs can be used as a general abstraction technique for structuring complex concurrent systems. Although a design based on DMIs can decrease the overall performance for some systems, we believe that the benefits gained by a simple design, built using re-usable components, and providing system fault tolerance in a disciplined way, etc. are more important for safety-critical applications.

Conventional design approaches do not usually support the structuring of concurrent system dynamic behaviour and do not have any features for describing both component cooperation and competition in complex systems. Systems built using DMIs have a very different design from systems built without DMIs. Using conventional techniques, component cooperation is usually flat and consists only of component synchronisation (e.g. message passing). With our approach, cooperation can have several levels. See, for example, the design of the control software for the FZI Production Cell II, in which on one level we have the controllers cooperating to achieve better performance, and on a second level, the devices cooperating to achieve safety when passing plates. In each level, part of the fault tolerance requirements were satisfied.

Compared with conventional methods, there are obviously additional (mainly performance) overheads for using DMIs: additional messages and increased overall system synchronisation. It would be possible to design a faster system by taking advantage of some low level knowledge of the application and by breaking the information and concurrency encapsulation imposed by DMIs. However, this would be at the cost of simplicity, maintainability and safety.

Chapter 7

Conclusions

In this thesis, we have surveyed a set of languages that allow multiparty interactions and exception handling as basic constructs. Despite all the work in multiparty interactions and exception handling, none of the languages surveyed provide the integration of both concepts in a single mechanism. Therefore, the first major contribution of this thesis is the integration of multiparty interaction and exception handling in such a single mechanism called *dependable multiparty interaction* (DMI), and the embedding of this mechanism in a programming language. Furthermore, none of the languages surveyed provides a structured way of handling concurrent exceptions. The mechanism proposed in this thesis thus provides a structured way of dealing with concurrent exceptions when inside a multiparty interaction.

The second major contribution of this thesis relates to the way DMIs can be implemented in object-oriented languages like C++, Java, or Eiffel. Indeed, a complete object-oriented framework was implemented in Java.

Other small contributions, such as the application of the DMI concept to implement safety-critical systems, or different computation abstraction, are included in this thesis.

7.1 Summary

In Chapter 2, a complete state-of-the-art survey about multiparty interactions has been presented. It includes the main requirements and design choices made in several languages that provide multiparty interaction mechanisms. A taxonomy based on the way participants can enroll in an interaction. In addition, a safety-critical system is designed using one of the languages that provide multiparty interactions.

Exception handling is fully discussed in Chapter 3. First some basic definitions are presented. The whole chapter is devoted to show how exception handling is implemented in programming languages. Furthermore, the chapter shows that concurrent exception handling is provided in different models, but not in programming languages.

The concept of *dependable multiparty interactions* (DMIs) is introduced in Chapter 4. The key idea to build DMIs, is to use unreliable multiparty interactions and chain them together, so that each multiparty interaction in the chain is the exception handler for the previous multiparty interaction in the chain. Also important is the enclosing of the unreliable multiparty interaction to protect them from being interfered with during their normal execution or exception handling. A full language that has DMIs as a basic construct is presented in the same chapter, i.e. Dependable Interaction Processes (DIP). Formal semantics of part of this language is also provided using

TLA$^+$.

Although DIP can be used to design, or implement, a computer system, there are a lot of object oriented languages that are used for designing and implementing the same systems. Chapter 5 has presented different approaches for implementing DMIs in an object-oriented language: Java. Even though the framework presented imposes an overhead burden in the final system, depending on the application, the burden will be negligible, e.g. for a production cell case study it incur only 0.0164 percent of overhead for a 2-party interaction.

The last chapter presented three case studies. DMIs were applied to implement them. The first system is a safety-critical production cell, which involves several devices and represent a real industrial installation. The second and third case studies showed how DMIs can be used to implement different computational abstractions.

7.2 Future Work

The research presented in this thesis is far from being complete. Actually, it has opened the horizon to several research topics:

- DIP as a new language needs a compiler and a runtime system. Both compilation and execution of DMIs have yet to be studied. Although there has been some work on scheduling of multiparty interactions [Joung & Smolka 1996] [Forman & Nissen 1996], further issues arise with the introduction of exception handling in multiparty interactions.

- The framework presented in this thesis uses a simple transaction system. Studies on the use of a real transaction system have to be taken.
- Formal specification of the whole DIP language. At the end of the chapter in which DIP was introduced, a formal semantics for DMIs was presented. Although it provides a clear specification of major aspects of DMIs in DIP, there is still some issues to be formally specified, such as the interruption of nested DMIs.
- The case studies presented in the last chapter showed that DMIs can be used to implement those systems. Once a compiler and runtime system are ready for DIP, performance issues have to considered, so all the case studies have to be tested using the compiler and runtime system provided.

Appendix A

Multiparty Interaction Mechanisms - A Comparison using the Dining Philosophers Problem

The dining philosophers problem was introduced [Dijkstra 1965] to show how to achieve synchronisation[1] when a process needs more than one resource to execute an activity, e.g., one philosopher needs two forks to start the operation eating. Note that in the original specification of the problem only one process would start the operation. For our purpose, i.e. multiparty interaction, we will consider forks as being processes rather then resources,

[1]Although synchronisation is defined as "cause to occur at the same time; be simultaneous; coordinate, combine", "coordinate, combine" is considered wrong by several people (*The Concise Oxford Dictionary*) and should be avoided, so we will not use it in that sense.

and an eating activity will only happen when the 3-parties are ready to execute it.

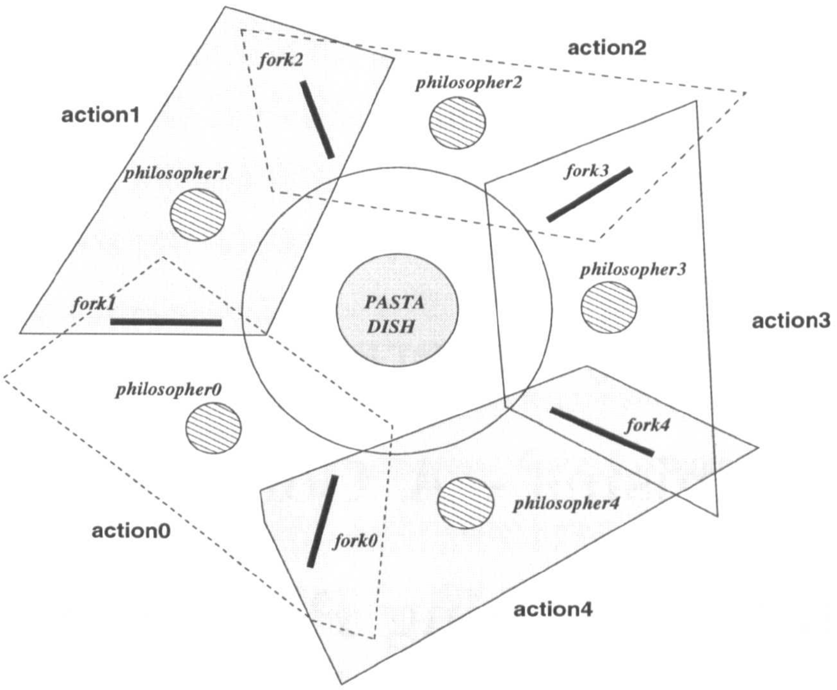

Figure A.1: Dining Philosophers Problem

Definition: Let us consider a set of 10 processes divided in two groups of 5 processes: *forks* and *philosophers*, where *forks*= $\{fork_0, ..., fork_4\}$ and *philosophers*= $\{philosopher_0, ..., philosopher_4\}$. These processes will, during the execution of the system, execute joint activities, called ***actions***= $\{ action_0, ..., action_4 \}$ that will be composed of 2 *fork* components and 1 *philosopher* component. Each component of *actions* is a tuple in the format $action_i = \{ fork_i, philosopher_i, fork_{i+1})$, where $0 \leq i \leq 4$, and $i + 1$ is equivalent to $(i + 1) mod 5$. Let us also consider that each $action_i$ will need an external data called *pastaDish* in order to execute the joint activity, and that such external data can be accessed only by one action at a time. To better understanding of the problem see Figure A.1.

Every *$action_i$* will execute a generic activity called *EatingAction* = (*leftfork, philosopher, rightfork*). This activity is executed in a coordinated[2] way by the three participant processes. Firstly, the *philosopher* and the *leftfork* will synchronise their execution, and such synchronisation will represent the philosopher grabbing the left fork. A similar synchronisation will occur later representing the philosopher grabbing the right fork. After those, both forks will synchronise their execution in order to access the external data *pastaDish*. Such execution will consume part of the external data. When the *leftfork* and the *rightfork* have got the "pasta", i.e. part of the external data, the three participants will synchronise their effort to "eat the pasta", i.e. consume the data. See Figure A.2.

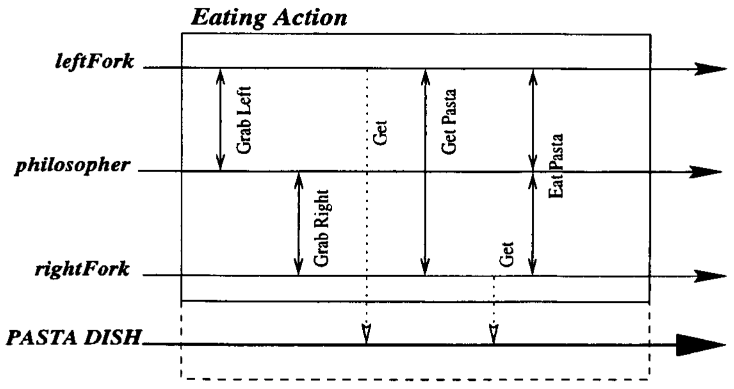

Figure A.2: Eating Action

In the next sections, parts of this case study will be used to show properties of the multiparty interaction mechanisms in several languages. An

[2] *The Concise Oxford Dictionary* defines coordinate as "Bring (various parts, movements, etc.) into a proper or required relation to ensure harmony or effective operation etc.; work or act together effectively".

implementation of this case study using the DIP language (Chapter 4) and the framework presented in Chapter 5 is described in Appendix E.

A.1 IP

Interacting Processes (IP) [Forman & Nissen 1996] is an (executable) specification language for the expression of distributed programs (IP is executable if failure-free hardware and software system is assumed, since a failure can negate the atomicity of IP's multiparty interaction). The main ideas of IP come from other two languages, Script [Francez *et al.* 1986] and Raddle [Forman 1986].

IP core language

A program in IP is a sequence of processes in the format of:

$$\mathrm{P} :: [P_1 \parallel P_2 \parallel \cdots \parallel P_n] \text{ or } [_{i=1,n}\parallel P_i]$$

with $n \geq 1$ processes which have disjoint local states, i.e. they do not share variables. Processes can share/interchange information through inter-processes communication, which is achieved in IP via a multiparty interaction.

P_i, $1 \leq i \leq n$, also has a statement S, called body. S can be composed of statements of the forms:

- *Skip*: has no effect on the state

- *Assignment*: $x := e$, where x is local do P_i and e is an expression over the local state of P_i.

- *Interaction*: $a[\overline{v} := \overline{e}]$, where a is the interaction name, and $a[\overline{v} := \overline{e}]$ is an interaction part containing an optional parallel assignment. P_i is a participant of the interaction a. All variables in $\overline{v}$ belong to the local state of P_i, and e may involve variables not local to P_i, i.e. e can contain any variable from any local state of any participant of interaction a. The interaction a is **enabled** only when all of its participants have arrived at a point where executing a is one of the possible continuations (when P_i arrives in that point, it is said to ready a). When a local interaction part is finished, the participant process can resume its local thread of control, i.e. no synchronisation at the end of an interaction is assumed.

- *Sequential composition*: $S_1; S_2$: first S_1 is executed; when it terminates, S_2 is executed.

- *Nondeterministic selection* $[[]_{k=1}^{m} B_k \& a_k[\ \overline{v} := \overline{e}] \rightarrow S_k]$: where $B_k \& a_k[\overline{v} := \overline{e}]$ is a guard. B_k is a boolean expression, called **guarding predicate**, over the local state of participant P_i. The $a_k[\ \overline{v} := \overline{e}]$ part is called **guarding interaction**. S_k is any statement. The guard is enabled when the guarding predicate is passable, i.e. its evaluation is true, and when the guarding interaction is enabled. During the execution of the guard, first all boolean expressions are evaluated, if one of them is true then an interaction is a possible continuation. If an enabled guard is passed then S_k is executed. If no guard is enabled, the execution is blocked until some guard is enabled. Note that m is fixed and cannot be changed during execution.

- *Nondeterministic interaction* $*[[]_{k=1}^{m} B_k \& a_k[\ \overline{v} := \overline{e}] \rightarrow S_k]$: this is similar to nondeterministic selection, but the whole statement is repeated until none of the guards is passable.

Teams in IP

A program in IP is composed by a set of *teams declaration.* The execution of a program starts by a single instantiation of a team followed by an execution of the instantiated team. When declaring a team, the following items are specified:

- A unique team name.
- Team parameters and their respective types.
- A prologue used as the initialisation section of the team.
- A body, composed by concurrent *roles* and *processes.* Processes are the executing agents in a team, while roles are formal processes, to which processes may *enrole* in order to activate their computations.

After instantiating a team, the statements in its prologue are executed, and then all its processes start execution. We can compare an instantiation of a team with an instantiation of an object, where a unique **team instance identifier** is returned. A process is local to a team and no one can interfere in its computation, unless they are participating in the same interaction. A role can be parameterised, and is composed of a statement, which is basically of the core language. In order to activate a role, some role or process in another team has to enrole in that role.

```
team EatingAction(value i:integer)::
[
  role philosopher(value pasta)::
     getForks_i[s_i := 'e']&consume[ pastaDish := pastaDish - 2 ]
     → giveForks_i[]
||
  role leftFork(result x)::
     getForks_i[ ]&consume[ x := pasta ] → giveForks_i []
||
  role rightFork(result y)::
     getForks_i[ ]&consume[ x := pasta ] → giveForks_i []
]
```

```
PHIL :: [P0 || ··· || P4 || F0 || ··· || F4]

action : array[k:0..4] of team designator := new EatingAction(k);

P_i :: s_i := 't';
  *[s_i='t' → s_i := 'h'
    []
    s_i='h'&philosopher(pasta)@action(i) → s_i := 't'
  ]
F_i ::*[ leftFork@action(i)
     []
       rightFork@action(i+1)
     ]
```

The above IP code shows how the dining philosopher problem could be implemented. In the example, forks are represented by processes F_i. The state of the philosophers is represented by the variable s_i. When the *ith* philosopher P_i is hungry (s_i = 'h') then it enroles in action$_i$ playing the role philosopher. In the team action$_i$ the philosopher will interact in a three-party interaction getForks$_i$ (together with F_i and F_{i+1}). After that, the pasta dish will be consumed in the three-party interaction consume. After the consume interaction is finished, the three processes leave the action team, and the state of the philosopher P_i becomes thinking again (s_i = 't'). Atomicity in the access to the pasta dish is not represented in the above example. (In the example i+1 = ((i+1) mod 5).)

A.2 Action Systems

Action Systems [Back & Kurki-Suonio 1988] provide a method to program distributed systems. In Action Systems processes can interact through actions, called *joint actions.* All processes that are participating in a joint action will be synchronised by a common handshake.

Basically, an action system consists of a collection of processes and a collection of actions in the following format:

```
process p: var
y_a; statement_p.

action a by processes_a:guard_a → statement_a.
```

where y_a is the set of variables of process p, while $statement_p$ assigns initial values to these variables. The action statement indicates that a may be executed jointly by the processes in the set $processes_a$, provided that $guard_a$ is satisfied and the processes are not participating in other action. The program state is changed in accordance with $statement_a$, and it is called *body* of the action. The body of the action can only refer to local variables of the processes participating in the action.

```
process Philosopher[i:0..4];
 var hungry : boolean;
   hungry := false;

process fork[i:0..4];

action Thinking[i:0..4] by Philosopher[i]
¬Philosopher[i].hungry →
 // thinking
 Philosopher[i].hungry := true;

action EatingAction[i:0..4] by Philosopher[i],fork[i],fork[(i+1) mod 5],pasta
 Philosopher[i].hungry →
 // eating pasta dish
 Philosopher[i].hungry := false;
```

The previous code shows an example of a set of actions and processes in Action Systems. The dining philosophers case study in Action Systems is composed of 5 philosophers, 5 forks, and two kinds of actions. First, a private action Thinking[i] is executed by the philosopher[i] when the philosopher is not hungry. The action execution consist of a non-specified thinking being done, after which the philosopher becomes hungry. The second action EatingAction, a joint action, will be executed when a philosopher is hungry, and both forks are not being used in another action. At the end of EatingAction the state of the philosophers becomes not hungry. An external data, e.g. the pasta dish we want to represent in the interaction, has to be defined as a participant of the action, thus guaranteeing that it will be used in only one action.

A.3 Ada

In the rendezvous model of Ada one task, the *server* task, declares a set of services that will be used by other tasks, the *client* tasks [Burns & Wellings 1995]. The *server* task does this by declaring one or more public **entries** in its task specification. Each entry identifies the name of the service, the parameters that are required with the request and the results that will be returned. The *client* task issues an "entry call" on the *server* task by identifying both the *server* task and the required entry. For the interaction to occur between the *server* and the *client*, both tasks must have issued their respective requests (this is the rendezvous). When the rendezvous takes place, any in or out parameters are passed to the server task by the client. The server task then executes the code inside the accept statement. When this statement finishes both tasks proceed independently and concurrently.

```
task body Philosopher(P:0..4) is
   chopLeft, chopRight : 0..4;
begin
   chopLeft := P; chopRight := (P+1) mod 5;
   loop
   // thinking
     forks(chopLeft).PickUp;
     forks(chopRight).PickUp;
     // eating action
     forks(chopLeft).PutDown;
     forks(chopRight).PutDown;
   end loop;
end Philosopher;
```

In the above example, we do not create an action to represent the eating action as discussed in the beginning of this appendix. Instead, we represent

```
task body fork is
begin
  loop
    accept PickUp;
    accept PutDown;
  end loop;
end fork;

forks : array (0..4) of fork;
philosophers : array (0..4) of Philosopher;
```

the eating action inside the body of the philosopher task. In the above code, task **fork** represents a server while task **Philosopher** represents a client that issues an entry call. A server, i.e. a fork, is ready to accept either a **PickUp** or a **PutDown** entry. A philosopher tries to pick up both forks, representing the synchronisation between the philosopher and the forks, before it starts the eating action. We did not represent the pasta dish in the above example.

A.4 CSP

Communicating Sequential Processes (CSP) [Hoare 1985] is a distributed language that relies on a channel mechanism for interaction between two processes. The interaction is established via the execution of "complementary" pairs of input and output command. For example, the input command $P_i?y$ of processes P_j, which inputs a value from process P_i into variable y, is complementary to the output command $P_j!x$ of process P_i, which outputs the value of expression x to P_j. The joint execution of these commands is equivalent to the assignment of x to y ($y \leftarrow x$). The following code shows how synchronisation between the processes of the dining philosopher problem could be implemented in CSP.

```
PHIL =
  *[ ... →
     // thinking
     fork(i)!enter();
     fork((i+1) mod 5)!enter();
     // eating action
     fork(i)!leave();
     fork((i+1) mod 5)!leave();
  ]

fork =
  *[ phil(i)?enter() → phil(i)?leave()
    []
    fork((i+1) mod 5)?enter()→ fork((i+1) mod 5)?leave()
  ]
```

A.5 LOTOS

LOTOS - Language Of Temporal Ordering Specifications [Brinksa 1988] is a specification language for distributed systems. It is based on an extended version of Milner's calculus of communicating systems (CCS) and it is also influenced by the related work on CSP. In LOTOS, interprocess communication occurs by means of gates. When processes want to interact, each one of them specifies the name of a gate and whatever information it wishes to provide on the values to be established. Processes can participate in an interaction only if, at the same time, they are ready for the interaction where they specify the same gate and compatible types. The following code shows how synchronisation between the processes of the dining philosphers problem could be achieved before executing the eating action.

```
process DiningPhilosophers : noexit :=
hide l0,l1,l2,l3,l4,r0,r1,r2,r3,r4 in
  P[l0,r0]—[]—F[l1,r0]—[]—P[l1,r1]—[]—F[l2,r1]—[]—P[l2,r2]—[]—
  F[l3,r2]—[]—P[l3,r3]—[]—F[l4,r3]—[]—P[l4,r4]—[]—F[l0,r4]
where
  process P[left, right] : noexit :=
        (* think *)
        left; (* grab left *)
        right; (* grab right *)
        (* eat *)
        right; (* release right *)
        left; (* release left *)
        P[left, right]
  endproc (* Philosopher *)

  process F[left, right] : noexit :=
        left; (* grabbed by left philosopher *)
        left; (* release by left philospher *)
        F[left, right]
     []
        right; (* grabbed by right philosopher *)
        right; (* released by right philospher *)
        F[left, right]
  endproc (* Fork *)

endproc (* Dining Philosophers *)
```

A.6 Multiway Rendezvous

The *multiway rendezvous* is a generalisation of the rendezvous in which more that two processes may participate [Charlesworth 1987]. The central idea of the multiway rendezvous is that a block of statements is executed when, and only when, a specified number of processes reach a corresponding stage of processing.

The previous code shows the *compact* construct for the dining philosophers problem. The system is composed of 5 philosophers and 5 forks. Every

```
declare
  EATING: array (i:0..4) of compact is
    acquire request P(i:0..4); C(i); C((i+1) mod 5);
      ... eating action
  PHILOSOPHER: array (i:0..4) of process is
    declare hungry: BOOLEAN;
    begin
      hungry := false;
      loop
        if hungry then
          EATING[i].P(i);
          hungry := false;
        else
          ... thinking
          hungry := true;
      end loop;
end PHILOSOPHER;
fork: array (i: 0..4) of process is
begin
    loop
      if (i mod 2) = 0 then
        EATING[i].C(i); EATING[(i+1) mod 5].C(i);
      else
        EATING[(i+1) mod 5].C(i); EATING[i].C(i);
    end loop;
end;
begin
  cobegin PHILOSOPHER(0) // ... // PHILOSOPHER(4)
                    // fork(0) // ... // fork(4)
  coend;
end;
```

fork is ready to join into an interaction all the time. In order to avoid starvation or deadlock, $fork_i$, where i is an even number, enables interaction i first, while $fork_i$, where i is an odd number, enables interaction *((i+1) mod 5)* first. The interaction will occur only when two forks are available and the philosopher is hungry. An interaction is expressed in the **compact** construct and the participants of such interaction are specified in the **acquire request** statement.

A.7 SR

The Synchronizing Resources (SR) language [Andrews *et al.* 1988] was designed for programming distributed systems. Its programs are composed of three kinds of components: *resource specifications*, *resource bodies*, and *globals*. Resources are the main building blocks; they are the unit of abstraction and encapsulation. Globals contain declarations of constants and types shared by resources.

Resources define operations and are implemented by one or more processes that execute in the same address space. Processes interact by means of operations; processes in the same resource may also share variables. A resource has a *specification* and a *body*. The specification identifies other components the resource uses and declares operations, constants, and types exported by the resource. The body contains the processes that implement the resource, declarations of objects shared by those processes, and initialisation and finalisation code.

SR allows both conjunctive parallelism (IN operation []..[] operation NI) and disjunctive parallelism (CO concurrent_invocation ||... || ... OC). An operation typically involves argument passing between the caller and the callee, which can be by value, result, value/result, or reference. Care must be taken when the same variable is passed twice within a CO statement by reference. For example, suppose two operations both pass a reference to

```
resource Philosopher
body Philosopher ( forkRight , forkLeft : cap fork )
 process phil
  do true →
   // think
   send forkLeft.left()
   send forkRight.right()
   // eating action
  od
 end
end Philosopher

resource Fork
body Fork
 process fork
  do true →
   in left() →
      // eating Action as left fork
     []
     right () →
      // eating Action as right fork
   ni
  od
 end
end Fork
```

variable x to the corresponding callees, which in turn modify the content of x. Then, the final outcome of the operations depends on their relative execution speeds, since the actual value of x seen by the callee may be the original value or the value as modified by the other callee.

A.8 SC++

Synchronous C++ (SC++) [Petitpierre 1998] is an extension to C++ that provides it with features to develop concurrent applications within an object-oriented language. SC++ contains active objects that have their own thread of control, and communicate with each other by means of synchronising method calls.

```
active class Philosopher {
  int i;
  @Philosopher() {
    for (;;) {
      // thinking
      chop[i].get();
      chop[(i+1)%5].get();
      // eating action
      chop[i].put();
      chop[(i+1)%5].put();
    }
  }
}

active class fork{
  bool free = true;
public :
   void get() {  free = false; // eating action }
   void put() {  free = true; }
   @fork() {
     for (;;) {
       if free accept get;
       else accept put;
     }
   }
}
```

In the previous example, we showed how to implement the philosophers and forks processes of the dining philosophers problem. Processes are active objects in SC++ and are defined as classes. SC++ uses a rendezvous mechanism to attend the requests to its methods. The rendezvous is represented by the command **accept** that is coded inside the objects main body (the main body of an active object is described using the class name and the prefix @ - see the previous code).

The eating action is described in three different places in the following example. The philosopher class will host the code for the philosopher role in the eating action, and each fork will host the code for the fork role in the eating action.

A.9 Discussion

In this appendix, we have presented (the description of) several languages that have a multiparty interaction as a basic construct. We have used the dining philosphers problem to show how the multiparty interaction construct could be used in these languages. Even though the example is very simple, we can draw several conclusions from the use of the multiparty interaction mechanism in these languages. For example, while all the languages provide means of synchronisation upon entry, only a few provide some kind of pre-condition in the multiparty interaction, e.g. Action Systems and DisCo. In addition, languages that provide split bodies in the multiparty interaction, e.g. IP, provide a better structuring mechanism than those that do not, e.g. Action Systems, Ada, Multiway Rendezvous, and DisCo. In some languages, the multiparty interaction mechanism does not provide split bodies, and the mechanism is used only for synchronisation and communication, e.g. CSP, LOTOS and SR. Split bodies in those languages can be simulated by using several of their multiparty interaction mechanisms, e.g. Ada, SC++, SR, LOTOS, and CSP.

Appendix B

Exception Handling - Survey

B.1 Exception Handling in PL/I

PL/I was the first programming language had special constructs for exception handling. PL/I allows the programmer to write exception handler for a list of pre-defined exceptions, and also for user-defined exceptions.

Exception handlers are declared by **ON** statements. Handlers in PL/I are divided in three types. The first type, which includes **SUBSCRIPTRANGE, SIZE**, and **CHECK**, are ignored unless an **ON** statement has been set. The second type, which includes **OVERFLOW, ZERODIVIDE**, and **CONVERSION**, are normally enabled but can be disabled by a **NO** prefix, e.g **NOOVERFLOW**. The third type, which includes **ENDFILE, UNDEFINEDFILE**, and **KEY**, are always enabled and cannot be disabled.

An **ON** statement have the following form:

```
ON ZERODIVIDE BEGIN;
   statements
END;

ON OVERFLOW;
```

The *statements* represent the code of the exception handler. If programmers want to ignore an exception occurrence, they can include a statement like the OVERFLOW above. In that case the exception handler is a null statement, therefore the exception is ignored.

An exception is raised by the statement: **SIGNAL CONDITION** *exception_name*;

Exceptions in PL/I are bound to exception handlers dynamically. When an **ON** statement is encountered during the execution, a new binding takes place. This binding will remain valid until it is overridden by the execution of another **ON** statement for the same exception, or until the block, in which the **ON** statement was declared, terminates.

In PL/I the place in the program the flow of control continues after an exception handler has completed its execution depends on the built-in exception handlers. For some exceptions, execution continues with the statement that caused the exception; for other exceptions the program terminates. Handlers written by programmers can continue wherever the programmer wants to using a **GOTO** statement, but there is no mechanism that provides the address of the statement that raised the exception.

B.2 Exception Handling in CLU

In CLU, exceptions are associated with procedures. Only procedures can raise exceptions and the handler is declared in the calling unit. A procedure cannot handle the exceptions it raises.

Exception handlers can be bound to any statement in CLU. When bound to a statement, they handle only exceptions raised by routines called by that statement. Exception handlers in CLU are declared as:

```
statement
   except
      when exception_list_1: statement_1
      ...
      when exception_list_n: statement_n
   end
```

If the execution of a procedure call in ***statement*** raises an exception then the control is transfered to the **except** clause. If the raised exception belongs to *exception_list*$_i$, then the ***statement***$_i$ is executed. If the statement in which the exception was raised does not have an handler for that exception, then the exception is propagated to a larger static scope within the procedure. If no handler is found, then the built-in exception **failure** is raised and control returns to the caller.

Exceptions in CLU can be explicitly raised with the **signal** instruction. A **signal** instruction can include optional parameters that can be used to pass information to the exception handler.

When the execution of the handler is completed, control passes to the statement that follows the one to which the handler is attached.

B.3 Exception Handling in Ada

Ada provides a set of four predefined exceptions that can be raised by the underlying system: Constraint_Error, e.g. an array subscript out of range, or when there is a range error in a variable with a range description, division by zero, etc.; Program_Error, e.g. a function is required to complete normally by executing a return statement that transmits a result back to the caller; Storage_Error, e.g. run out of memory during the invocation of **new**; or Tasking_Error.

Ada exception handlers are usually local to the code in which the exception can be raised. Because this provides them with the same referencing environment, parameters for handlers are not necessary and, in fact, are not allowed.

An exception clause has the following format:

```
begin
    statements
exception
    when exception_1 => handler code
    :
    when exception_n => handler code
    when others => handler for any other exception that
                   is not exception_i, for i=1 to n
end;
```

In the above example, a list of handlers is attached to the block. The list is prefixed by the keyword **exception** and the handlers for each **exception**$_i$ is prefixed by the keyword **when**. The keyword **others** indicates that this handler is meant to handle exceptions that were not named locally, or exceptions that were propagated to this block.

User-defined exceptions are declared as:

```
exception_name: exception;
```

An exception is raised by the underlying system, producing one of the exceptions listed above, or is a user-defined exception raised with the following command:

```
raise exception_name;
```

If the block that raises the exception provides a handler for that exception, then the control is transfered to the handler. The control continues as if the block had terminated normally, i.e. the control continues after the **end** of the block. If an exception is raised and the unit that raised the exception does not provide a handler for that exception, then one of the following four possibilities may happen:

i) if the unit that raised the exception is a block, then the termination of the block propagates the exception in the enclosing unit;

ii) if the unit is a procedure, then the control returns to the caller of the procedure and the exception is propagated at the point of the call;

iii) if the unit is a package, then the exception is propagated in the unit that contains the package;

iv) if the unit is a task body, then the exception is not propagated and the task terminates abnormally.

B.4 Exception Handling in C++

In C++, exceptions are raised by the run-time environment, e.g. due to a division by zero, or raised by the program. An exception is raised by a **throw** instruction, which sends an object to the corresponding handler. A handler is attached to a **try** block command. A **try** block is a group of C++ statements, normally enclosed in braces { }, which may cause an exception. This grouping restricts exception handlers to exceptions generated within the **try** block. The exception handler is declared in a **catch** block. A **catch** block is a group of C++ statements that is used to handle a specific exception. Catch blocks, or handlers, should be placed after each try block. A **catch** block is specified by: the keyword **catch**; the type of exception that may be thrown in the **try** block; and, a group of statements, enclosed in braces { }, whose purpose is to handle the exception. See example below:

```
class Overflow {
    // ...
  public:
    Overflow(char,double,double);
};
void f(double x) {
   // ...
   throw Overflow('+',x,1.23e123);
}
int main() {
   try {
      f(6.6);
      //...
   } catch(Overflow& oo) {
        // handle exceptions of type Overflow here
   } catch // ...
}
```

In the above example, the function call in the **try** block passes control to f(), which throws an exception of type Overflow. This exception is handled by the **catch** block, which handles type Overflow exceptions.

In C++, exception handlers use the termination model. When an exception is generated, control is passed out of the function that threw the exception, out of the **try** block that anticipated the exception, and into the **catch** block whose exception declaration matches the exception thrown. The **catch** block handles the exception. It either re-throws the exception or ends normally. If a **catch** block ends normally, without a **throw**, the flow of control passes over all subsequent **catch** blocks.

In C++, a function declaration can include an exception specification, a list of exceptions that a function may directly or indirectly throw. The following declaration indicates to the caller that the function *func_name* may raise two types of exceptions, and that they are caught by a handler bound to types ExceptionType1 or ExceptionType2:

```
void func_name(parameters) throw(ExceptionType1,ExceptionType2);
```

If programmers want to specify that no exception is generated by the function *func_name*, they use the following command:

```
void func_name(parameters) throw();
```

B.5 Exception Handling in Java

Java exceptions are instances of classes. All objects that represent a Java exception must be instance of a class that extends the class Throwable. The Throwable class describes anything that can be thrown as an exception. There are two general sub-classes of the Throwable class: Error and Exception. The Error class represents compile-time and system errors. The Exception class is the basic class that can be thrown from any of the standard Java library class methods and from user-defined methods.

An example of how to define an exception in Java is given below:

```
Exception exception = new Exception();
```

A Java exception is raised using the **throw** instruction followed by the exception object. The object, which was thrown, is then transfered to a corresponding exception handler. A handler may be attached to any segment of code (a block) that needs to handle exceptional occurrences. In the next example, the **try** block might execute some commands that may raise exceptions. If an exception is raised, e.g. the computation of an angle that does not conform to some specification, the **catch** block corresponding to that exception, e.g. **catch** (AngleException) is invoked. In the above example, if an exception was raised and there was no handler specified for that exception, i.e. a **catch** block, then the block corresponding to Exception would be invoked.

If programmers wanted that exception to be re-thrown, they could either get rid of the block to handle Exception or define a **catch** block that re-throws

```
class AngleException extends Exception {
    public AngleException() {}
    public AngleException(String msg) {
        super(msg);
}
}
:
:
try {
// any code that may raise AngleException or
// other Java exceptions
} catch (AngleException e) {
// handler code for AngleException
} ...
} catch (Exception e) {
// handler for any other exception that is not an Error
// and has not been caught by the previous catch blocks
}
:
```

the exception. For example the following code could have been inserted in the previous example (before the block for Exception):

```
:
} catch (AnotherException e) {
    throw e;
} ...
:
```

The masking of an exception, i.e. catching an exception and re-throwing a different one can be done in the same way as re-throwing the caught exception. Consider the above example. If we want to catch AnotherException and re-throw a different exception, we would have to change the **throw** command to something like: **throw new** DifferentException(). In this way the previous object representing AnotherException will be garbage collected and a new object will be thrown.

Java also provides a **finally** clause to be attached as the last part of a **try** command. The code in the finally clause is always executed; it does not matter whether the commands in the **try** block have finished normally or raised an exception.

When exceptions are raised inside a Java method, those exception have to be declared in the interface of the method. Consider a method that can raise the AngleException declared previously. If you have a method that may raise that exception during the execution, then the interface of that method has to be declared as:

```
visibility type method_name ( parameters ) throws AngleException {
... // code for the routine
}
```

The rationale for specifying the exceptions in the method's interface is to allow the programmers to know which exceptions are raised by the method, so they can properly react in their code by catching the exception and handling it, or catching the exception and mapping it to one of their throwable exceptions, or listing the exception in the declaration of their enclosing method and allowing the exception to propagate.

B.6 Exception Handling in Eiffel

In Eiffel an exception is raised when an assertion is violated, or the underlying system detects an anomalous behaviour, e.g. run out of memory space. When an exception is raised the flow of control is passed to the **rescue** clause. Every routine in Eiffel has this clause for handling exceptions; it can be implicit or explicit. An explicit **rescue** clause is written by the programmer, while the implicit **rescue** clause is inherit from class ANY. The default **rescue** clause does nothing. A **rescue** clause comes right at the end of a routine and has the following format:

```
routine_name is
   statement
   rescue
       rescue_statement
end
```

Information about the exception that was raised is provided by the EXCEPTION class in Eiffel. This class provides the follow features:

- signal codes that are set when a hardware or operating system exception occurs;
- exception codes that are set when a program exception occurs;
- procedures catch_signal and raise_on_signal which enable the programmer to switch signals on and off; and the procedure raise which enables the programmer to raise a user-defined exception;
- attributes, e.g. last_ecode - code of the last exception, which enables the **rescue** clause to obtain information about the last exception.

The Eiffel exception handling mechanism provides two different approaches for a program to continue its execution. They are usually referred as: *organized panic* and *retrial (resumption)*. In the organized panic approach, the routine raising an exception fails, i.e. as an exception is raised, the routine execution is terminated and the control is passed to the **rescue** clause. The handler performs some type of recovery and then terminates signaling failure. In the retrial approach a programmer can write a **rescue** clause to cause an alternate part of the routine to be executed. This is achieved by the **retry** clause that may appear in the **rescue** clause and causes the routine to be re-executed. The following code exemplifies this case.

```
routine_name is
   require
   ...
   local
      version_two: BOOLEAN
   do
      if not version_two then
         version1
      else
         version2
      end
   require
   ...
   rescue
      if not version_two then
         version_two := true
         retry
      end
end
```

In the above example, ***routine_name*** will first execute the ***version1***, since the boolean variable version_two will be initialised to **false** (default value for boolean variables). If ***version1*** is not able to satisfy the post-condition (**require** clause) or raises an exception, then an exception will be raised in ***routine_name*** and the **rescue** clause will be invoked. The first time the **rescue** clause is executed, the version_two variable will be **false** and the commands version_two := true and **retry** will be executed. The effect is to execute the **do** clause of the routine again (without reinitialising local variables) and *version2* will be executed. The exception that caused the **rescue** clause to be invoked is considered suspended after the **retry** was executed. If no further exception is raised inside *routine_name*, then the flow of control continues normally.

B.7 Exception Handling in ML

The ML exception mechanism has three parts: exceptions declarations, which identify certain names as exception names and declare the type of data carried by each exception; raise expressions, which raise an exception and pass data to the handler; and handler declarations, which establish handlers for specific expressions.

An ML exception declaration has the form:

```
exception name of type
```

Where *name* is the name of the exception and *type* is the type of the data that is carried to an exception handler when this exception is raised. For example, in the code below, the exception Neg does not carry data, but the exception Signal must be raised with an integer parameter, which may be used by the exception handler.

```
exception Neg;
exception Signal of int;
```

An ML raise expression has the form

```
raise name params
```

An ML expression with handler declaration has the form:

```
exp handle exception_1 => exp_1
     ...|  exception_n => exp_n
```

In the code above, the *exp* is first evaluated. If the expression raises an exception that matches *exception*$_i$, then the handler is invoked and the corresponding *exp*$_i$ is evaluated. If there is no handler for a exception that was raised in *exp*, then this exception is propagated to the caller of the function where *exp* is declared.

```
exception Neg;
:
fun f(x) = if x <= 0
           then raise Neg
           else 1/x;
:
( f(x) handle Neg => 0 ) / (f(x) handle Neg => 1)
```

In the code above, we present a small example of how to raise an exception in a function, and how the same exception could be handled in two different ways. After the exception has been handled the control continues after the expression that raised the exception.

B.8 Discussion

In the languages surveyed in this appendix, almost all languages have similar approaches for the type of exceptions that can be handled and how they can be defined. They all provide built-in and user-defined exceptions. One of the major differences is whether an exception can carry information or not. In PL/I, an exception is a named signal that does not allow any additional information to be carried with the exception to the handler. Languages like Ada95, CLU, Java, C++ or ML allow data to be carried along with the exception. C++, for example, allows any kind of object to represent an exception.

Exception handlers in Java and C++ can be attached to any block of commands. Eiffel allows exception handlers to be only attached to routines. In CLU, an exception handler cannot be attached to the routine in which the exception was raised. In ML exception raising and exception handlers are attached to expressions or functions. In PL/I the binding between an exception and an exception handler takes place when an **ON** condition is encountered during the execution. The handler is valid then during the execution of the block of commands in which the **ON** condition was declared, until the termination of the block.

Appendix C

Syntax of DIP

In this appendix we present the syntax of DIP using *Backus-Naur Form* - BNF. Names in between <> are non-terminals, names in **bold** font are terminals (keywords), and ε represents an empty transition.

```
<program> :: program <name> is
                <class-definition>
                <action-definition>
                <process-definition>
                <objects-declaration>
                <teams-declaration>
                <player-declaration>
             end program

<class-definition> :: class <class-name> <inherit-part> is
                          <class-components>
                          <class-assertion>
                      end class ; <class-definition>
                      | ε

<inherit-part> :: inherit <class-name>
                  | ε
```

```
<class-components> :: <basicvar-declaration> <class-components>
                    | <objects-declaration> <class-components>
                    | state <list-of-names> ;
                    | ε

<class-assertion> :: assertion <boolean-expression> : <exception-name> ;
                  <class-assertion>
                  | ε

<basicvar-declaration> :: <basic-type> <var-name> ; <basicvar-declaration>
                        | ε

<objects-declaration> :: <class-name> <obj-name> ; <objects-declaration>
                       | ε

<action-definition> :: action <action-name> is
                       <body-definition>
                       <handler-definition>
                       end action ; <action-definition>
                     | ε
<body-definition> :: body is
                       <guard-definition>
                       <objects-declaration>
                       <teams-declaration>
                       <roles-definition>
                       <assertion-definition>
                       <exception-definition>
                   end body

<guard-definition> :: guard <boolean-expression> ; | ε

<teams-declaration> :: <action-name> <team-name> ; <teams-declaration>
                     | ε

<role-definition> :: role <role-name> same ;
                   | role <role-name> <parameters> is
                        <objects-declaration>
                        <basicvar-declaration>
                        <command>
                   end role ; <role-definition>
                   | ε

<parameters> :: with <list-of-objects>
              | ε
```

```
<assertion-definition> :: assertion <boolean-expression> | ε

<exception-definition> :: exceptions <exceptions-list> | ε

<exceptions-list> :: <exception-name>
                   | <exception-name> , <exceptions-list>
                   | <exception-name> ( <exceptions-list> )
                   | <exception-name> ( <exceptions-list> ) ,
                     <exceptions-list>

<handler-definition> :: handler for <exceptions-list> is
                          <guard-definition>
                          <objects-declaration>
                          <teams-declaration>
                          <roles-definition>
                          <assertion-definition>
                          <exception-definition>
                        end handler ;
                        <handler-definition> | ε

<process-definition> :: process <process-name> is
                        <basicvar-declaration>
                        <objects-declaration>
                          body is
                          <command>
                          end body
                        end process ; <process-definition> | ε

<players-declaration> :: <process-name> <player-name> ; <players-declaration>
                       | ε

<command> :: begin <list-commands> end
           | <assignment-command>
           | <iteration-command>
           | <selection-command>
           | <state-change-command>
           | <role-activation-command>
           | <message-sending-command>
           | <message-receiving-command>
           | <requeue-command>
           | <exception-handling-command>
           | <exception-raising-command>
           | ε
```

<list-commands> :: <command> **;** <list-commands>
| ε

<assignment-command> :: <obj-name> **:=** <expression>

<iteration-command> :: **loop** <while-clause> <until-clause> **do** <command>

<while-clause> :: **while** **(** <boolean-expression> **)**
| ε

<until-clause> :: **until** **(** <boolean-expression> **)**
| ε

<selection-command> :: **if** **(** <boolean-expression> **)** **then** <command>
<else-clause>

<else-clause> :: **else** <command>
| ε

<state-change-command> :: − > <obj-name>.<state-name>

<role-activation-command> :: <role-name>**@**<team-name> <with-clause>

<with-clause> :: **with** <list-objects>
| ε

<message-sending-command> :: <obj-name> **to** <role-name> <block-clause>

<block-clause> :: **block** | ε

<message-receiving-command> :: <obj-name> **from** <role-name>
<block-clause>

<requeue-command> :: **requeue** <obj-name>

<exception-handling-command> :: **try** <command>
on exception **(** <exception-name> **)** <command> <catch>

<catch> :: **on exception** **(** <exception-name> **)** <command> <catch>
| ε

<exception-raising-command> :: **raise** <exception-name>

<basic-type> :: **integer** | **float** | **char** | **string** | **boolean** | **list**

Appendix D

Operators of TLA$^+$

These operators are explained in [Lamport 1994].

Predicate and Action Operators

p'	[p true in final state of step]
$[A]_e$	[$A \lor (e' = e)$]
$\langle A \rangle_e$	[$A \land (e' \neq e)$]
ENABLED A	[An A step is possible]
UNCHANGED e	[$e' = e$]
A . B	[Composition of actions]

Temporal Operators

$\Box F$	[F is always true]
$\Diamond F$	[Eventually $\neg \Box \neg F$]
$WF_e(A)$	[Weak fairness for action A, $\Box \Diamond \langle A \rangle_e \lor \Box \Diamond \neg$ENABLED A]
$SF_e(A)$	[Strong fairness for action A, $\Box \Diamond \langle\langle A \rangle\rangle_e \lor \Diamond \Box \neg$ENABLED A]
$F \leadsto G$	[Leads to: $\Box(F \Rightarrow \Diamond G)$]
$\exists\!\!\exists\, x : F$	[Temporal existential quantification (hiding)]
$\forall\!\!\forall\, x : F$	[Temporal universal quantification]

Logic

$\lor \quad \land \quad \neg \quad \Rightarrow \quad <\!\!\Rightarrow$

TRUE FALSE BOOLEAN	[the set {TRUE, FALSE}]
$\forall x : p \quad \exists x : p$	$\forall x \in S : p \quad \exists x \in S : p$
CHOOSE $x : p$	[An x satisfying p]
CHOOSE $x \in S : p$	[An $x \in S$ satisfying p]

Sets

$= \neq \in \notin \cup \cap \subseteq$

$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p\}$	[Set of elements x in S satisfying p]
$\{e : x \in S\}$	[Set of elements e such that x in S]
SUBSET S	[Set of subsets of S]
UNION S	[Union of all elements of S]

Functions[1]

f[e]	[Function application]
DOMAIN f	[Domain of function f]
$[x \in S \mapsto e]$	[Function f such that $f[x] = e$ for $x \in S$]
$[S \rightarrow T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]

Records

e.x	[The x-component of record e]
$[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$	[The record whose x_i component is e_i]
$[x_1 : S_1, \ldots, x_n : S_n]$	[Set of all records with x_i component in S_i]

Tuples

e[i]	[The ith component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose ith component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with ith component in S_i]

Miscellaneous

"$c_1 \ldots c_n$"	[A literal string of n characters]
IF p THEN e_1 ELSE e_2	[Equals e_1 if p true, else e_2]
CASE $p_1 \rightarrow e_1 \; \Box \ldots \Box \; p_n \rightarrow e_n$	[Equals e_i if p_i true]
CASE $p_1 \rightarrow e_1 \; \Box \ldots \Box \; p_n \rightarrow e_n \; \Box \; e$	[Equals e_i if p_i true, or e if all p_i are false]

LET $d_1 \triangleq e_1 \ldots d_n \triangleq e_n$ IN in e [e in the context of the definitions]

$\wedge\, p_1$ [the conjunction $p_1 \wedge \ldots \wedge p_n$] $\quad \vee\, p_1$ [the conjunction $p_1 \vee \ldots \vee p_n$]

.
.
.

$\wedge\, p_n \quad \vee\, p_n$

[1]Only functions with one argument is described; TLA+ also allows functions with multiple arguments

Appendix E

The Dining Philosopher System Using DMIs

In this appendix, we list the code for one complete system implemented using the DMIs framework presented in Chapter 5.

Each philosopher i in the system creates manager and role objects as shown in Figure E.1. The manager object for the philosopher will act as the leader of the corresponding eating DMI.

```
Manager pmgr[i] = new Manager("philM"+i,"eating"+i);
Role      phil[i] = new Philosopher("Philosopher"+i,pmgr[i]);
```

Figure E.1: Instantiation of Philosopher role objects in the framework

Each fork j in the system creates manager and role objects as shown in Figure E.2 (one fork role to the DMI with philosopher j, and one fork role to the DMI with philosopher $(j + 1)\%5$). Notice that each manager for the forks are created with the corresponding philosopher manager object, which acts as the leader of the eating DMI.

```
Manager rmgrL[j] = new Manager("forkL"+j,"eating"+j, pmgr[j]);
Role    forkL[j] = new LeftFork("LeftFork"+j,rmgrL[j]);

Manager rmgrR[j] = new
                 Manager("forkR"+((j+1)%5),"eating"+((j+1)%5), pmgr[(j+1)%5]);
Role    forkR[j] = new RightFork("RightFork"+((j+1)%5),rmgrR[j]);
```

Figure E.2: Instantiation of Fork role objects in the framework

In Figures E.3 and E.4, we represent in DIP the processes that will execute the eating DMIs (action). Each of these processes receives a number upon instantiation that specifies its number, e.g. Philosopher p(1); creates a philosopher DIP object and the pn variable parameter is set to 1 to this philosopher object.

```
process Philosopher(pn:integer) is
  body
    loop
      action[pn].Philosopher with pasta
      // sleep ...
    end;
  end body
end process
```

Figure E.3: Philosopher process in DIP

A fork process also is instatiated with a parameter that specifies what eating DMI it will act as right or left fork. If the parameter rn is an even number, then the fork process first execute the role on its right eating DMI, playing as the right fork role.

```
process Fork(rn:integer) is
 body
   loop
     if rn%2 == 0 then
       begin // action on the right first
         action[(rn+1)%5].RightFork
         action[rn].LeftFork
       end
     else
       begin action on the left first
         action[rn].LeftFork
         action[(rn+1)%5].RightFork
       end;
   end;
 end body
end process
```

Figure E.4: Fork process in DIP

The following Java code shows the implementation of the roles for the DMIs in the dining philosopher problem described in Appendix A. These role objects were instantiated in the code shown in Figures E.1 and E.2.

```
/**********************************************************************
 * Class: LeftFork
*/
package phil.eating;

import classes.*;
import drip.*;

public class LeftFork extends Role {
  Synchronous getFood;
  /**
   * Constructor.
  */
  public LeftFork(String n, Manager mgr, Manager leader) {
     super (mgr, leader, n);                    // initialise role
     getFood    = new Synchronous();            // shared local object
     mgr.sharedObject("getFood", getFood); // export object
  }

  public void body(Object list[]) throws Exception {

       Synchronous grabLeft  =  mgr.getSharedObject("grabLeft");
       Synchronous eatFood   =  mgr.getSharedObject("eatFood");

       // inform the philosopher that LeftFork is
       // ready
       grabLeft.synchronize();

       // when the RightFork is ready, then
       // get the food
       getFood.synchronize();

       // get food

       // synchronize with RightFork and Philosopher
       // so the Philosopher can eat the food
       eatFood.synchronize();

       // Philosopher is eating the food
  }
}
```

```
/*****************************************************************
 * Class: RightFork
*/
package phil.eating;

import classes.*;
import drip.*;

public class RightFork extends Role {
  /**
   * Constructor.
  */
  public RightFork(String n, Manager mgr, Manager leader) {
     super (mgr, leader, n); // initialise role
  }

  public void body(Object list[]) throws Exception {

       Synchronous grabRight =  mgr.getSharedObject("grabRight");
       Synchronous eatFood   =  mgr.getSharedObject("eatFood");
       Synchronous getFood   =  mgr.getSharedObject("getFood");

       // inform the philosopher that RightFork is
       // ready
       grabRight.synchronize();

       // when the LeftFork is ready, then
       // get the food
       getFood.synchronize();

       // get food

       // synchronize with LeftFork and Philosopher
       // so the Philosopher can eat the food
       eatFood.synchronize();

       // Philosopher is eating the food
  }
}
```

```
/****************************************************************
 * Class: Philosopher
*/
package phil.eating;
import dip.*;
import classes.*;

public class Philosopher extends Role {
  Synchronous grabLeft;
  Synchronous grabRight;
  Synchronous eatFood;
  /**
   * Constructor.
  */
  public Philosopher(String n, Manager mgr, Manager leader) {
     super (mgr, leader, n);                     // initialise role
     grabLeft  = new Synchronous();              // create share local objects
     grabRight = new Synchronous();
     eatFood   = new Synchronous(3);
     mgr.sharedObject("grabLeft" , grabLeft);  // export objects
     mgr.sharedObject("grabRight", grabRight);
     mgr.sharedObject("eatFood"   , eatFood);
  }

  public void body(Object list[]) throws Exception {

       // synchronize with the left fork, representing
       // the grabbing of the fork by the philosopher
       grabLeft.synchronize();

       // synchronize with the right fork, representing
       // the grabbing of the fork by the philosopher
       grabRight.synchronize();

      // synchronize with both forks to eat
       eatFood.synchronize();

       // eating
       Thread.currentThread().sleep(3000);
  }
}
```

Bibliography

[Andrews *et al.* 1988] G. R. Andrews, R. A. Olsson M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, 1988.

[Attie *et al.* 1993] P. C. Attie, N. Francez, and T. X. Austin. Fairness and hyperfairness in multiparty interactions. *Distributed Computing*, 6(4):245–254, 1993.

[Avizienis 1985] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[Back & Kurki-Suonio 1988] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[Banatre & Métayer 1993] J.-P. Banatre and D. L. Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[Banatre & Métayer 1996] J.-P. Banatre and D. L. Métayer. GAMMA and the chemical reaction model: Ten years after. In J.-M. andréoli, C. Hankin, and D. L. Métayer, editors, *Coordination Programming: Mechanisms, models and semantics*, pages 3–41. World Scientific Publishing, 1996.

[Beek 1993] D. A. van Beek. *Exception Handling in Control Systems.* PhD thesis, Eindhoven University of Technology, the Netherlands, 1993.

[Best *et al.* 1992] E. Best, R. Devillers, and J. G. Hall. *The Box Calculus: A new causal Algebra with multi-label communication*, volume 609 of *Lectures Notes in Computing Science*, pages 21–87. Springer Verlag, Berlin, Germany, 1992.

[Bolognesi & Brinksma 1987] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks ISDN Systems*, 14:25–59, 1987.

[Brinksa 1988] E. Brinksa. *On the Design of Extended LOTOS - A Specification Language for Open Distributed Systems.* PhD thesis, University of Twente, the Netherlands, 1988.

[Burns & Wellings 1995] A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge University Press, 1995.

[Burns & Wellings 1996] A. Burns and A. Wellings. *Real Time Systems and Programming Languages.* Addison Wesley, Reading, MA, USA, 1996.

[Campbell & Randell 1986] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[Canver 1997] E. Canver. Formal verification of the CAA-design of the Production Cell. In *DeVa - Design for Validation*, 2nd year, pages 357–383. ESPRIT Long Term Project 20072, 1997.

[Canver *et al.* 1998] E. Canver, D. Schwier, A. Romanovsky, and J. Xu. Formal verification of the CAA-designs: The Fault-Tolerant Production Cell. In *DeVa - Design for Validation*, 3rd year, pages 229–258. ESPRIT Long Term Project 20072, 1998.

[Charlesworth 1987] A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, 1987.

[Cox & Gehani 1989] I. J. Cox and N. H. Gehani. Exception handling in robotics. *Computer*, pages 43–49, March 1989.

[Cristian 1982] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, C-31:531–540, 1982.

[Cristian 1989] F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, chapter 4, pages 68–97. BSP Professional Books, Oxford, UK, 1989.

[Davies 1978] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[Digital 1986] Digital Equipament Corporation, Massachusetts, USA. *VAX-ELN Pascal language reference manual: Programming*, 1986.

[Dijkstra 1965] E.Dijkstra. Solution of a problem in concurrent programming control. *Communications of ACM*, 8(9):569, 1965.

[DiMarzo & Guelfi 1998] G. DiMarzo and N. Guelfi. Formal development of Java based web parallel applications. In *Hawaii International Conference of System Science*, pages 604–613. IEEE CS Press, 1998.

[DiMarzo *et al.* 1999] G. DiMarzo, N. Guelfi, A. Romanovsky, and A. F. Zorzo. Formal development and validation of Java dependable distributed systems. In *5th IEEE International Conference on Engineering of Complex Computer Systems.* IEEE CS Press, 1999.

[Dony 1990] C. Dony. Exception handling and object-oriented programming: towards a synthesis. In *OOPSLA '90*, pages 322–330. ACM Press, 1990.

[Evangelist *et al.* 1989] M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, 1989.

[Forman 1986] I. R. Forman. On the design of large distributed systems. In *1st International Conference on Computer Languages*, pages 84–95. IEEE CS Press, 1986.

[Forman & Nissen 1996] I. Forman and F. Nissen. *Interacting Processes - A multiparty approach to coordinated distributed programming.* ACM Press, 1996.

[Foster 1996] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.

[Francez *et al.* 1986] N. Francez, B. Hailpern, and G. Taubenfeld. Script: A communication abstraction mechanism. *Sci. Comput. Prog.*, 6(1):35–88, 1986.

[Ghezzi & Jazayeri 1998] C. Ghezzi and M. Jazayeri. *Programming Language Concepts.* John Wiley & Sons, Chichester, UK, 1998.

[Goldberg & Robson 1983] A. Goldberg and D. Robson, editors. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, Reading, MA, USA, 1983.

[Goodenough 1975] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[Gosling *et al.* 1996] J. Gosling, J. Bill, and G. Steele. *The Java Language Specification.* The Java Series. Addison Wesley, Reading, MA, USA, 1996.

[Gray & Reuter 1993] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 2nd edition, 1993.

[Gray 1978] J. Gray. *Notes on Database Operating Systems*, volume 60 of *Lectures Notes in Computing Science*, pages 393–481. Springer Verlag, Berlin, Germany, 1978.

[Hennessy 1990] M. Hennessy, editor. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* John Wiley & Sons, Chichester, UK, 1990.

[Hoare 1984] C. A. R. Hoare. *Occam Programming Manual.* Prentice Hall, Englewood Cliffs, NJ, USA, 1984.

[Hoare 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

[Horning 1979] H. Horning. Programming languages. In T. Anderson and B. Randell, editors, *Computing Systems Reliability.* Cambridge University Press, 1979.

[Horowitz 1983] E. Horowitz. *Fundamentals of Programming Languages.* Springer Verlag, Berlin, Germany, 1983.

[ISO 1995] International Standard for Organization, editor. *Ada 95 Reference Manual - ISO/8652-1995.* ISO, 1995.

[Issarny 1993] V. Issarny. An exception handling mechanism for parallel object-oriented programming: Toward reusable, robust distributed software. *Journal of Object Oriented Programming,* 6(6):29–39, 1993.

[Jårvinen & Kurki-Suonio 1990] H.-M. Jårvinen and R. Kurki-Suonio. The DisCo language and Temporal Logic of Action. Technical Report 11, Software Systems Laboratory, Tampere University of Technology, Finland, 1990.

[Jårvinen & Kurki-Suonio 1991] H.-M. Jårvinen and R. Kurki-Suonio. DisCo specification language: Marriage of actions and objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE CS Press, 1991.

[Joung & Smolka 1990] Y.-J. Joung and S. A. Smolka. A completely distributed and message-efficient implementation of synchronous multiprocess communication. In *19th International Conference on Parallel Processing*, volume III, pages 311–318, 1990.

[Joung & Smolka 1994] Y.-J. Joung and S. A. Smolka. Coordinating first-order multiparty interaction. *ACM Transactions on Programming Languages and Systems*, 16(3):954–985, 1994.

[Joung & Smolka 1996] Y.-J. Joung and S. A. Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of ACM*, 43(1):75–115, 1996.

[Knudsen 1987] J. L. Knudsen. Better exception handling in block structured systems. *IEEE Software*, pages 40–49, May 1987.

[Lamport 1994] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lamport 1996] L. Lamport. The Windows Win32 threads API specification. Technical report, Digital - Systems Research Center, Palo Alto, CA, USA, 1996.
http://www.research.digital.com/SRC/personal/lamport/tla.

[Lamport 1999] L. Lamport. Specifying concurrent systems with TLA$^+$. In M Broy and R. Steinbruggen, editors, *Calculational System Design*. IOS Press, Amsterdam, 1999.

[Laprie 1995] J.-C. Laprie. Dependability - its attributes, impairments and means. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, chapter 1, pages 1–24. Springer-Verlag, Berlin, Germany, 1995.

[Lee & Anderson 1990] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer Verlag, Berlin, Germany, 1990.

[Lewerentz & Lindner 1995] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes in Computing Science*. Springer Verlag, Berlin, Germany, 1995.

[Liskov & Snyder 1979] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, pages 546–558, November 1979.

[Lötzbeyer & Muhlfeld 1996] A. Lötzbeyer and R. Muhlfeld. Task description of a fault-tolerant production cell. Technical report, Forschungszentrum Informatik, Karsruhe, Germany, 1996.
http://www.fzi.de/divisions/prost/projects/korsys/korsys.html.

[Lynch *et al.* 1994] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, CA, USA, 1994.

[Meyer 1992] B. Meyer, editor. *Eiffel: The Language.* Prentice Hall, Englewood Cliffs, NJ, USA, 1992.

[Meyer 1997] B. Meyer. *Object-oriented software construction.* Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1997.

[Mitchell *et.al.* 1998] S. E. Mitchell, A. J. Wellings, and A. Romanovsky. Distributed atomic actions in Ada95. *Computer Journal* 41(7):468–502, 1998. (Also as Technical Report YCS-98-299, Department of Computer Science, University of York, York, UK, 1998.)

[Muller 1997] P-A. Muller, editor. *Instant UML.* Wrox Press Ltd., Birmingham, UK, 1997.

[Neumann 1996] P. G. Neumann. Distributed systems have distributed risks. *Communications of the ACM*, 39(11):130, 1996.

[Petitpierre 1998] C. Petitpierre. Synchronous C++: A language for interactive applications. *IEEE Computer*, 31(9):65–72, 1998.

[Pnueli 1977] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE CS Press, 1977.

[Randell 1975] B. Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[Randell *et al.* 1997] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated Atomic actions: from concept to implementation. Technical Report 595, Department of Computing Science, Newcastle upon Tyne, UK, 1997. http://www.cs.ncl.ac.uk/research/trs.

[Roman & Day 1984] G.-C. Roman and S. Day. Multifaceted distributed systems specification using processes and event synchronization. In *7th International Conference on Software Engineering*, pages 44–55, 1984.

[Romanovsky 2000] A. Romanovsky. Extending conventional languages by distributed/concurrent exception resolution. *Journal of Systems Architecture*, 46(1):79–95, 2000.

[Romanovsky *et al.* 1996] A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object-oriented systems. In *16th IEEE International Conference on Distributed Computing Systems*, pages 545–552. IEEE CS Press, 1996.

[Romanovsky & Zorzo 1999] A. Romanovsky and A. F. Zorzo. Coordinated Atomic actions as a technique for implementing distributed GAMMA computation. *Journal of Systems Architecture - Special Issue on New Trends in Programming*, 45(15):1357–1374, 1999.

[Schwier *et al.* 1997] D. Schwier, F. von Henke, J. Xu, R. J. Stroud, A. Romanovsky, and B. Randell. Formalization of the CA action concept based on Temporal Logic. In *DeVa - Design for Validation*, 2nd year, pages 3–15. ESPRIT Long Term Project 20072, 1997.

[Sebesta 1989] R. Sebesta, editor. *Concepts of programming language*. The Benjamin/Cummings Publ., Redwood City, CA, USA, 1989.

[Shrivastava *et al.* 1991] S. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):63–73, 1991.

[Simone 1985] R. De Simone. Higher-level synchronizing devices in MEIJE-SCCS. *Theoretical Computing Science*, 37:245–267, 1985.

[Stroustrup 1994] B. Stroustrup. *The C++ programming language*. Addison Wesley, Reading, MA, USA, 1994.

[Szalas & Szcepanska 1985] A. Szalas and D. Szcepanska. Exception handling in parallel computations. *Sigplan Notices*, pages 95–104, October 1985.

[Tennent 1991] R. D. Tennent, editor. *Semantics of Programming Languages*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[Ullman 1994] J. D. Ullman, editor. *Elements of ML programming.* Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[Xu *et al.* 1995] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *25th International Symposium on Fault-Tolerant Computing*, pages 450–457. IEEE CS Press, 1995.

[Xu *et al.* 1998] J. Xu, A. Romanovsky, A. F. Zorzo, B. Randell, R. J. Stroud, and E. Canver Developing control software for Production Cell II: Failure analysis and system design using CA actions. In *DeVa - Design for Validation*, 3rd year, pages 167–188. ESPRIT Long Term Project 20072, 1998.

[Xu *et al.* 1999] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, E. Canver A. F. Zorzo, and F. von Henke. Rigorous development of a safety-critical system based on Coordinated Atomic actions. In *29th International Symposium on Fault-Tolerant Computing.* IEEE CS Press, 1999.

[Wellings & Burns 1996] A. J. Wellings and A. Burns. Implementing atomic actions in Ada95. *IEEE Transactions on Software Engineering*, 23(2), 1997. (Also Technical Report YCS-96-264, Department of Computer Science, University of York, York, UK, 1996.)

[Wirth 1977] N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.

[Young 1982] S. J. Young. *Real Time Languages: Design and Development.* Ellis Horwood Ltd., Chichester, 1982.

[Zorzo 1999] A. F. Zorzo. Dependable multiparty interactions: A case study. In *29th Conference on Technology of Object-Oriented Languages and Systems - TOOLS29-Europe*, Nancy, France, 1999. IEEE CS Press.

[Zorzo *et al.* 1999] A. F. Zorzo, A. Romanovsky, B. Randell J. Xu, R. J. Stroud, and I. S. Welch. Using Coordinated Atomic actions to design safety-critical systems: A production cell case study. *Software, Practice and Experience*, 29(8):677–697, 1999.

[Zorzo & Stroud 1999] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *14th ACM Conference on Object-Oriented Programming Systems, Languages and Applications - OOPSLA'99*, Denver, CO, USA, 1999. ACM Press.