Exception Handling in Control Systems

Print: Wibro, Helmond.

Cover design: D. van der Pol.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Beek, Dirk Albert van

Exception handling in control systems / Dirk Albert van Beek. - Eindhoven : Eindhoven University of Technology
Thesis Eindhoven. - With index, ref. - With summary in Dutch.
ISBN 90-386-0262-6
Subject heading: control systems ; exceptions.

Exception Handling in Control Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof. dr. J.H. van Lint,
voor een commissie aangewezen door het College van
Dekanen in het openbaar te verdedigen op
donderdag 1 juli 1993 om 16.00 uur

door

DIRK ALBERT VAN BEEK

geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotoren

prof. dr. ir. J.E. Rooda

en

prof. drs. C. Bron

Acknowledgement

The research that I have done in the Vredestein factories in Doetinchem, where the ROSKIT kernel [Rossingh and Rooda, 1985] is used for machine control, has been of great importance for the development of the concepts presented in this dissertation. I would like to thank dr.ir. J.H.A. Arentsen who made my stay there possible, and ing. G. Huizing, ir. T.J. Rossingh and ing. M.F.M. Seinhorst for the stimulating discussions with them.

Summary

This thesis deals with the required concepts and mechanisms for exception handling in control systems.

There is much confusion in the literature about the meaning of exceptions and the relationship of exceptions and errors. In this thesis, the most important terms relating to errors and exceptions are accurately defined, while retaining a high level of compatibility with the way these terms are used in the literature.

A treatment is presented of the most important concepts relating to the three stages of error handling: error detection, error diagnosis and confinement, and error recovery and fault repair. Only forward error recovery is covered in this thesis.

Several satisfactory and more or less similar exception handling mechanisms exist for the handling of internal exceptions. These mechanisms and a traditional mechanism are briefly evaluated. The resume response from an exception handler is rejected as being undesirable in both single and multi-process environments.

A literature search has yielded several proposals and existing mechanisms for exception handling in controlling systems or, more generally, in a multi-process environment. No publications, however, have been found which describe the essence of the required functionality of such mechanisms. Therefore, 'constraint of an operation' and 'constraint violation' have been introduced as new concepts. The constraint of an operation is that part of its precondition which is invariant over the operation: it has to be valid throughout the execution of the operation. A violation of an operation's constraint causes an exception occurrence in the process executing the operation and should result in the raising of an exception.

The concepts constraint and constraint violation have been used to describe the required functionality of mechanisms for the handling of exceptions in controlling systems. Several existing and proposed mechanisms have been evaluated using this functionality. The mechanisms have been evaluated as

either offering a functionality which is too restricted for controlling systems, as offering an incorrect or undesirable functionality, or as inadequate in other ways.

A new mechanism for the handling of constraint violations has been introduced. The mechanism has been realized by means of constraint monitors which are used to specify and monitor constraints of operations independently of other operations, which is an important requirement for the creation of modular subprograms. A constraint monitor bound to a single operation can also be used to specify a constraint which is common to several operations.

A constraint monitor is bound to an operation, and consists of a constraint and an exception. The violation of the constraint which is monitored by a constraint monitor results in the creation of a pending exception. The exception is not immediately raised, since this can result in time-dependent run-time errors due to violations of the internal invariants of a process. Pending exceptions are raised at interaction points, which are natural places for internal invariants to hold, but they are not raised in exception handlers.

Several constraints can be violated at the same time by concurrently executing processes. This can result in more than one pending exception in a process. Some criteria for the selection of a pending exception have been evaluated. The pending exception that should be selected is the one belonging to the constraint monitor which was enabled first, and thus at the outermost level. The other pending exceptions can be discarded.

The only systems considered are systems that can be modeled as discrete event systems.

The new mechanism is independent of a particular programming language. The functionality of the mechanism therefore deals with the common requirements of languages for the control of industrial systems. Language-specific elements are not treated.

Constraint monitors have been successfully implemented in Process Calculus, which is a language for the specification, simulation and control of industrial systems. The simplicity and power of the new mechanism is illustrated with a case concerning the control of a transport system.

Samenvatting

Dit proefschrift beschrijft een studie naar de vereiste concepten en mechanismen voor de afhandeling van excepties in besturingssystemen.

Er bestaat veel verwarring in de literatuur ten aanzien van de betekenis van excepties en de relatie tussen excepties en fouten. In dit proefschrift zijn de belangrijkste termen betreffende fouten en excepties nauwkeurig gedefinieerd, waarbij een hoge mate van compatibiliteit is behouden met het gebruik van deze begrippen in de literatuur.

De belangrijkste concepten betreffende de drie stadia van foutafhandeling namelijk foutdetectie, foutdiagnose en schadebeperking, en tenslotte foutherstel zijn behandeld. Dit proefschrift gaat uit van voorwaarts foutherstel.

Er bestaan verschillende bevredigende en min of meer gelijkwaardige mechanismen voor de afhandeling van interne excepties. Deze mechanismen en een traditioneel mechanisme zijn kort geëvalueerd. De hervattingsresponsie vanuit een exceptie-afhandelaar (Eng. exception handler) is verworpen als zijnde ongewenst, zowel in een enkel sequentiëel proces als in een omgeving met parallelle processen.

Een literatuuronderzoek heeft verschillende voorstellen en bestaande mechanismen opgeleverd voor het afhandelen van excepties in besturingssystemen of, meer in het algemeen, in een omgeving van parallelle processen. Er zijn echter geen publicaties gevonden waarin de essentie van de gewenste functionaliteit van zulke mechanismen is beschreven. Daarom zijn de 'constraint van een operatie' en 'constraint-schending' als nieuwe concepten geïntroduceerd. De constraint van een operatie is dat deel van haar preconditie dat invariant is over de operatie: hij moet gelden gedurende de uitvoering van de operatie. Een schending van een constraint van een operatie veroorzaakt een exceptiegeval (Eng. exception occurrence) in het proces dat de operatie uitvoert, hetgeen zou moeten leiden tot het activeren (Eng. to raise) van een exceptie.

De concepten constraint en constraint-schending zijn gebruikt om de gewenste functionaliteit van exceptie-afhandelingsmechanismen in besturingssystemen te beschrijven. Verschillende bestaande en voorgestelde mechanismen zijn aan de hand van deze functionaliteit geëvalueerd. De mechanismen blijken na evaluatie ofwel een functionaliteit te bieden die te beperkt is voor besturingssystemen, die foutief of ongewenst is, ofwel die in andere opzichten inadequaat is.

Een nieuw mechanisme voor het afhandelen van schendingen van constraints is geïntroduceerd. Het mechanisme is gerealiseerd door middel van 'constraint monitors' die worden gebruikt om constraints van operaties onafhankelijk van andere operaties te specificeren en te bewaken, wat een belangrijk vereiste is voor de onwikkeling van modulaire subprogramma's. Een constraint monitor die gebonden is aan een enkele operatie kan ook worden gebruikt voor de specificatie van een constraint die gemeenschappelijk is voor verscheidene operaties.

Een constraint monitor wordt gebonden aan een operatie, en bestaat uit een constraint en een exceptie. De schending van de constraint die wordt bewaakt door een constraint monitor resulteert in de creatie van een hangende exceptie. De exceptie wordt niet onmiddellijk geactiveerd, aangezien dit aanleiding kan geven tot tijdsafhankelijke executie-fouten ten gevolge van schendingen van de interne invarianten van een proces. Hangende excepties worden geactiveerd op interactie-punten, wat natuurlijke plaatsen zijn waar interne varianten gelden, maar zij worden niet geactiveerd in exceptie-afhandelaars.

Verschillende constraints kunnen op hetzelfde moment worden geschonden door gelijktijdig uitgevoerde processen. Dit kan resulteren in meer dan een hangende exceptie in een proces. Een aantal criteria voor de selectie van een hangende exceptie is geëvalueerd. De hangende exceptie die geselecteerd zou moeten worden, is degene die behoort bij de constraint monitor die als eerste, en dus op het buitenste niveau, is geactiveerd. De andere hangende excepties kunnen worden verwijderd.

De enige systemen welke zijn beschouwd zijn systemen die kunnen worden gemodelleerd als 'discrete event' systeem.

Het nieuwe mechanisme is onafhankelijk van een specifieke programmeertaal. De functionaliteit van het mechanisme betreft daarom de

gemeenschappelijke vereisten van talen voor het besturen van industriële systemen. Taal-specifieke elementen zijn niet behandeld.

Constraint monitors zijn met succes geïmplementeerd in Procescalculus, een taal voor het specificeren, simuleren en besturen van industriële systemen. De eenvoud en kracht van het nieuwe mechanisme is verduidelijkt aan de hand van een voorbeeld betreffende de besturing van een transportsysteem.

x

Table of contents

Chapter 1
Introduction

1.1 Background

An important aspect in the design of industrial control systems is the handling of errors. The amount of code required for error recovery is usually many times greater than the amount needed to control the system under error-free circumstances. In [Gini, 1985] it is observed that the amount of code for error recovery in a robotic environment often amounts to 80% of the total amount of code. This enormous amount of code for error recovery is specific to industrial control systems. Controlled physical systems suffer from deterioration due to wear and ageing; they also exhibit a stochastic behaviour in certain respects. Components, for instance, have tolerances and robots suffer from imprecise positioning. Such characteristics will lead to errors.

The terminology in regard to control systems, controlling systems and controlled systems is taken from [IEC 50, 1975]. According to this standard, a control system may be divided into two interdependent parts: the controlled system, which comprises the operative equipment executing the physical process; and the controlling system, which interacts with the supervisor, the process to be controlled and possibly other controlling systems in the control system's environment. The controlling system receives feedback information from the controlled system, and controls this system by means of output commands.

An important concept which facilitates the handling of errors in a structured way is the concept of exception. Most research concerning exception handling has focused on the use of exceptions in sequential systems. This is reflected in the definition of programming languages that offer advanced exception handling mechanisms. These mechanisms are usually restricted to exceptions within a sequential process.
Because of the inherent parallelism of controlled systems and their associated controlling systems, an exception handling mechanism is needed for the handling of exceptions in a multi-process environment.

1.2 Scope of the thesis

There is much confusion in the literature about the meaning of exceptions and the relationship of exceptions and errors. Many definitions are imprecise or incorrect, or contain undesirable subjective elements. In Chapter 3 and 4, the most important terms relating to errors and exceptions are accurately defined. The important characteristics of the most frequently used exception handling mechanisms in sequential processes are evaluated. We concentrate on forward error recovery; backward error recovery and redundancy techniques that aim to provide fault tolerance in the presence of incorrect components in a controlling system are not covered. Apart from the new definitions and the evaluation of exception handling mechanisms, Chapters 3 and 4 mainly give a general treatment of the state of the art concepts regarding error and exception handling.

The exception handling mechanisms for sequential processes can also be used in control systems. Their usefulness, however, is restricted to exceptions related to a single process. A different or additional mechanism is needed in a multi-process environment. The development of such a mechanism is the objective of this thesis. The new mechanism is described in Chapter 6.
The most important aspect in the development of an exception handling mechanism is a clear definition of the desired functionality. In order to be able to describe the essence of the desired functionality, 'constraint of an operation' and 'constraint violation' are introduced as new concepts in Chapter 5. The desired characteristics of the new mechanism are defined using these concepts. Some important existing and proposed mechanisms are evaluated against this framework.

The only systems considered in this thesis are systems that can be modeled as discrete event systems, such as robotic systems, manipulators, transporting systems, etc. Continuous systems, such as chemical reactions and the continuous flows of liquids, are not considered. Nevertheless, most of the theory developed in this thesis is independent of the kind of controlled system.

The new mechanism is independent of a particular programming language. The functionality of the mechanism therefore deals with the common requirements of languages for the control of industrial systems.
In this thesis, parallel controlling processes are considered to exist throughout the life of an executing control program. Some languages allow

parallel constructs such that father processes create concurrently executing children and wait for their – possibly exceptional – termination. They require the same newly-developed mechanism if they are used for the control of systems. The additional features that are necessary to deal with the specific problems introduced by children terminating with an exception have been disregarded.
Exceptions occurring during the execution of an interaction, such as during a rendezvous, are also language specific and are therefore also not covered.

Process Calculus [Rooda, 1991], which is described in Chapter 2, has been complemented with the new mechanism. Process Calculus is a powerful language for the specification, simulation and control of industrial systems.

Finally in Chapter 7, the mechanism developed is illustrated with reference to a case.

Chapter 2
Modeling control systems using Process Calculus

The concepts, theories and mechanisms developed in this thesis are independent of a specific programming language. Only the implementation of the developed mechanism is realized in Process Calculus. Many examples are used throughout this thesis in order to illustrate the different concepts, theories and mechanisms. These examples are mainly implemented in Process Calculus.

Process Calculus is treated in [Rooda, 1991a and 1991b]. More information can be found in [Rooda, 1981; Overwater, 1989; Wortmann, 1991] where the respective terms SOLE, Process Interaction Approach and ProcessTalk are used instead of Process Calculus.

In this chapter, Process Calculus is treated in such a way that the examples using Process Calculus can be understood. It is also shown how Process Calculus can be used to specify controlling systems and test them by means of simulation, and how controlling systems are interfaced with the controlled system. Finally, a transport system is considered, together with the specification of its controlling system without error handling. The transport system will be used as the basis of many subsequent examples.

2.1 Process Calculus

2.1.1 Processors and interactions

Using Process Calculus, an industrial system is specified or modeled as a collection of processors and interaction paths. Interaction paths are connected to processors by means of ports on the processor. A processor can have send ports and receive ports. Interaction paths establish a connection between a send port on a processor and a receive port on another processor. The interaction path is used to transfer an object from one processor to

another processor. An object leaves a processor through a send port and enters through a receive port. When an object is actually being transferred from one processor to another processor, an interaction is said to take place. Processors do not refer to surrounding processors; they only interact via their ports.

The main interaction mechanism used is the synchronous interaction mechanism. This mechanism stipulates that an interaction can take place between two processors if they are both prepared to interact: i.e. one of them must be executing a send action to a send port and the other processor must be executing a receive action from a receive port. Naturally, there must exist an interaction path between the two ports.

More than one interaction path may be connected to a single port. In this case, the interaction can take place between the single port and any one of the other ports that are connected to it by means of interaction paths. If a processor executes a send or receive action to a port, and no other processor connected to that port is executing a corresponding receive or send action, then the processor is blocked until the send or receive action can take place.

There are two kinds of processors: leaf processors and expanded processors.
The model of a leaf processor is a process description. Processes are only associated with leaf processors. Only leaf processors can execute send and receive actions.
The model of an expanded processor is a collection of processors – known as the child processors of the expanded processor – and interactions. The expanded processor is known as the parent of its child processors. The ports of an expanded processor are connected through interaction paths with the ports of its child processors. Expanded processors do not execute a process, but merely act as an abstraction of a collection of processors and interactions.

2.1.2 Graphical representation of models

Processors are represented graphically by a circle. The name of the processor is presented within the circle. An interaction path is represented by an arrow, starting at a send port and ending at a receive port. The ports are situated on the edge of the circle. The name of the port to which an arrow is connected can be displayed near the end of the arrow, where it

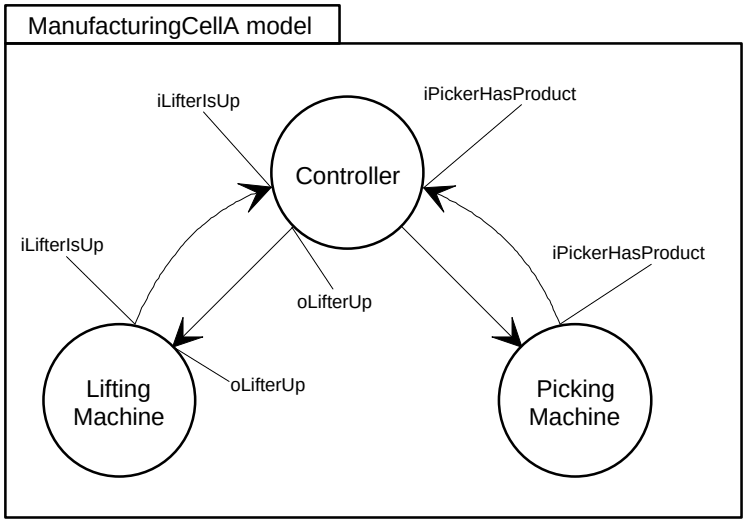

Figure 2.1.1 *Model of a manufacturing cell without expanded processors.*

connects to the circle. A dotted line may be used to connect the name of the port to the port itself, at the end of the arrow (see Figure 2.1.1).
A model of an expanded processor is graphically depicted by means of a rectangle with a label bearing the name of the processor. The model of an expanded processor contains further processors and interactions. This model of an expanded processor can be referred to as its expansion. The ports of the expanded processor are depicted graphically in the expansion by their port names. If more than one arrow is connected to a port name in the expansion of a processor, the port name can be copied to different locations in the expansion and other arrows can be connected to the copied port names in order to achieve a clearer layout.

Figures 2.1.1 and 2.1.2 show examples of two functionally equivalent models of a manufacturing cell. The only difference is the hierarchical ordering of the processors. The manufacturing cell consists of a controller and two physical machines: a lifting machine and a picking machine. In this model of the manufacturing cell, the processor for the controller is connected with the processors for the controlled machines by means of interaction paths. The lifting machine has actuator oLifterUp and sensor iLifterIsUp, while the picking machine has actuator oPickerPickProduct and sensor iPickerHasProduct. The actuators and sensors are modeled by ports with corresponding names.

The first model uses no expanded processors. In the second model, the expanded processor Machines is added. The processors Controller, LiftingMachine and PickingMachine in Figure 2.1.1 have the same model as the corresponding processors in Figure 2.1.2.

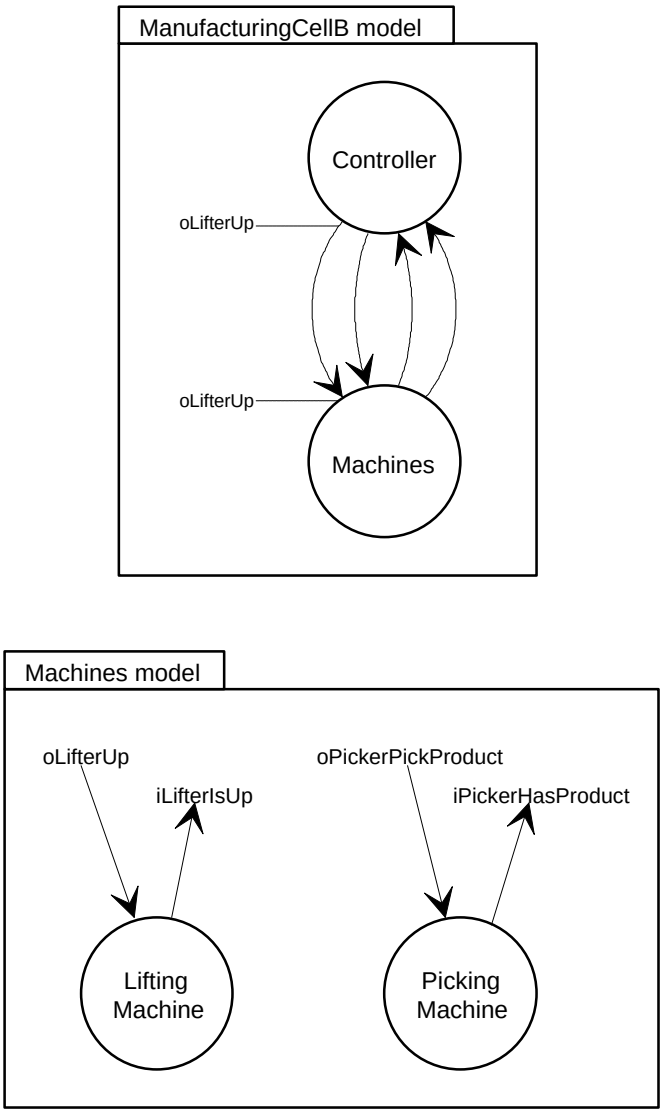

Figure 2.1.2 *Model of a manufacturing cell with an expanded processor.*

2.1.3 The use of classes for the specification of processor models

The model of an expanded processor is specified graphically, which has been shown in the previous section.

The model of a leaf processor is a process description. The language used for these process descriptions is the task language, which is based on the object-oriented programming language Smalltalk-80. For a more detailed description of Smalltalk, see for example [Goldberg and Robson, 1989]. An introduction to Smalltalk is given in Appendix A. The most important elements of the task language are given in Appendix B.

Ports are referred to by name in process descriptions. Names in Smalltalk are represented by strings (see Appendix A), such as 'oLifterUp'. An example of the representation of a send action by the processor Controller (from Figure 2.1.1) which sends the object true to the port oLifterUp is self send: true to: 'oLifterUp'.

Some classes of the Smalltalk-80 system cannot be used in the task language, and new classes for the creation of Process Calculus models have been added. The most important class which has been added is the class Bubble. This class includes, amongst others, methods used to send objects to, and to receive objects from, ports. These methods are used by leaf processors. The class Bubble also includes methods which are used for the specification of the models of expanded processors.

Bubble is an abstract class, which means that instances of Bubble are normally not created. Instead, additional subclasses of Bubble are created for all processors with a different functionality. Subclasses of Bubble can also be abstract classes. An example of a class hierarchy is shown below:

```
Bubble
    Buffer
        Fifo
        Stack
    WaferProcessingModule
        Cleaner
        Furnace
```

The classes Bubble, Buffer and WaferProcessingModule are abstract classes. In these classes, the methods which are common to their subclasses

are specified. So in the class Bubble, which is at the root of all processor classes, the methods are specified which are common to all processor classes. The classes Fifo, Stack, Cleaner and Furnace are not direct subclasses from Bubble, but they do inherit from Bubble.

The model of a processor is determined by the class of the processor. A processor is an instance of a class which inherits from the class Bubble. A class can have several instances. Processors which are instance of the same class have the same model. Such processors can be used in different models, or several times in the same model. They have the same functionality, but they need not have the same internal state. Buffer processors of the same class, for instance, can be used at various places in the same model: they will all have the same functionality, but they can each contain other buffered elements, depending on their environment.

2.1.4 Compound ports and interaction paths

An expanded processor serves as an abstraction of the detailed description of its model, which consists of other processors and interactions. This is an important concept which helps to make the complexity of systems manageable by showing only the relevant amount of detail at each level of abstraction.

Process Calculus does not provide a similar mechanism for ports and interaction paths. Ports and interaction paths cannot be 'expanded'.
In many systems, especially in control systems, it is essential that irrelevant detail in the presentation of ports and interaction paths can be hidden by using abstraction. Consider, for example, the interactions between the controlling processor ControllerC and the model of the controlled machines MachinesC in Figure 2.1.3. In the model of ManufacturingCellC, we are not interested in the exact representation of all the sensors and actuators, since there could be hundreds of them.

In order to make it possible to refer to a collection of ports or interaction paths with a single entity, we have introduced a new kind of port and interaction path: a compound port and a compound interaction path. The 'old' ports and interaction paths will be referred to as simple ports and simple interaction paths. When the type of a port or interaction path is not explicitly specified as simple or compound it can be either of the two.

Ports can be hierarchically ordered by means of compound ports. A compound port is a collection of other simple or compound ports. Objects can only be sent to and received from simple ports.
Interaction paths can be hierarchically ordered by means of compound interaction paths. A compound interaction path is a collection of other compound or simple interaction paths.

Compound ports on different processors can only be interconnected by means of compound interaction paths. Simple ports on different processors can only be connected with each other by means of simple interaction paths.
Simple interaction paths are always unidirectional. A compound interaction path can be unidirectional, bidirectional or nondirectional; the direction depends on the interaction paths which it contains. A compound interaction path is nondirectional if it either contains no interaction paths or only nondirectional interaction paths; it is unidirectional if it contains only unidirectional interaction paths with the same direction, and possibly some additional nondirectional interaction paths; and it is bidirectional if it contains bidirectional interaction paths or unidirectional interaction paths in opposite directions.

Compound ports are represented graphically by their names with hyphens as suffixes. Compound interaction paths are represented by arrows which can be unidirectional, bidirectional or nondirectional.

Figure 2.1.3 shows a model which is functionally equivalent with the models shown in Figures 2.1.1 and 2.1.2. The only difference is that Figure 2.1.3 uses compound ports and compound interaction paths. The sensors are modeled by the compound port i-. This port contains the compound ports i-lifter- and i-picker-. The port i-lifter- contains the simple port i-lifter-isUp and the port i-picker- contains the simple port i-picker-hasProduct. The actuators are likewise modeled by the compound port o-.

In the expansion of MachinesC, the names of the ports i- and o- are shown in the right-hand part of the model. No interaction paths are connected to these compound ports. The ports i- and o- contain other compound ports. These can also be used in the model to connect interaction paths. In the example, the compound ports i-picker- and o-picker- are connected by means of compound interaction paths to the processor PickingMachineC. In this way, all the simple ports contained in i-picker- (in this case only one, viz. i-picker-hasProduct) and o-picker- are connected by means of simple interaction paths to PickingMachineC. The processor LiftingMachineC is

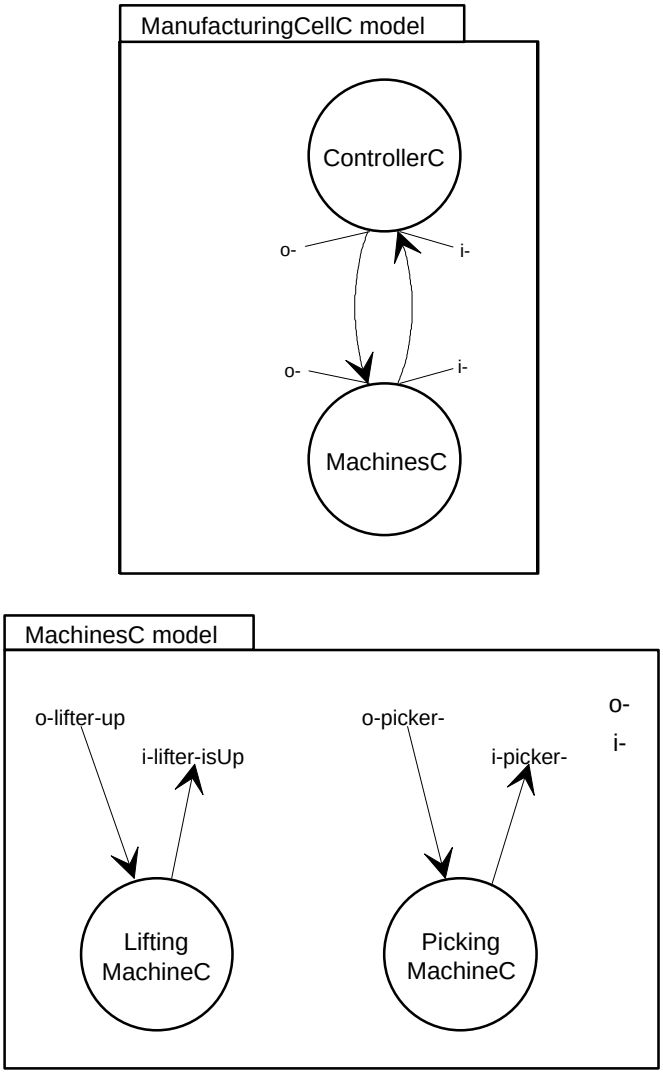

Figure 2.1.3 *Model of a manufacturing cell with an expanded processor and compound ports and interaction paths.*

directly connected by means of simple interaction paths to the ports o-lifter-up and i-lifter-isUp. These different possibilities show, by way of example, how compound and simple ports can be used in a model.

Send and receive actions can only take place on simple ports such as in, out, o-lifter-up or i-lifter-isUp. An example of a representation of a send action by

the processor ControllerC (from Figure 2.1.3) which sends the object true to the port o-lifter-up is self send: true to: 'o-lifter-up'.

2.2 The realization of controlling systems

2.2.1 Using simulation to test controlling systems

Controlling systems can be tested in two ways: they can be connected to the actual controlled system or they can be tested by means of simulation using a Process Calculus model of the controlled system.
Testing by means of simulation can have significant advantages in the following situations:

- Testing using the actual controlled system is hazardous because of the possibility of damage to the controlled system due to errors in the software of the controlling system.
- The actual controlled system is already operative and has to be taken out of production to test the new controlling system.
- The actual controlled system is not available. It may not yet have been built, or it may be at a remote location.
- The actual controlled system has a long cycle time. In this case, using the actual system for testing will take a long time. The time to simulate a control system is unrelated to the real time in the actual system, and can therefore be done more efficiently.

The main disadvantage of simulation-based testing is the time and effort needed to model the controlled system. The advantage gained by simulation-based testing of the controlling system must outweigh the effort needed to model the controlled system. In order to successfully use a model of the system, it is necessary that the model is sufficiently accurate.

2.2.2 The transition from simulation to the control of the actual system

When the controlling system has been tested using a simulation model of the controlled system, the transition from simulation to actual control must be made. This can be done relatively simple using Process Calculus. Two models can be used: one for simulation-based testing and one for the control of the actual controlled system. In both of these models the same controlling processor is used.

This approach is demonstrated in Figure 2.2.1. The model SimulatedSystem contains the processors Controller, which is the controlling processor, and the processor Machines which is a model of the controlled system. After the SimulatedSystem model has been tested using simulation, the ActualControl model can be used for the control of the actual controlled system, which system is indicated by a dotted rectangle named Machines. The two processors Controller have the same model: they are both instance of the same class Controller. Therefore their functionality is identical. In the model of ActualControl, the processor Driver replaces the processor Machines from the SimulatedSystem model. The Driver converts the objects it receives from the Controller into controlling signals to the corresponding actuators on the actual controlled machines. The Driver likewise converts the signals from the sensors to objects which are sent to the Controller.

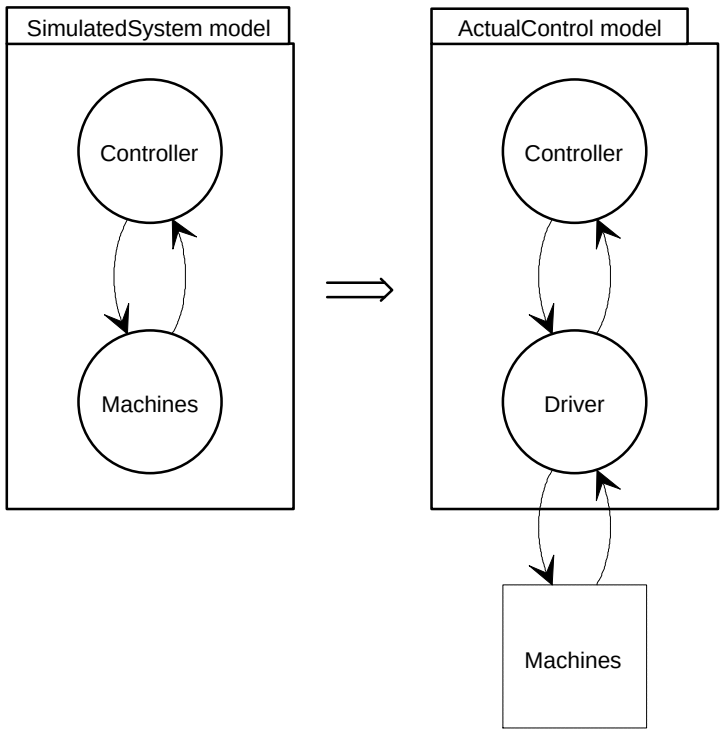

Figure 2.2.1 *Models for simulation and actual control.*

2.2.3 The interaction mechanism between the controller and the driver

The interaction mechanism between the control processor and the driver will not always be the same: it is partly determined by the type of physical interface, and types of actuator and sensor used. A motor with an accurate positioning unit which is controlled through an RS232 interface is interfaced differently from binary actuators and sensors.

The interfacing of most actuators can be done simply by sending an object to a specific port on the driver processor. The port name determines the actuator. The object received determines the desired new state of the actuator. For a binary actuator, the objects to be sent could be true and false, meaning that the actuator concerned should be turned on or off respectively.

The sensors can be periodically polled by the driver. The polling period is contained in the driver. The sensors that need to be polled are determined by the send ports present on the driver. Every send port is associated with a sensor. After every poll, the driver should send the new values of the sensors whose values have changed to the corresponding ports on the driver processor.

The controlling processors should be able to read the values of all sensors at any time, using a non-blocking receive operation. This means that the driver should at all times be ready to send the values of all sensors to all sensor ports. The easiest way to do this is to use the method Bubble >> send:continuousTo: in the driver (see Appendix B.2). In a controlling processor, a receive action from a port connected with a sensor port on the driver would then yield the value of the sensor. If a blocking receive action is desired, the required object to be received can be specified in the receive action. For example, the way to wait until the sensor connected to the port productPresent becomes active, would be to use the expression:

```
self receive: true from: 'productPresent'.
```

This receive action will remain blocked until the true object can be received from the port productPresent. The precise semantics of the method Bubble >> receive:from is explained in appendix B2.

2.3 An example: The control of an error-free transport system

2.3.1 Description of the system

The concepts treated in the previous sections will now be illustrated with an example. The example is based on an actual system for the movement of tyres around a bicycle tyre factory. The actual system, however, has been simplified and changed to yield a system which is easier to understand.
Tyres are made of rubber. A part of the production process is the vulcanization of the newly-made rubber tyres in a furnace. Tyres are transported on trays, which each carry seven tyres. The furnace operates in batch mode, so the trays are stacked up approximately twenty high.

A schematic diagram of the system is shown in Figure 2.3.1. Sensors are represented by the symbol: '>', '<', or '∧'. In the control program, the sensors will be represented by ports with the name of the sensor, prefixed by the machine part that the sensor belongs to, prefixed by an 'i' indicating input. The sensor opened in the figure, for example, is represented by the port i-holder-opened.

The trays are transported on multiple-section conveyor belts. When the trays arrive at the traversing-shuttle they are stacked. The traversing-shuttle will be referred to as 'traverse' for ease of reference. Every time a new stack of, say, twenty trays is ready, the traverse moves the stack onto a fork-lift truck. The fork-lift truck then moves the stack to the furnace and deposits it there.
The sections are meant to buffer the incoming trays when the stack is transported to the fork-lift truck. In reality, there are more sections than just the two drawn in the figure. Each section is equipped with a sensor to stop the section when the tray cannot be moved to the next section.

The traverse is equipped with four holders which hold the stack. Two of these are shown in Figure 2.3.1. The pusher consists of a plate which can move in a vertical direction to push a tray against the stack. The plate comes up between the two small side-belts of the last section. The pusher is driven by two cylinders. One of them has a large stroke to push the tray up against the stack. The stack holders will then be opened. Consequently the pusher cylinder with the small stroke will push the complete stack up, so that the stack holders can close under the new tray at the bottom of the stack. After this the pusher will

go down. The three positions of the pusher plate are detected by the sensors pusher-isDown, pusher-isMiddle and pusher-isUp.

When the pusher passes the sensor pusher-isMiddle while going down, and the stack has reached its maximum size, the traverse will transport the stack to the fork-lift truck. The traverse is movable, because in the actual system there is also a sampling station. This station can remove a single tray from the stack at approximately midway between the sensors traverse-atPusher and traverse-atFork. This station is omitted in the simplified diagram of Figure 2.3.1.

When the traverse reaches the fork-lift truck, the fork will go up, lifting the trays above the traverse. After that, the truck will go backwards in the direction of the furnace. When it reaches the position indicated by the sensor truck-canTurnToFurnace, the fork will start rotating 180 degrees to the furnace, while at the same time going down to the middle position. The truck will stop when it has reached the furnace, where it will deposit the stack. Finally, the truck will go back to the traverse.

The movements of the traverse, fork lifter, fork turner and the truck are all implemented using bidirectional motors. Each motor is controlled using two binary actuators: one for the direction and one for the power (on/off function). Sensors are installed at the extremes of all trajectories.

The part of the controlling system to be analized is restricted to the part which concerns the transfer of the stack from the traverse to the fork-lift truck and the transportation of the stack to the furnace.

The model is shown in Figure 2.3.2a. The TransporterDriver interfaces the CtrlTransporter with the sensors and actuators on the physical machine. The CtrlInterface processor interfaces with the controlling processors of the previous machines. The control processors are shown in Figures 2.3.2b and 2.3.2c. In order to keep the control simple, the machines will be initially treated as ideal: errors will thus not be taken into account. At the start of the production cycle, the machine is supposed to be in its reset position as shown in Figure 2.3.1. Chapter 7 will present a control system which takes full account of error handling.

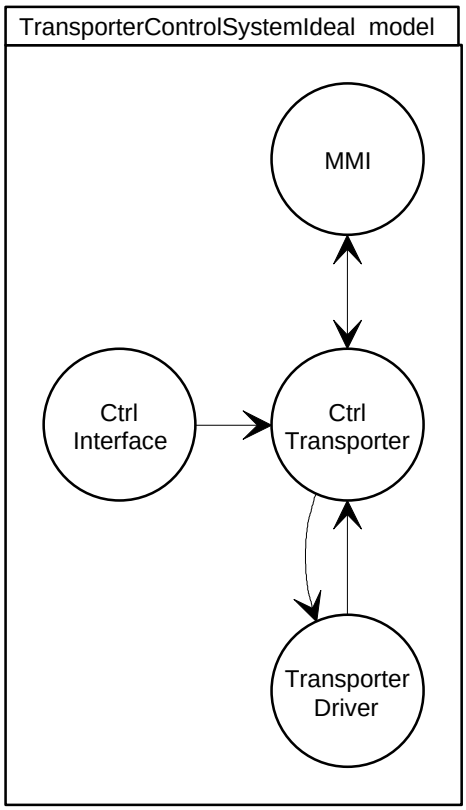

Figure 2.3.2a *Model of the controlling system of an error-free transporter.*

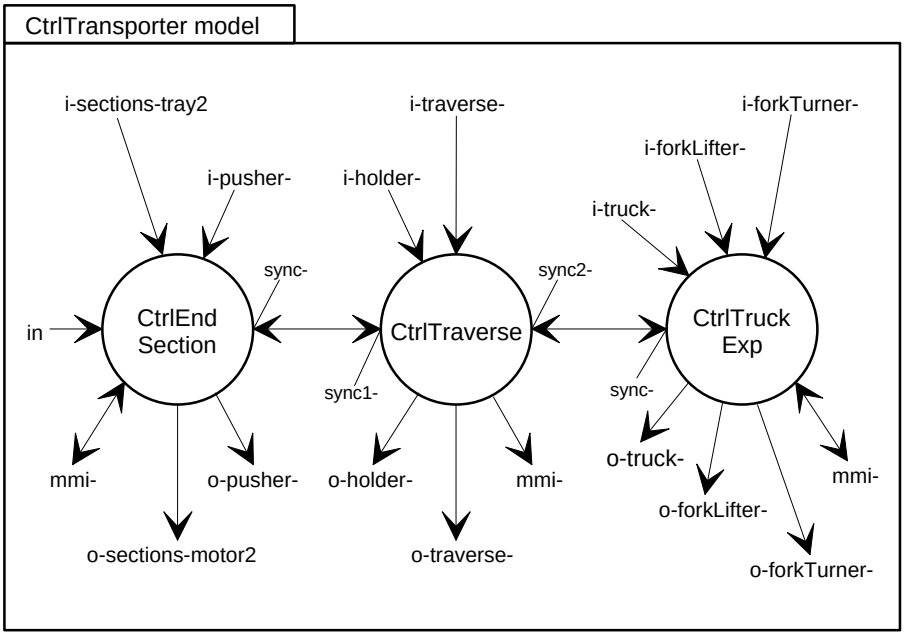

Figure 2.3.2b *Model of the CtrlTransporter processor.*

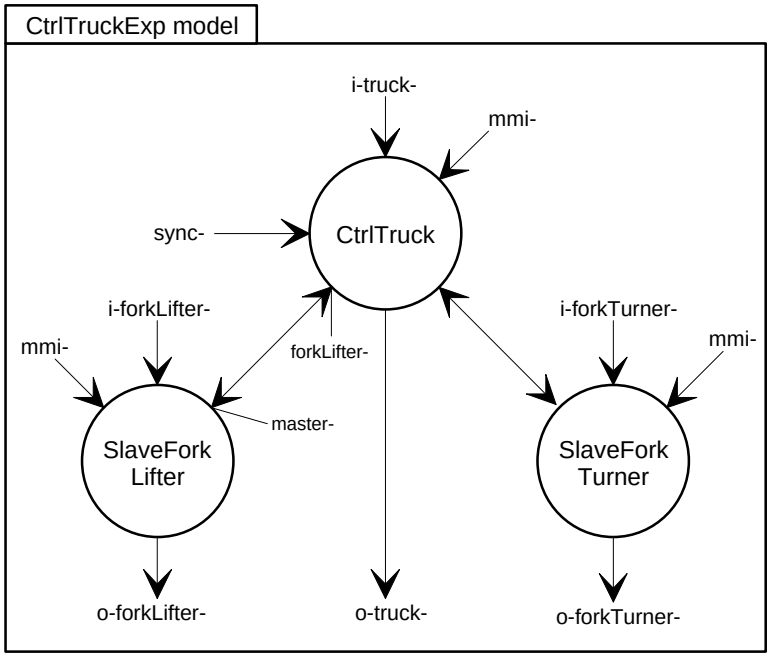

Figure 2.3.2c *Model of the CtrlTruckExp processor.*

2.3.2 Conventions used in the control model

Synchronization between controlling processors without using sensors

As is apparent from the figures, each machine part (such as the traverse, the holders and the fork lifter) is directly controlled by only one processor. This is done in order to keep the control system clearly structured. Furthermore, this approach makes it easier to reset a system, a topic which will be returned to in Chapter 7.

The controlling system is interfaced with the control system through actuators and sensors. The machine is made to change state by driving the actuators. The controlling system can then wait for the desired state change of the machine by waiting for the desired state change of the sensors.

In Process Calculus, the parallelism of control systems is modeled by separate processors. These controlling processors can have interactions amongst themselves, as well as with the actuators and sensors, by means of a

driver processor (see Figure 2.2.1). Many controlling systems can be modeled using controlling processors that only interact with the sensors and actuators and not with each other. In Process Calculus models, controlling processors usually interact both with the controlled machine and amongst themselves. The reason for this is that each machine part is controlled by a single processor. It is possible that a processor controls several sequentially-operating machine parts, but a machine part is normally not controlled by more than one processor. If a processor has to wait for a machine part controlled by another processor to reach a certain state, this synchronization is achieved by means of an interaction with the processor controlling the machine part in question, rather than by synchronization with the machine part's sensors. This is done in order to achieve a safer system in the presence of errors. This will be dealt with in greater detail in Section 7.5.5.

The connection of compound ports with compound interaction paths

Two compound ports are connected by means of a compound interaction path. The convention adopted in this case is that, unless apparent otherwise, both compound ports correspond and each of the ports of one compound port is connected through an interaction path with a corresponding port of the other compound port. Simple ports correspond when they have the same name. Compound ports correspond when they consist of the same number of ports and for each port there is a corresponding port in the other compound port. An exception to this rule is that a send port out can correspond with a receive port in. Consider Figure 2.3.2b, for example. In this figure, there is an interaction path connecting the ports sync2- on CtrlTraverse and sync- on CtrlTruckExp. The port sync2- has, amongst other ports, a send port traverseAtFork. This port is connected by means of an interaction path with a receive port traverseAtFork in the port sync- on CtrlTruckExp. When these ports are used in send or receive actions they have to be referred to using their full name as, for example, in the message: self sendTo: 'sync2-traverseAtFork'.

The grouping of methods in protocols

The methods of a processor class are grouped into protocols. The protocols used in this example are the protocols 'process control' and 'machine io'. Protocols are only used for grouping related methods. In this way methods

can be more easily located during the development of programs. Protocols are not part of the Smalltalk language.
The methods in which synchronization is only achieved by interactions with the machine's sensors and actuators are grouped under the protocol 'machine io'. The protocol 'process control' contains methods that preferably only contain interactions with other control processors and no interactions with sensors and actuators. This separation introduces a form of structuring into the code. In cases where it is desirable to mix interactions with the controlled machine and other controlling processors in a single method, these methods are also grouped under the protocol 'process control'.

Reference to methods

A method methodName defined in a class ClassName can be referred to as follows: ClassName >> methodName. Methods that belong to the same protocol of a certain class can also be listed with a heading in italics, defining the class and the protocol, followed by a list of method definitions. Each method definition begins with the name of the method in bold type, followed on the next line by the indented code for the method. Comments in methods are typed in italics between quotes: "".
For example:

```
------  ClassName protocol:  protocolName   -------

methodName1
    "code for first method"

methodName2
    "code for second method"
```

2.3.3 The implementation of the model

The send and receive actions in CtrlTraverse >> stackToTruck (shown below) are synchronization actions with the processors CtrlEndSection and CtrlTruck to achieve synchronization between the machine parts pusher, traverse and forkLifter that are controlled by the respective processors CtrlEndSection, CtrlTraverse and CtrlTruck. The methods traverseToFork and traverseToPusher that are used in the method CtrlTraverse >> stackToTruck implement a direct

control of the traverse by means of the sensors and actuators, and are not shown here.

```
------  CtrlTraverse protocol:  process control   -------

body
    stackSize >= self maxStackSize ifTrue: [self stackToTruck].
    self stackTray          "add a new tray to the stack"

stackToTruck
    self receive: 'belowMiddle' from: 'sync1-pusherState'.
    self receiveFrom: 'sync2-truckResetAtTraverse'.
    self traverseToFork.
    self sendTo: 'sync2-traverseAtFork'.
    self receiveFrom: 'sync2-forkIsUp'.
    self traverseToPusher
```

The following code is part of the code for the processor CtrlTruck. This processor drives the truck directly by means of methods in the protocol 'machine io'. Due to the parallelism, the fork lifter and the fork turner are controlled by separate processors: SlaveForkLifter and SlaveForkTurner. The coordination of the movements of the truck, fork lifter and fork turner is determined by the CtrlTruck processor. The processors SlaveForkTurner and SlaveForkLifter receive their commands from CtrlTruck. The command is translated into an action performed by the machine part that they control and an acknowledgement is sent to CtrlTruck when the action is finished.

```
-------- CtrlTruck protocol: process control --------

body
    self receiveStackFromTraverse.
    self transportStackToFurnace.
    self giveStackToFurnace.
    self goBackToTraverse

receiveStackFromTraverse
    self sendTo: 'sync-truckResetAtTraverse'.
    self receiveFrom: 'sync-traverseAtFork'.
    self send: #forkUp to: 'forkLifter-command'.
    self receiveFrom: 'forkLifter-ack'.
    self sendTo: 'sync-forkIsUp'

transportStackToFurnace
    self toTurnPointAtTraverse.
    self send: #forkDownToMiddle to: 'forkLifter-command'.
    self send: #turnToFurnace to: 'forkTurner-command'.
    self continueToFurnace.
    self receiveFrom: 'forkLifter-ack'.
    self receiveFrom: 'forkTurner-ack'


---------- CtrlTruck protocol: machine io ----------

toTurnPointAtTraverse
    self putOn: 'o-truck-toFurnace'.
    self putOn: 'o-truck-power'.
    self receive: true from: 'i-truck-canTurnToFurnace'

continueToFurnace
    self receive: true from: 'i-truck-atFurnace'.
    self putOff: 'o-truck-power'
```

In the method SlaveForkLifter >> body (shown below), the command is received from the CtrlTruck processor. The command is, for instance, the symbol #forkUp. The message self perform: #forkUp will result in the method forkUp being executed by the SlaveForkLifter processor.

```
--------- SlaveForkLifter protocol: process control ---------

body
| command |
command := self receiveFrom: 'master-command'.
self perform: command.
self sendTo: 'master-ack'


--------- SlaveForkLifter protocol: machine io ---------

forkUp
    self putOn: 'o-forkLifter-up'.
    self putOn: 'o-forkLifter-power'.
    self receive: true from: 'i-forkLifter-isUp'.
    self putOff: 'o-forkLifter-power'
```

Chapter 3
Errors

An ideal manufacturing system will generate no errors: machines never malfunction; operators do not come into dangerous zones near the machines; and once a machine has started producing an order, it will continue without interruption until the order is finished. Unfortunately, real world systems are not ideal: errors in manufacturing processes are unavoidable. In order to ensure the safety of the operators and machines and to enable efficient continuation of the production process after an error, errors will have to be handled. In fact, error handling has a major effect on the safety, reliability and efficiency of a manufacturing system.

This chapter starts with a definition of errors and related terms. Thereafter, successive sections will deal with other aspects associated with errors. This will be done in an abstract way, independent of specific implementations.

3.1 Definition of terms

3.1.1 Systems and states

In order to be able to define precisely terms like error and failure, it is necessary to start with the definition of a system. Many different definitions of a system can be found in the literature. The definition given here is based on the definitions of [Melliar-Smith and Randell, 1977] and [Lee and Anderson, 1990]. It is useful for an analysis of system correctness, errors, and faults. Since errors and faults are only relevant when the desired behaviour of a system is known, the definition is restricted to systems which have been designed to provide a specified service. The definition is as follows:

A **system** has been designed to provide a specified service to its environment and is either atomic or consists of a set of cooperating components. Components themselves are systems, so the definition is recursive.

A system is considered to be **atomic** when its internal structure cannot be discerned or is not of interest. Therefore, a system may be considered as atomic when only its specification is of interest; how its functionality is implemented is not relevant.

The behaviour of a system is determined by its design and its state.

The **design** of a system consists of the selection of the components (if present) and the interrelationships between the components themselves and between the components and the environment. The design of the system can be regarded as the algorithm of the system. The initial internal state of the system, that is the state of the system prior to any inputs, is also considered to be part of the design.

System designs are not restricted to software systems. The design of a machine without a controlling system, for example, is the mechanical construction of the machine.

There are two kinds of state: *external* and *internal* states. External states are the states of the system which are relevant for the system's environment. Internal states are relevant for the internal operation of the system. The relationship between internal and external states is as follows:

The **internal** state of a system is defined as the aggregation of the *external* states of its components. The **external** state of a system is an abstraction of its *internal* state.

It follows directly from the definitions that the external state of a system is also an abstraction of the aggregation of external states of its components.

When a system is active, it will change from one internal state to another. As a result its external state will change, but it may take several transitions from one internal state to another in order to effect one external state transition. The external states are defined by an abstraction function which maps the internal states onto external states. The mapping is done in such a way that one or more internal states are mapped onto the same external state.
External states are important if we are interested in *what* a system does, and internal states are important if we are interested in *how* a system does what

it does. The relationship between the internal states of a system and between the internal and external states is determined by the design.
The specification of a system concerns only the external behaviour of the system. As Randell et al. (1978) put it: 'The specification only defines the external states of the system, the operations that can be applied to the system, the results of these operations, and the transitions between external states caused by these operations, the internal states being inaccessible from outside the system'.

3.1.2 Specifications, goals, preconditions and failures

The **specification** of a system is the agreed description of the service that the system is designed to provide to its environment. It determines the required behaviour of the system in terms of a relationship between the inputs or input sequences to the system and the associated responses. It can be viewed as a contract between the designer and the user of the system.
A system will be designed for a specific purpose. This purpose is specified by the **goal** of the system. The goal is specified as a relationship between the inputs to the system and the desired resulting responses of the system. The goal is also referred to as the **primary goal** of the system.

Usually the specification will place certain restrictions on the system's inputs and on the behaviour of the system's environment, so that the system can realize its goal when those conditions are met. These restrictions are specified by the **precondition** of the system. The set of inputs or input sequences with the states of the system's environment that satisfy the precondition is termed the **standard domain**.

The precondition refers to the external state of the system if the system interacts with other systems. In such a case, the precondition specifies for all interactions the kind of behaviour that the system requires from the other systems, so that the system will be able to achieve its goal. The precondition requires more than just correct behaviour of the systems with which the system interacts. Consider, for example, a system which controls a cylinder. The precondition of the controlling system will specify that the air pressure supplied to the cylinder is sufficiently high. This requires more than just correct operation of the compressor, because correct operation of the compressor does not guarantee the delivery of air. The precondition of the compressor will require the availability of mains voltage. If there is no mains voltage, then the compressor will not be able to supply air. This is

still correct behaviour of the compressor, since the specification of the compressor does not require it to operate without mains voltage.

Note that the precondition is not necessarily the weakest precondition. This point is explained at the end of this section.

A system often also has a **secondary goal** which should be achieved in the event of the precondition not being met. The (primary) goal of a manipulator, for example, might be to pick up a product and deposit it somewhere else. In the case of a power failure, the secondary goal could specify that the product must not be dropped and that the manipulator remain in position.

A system is said to *operate in its standard domain* if the inputs that are applied to the system and the state of the system's environment belong to the system's standard domain.

The specification of a system thus consists of a goal, together with the associated standard domain, possibly complemented by a specification with a secondary goal for operation outside the standard domain. This complementary specification need not be defined for the complete complement of the standard domain, but can be restricted to a subset. The complete domain for which the specification is defined is the **defined domain** of the system.

In this thesis, we will only consider specifications that specify a deterministic effect for the inputs. Therefore, reliability specifications, such as the specification that a testing unit will correctly identify faulty items in 99% of the cases tested, are not considered. With such a specification it is impossible to determine whether the unit has functioned according to its specification if, for example, only one test is considered in which the unit mistakenly determines that a faulty item is satisfactory.

The specification of a system permits a comparison to be made between the actual behaviour and the required behaviour of the system, and can thus be used to define system failure. The following definition is based on the one from [Lee and Anderson, 1990].

A **failure** of a system occurs when the behaviour of the system deviates for the first time from that required by its specification.

Failure is defined only for inputs from the defined domain. If inputs from outside the defined domain are presented to a system, this is an erroneous action by the user or from the environment of the system, since the system is not being operated within its specification. Failure of the system itself is not defined for such cases: its response is undefined. The undefined system response could, however, easily result in a failure of the system which presented the erroneous inputs to the original system.

In [Lee and Anderson, 1990] it is required that the behaviour of a system is specified for all inputs. For many systems, however, this requirement is too strong. It contradicts the 'programming by contract' principle [Meyer, 1988], which implies a clear agreement about the responsibilities of the user and the implementer of an operation. If, for example, the precondition of an operation is always explicitly tested by the invoker of the operation, it is not necessary to define the operation's behaviour for invocations outside of its standard domain. Duplication of the precondition test in the operation itself would, in this case, serve no purpose and would, in fact, lead to a more complex and therefore more error-prone program.

It may, however, be interesting to observe whether or not the goal can still be satisfied outside the standard domain. This depends on how the precondition is defined. If the precondition is the weakest precondition for the specified goal, the goal cannot be satisfied for any inputs outside the standard domain. This is a consequence of the definition of the weakest precondition, which determines all inputs for which the goal can be satisfied. If the precondition is stronger than the weakest precondition, then the goal could be satisfied for those inputs outside the standard domain that satisfy the weakest precondition. Such a situation can occur when the weakest precondition is not exactly known. An example of this is the specification of a temperature range over which a component is guaranteed to achieve its goal. In this case, the goal may also be achievable for temperatures which are slightly outside the given range.

3.1.3 Correctness and errors

A **system** is **correct** if its behaviour conforms to its specification for all inputs that belong to its defined domain. Therefore, correctness of a system is only defined for the inputs belonging to its defined domain: it means that for those inputs it will never fail.

Note that this definition of correctness makes no assumption whatever about the correctness of the system's design or its components. The correctness of the system can be determined by the user of the system, using only the system's specification. The correctness of the system, its design and its components are linked in the following definition.

The **design** of a system is **correct** if, under the assumption that the system's components are correct, the system is correct when started from its initial state.

This means that an incorrect design can be corrected in three ways. First, the specification of the system can be changed. This is usually undesirable, unless the specification does not correctly express the expectations and ideas of both user and designer. In that case, there is an error in the specification. Second, the design can be changed by exchanging one or more of the system's components for a component with a more appropriate specification. Third, the design can be changed by correcting the interrelationships.
Now that the correctness of a system and design are defined, the definition of correct and erroneous internal states, and internal state errors can be given.

The **internal state** of a system is **correct** if it cannot cause system failure. More specifically, there may not exist a sequence of inputs from the system's defined domain that will lead to failure of the system, assuming that the system's components and design are correct.

The **internal state** of a system is **erroneous** if it can cause system failure. More specifically, there must exist a sequence of inputs from the system's defined domain that will lead to failure of the system, assuming that the system's components and design are correct.

An **error** in the **internal state** of a system (or an internal state error) is a part of an erroneous internal state which needs to be different in order to make the erroneous internal state correct.

The above given definition of internal state errors is more restricted than the definitions usually found in the literature such as in [Laprie, 1992] and in [Lee and Anderson, 1990]. In [Laprie, 1992] an error is defined as being *liable* to lead to a subsequent failure even though, due to (for example) redundancy the error can in practice never lead to system failure. In the view

of Lee and Anderson (1990), an internal state is erroneous if it can lead to system failure or if corrective action is needed to prevent system failure, even though failure can no longer occur due to the corrective action. They consider corrective action to be an essentially subjective element in the definition of internal state errors. Although these definitions seem to reflect the way that the term error is used by many people in the field, they cannot be used objectively in order to determine whether the internal state of a given system with a given specification is erroneous. This is due to the use of the subjective elements of *being liable*, *redundancy* and *corrective action* in the definitions. Hence the restricted definition of internal state errors is used in this thesis.

Two examples are given to illustrate the inadequacy of the unrestricted definitions of (internal state) errors.

The first example concerns an automatic insertion system which places components on a printed circuit board. Suppose that the components to be placed have pre-formed leads. If the leads are not pre-formed within certain tolerances, they cannot be directly inserted into the holes in the printed circuit board. Their leads must be correctly formed first. For this purpose, the system is equipped with a vision system to determine whether the leads are correctly formed. If they are not, the component leads are reshaped. Whether or not incorrectly formed leads are considered to be an internal state error would depend on the view of corrective action. If reshaping the leads is considered to be corrective action, then the incorrect leads would be an error. If, however, reshaping the leads is considered to be part of the normal processing, then leads that need to be reshaped would not be an error.

The second example is the fragment of a Pascal program shown below. It is used to get the name of a person. A person is asked to type his name. If the name is mistyped, for example because it contains a digit, the person is asked to type his name again.

```
repeat
    name := getName()
until nameIsCorrect(name)
```

If the name is mistyped and the second call to getName is considered to be either redundant or a corrective action necessary to prevent system failure, then the mistyped name would be an error. If it were to be considered as part of the normal processing, then the mistyped name would not be an error.

Using the restricted definition of internal state errors, the internal states in both examples are correct and therefore do not contain errors, because the internal states cannot lead to system failure.

The failure of a component of a system need not lead to errors in the system's internal state. In practice, however, the external state of such a failed component is often termed erroneous. Erroneous external states and external state errors are therefore introduced as new terms.

The **external state** of a system is **erroneous** if it deviates from the external state as required by the system's specification.

An **error** in the **external state** of a system (or an external state error) is a part of the external state which deviates from the external state as required by the system's specification.

Therefore, failure of a component will lead to errors in its external state. The internal state of a system is a collection of the external states of its components. Therefore, an error in the external state of one of a system's components can be an internal state error in the system itself, depending on whether or not it can lead to failure of the system.

An error in the internal state of a system does not necessarily lead to system failure. Certain internal state errors may go unnoticed for a long time and lead to failures only when certain input sequences are applied to the system. Consider, for example, a system with a buffer which buffers components of a certain type in a first-in last-out sequence. If an erroneous component is supplied to the buffer, followed by many correct components, the incorrect component may never be retrieved from the buffer at all, in which case the system will not fail.

In practice, it is not always possible to uniquely define internal state errors. If the design of a system is not correct, the assumption that the system's design is correct will yield a design which is different from the original one. An incorrect design can often be corrected in more than one way. These different correct designs can lead to the determination of different sets of internal state errors.

Another type of error is a precondition error, which is defined as follows:

A **precondition error** is that part of the input to a system or part of the state of the system's environment which does not satisfy the system's precondition.

Precondition errors are not internal state errors, since they do not refer to a system's internal state.
The term precondition error is taken from [Srinivas, 1978]. The use of preconditions is especially associated with Hoare, who introduced pre- and postconditions in [Hoare, 1969] to specify the meaning and prove the correctness of programs. Note that a precondition error is *not* an error in a precondition. It is the input that is erroneous, and not the precondition itself.

The specification of the goal of a system determines the precondition necessary to enable the goal to be achieved and thus directly affects the presence of precondition errors in inputs. Consider for example a robot that assembles two components. If assembly is not possible because of inaccuracies in the component's dimensions, the components could be placed in a separate container. If the goal of the assembly process is specified to be the assembly of the components, then inaccurate dimensions would be a precondition error. If the goal were to be specified as the assembly of the components when possible and the placement of those components with inaccurate dimensions in a separate container, then inaccurate dimensions would not be a precondition error.

The concepts defined so far will be clarified using an example of a simple transport system consisting of a conveyor belt with controller. The belt carries similar products that are spaced apart at a certain distance. When a product is detected near the end of the belt, the belt is stopped for a predefined period of time and a pneumatic cylinder will extend to push the product off the belt. The system consists of the following components: controller, cylinder, belt, component detector and the products.

Suppose the air supply to the cylinder ceases. When the first product is detected near the end of the belt, the belt will be stopped. The controller will then drive the cylinder valves in order to make the cylinder push. Since there is no air pressure, the cylinder will not extend. The absence of pressure is a *precondition error* for the cylinder: it makes it impossible for the cylinder to satisfy its goal. The cylinder does not fail, however, because it is specified not to operate when the air pressure is too low. If the system is specified to push a product off the belt within half a second after the belt has been stopped, the system will *fail* if the cylinder does not extend on time. The

internal state error which caused system failure is the absence of air pressure. The position of the cylinder shaft, which is retracted instead of extended, will be an *external state error* of the system.
The controller could detect that the cylinder does not extend. For this purpose, the cylinder can be supplied with a limit detector to indicate when it is fully extended. If, after activation of the cylinder valves, the limit detector is not activated within a predefined period of time, the controller can detect this external state error of the transport system by means of a time-out. It can then alert the operator. After restoration of the air supply by the operator, the controller can retry activating the cylinder so that the system can continue successfully. If the transport system is considered to be a component in the encompassing production system which also includes the operator as a component, then failure of the transport system need not lead to internal state errors in the production system. This is the case when the operator can recover from the external state error of the transport system and keep the production system operating within the specifications.

3.1.4 Faults

There are some related definitions of faults in the literature. In [Lee and Anderson, 1990] faults are considered to be errors at a certain hierarchical level. They state that a *fault in a system* is an error in the internal state of one of its components – which is termed a *component fault* – or an error in the design of the system – which is termed a *design fault*. In [Melliar-Smith et al., 1976] and [Randell et al., 1978] faults are defined to be the *mechanical or algorithmic cause of an error*, whereas errors are only defined in internal states. In [Laprie, 1989] faults are defined as the *adjudged or hypothesized cause of an error*.

A fault in a system can remain undetected for a long time. A fault will only affect the operation of the system when the part of the system containing the fault is used. When a fault causes an erroneous internal state or failure, this is known as the *manifestation* of the fault.

3.1.5 Robustness

The robustness of a system concerns the correctness of the system and the way in which the system responds to inputs that do not belong to its

standard domain. A system is **robust** when it satisfies the following two requirements:

- All inputs that can possibly be offered to the system are in the defined domain, so that for all possible inputs to the system the system's response is defined.
- The system is correct.

This means that a robust system has a defined response, which conforms to the specifications, for all possible inputs.

Robustness and correctness can be achieved for small and simple systems like known mathematical algorithms, which remain unchanged over a long time: correctness can even be proved. For complex, practical systems the realization of a robust system is a target which should be aimed at. It is, however, practically impossible to actually achieve such a target. Even achieving correctness of complex systems is extremely difficult: in practice, all large and complex systems contain some residual errors. Achieving robustness is even more difficult, because large and complex systems can take an enormous number of undesirable inputs for which the system's response must nonetheless be defined.

Current research thus does not only focus on achieving 100% error-free software. It is accepted that some errors will inevitably remain undetected in complex systems. However, the employment of redundancy can lead to the achievement of fault tolerance, so that higher levels of a system can correct failures at lower levels, making it possible for the top level of the system to continue to function (within reasonable limits). See for example [Bendell and Mellor, 1986]. Such techniques can help in creating robust systems. They are, however, not treated in this thesis.

3.2 Some general concepts regarding errors

3.2.1 The causes of precondition errors and internal state errors

We first consider the behaviour of a correct system consisting of correct components and thus also of a correct design. Even if all inputs belong to the defined domain of the system, inputs that do not satisfy the system's precondition can still cause precondition errors. It seems useless to offer such inputs to a system, since the system's goal cannot be satisfied in that

case. It cannot, however, always be avoided. Firstly, when the same input to the system can be offered from several places in the system's environment, it can be more economical to test the precondition in the system itself, where the test need only be performed at one place, rather than test the precondition at every place where the input is offered to the system. Secondly, it may be impossible (or highly impractical) to explicitly test the precondition. When a robot needs to assemble two components, it can be easier to just try to assemble the two components and see whether they fit, rather than to check the components in advance. Even if they are checked in advance – by (say) using vision techniques – the tester can fail by letting faulty components pass, which will lead to a precondition error in the assembly process.

Incorrect designs are an evident cause of internal state errors. This can be illustrated by the following example. Referring to the conveyor belt example in Section 3.1.3, let us assume that the cylinder is misaligned with the position of the product detector. This will cause the belt to stop at the wrong moment, so the cylinder will miss the product when it extends. The external states of the cylinder, the belt and the product are all correct when regarded individually; it is their incompatibility which causes the internal state of the control system to be erroneous.

A simple error can lead to many other errors; this is known as error propagation. The process of error propagation is a chain which starts with an error, which error leads to failures, which in turn lead to further errors, and so on.
The following situation is an example of error propagation. If the product detector fails due to a fault in the detector, the controlling system will not be able to detect an arriving product. The error in the external state of the product detector is an internal state error in the control system, causing its failure when the belt does not stop and the cylinder does not extend when the component is in front of the detector. This leads to errors in the internal and external states of the control system. Note that, as in the previous example, the internal state of the control system is erroneous, but the external states of the belt, the cylinder and the product themselves are correct.

3.2.2 Errors in the controlling and controlled system

Errors in the controlling system

These errors can be divided into hardware and software errors.

Hardware errors are errors in the computer hardware. They will normally be very rare. They are possible, however, and should be taken into account in order to avoid serious damage to the machine, should they occur. An example of this type of error is a bus error (a conflict on the internal computer bus) or a memory read error. Errors of this kind should be detected by the operating system of the computer and brought to the attention of the user control program. In this way, the control program can take the necessary actions to bring the machine to a safe position and report the error. Often specialized hardware, such as watchdog timers, is used in order to detect and handle fatal program failures or crashes.

Software errors are errors in the control program. They can appear in many different forms. Some examples of software errors are:

- Parameters may be incorrectly adjusted. A product may be placed at an incorrect position. This may lead to faulty products or to actual errors that cause part of the production process to fail.
- Synchronization of processes may be incorrect. Since control systems are inherently parallel, the state of the machine will be changed by a number of different processes. If a process controls a machine part, it must be certain that the operations of that part do not conflict with operations of machine parts controlled by other processes. Incorrect synchronization may lead to collisions of machine parts.
- Mathematical algorithms may be incorrect. This may lead to precondition errors such as division by zero or reference to a non-existent element in an array. If incorrect algorithms lead to such precondition errors, they can be detected by the program and brought to the attention of the user by means of an error message. If incorrect algorithms do not lead to detectable precondition errors or to a violation of invariants that are checked by the program, they will lead to failure of the system without warning.

Obviously, it is preferable not to have errors resulting from incorrect designs, rather than to detect and correct them. Prevention of such errors can be achieved through the careful analysis and specification of the system.

This should be followed by careful implementation of the control program, and possibly by simulation of the system.

Errors in the controlled system

Errors in the controlled system can have many different causes. They can be a result of errors in the machine parts, errors in the materials used in the production process, or an incorrect design of the controlling or controlled system.

Machine parts may fail in various ways. Detectors may not be correctly adjusted, causing them to detect at the wrong moment. A motor may not respond when power is applied, or it may suddenly cease to operate due to overheating.
Failures may also be caused by wear, lack of maintenance or incorrect adjustment. A cylinder may start to extend and retract more slowly over a period of time. This may be fatal in time-critical applications.
Tolerances in the dimensions of materials can result in errors, for example resulting in impossible assembly operations. Tolerances in the positioning of products when handled by machines can result in errors. A robot may not be able to pick up a product if the gripper is not properly aligned with the product.

Comparison of errors in the controlling and controlled system

Errors in the controlling system are usually less frequent than errors in the controlled system. The computer hardware is usually much more reliable than the machines it controls. Many errors in the computer software can be found using simulation models and through real-time testing. The problem with testing, be it with the aid of simulation models or in real-time, however, is that it can only be used to indicate the presence of errors and not their absence.
Since computer software does not suffer from deterioration over time, software errors will not recur once they have been corrected. The only cause of new software errors over time is a changed demand, resulting in the making of new specifications and changes to the software.
Unfortunately, the controlled machines do suffer from deterioration over time due to wear and ageing, possibly introducing new errors. They also exhibit a stochastic behaviour in certain respects. Machines and robots

inevitably suffer from imprecise positioning, and all components have tolerances in their dimensions. In the controlled systems, faults can recur after they have been corrected. Machine errors are therefore more frequent than other kinds of errors. Fortunately, machine errors can be handled more easily than software errors. This is because the designer of a system knows in advance that certain machine errors can occur and can then take measures to handle the errors. It is also known that software errors may occur, but one does not know exactly what kinds of error will occur. If one were to know this, immediate action would be taken to correct the errors. Therefore, in this work emphasis will be placed on the handling of errors in the controlled system.

3.2.3 The three stages of error handling

The handling of errors can be divided into three consecutive stages:

- error detection
- error diagnosis and damage confinement
- error recovery and fault repair.

Error detection deals with the detection of errors, either by an operator or, preferably, by the control program. Error diagnosis deals with the determination of all errors in the system which are related to the detected error. Damage confinement is used to prevent further propagation of errors. Error recovery is the part of the error handling process in which the system is transformed from an erroneous state to an error-free state, so that normal system operation can continue.

Practical systems vary greatly in respect of the complexity of the error handling techniques employed. The simplest systems may rely entirely on operators for the detection and recovery of errors. In these systems, the whole system often needs to be reset after an error. The most advanced systems use artificial intelligence techniques. These systems employ fully automatic error detection, diagnosis and recovery for most errors. Only the parts of the production system that cannot sensibly continue due to the error will be affected.

In this thesis, attention will focus primarily on control systems that do not employ artificial intelligence techniques for recovery from errors. In the

ensuing Sections 3.3, 3.4, and 3.5, the three stages in the handling of errors will be further elaborated on.

3.3 Error detection

3.3.1 The importance of early error detection

A fault or error in a component can lead to the failure of that component, leading to an error in its external state. If the component is a part of another component, then the error in the external state of the first component can be an error in (the internal state of) the second component. This error can, in turn, lead to failure. In such a way faults, failures and errors can spread throughout the components of a system. Clearly, errors should be detected as early as possible, in order to prevent their propagation through the system.
Even if there is no danger of errors spreading throughout the system, errors should be detected as early as possible for reasons of damage confinement and efficiency. It may seem that certain control actions can be effected without error detection. A control program which is waiting for a detector to be activated will be blocked if the detector is not activated due to an error. This will eventually be noticed by the operator, who has to discover the cause of the error himself. This is, however, very inefficient in terms of the resultant throughput of the system.

3.3.2 The use of sensors

In order to detect errors in the controlled system, its state must be known. The control program can determine this state by means of its sensors, or by enquiring about the status of intelligent machine parts with built-in controllers. Apart from this latter case, the detection of errors in the production process will be accomplished by reading the sensors. If an error is detected, it can be very difficult for the control program to determine the cause of the error. This depends on the number of sensors used for error detection.

Controlling systems use two main methods to detect errors in the controlled system: time-outs and state checks.

3.3.3 Time-outs

It is relatively easy to detect errors using time-outs. This notion is inspired by the observation that most errors do not occur at random moments, but rather in response to stimuli from the controlling system. A cylinder, for example, could be obstructed when extending, so that its end position detection switch would not be activated. Clearly, this error can only occur when the cylinder is first made to extend by the control program. The error can be detected by the control program by specifying a maximum time to wait before the switch is activated. Since the program will need to wait for the switch in any case, the only extra thing to do is to specify a maximum time to wait. This method of error detection is thus based on the assumption that certain operations performed by the machine should be finished within a certain period of time. The controlling system will specify a maximum time to wait for all such operations. When this period of time is exceeded, a time-out will occur, indicating an error. The time-out method can only be used in sensor-driven systems in which the computer uses sensors for every action it orders the machine to do. Using sensors is the only way to check the completion of the actions of the machine. If, for example, a robot is made to pick up a product, it could be made to close its fingers, wait a short time and then continue, but, without a sensor, there is no way of telling whether there is actually a product in the robot's hand.

3.3.4 State checks

Detecting errors using state checks is accomplished by testing the state of the controlled system by means of the sensors. If the established state is different from what is expected, an error is detected. The error can be due to an incorrect state of the controlled system, or to an error in the controlling system itself.

The testing of the controlled system's state can take place at explicit points in the program. Consider, for example, two machine parts that move through the same space. Before moving into the shared space, the controller of each part could check that the other part is not in the shared space.

The testing could also be effected during an operation. Consider a robot moving a part to another place. During the movement, the detector which indicates the presence of the part in the robot's hand could be continuously tested.

Some checks must be performed throughout the control program. This is necessary to detect errors that can take place in large parts of the production cycle. The emergency switch, for example, can be pressed at any time and requires immediate action. Thus this event must be monitored at all times. Intervention of an operator can also take place at any point in the control cycle. Therefore, all actions of the control program that take more time than the operator is prepared to wait must be interruptible.

3.3.5 Error detection by the supporting system

Errors in the supporting hardware cannot be detected by user programs. Examples of this kind of error are memory faults. Other errors, such as precondition errors in the inputs of system calls, are more easily detected by the supporting system than by the user program. Examples of this kind of error are division by zero and indexing a non-existing element in an array. Not only should these errors be detected by the supporting system, but a mechanism must also exist to bring such errors to the attention of the user program so that it can take the necessary actions to handle the error.

3.4 Error diagnosis and damage confinement

3.4.1 Definitions

As shown in Section 3.3.1, a detected error can be the result of other errors in the system. Errors propagate through the system, and only when they are detected can further error propagation be prevented. This is known as damage confinement.

In the context of error handling in manufacturing processes, damage can have two meanings.
The first meaning of damage is the total of all errors in the system that are related to the detected error. As shown in Section 3.3.1, a single fault can cause many errors to spread through the system. Therefore, when one of these propagated errors is detected, many more may exist due to the original fault. Determining all the related errors, and the fault which caused them, is known as error diagnosis.
Damage can also mean physical damage, such as personal injury, damage to machines or to the products being processed.

3.4.2 Error diagnosis

Error diagnosis need not be an explicit phase in the handling of errors: it can take place implicitly. The point in the controlling process at which a machine error is detected approximately determines the state of the machine. A direct approximation of the resultant damage can be made from this approximated state of the machine, together with the detected error. Consider, for example, a robot holding a product in its hand. The robot takes the product to another station. If the product-sensing detector were suddenly no longer to indicate the presence of the product, this would imply an error. If the error occurred while the robot's hand moved over a transporting belt, the robot hand controller would assume that the product was dropped onto the belt, possibly causing other errors. Without further testing, the controller would signal the error to the belt controller. Here the outcome of the error diagnosis stage was that the belt was also affected by the error. The only information used to come to this conclusion was the detected error itself and the implicitly available state of the error detecting process. If the damage cannot be assessed accurately enough because, for example, not enough detectors are available, then the human operator can be used to assess the damage.

3.4.3 Damage confinement

It is the task of the control program to detect errors as soon as possible in order to prevent further damage. The control program cannot usually determine the exact cause of a detected error. Therefore, it should immediately bring all parts of the machine that could lead to further damage into a safe state. This should be done in response to all errors interrupting the normal flow of control in a process.

Let us consider a simple automatically guided vehicle which goes back and forth between two positions. Suppose that a controlling process switches the motor of this vehicle on. The motor must be switched off when the vehicle reaches its destination. If, however, the normal flow of control of the controlling process is interrupted due to an error shortly after the motor has been switched on, the vehicle would continue uncontrolled. To prevent this from happening, the controlling process should be designed in such a way that the vehicle is always stopped when the flow of control is interrupted due to an error.

Hardware measures for damage confinement are needed, in addition to the software measures. A frequently used device for this purpose is a watchdog timer that should be periodically reset by the software. If it is not reset in time, for example due to a software crash, the timer automatically disables the outputs of the computer system and sets them to a predefined state.

In many cases, the damage cannot be exactly determined. In these cases, the control program should make a realistic worst-case assumption to prevent further damage. Consider, for example, a simple fork-lift truck carrying a heavy load. Suppose that the truck moves a short distance to another location and simultaneously the fork moves upwards. If the controller detects an error because the truck does not reach its destination in time, this could be due to failure of the truck detector. In this case, stopping the truck would suffice. But the controlling system should stop both the truck and the fork movements, since a collision with another object is also possible.

In most situations, the error should be reported to the operator. The operator can then check the system to see if the damage prevention actions of the controlling system have been sufficient. If the controlling system cannot automatically recover from the error, the operator can assess the damage and decide how to recover from the error.

It is not always obvious how machine parts should be brought into a safe state. The control program usually does not know exactly what has happened. In many cases, however, there is a rather straightforward solution to this problem, based on the safety requirements regarding emergency stops by the operator. This is explained in the following two sections.

3.4.4 Emergency stops

Since the control system has a limited number of detectors, it can only have a limited view of the production process and cannot detect all errors. Therefore, any production system must have emergency buttons which can be pressed by the operator in the case of a serious error which has not been detected by the controlling system. The emergency button will be pressed, for example, when the operator sees that another person is in danger of being injured by the machine.

The pressing of an emergency button must bring the machine or a part of it to an immediate stop. This can be done in two ways: electromechanically,

by having the emergency switch directly switch off the power supply of all machine parts; or the control program can detect the activation of the emergency button and then bring the machine into a safe state by control commands.
The safest way to deal with emergency stops is to use both ways. This has two advantages. If the control program were to fail because of a software error, the machine is switched off anyway by direct interruption of the power supply. On the other hand, a person could accidentally deactivate the emergency button prematurely. In this case, the machine will not suddenly become operational again, because it is also kept in a safe state by the control software. It is not always possible to use this kind of error handling in the case of an emergency stop: in complex and critical systems such as nuclear power plants, it is not possible to simply switch off the power supply in the case of an emergency stop. In such systems, the control software should be reliable enough to handle emergency stops by itself.

3.4.5 The safe state of machine parts

Pressing the emergency button will often switch off all energy supplies to the controlled system. As a result of this, the machine parts should automatically go into a safe state. A safe state is a state from which no damage to machines or products and no human injury will result and which will not lead to any additional errors. It follows that the machine parts should be constructed in such a way that they automatically go to a safe state when the power supply is disconnected. The safe state of a machine part depends on its function. Preferably the safe state of a moving machine part should be such that it can be freely moved by manual power. In this way, a person who is stuck in the machine can be released.

In the case of a cylinder, for example, a type with two valves could be chosen. Under normal operating conditions, one of the two valves will be closed and the other one open. If power is removed from the valves they will either both open or both close. The cylinder shaft should then be freely movable.
In some cases, a freely movable part is impractical. A machine part which lifts a heavy load, for example, should not be left to move freely after an emergency stop: rather, it should be blocked to prevent the heavy load from falling. In this case, a brake could be put on the cylinder shaft. The brake should be free when power is applied to it, and it should be on when the power is switched off.

The safe state that a machine part should be brought into in the case of an error can often be precisely the state it goes to after an emergency stop. This follows from the fact that both after an emergency stop and after another error during the production process the main goals are often the same: the prevention of possible personal injury and damage to the machine. There are also some differences. An emergency stop should only be used in emergencies. Therefore, it really does not matter if production time is lost due to an emergency stop. For less serious errors, however, it is important to handle errors as efficiently as possible. In many cases, bringing a machine part to the single safe state is a simple and reliable concept.

3.5 Error recovery and fault repair

If a system is in an erroneous state, there are two possibilities for returning the system to a correct state.

Selective changes can be made to the state in order to remove the errors. This is known as forward error recovery, and it is suitable for control systems. It is mainly used to recover from anticipated errors.
The other possibility is to discard the current state completely and return to a previously recorded state of the system. This is termed restoring the state of the system. If the errors which caused the state to be erroneous occurred after the recording of the state, then the restored state will be error-free. This is known as backward error recovery, and it can be used effectively to recover from unanticipated damage. It is, however, difficult to use when physical processes are controlled.

The following sections will deal with state restoration and forward and backward error recovery.

3.5.1 Backward error recovery and state restoration

The simplest form of state restoration uses fixed states to which a system can return. If there is only one fixed initial state to which the system can return, this state is known as the reset state and state restoration will mean resetting the system. An example of this is found in personal computer systems: if a fatal error occurs, such that the system no longer responds to the user, then the user can reset the computer.

A more powerful form of state restoration can restore the state of the system to certain states which have previously occurred, taking into account all events that have happened in the system. The point in time at which the state of a system is recorded for a possible subsequent restoration is a recovery point. Restoring the recorded state belonging to a recovery point is known as restoring the recovery point.

Backward error recovery can be used effectively to recover from unanticipated errors. This is because no knowledge is required about the erroneous state. The only thing that is important is that the state at the recovery point to be restored is error-free and consistent.

The problem with backward error recovery is that it only works with recoverable components. This is not a problem in pure software systems, where values can be automatically restored independently of their current values. Physical systems, however, are usually not so easily recoverable: if a hole has been drilled, it cannot be undrilled. In order to make such a component recoverable, it could be replaced by another still intact component. The recovery of a robot to a previous state would imply moving the robot itself, and all its parts, back to their previous positions.

Other problems arise when concurrent processes are used. These problems arise from the fact that processes interact for the purposes of synchronization and information exchange. If a recovery point is restored in a process which has interacted with other processes after the recovery point, then these other processes will also need to be restored to previous recovery points. The recovery of these processes may again trigger recovery in other processes, including the original one. This effect is known as the *domino* effect [Randell, 1977].

The technique of recovery blocks [Horning et al., 1974] uses a recovery point. This technique is based on supplying redundant algorithms for the same task. A recovery point is established before entering the recovery block. The task is first executed by the primary algorithm. An acceptance test must be specified to determine if the task has been successfully fulfilled. If not, the recovery point is restored and another algorithm is tried.
Because recovery blocks use a recovery point, they share the problems associated with recovery points discussed above.

3.5.2 Forward error recovery

Forward error recovery methods make incremental changes in the current erroneous state in order to come to a correct state. In order to be able to make the right changes in the erroneous state, the state has to be fairly accurately known. Therefore, forward error propagation is especially useful for recovering from anticipated errors. Consider, for example, an automatic part feeder. If the feeder were regularly to suffer from jammed parts, it could be equipped with an air jet propulsion system. Upon detection of a jammed part, a forward error recovery mechanism could automatically blow the jammed part out of the way.

As shown in the previous section, resetting a system to a predefined state is a form of state restoration. Resetting a system is also possible using forward error recovery. In fact, this is a more powerful form of resetting. Resetting a system using state restoration completely discards all information relating to the previous state. In many cases, it is necessary or useful to take the present state into account when resetting a system. Consider, for example, a process which takes finished products from a transporting belt and puts them into a container at predefined places. If an error occurs which forces the resetting of the system, the initial state of the process cannot be completely restored, since the system must remember the places in the container that are already occupied.

Because of the simplicity of forward error recovery and the problems associated with backward error recovery, this thesis will be limited to a treatment of forward error recovery.

3.5.3 Fault repair

Error recovery and fault repair can be viewed as two distinct phases. Error recovery enables the process to recover from the error, and fault repair will correct the fault which caused the error, thereby preventing recurrence of the same or a similar error. Fault repair is usually part of the error recovery process. Consider, for example, a vehicle which is blocked by an obstacle. If the obstacle is removed so that the vehicle can continue, fault repair is part of the error recovery. If, however, the vehicle recovers from the error by moving around the obstacle, fault repair could take place later by removing the obstacle.

3.6 Summary

The definition of errors is based on the definition of a *system* as either being atomic or consisting of a set of cooperating components. The behaviour of a system is determined by its design and its state. There are two kinds of state: *external* and *internal* states. External states are the states of the system which are relevant for the system's environment. Internal states are relevant for the internal operation of the system. The *internal state* of a system is defined as the aggregation of the external states of its components. The *external* state of a system is an abstraction of its internal state.
A system will be designed for a specific purpose. This purpose is specified by the *goal* of the system. Usually, the specification will place certain restrictions on the system's inputs and on the state of the system's environment, so that the system can realize its goal when those conditions are met. These restrictions are specified by the *precondition* of the system. The complete domain for which the system's specification is defined is the *defined domain* of the system. A *failure* of a system occurs when the behaviour of the system deviates for the first time from that required by its specification.

The definitions given above are used in the definition of errors. In order to avoid the ambiguity in many of the definitions of errors in the literature, different kinds of error have been distinguished. The definitions are as follows.

The **internal state** of a system is **erroneous** if it can cause system failure. More specifically, there must exist a sequence of inputs from the system's defined domain that will lead to failure of the system, assuming that the system's components and design are correct.

An **error** in the **internal state** of a system (or an internal state error) is a part of an erroneous internal state which needs to be different in order to make the erroneous internal state correct.

An **error** in the **external state** of a system (or an external state error) is a part of the external state which deviates from the external state as required by the system's specification.

A **precondition error** is that part of the input to a system or part of the state of the system's environment which does not satisfy the system's precondition.

Some different causes of errors have been treated, and errors in the controlling and controlled system have been distinguished. Errors in the controlled system have been shown to be more frequent than errors in the controlling system, mainly because errors in the controlled system can recur once corrected, due to their stochastic nature. Software errors will not recur once they have been corrected.

Finally, the three stages of error handling have been treated. These are error detection, error diagnosis and damage confinement, and finally error recovery and fault repair.

Chapter 4
Basics of exception handling

An error is a common concept, although the exact meaning of errors is not as easily defined as might appear at first sight, as has been shown in the previous chapter. Exceptions are related to errors. The definition of exceptions is even more difficult than the definition of errors. It appears that many papers differ in their definitions of exceptions, sometimes exceptions are not defined at all. Some existing definitions of exceptions are evaluated in this chapter, resulting in a definition of exceptions and related terms. The relationship between errors and exceptions is also investigated. Finally, several mechanisms are treated for the handling of exceptions in a sequential process.

4.1 Definition of terms

4.1.1 Operations

An exception is always related to an operation in a sequential process. An operation is a special kind of system. The meaning of operation is not restricted to procedures or functions. An **operation** is a logically related group of statements or expressions in a sequential process with a single entry.

The meaning of goal and precondition has already been given in relation to systems in Section 3.1.2. They will be repeated briefly here.
An operation is designed for a specific purpose. This purpose is specified by the **goal** of the operation. The goal is a relation between the inputs to the operation and the resulting responses of the operation.
The **precondition** of an operation specifies the restrictions on the operation's inputs and on the state of the operation's environment, so that the operation can realize its goal when those conditions are met. The precondition refers to the external state of the operation if the operation interacts with operations in other processes.

4.1.2 Exceptions, exception occurrences and exception conditions

Unfortunately, many different definitions of exceptions exist. Some of these definitions are cited in order to give the reader an impression of the different ideas about exceptions in the literature.

In [Young, 1982] the term exception is used to denote the occurrence of an error. Horowitz (1983) explains that the term exception is chosen to emphasize that the condition which arises need not be an error, but merely some event whose possibility requires versatility of action. Dony (1990) defines an exception as a situation leading to the impossibility of finishing a computation, whereas Cox and Gehani (1989) define an exception as an event that occurs infrequently, possibly indicating an error. A quite different definition is given by Szalas (1985): 'Exceptions are rare situations detected by a run-time system or by a user program.' Lee and Anderson (1990) use the component model of a system for the definition of exceptions. They do not give a formal definition of exceptions, but rather state that: 'The abnormal responses from a component are commonly referred to as exceptional responses or *exceptions*, particularly in software systems.' In [Knudsen, 1987] the following definitions are given: '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.'

Probably the most frequently cited article about exceptions is [Goodenough, 1975], which covers many exception handling concepts. Goodenough defines exception conditions as follows: 'Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker.' Another frequently cited article is [Liskov & Snyder, 1979]. Like Goodenough, these authors refer to exception conditions, which they also call exceptions. They do not give a formal definition, but state that: "The term 'exception' is chosen because, unlike the term 'error', it does not imply that anything is wrong." Christian, in his articles [Christian, 1982, 1984], defines an exception occurrence as an invocation of an operation or program in its exceptional domain, where the exceptional domain contains all initial states for which the goal of the operation or program cannot be reached by its normal execution.

A definition of exceptions should describe their most essential characteristics. It should also be consistent with the use of exception in the

terminology *exception condition*, *exception occurrence*, *exception declaration*, *exception handler* and *signaling, raising, catching* and *handling an exception*.
The most essential characteristic of exceptions is also the common part of many different exception definitions. It is the fact that, when an exception occurs during the execution of an operation, the operation cannot achieve its goal, for example, due to division by zero or due to the pressing of the emergency button by the operator.

Here exceptions are considered only in relation to programs, not in physical systems themselves. This is because the terminology of *signaling and handling of exceptions* is based on programs and is not relevant to physical systems.

The term exception is used in many different ways, which makes it difficult to define exceptions accurately. They can be defined more easily if the term exception occurrence is defined first. Its definition is related to the definition given by Knudsen (1987) and Christian (1982, 1984):

An **exception occurrence** is a computational state such that some invoked operation cannot realize its goal.

Clearly, this definition implies that the determination of exception occurrences is dependent on what is regarded as the goal of an operation. The word computation is used to emphasize that exception occurrences are related to the execution of a program. They occur and must be handled at run-time.
The computational state can be restricted to the state of the process executing the operation, but it can also include parts of the state of the process's environment.

The invoked operation which cannot realize its goal can be any operation in the call chain of the process: it need not be the operation called last. Consider, for example, the inversion of a singular matrix. If this operation is attempted, all states reached in this operation and in operations that are subsequently called will be exception occurrences. This is due to the fact that the inversion operation will not be able to reach its goal, even though operations called by the inversion operation to perform the desired calculations will be able to reach their goals.

This definition comes close to the definitions in [Christian, 1982, 1984]. There are, however, some important differences:

First, in our definition, an exception occurrence is not exclusively related to initial states, as in Christian's definition. This is because exception occurrences can also be related to the state of the environment of the process executing the operation, such as the state of an emergency button. The cause of this difference is that Christian's definition is based on *internal* exceptions and disregards *external* exceptions. (Internal and external exceptions will be defined later in this section.)

Second, the use of the term *normal execution* is avoided in our definition, because 'normal' is essentially a subjective term and, even more importantly, because it suggests that the operation could realize its goal in an abnormal way. The essence of an exception occurrence, however, is that some invoked operation cannot (in any way) realize its goal. It is important that the passive voice is not used in this statement, because it may be possible to realize the goal of the operation, but the particular operation *itself* cannot realize it. Consider the example of the memory allocation problem in a multi-process environment (which is often cited as an illustration of the use of the resume response). Suppose that the allocate function is invoked when all memory has already been allocated. The allocate function will try to allocate memory from the memory pool directly, which will not be possible, and so an exception occurrence results. This exception could be handled by making other process release some memory. After that, the handler could issue a resume response, and the allocate function could resume execution by allocating the desired memory from the pool. In this case, the goal of the allocate function is the *direct* allocation of the requested memory from the pool. An exception is raised only when this is not possible. When the memory is finally allocated after the resume response, the allocate function has realized a secondary goal, namely the allocation of the requested memory after the deallocation of memory by other processes. If the allocate function were to be rewritten in such a way that it would itself request other processes to deallocate some memory in the case of insufficient memory in the pool, then this allocate function would have a different goal, namely the allocation of the request memory at the time when enough memory is available.

Christian's definition could be changed by taking account of external exceptions and leaving out the term 'normal execution'. This would lead to the following definition: an *exception occurrence* is an invocation of an

operation in its exceptional domain, where the exceptional domain contains all states for which the operation cannot reach its goal. In this thesis, however, the definition of an exception occurrence as a computational state will be used, which has been defined earlier in this section.

All attempts to define an exception simply as an error, event, condition or response are bound to be unsatisfactory because of the many different uses of exceptions. Errors, events and responses, for example, cannot be raised and do not have a condition. Therefore, we will not explicitly define what an exception is, but rather give its characteristics and, in the next section, explain its use.

An **exception** is associated with a class of exception occurrences [Knudsen, 1987].

The **exception condition** describes the common aspect of the exception occurrences associated with the exception.

Exception occurrences and exceptions are either internal or external. The definitions are as follows.

An **internal exception occurrence** is an exception occurrence of which the computational state is completely determined by the internal state of a sequential process.

An **external exception occurrence** is an exception occurrence of which the computational state is determined by the internal state of a sequential process and the external states of one or more other processes.

An **internal** or **external exception** is an exception which is associated with a class of internal or external exception occurrences respectively.

External exception occurrences can appear to be internal. This is due to the fact that controlling systems must sample the state of the controlled system. The sampled values of the sensors constitute a part of the internal state of the controlling system. An exception occurrence caused by the (sampled) value of the state of a sensor may therefore seem to be an internal exception occurrence. In reality, however, the sampled values are used under the assumption that they are equivalent to the corresponding actual values of the physical sensors. So, the exception occurrences are, in fact, determined by the actual values of the sensors and are therefore external.

In other cases, the inputs supplied to a process will actually cause internal exception occurrences. This is the case, if the inputs need only be received once and can consequently be used throughout the process, eventually resulting in the production of outputs. If such inputs contain precondition errors, they will lead to internal exception occurrences in the process.

In the literature, the terms *asynchronous* and *synchronous* exceptions are sometimes used instead of external and internal exceptions respectively.

4.1.3 Signaling, handling, declaring and raising exceptions

When an exception occurrence is detected in an operation, the corresponding exception or exception condition should be **signaled**. By signaling the exception, the specific information about the exception occurrence is lost. How the signaling of an exception is actually implemented depends on the programming language used. When an exception is signaled, it must be **handled**. The handling of exceptions refers to the way a program recovers from a situation in which some invoked operations cannot realize their goal, to a situation where all invoked operations can, in principle, realize their goal. There are several exception handling mechanisms in programming languages. In the simplest programming languages, signaling an exception is done by returning an error code. This is described in Section 4.3.1. Some programming languages offer a more sophisticated mechanism for exception handling.

In some languages, exceptions can be declared in an **exception declaration** using the 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. Other language constructs offered by these languages to support the exception handling mechanism are constructs for the raising of exceptions, which are comparable with predefined **raise** procedures, and the declaration of **exception handlers**. An exception is signaled by **raising** the exception by means of the raise procedure. Raising an exception will result in the activation of an exception handler which was bound to the exception. The functionality of the exception handling mechanism of these languages will be elaborated further in Section 4.4.

In Smalltalk-80, instances of class Signal are used to denote exceptions. The raising of such a signal in the case of an exception occurrence leads to the

creation of exception objects, which are instances of the class Exception. The functionality of the exception handling mechanism in Smalltalk-80 will be elaborated further in Section 4.5.

The detection of an exception occurrence should lead to the signaling of the corresponding exception. This can be done by raising the exception or, for example, by returning an error code. On the other hand, exceptions should only be raised in the case of an exception occurrence. In fact, we regard the raising of exceptions in other cases as bad programming practice.

When the invocation of a program unit results in an exception occurrence which is either handled in a handler bound to the unit, or propagated to the unit's invoker, the unit is said to **terminate with an exception**. Such a termination of the unit can also be referred to as an **exceptional termination**. This definition assumes that exception handling follows the termination model or, in the case of the resumption model (see Section 4.4.4), that it does not use the resume response. The other possibility is normal termination of the unit, by executing the statement just before the block's end identifier (or other symbol or identifier which closes the block), or by executing a return statement.

4.1.4 The relationship between exceptions and errors

Exceptions are used in programs in order to facilitate the handling of precondition errors in a structured way, and in order to facilitate the creation of robust programs in the presence of precondition errors. Exceptions and exception occurrences are only defined in relation to the execution of programs. Errors, on the other hand, are not restricted to any particular kind of system.

Exceptions can, to some extent, also be used for the handling of errors in the design of a controlling or controlled system; but only if the incorrect design leads to precondition errors or to the violation of invariants (which can be detected by means of assertions).

4.1.5 The relationship between exception occurrences and errors

An exception occurrence will ultimately always be caused by an error. This error can either be an internal or external state error, or a precondition error.

If a system is correct and (recursively) all its components are correct, then all exception occurrences will be the result of precondition errors. If the system and its components are also operated in their standard domain, so that no precondition errors can occur, then the goals of all components can be realized and so no exception occurrences will take place. The relation between exception occurrences and errors will be elaborated further below.

If the weakest precondition of a system is not satisfied, then it will be impossible to realize the system's goal and so an exception occurrence will result. Therefore, precondition errors which violate the weakest precondition of a system always lead to exception occurrences. Precondition errors which do not violate the *weakest* precondition of a system may lead to exception occurrences.
Exception occurrences need not always be caused by precondition errors. They can also be caused by incorrect designs that lead to internal state errors.

Internal state errors can lead to exception occurrences, depending on the inputs to the system. As explained in Section 3.1.3, internal state errors need not always lead to failure of a system. Certain internal state errors will only lead to failure in the presence of certain inputs to the system. If the inputs are such that failure will result, it will obviously also be impossible for some operation to realize its goal, and so there will also be an exception occurrence. If the inputs are such that failure does not result and the goal of the system can be realized, then the internal state error will not lead to an exception occurrence.

External state errors can lead to exception occurrences. However, this is not always the case, because the external state errors of a component can be corrected by the encompassing system, for example by using redundancy.

Exception occurrences can coincide with internal or external state errors, but cannot cause them.

4.2 Basic requirements for a mechanism for the handling of internal exceptions

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 would be to stop execution of the program and enter a debugger. In real-time systems, this response would be unacceptable. The requirements that should be met by a general-purpose mechanism for the handling of internal exceptions are given below. These requirements should also be met by exception handling mechanisms used in languages for the control of industrial systems, together with additional requirements concerning constraint violations. Constraint violations will be treated separately in Chapter 5.

1. The mechanism should facilitate the creation of robust programs.

2. All exception occurrences which are detected should allow for damage confinement and error recovery code to be executed.

3. The mechanism should be easy to use and understand. It should consist of a small number of orthogonal elements. This means that the elements can be used independently of other programming elements and do not overlap. This is a criterion which is often used in the design and analysis of programming languages.

4. There should be a clear separation of the code for normal program operation and the code for exception handling.

5. 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.

6. If an exception occurrence results in the termination of a call chain of several levels, each level should be allowed to fulfil its own finalization obligations.

A consequence of the second requirement is that the often encountered way of error handling which simply results in an error message and a user process being killed is unacceptable.
In addition to requirement number 3, it should be noted that is not necessary to strive for absolute orthogonality.

Requirement number 6 is a result of the use of different levels of abstraction which makes complex programs manageable. These different levels of abstraction should be used both for the normal operation of a program and for the exception handling operation.

Finalization obligations are the actions that have to be taken in order to bring a component in which an exception has occurred to a safe and consistent state after the exception occurrence, so that the component's invariants that have been invalidated due to the exception occurrence are restored.

Consider the fork-lift truck example from Section 2.3. This example is slightly modified so that CtrlTruck directly controls the fork-lift, see Figure 4.2.1. Here there are three levels of abstraction. The first level is the global process cycle represented by the body. The second level consists of the taking of the stack with the necessary synchronization with the stack supplying process. The actual control of the actuators and detectors of the

```
CtrlTruck >> body
    "Finalization obligations:
    Return the fork-lift truck to the predefined reset position"
    self receiveStackFromTraverse.
    self transportStackToFurnace.
    self giveStackToFurnace.
    self goBackToTraverse

CtrlTruck >> receiveStackFromTraverse
    "Finalization obligations:
    Correct the synchronization with the CtrlTraverse processor"
    self sendTo: 'sync-truckResetAtTraverse'.
    self receiveFrom: 'sync-traverseAtFork'.
    self forkUp.
    self sendTo: 'sync-forkIsUp'

CtrlTruck >> forkUp
    "Finalization obligations:
    Stop the motor of the fork lifter"
    self putOn: 'o-forkLifter-up'.
    self putOn: 'o-forkLifter-power'.
    self
        receive: true
        from: 'i-forkLifter-up'
        within: 6 seconds
        ifTimedOut: ["raise exception"].
    self putOff: 'o-forkLifter-power'
```

Figure 4.2.1 *Three levels of abstraction with different finalization obligations*

fork-lift takes place at the third level.
Suppose that the end detector is not activated in time while moving the fork up, possibly because of a defect in the detector. If the error cannot be corrected, it is not possible to continue the process cycle and all three levels should be terminated. At the third level the exception handling operation would consist of stopping the fork-lift motor. At the second level the synchronization with the CtrlTraverse processor would need to be corrected. This is necessary to prevent the CtrlTraverse processor from waiting for a signal that the fork is up, which signal will no longer arrive. Finally, in the body, the fork-lift truck could be returned to a predefined reset position. Each level executes only the finalization operations necessary due to its own premature termination.

4.3 Traditional ways of exception handling

4.3.1 Using returned values as exception codes

The simplest way of exception handling is by passing special codes through arguments or returned values of procedures or methods. This is the way internal exceptions are handled in the Unix operating system written in the C programming language. It is also the method that must be used in Sequential Function Charts [IEC 848, 1988] when goto-like jumps are to be avoided. Although the advantage of this method is that it is very simple to understand and requires no special support, the drawbacks to this way of exception handling are so severe that this method is unsuitable for use in industrial control systems. In this section, it will be shown that this method cannot satisfy the requirements defined in the previous section.
Using returned values as exception codes means that the results of all operations must be checked for the occurrence of exceptions. This has several severe consequences:

1. The creation of robust programs becomes problematic. The amount of code needed just to check error return codes becomes enormous. A single test can easily be forgotten. This will not hinder normal program operation and may thus remain unnoticed for a long time. Only when an error occurs, will the failure to test for the error occurrence cause the program to continue in an erroneous state with possibly disastrous consequences. Also, it can be very difficult to detect the cause of such errors.

2. Many system operations do not return any error values. Therefore, no error recovery code can be executed for these errors. Consider, for example, arithmetic operations such as division. A simple division could result in a division by zero error. It is unacceptable that such an error simply prints an error message on the terminal and stops the system, because this makes it impossible for the user program to do any damage confinement and error recovery.
   Error information could be supplied in global variables in an environment with a single user process. In a multi-process environment, this will lead to unpredictable and possibly erroneous program operation. In such a case, there should be a 'global' error variable for every process. The problem which still remains is that sequential errors overwrite the global error variable: if an error is undetected due to omission to test the global error variable, another error may result, which overwrites the original error information. More important even than this is the fact that such schemes are highly impractical, and that it is undesirable to use a second mechanism for the handling of errors that cannot be handled by returning error codes.

3. The exception testing code will be intermingled with the code for normal program operation, leading to programs that are hard to read.

Figure 4.3.1 gives an example of this way of exception handling. The send and receive operations are not checked for error return codes. If errors can occur during execution of these statements, then they should also be checked for error return codes.

4.3.2 Other mechanisms

There are several other traditional mechanisms for the handling of exceptions. The reader is referred to [Goodenough, 1975], which contains an extensive treatment of these mechanisms. They will not be dealt with further here. The VAXELN system [Digital, 1986] has some similarities with the exception handling mechanisms treated in Section 4.4. It also has special exception handling capabilities for multi-process environments. Its treatment will therefore be deferred until Section 5.5.2.

```
CtrlTruck >> body
    (self receiveStackFromTraverse) == #ok ifFalse: [↑ #error].
    (self transportStackToFurnace) == #ok ifFalse: [↑ #error].
    (self giveStackToFurnace) == #ok ifFalse: [↑ #error].
    (self goBackToTraverse) == #ok ifFalse: [↑ #error].
    ↑ #ok

CtrlTruck >> receiveStackFromTraverse
    self sendTo: 'sync-truckResetAtTraverse'.
    self receiveFrom: 'sync-traverseAtFork'.
    (self forkUp) == #ok ifFalse: [↑ #error].
    self sendTo: 'sync-forkIsUp'.
    ↑ #ok

CtrlTruck >> forkUp
    self putOn: 'o-forkLifter-up'.
    self putOn: 'o-forkLifter-power'.
    self
        receive: true
        from: 'i-forkLifter-up'
        within: 6 seconds
        ifTimedOut:
            [self putOff: 'o-forkLifter-power'.
            ↑ #error].
    self putOff: 'o-forkLifter-power'.
    ↑ #ok
```

Figure 4.3.1 *Exception handling by returning error codes.*

4.4 Advanced exception handling mechanisms

Exceptions have been defined in Section 4.1. In order to support the handling of exceptions, some programming languages have implemented advanced exception handling mechanisms. This section discusses these mechanisms, using the mechanisms of two imperative programming languages as an example. The advanced exception handling mechanisms of 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 ModPas [Bron and Dijkstra, 1987a] or Ada programming language [Ichbiah, 1983]. Modular Pascal, or ModPas, is a Pascal based language which includes modules. It was one of the first languages to include an advanced exception handling mechanism. The exception

handling mechanisms of Modular Pascal and Ada are based on [Bron and Fokkinga, 1976].

4.4.1 Exceptions

Exceptions can be declared in exception declarations. This is done as follows in ModPas and Ada respectively:

```
EXCEPTION xFatalError;                {ModPas exception declaration}

ILLEGALPARAMETER : exception;         -- Ada exception declaration
```

Exceptions are signaled by raising them in a raise statement. In ModPas, the raising of an exception is syntactically equivalent to the invocation of a parameterless procedure. In Ada, this is done by means of the predefined raise operation.

```
xFatalError;                  {raising a ModPas exception}

raise ILLEGALPARAMETER;       -- raising an Ada exception
```

4.4.2 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. 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 (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 ModPas, procedure and function bodies may be prefixed by handlers, the form of which is: BUT FOR <exception identifier>: <handler body> DO. In Ada, handlers can be attached to a block, a body of a subprogram, a package or a task. They are specified by means of the reserved word **exception** followed by the declaration of one or

more handlers at the end of a unit. The form of a handler is: **when** <exception identifier> => <handler body>.

```
BUT FOR xFatalError: write(’’ fatal error ! ’’) DO
BEGIN
    calculate;
    {rest of ModPas program}
END;

begin
    CALCULATE;
    -- rest of Ada program
exception
    when ILLEGALPARAMETER => PUT(’’ illegal parameter ! ’’);
end;
```

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 the previous example, the ModPas body has a handler for the xFatalError exception and the Ada block has a handler for the ILLEGALPARAMETER 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 kind of handler will be referred to as an *any handler* or *others handler*. This is necessary for the handling of 'out of scope exceptions' and to allow for finalization obligations. 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 ModPas, an any handler is specified by means of specifying xAny as the exception of a handler and in Ada others is used for this purpose:

```
PROCEDURE readInputsFromFile;
BUT FOR xAny: {close file}; xReraise DO
                                {xReraise is explained in Section 4.4.5}
BEGIN
    {open file};
    {process file}
END

begin
    -- Ada program
exception
    when others => PUT(’’ error ! ’’); raise;
                                -- raise is explained in Section 4.4.5
end;
```

4.4.3 The handling of exceptions

In order to understand exception handling mechanisms, the concepts of *invoker*, *invocation* and *activation point* should first be understood.

A subprogram is defined only once in a program, but it can be called at many different points. Each call of the subprogram results in a different subprogram invocation. The **invocation** of a subprogram is the activation of the subprogram's body in the environment of the subprogram's textual definition. The **activation point** of a subprogram invocation is the point in a unit from where the subprogram is called or invoked.

The **invoker** of a subprogram's invocation, SI, is the invocation of the smallest enclosing unit of SI's activation point.
In contrast to subprograms, the other units (namely blocks, modules and packages) cannot be called from different points in the program. They are defined and activated at one and the same place. Therefore, the **invoker** of a block's, module's or package's invocation is simply the invocation of its smallest enclosing syntactical unit.
We will often simply refer to the *invoker of a unit* instead of the more precise terminology *invoker of a unit's invocation*.

When an exception is raised explicitly, the exception will be handled in the invocation of the smallest enclosing unit of the exception raising statement.

An exception which has been raised is handled as follows in a unit's invocation, UI.
If the unit has a handler for the exception, control will pass to the handler. If the unit has no handlers for the exception, the exception will be propagated to UI's invoker, which means that the exception will be handled in UI's invoker.

Note that the definition of an invoker as the invocation of a smallest enclosing unit is, strictly speaking, not precise enough. Consider the following example:

```
for I in 1 .. 10 loop
    begin
        PROCESSPRODUCT(I);
    exception
        when FAIL => PUT(''Processing failed for product number ''); PUT(I);
    end;
end loop;
```

The procedure PROCESSPRODUCT is called for 10 products. If the exception FAIL is signaled during the processing of one of these products, an error message including the product number is printed by the exception handler. When the loop is executed, each value of I will result in a different invocation of PROCESSPRODUCT. There is, however, only one activation point: PROCESSPRODUCT(I). The smallest enclosing block of the activation point is the **begin end** block. Suppose the FAIL exception is signaled when the fifth product is processed. Then the invoker of the invocation of PROCESSPRODUCT(5) would be the invocation of the **begin end** block. But there have already been five subsequent invocations of the block. In this case, clearly the last invocation is meant, since the invocations of the block for the products 1 to 4 have already terminated. In the case of recursion, however, many nested invocations of a recursive procedure may exist. The invoker of subprogram's invocation, SI, is then obviously meant to be the invocation of the smallest enclosing unit of SI's activation point, from which SI was actually invoked.

4.4.4 The termination and resumption model

When an exception in a unit is propagated to the unit's invoker, the question arises of what happens with the unit's invocation. There are two possibilities. The simplest possibility is to terminate the invocation of the unit. This strategy is adopted in, for example, ModPas, Ada and Clu [Liskov and Snyder, 1979]. It is known as the termination model. The other possibility is to keep the invocation intact, so that, after the exception has been handled, the execution of the program can be resumed at the point where the exception was raised. This strategy is adopted in, for example, Smalltalk-80 and VAXELN. It is known as the resumption model. Conceptually, and from the viewpoint of the implementation, the termination model is much simpler than the resumption model. Also, in most cases, it is preferable in order to avoid goto-like programming, which is allowed in the resumption model. The disadvantages of the resumption model will be further discussed in Sections 4.6.3 and 4.6.4. A more

extensive evaluation of the resumption model versus the termination model can be found in [Liskov and Snyder, 1979].
Because of the simplicity and other advantages of the termination model, the explanation of the exception handling mechanism will assume the termination model unless explicitly stated otherwise.

4.4.5 Handler responses

When an exception is signaled in a unit for which the unit has a handler, control will pass to the handler. There are five principally different ways in which a handler can end. They each have 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. The functionality is similar to the goto statement. Since goto programming is generally acknowledged to be bad programming practice, the transfer response will be neglected as undesirable.
The functionality of all five responses will now be explained. Actual implementations can be more complex, but will exhibit the same functionality.

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 can only be used in the resumption model. It 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 signal 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 were propagated directly from the unit.

ModPas and Ada only support the return and propagate response. The propagate response is invoked in ModPas by raising the predefined exception *xReraise*. In Ada, this is done by calling the predefined *raise* operation with no arguments.

In theory, the responses treated above could be defined differently. If the handler of an exception is bound to a different unit than the unit in which the exception was raised, then the retry response, for instance, could cause the unit in which the exception was raised to be reinitialized and restarted, instead of the unit associated with the handler. The other responses likewise have different versions. All theoretically possible different versions of the above-mentioned handler responses are treated in [Feder, 1990]. Feder, however, does not indicate the necessity or value of the definitions other than the generally used definitions mentioned above.

4.4.6 The functionality of exception handlers in control systems

Exception handlers are bound to program units such as blocks. An exception handler is activated when the unit to which it is bound terminates with an exception. This premature termination of the unit can cause invariants to be invalidated. Some examples of the violation of invariants will be given in Chapter 5 in Figures 5.3.3a-c.

An important aim of the exception handler is to restore invariants. The actions that are necessary in order to restore the invariants can also be referred to as *finalization obligations*. In some languages, special constructs are available for dealing with finalization obligations, such as unwind blocks in Smalltalk-80. Such constructs will not be treated, because their functionality can be closely approached with exception handlers. Some invariants cannot be restored by the exception handler alone, for example when the exception causes the synchronization between processes to be incorrect. In this case, the exception handler should signal the other processes, so that the processes can be resynchronized and the invariant will be restored.

After restoring the invariants, the exception handler should try to realize the (secondary) goal of the program unit to which it is bound. If this goal can be realized, the exception handler will terminate with the return response. The exception handler may also try to create a situation where the unit's goal may be achieved by a renewed invocation of the unit; in this case the handler will terminate with the retry response. If neither of these responses are possible, the exception handler will terminate with an exception, under the assumption that exception occurrences are always signaled by means of raising exceptions, and not by returning error codes. This can be done by either propagating the handled exception or by raising a new exception. The resume response is not considered here, because it should not be used (see Sections 4.6.3, 4.6.4, and 6.9).

4.5 The exception handling mechanism in Smalltalk-80

The exception handling mechanism used in Smalltalk-80 is somewhat different from the mechanisms used in most imperative languages. This is partly due to the fact that Smalltalk is a truly object-oriented language, and also to the fact that the exception handling mechanism is not part of the Smalltalk language definition but is rather an addition to the language,

mainly by the addition of the classes Signal and Exception. The mechanism is more powerful and more complex than the mechanisms treated so far.

4.5.1 Exceptions and signals

In many languages, exceptions are defined and at the same time statically bound to an identifier in an exception declaration. In Smalltalk exceptions are defined when a signal is created. Signals are instances of the class Signal. Signals are usually created at the initialization of the class in which the signal is created. The signals created are made available to other objects by means of messages that can be sent to the class in which the signal is defined. For example:

```
Object errorSignal
ArithmeticValue divisionByZeroSignal
```

When a signal object is created, a new exception is defined which is denoted by the signal. The exception can be raised by sending a raise message to the corresponding signal. This will result in the creation of an instance of the class Exception. Such an exception object, however, does not denote an exception, since exceptions are denoted by signals. This fact can lead to confusion because raising an exception in Smalltalk is done by sending a raise message to a signal. In this thesis, the terminology of *raising an exception* and *raising a signal* will be considered equivalent for Smalltalk systems. The most important methods for raising a signal are raise and raiseErrorString:.

```
Object errorSignal raise
Object errorSignal raiseErrorString: 'an error has occurred'
```

Exception objects are used as arguments of exception handlers. They can be used to retrieve the signal that was raised to create the exception object. They are also used in the exception handler to retrieve any arguments supplied by the raiser of the signal and to control the handler response.

The hierarchy of signals

Smalltalk signals are hierarchical. A new signal is created with another, already existing, signal as its parent. The signal is said to be a child of its parent. This hierarchy is used to catch related exceptions without the need to

bind each exception explicitly to a handler. A handler which is bound to a certain signal will catch all exceptions represented by the signal's children. If a handler is bound to ArithmeticValue errorSignal for instance, it will catch, among others, the exceptions represented by the signals ArithmeticValue overflowSignal, ArithmeticValue underflowSignal and ArithmeticValue divisionByZeroSignal, since they are all (indirectly) children of ArithmeticValue errorSignal.
A small part of the hierarchy of signals is given below.

```
Signal genericSignal
    Object informationSignal
        Object userInterruptSignal
    Object errorSignal
        ArithmeticValue errorSignal
            ArithmeticValue rangeErrorSignal
                ArithmeticValue overflowSignal
                ArithmeticValue underflowSignal
            ArithmeticValue domainErrorSignal
                ArithmeticValue divisionByZeroSignal
```

This hierarchy is also used to specify 'any handlers'. All exceptions representing error conditions are supposed to be created with Object errorSignal as their parent. Therefore, a handler bound to Object errorSignal should catch all such exceptions. Object errorSignal has Signal genericSignal as its parent, which is the ancestor of all signals. Unfortunately this signal may not be bound to handlers. The Smalltalk user guide states that this signal is considered an abstract entity and should not be used to catch exceptions directly. Therefore, specifying Object errorSignal to create an any handler is only foolproof if all users conform to the convention of creating all new signals with Object errorSignal as their parent. This implementation is not as robust as the implementation of any handlers in ModPas and Ada.

4.5.2 Exception handlers

An exception handler in Smalltalk is a block. A block in Smalltalk is quite different from blocks in imperative languages. It has a greater resemblance to a subprogram. A block consists of a sequence of statements, enclosed in [ ] brackets, which are executed or invoked when the message value is sent to the block. Blocks can be conveniently used as arguments of methods. When, during execution or invocation of the method, the message value is sent to a method's argument which points to a block, the block is executed or invoked in the environment of the block's definition.

In contrast to imperative languages, where handlers are statically (i.e. at compile-time) bound to units and to exceptions, exception handlers in Smalltalk are dynamically (i.e. at run-time) bound to blocks and exceptions. The binding of an exception handler to a block and an exception is done by means of the handle:do: method which can be sent to a signal. The handle:do: method is invoked by evaluation of an expression such as:

```
signal handle: handlerBlock do: doBlock
```

This produces a binding of the handlerBlock to the doBlock and to the exception denoted by signal. The only purpose of the handle:do: method is to produce handler bindings. It is also the only way of binding a handler. Handlers cannot be bound to methods.
Invocation of the handle:do: method first results in the production of the bindings between the handlerBlock and the doBlock. Immediately after that, the doBlock is invoked by sending it the value message.

4.5.3 The handling of exceptions

When an exception is raised, the exception will be handled in the invocation of the smallest enclosing unit of the exception raising statement. A unit in Smalltalk is defined to be either a block or a method.
An exception is handled as follows in a unit's invocation, UI. If the unit has a handler for the exception, control will pass to the handler. If the unit has no handlers for the exception, the exception will be propagated to UI's invoker, which means that the exception will be handled in UI's invoker. Because methods cannot have handlers, the handling of an exception in the invocation of a method, MI, always implies the propagation of the exception to MI's invoker.

The invoker of an invoked doBlock or handlerBlock which are bound by a handle:do: method is the invocation of the smallest enclosing unit of the handle:do: message expression. In all other cases, the invoker of a unit's invocation, UI, is the invocation of the smallest enclosing unit of UI's activation point. The activation point of a method's invocation is the expression which invoked the method. The activation point of a block's invocation is the expression which invokes the block. So this is an expression where value (or value:, or any other of the value messages) is sent to the block.

4.5.4 Handler responses

All the handler responses mentioned in Section 4.4.5 are implemented in Smalltalk. The return, propagate, retry and resume responses are implemented by the messages return, reject, restart and proceed respectively. In order to effect these responses, the messages can be sent to the exception object which acts as the argument of an exception handler.

The resume response permits control to return from a handler to the statement following the statement where the exception was originally raised. Therefore, it is not possible to terminate an invocation when an exception is propagated. The invocation which handles the exception, and all other invocations that have issued a propagate response, are terminated only at the point that a return or retry response is issued or another exception is raised. The problems associated with the resume response in Smalltalk and in other languages will be discussed in Section 4.6.3.

The example of exception handling shown in Figure 4.5.1 is taken from the example presented in Section 2.3.3. It is rewritten to include exception handling.
ErrorSignal is another way of representing Object errorSignal. It is a global variable which is known throughout the system. When ErrorSignal is raised from within the time-out block, the exception will be caught by the handler. The run-time system will set the argument exc to the exception object. The error string with which the signal was raised is retrieved from the exception object by sending it the message errorString. The handler finishes with exc reject, which is the propagate response. Consequently the invoker of the forkUp invocation will handle the exception. In Figure 4.5.2 the method forkUp is rewritten for the retry response.

```
SlaveForkLifter >> forkUp
    ErrorSignal
SlaveForkLifter >> forkUp
    handle:
    ErrorSignal
        handle:
            [:exc |
            self putOff: 'o-forkLifter-power'.
            [:exc |
            self sendErrorMessageToOperator: exc errorString.
            self putOff: 'o-forkLifter-power'.
            self sendRestartMessageToOperator: 'fork lifter up'.
            self sendErrorMessageToOperator: exc errorString.
            self receiveRestartResponseFromOperator.
            exc reject].
            exc restart]
        do:
        do:
            [self putOn: 'o-forkLifter-up'.
            [self putOn: 'o-forkLifter-up'.
            self putOn: 'o-forkLifter-power'.
            self putOn: 'o-forkLifter-power'.
            self
            self
                receive: true
                receive: true
                from: 'i-forkLifter-isUp'
                from: 'i-forkLifter-isUp'
                within: 6 seconds
                within: 6 seconds
                ifTimedOut:
                ifTimedOut:
                    [ErrorSignal raiseErrorString: 'time-out fork lifter going up'].
                    ErrorSignal raiseErrorString: 'time-out fork lifter going up'].
            self putOff: 'o-forkLifter-power'.]
            self putOff: 'o-forkLifter-power']
```

Figure 4.5.1 *Exception handling with the propagate response.*
Figure 4.5.2 *Exception handling with the retry response*

4.6 Evaluation

4.6.1 A general evaluation of the advanced exception handling mechanisms

The most important aspect of the use of the exception handling mechanism as treated in the previous sections is that it facilitates the creation of robust programs. The advantages of the mechanism are as follows:

- The automatic propagation of unhandled exceptions prevents untreated errors from causing system crashes.
- Exceptions make it possible to fulfil finalization obligations at each level of abstraction. This is done by catching exceptions if necessary and propagating the caught exception after execution of the finalization obligations by the handler.
- The exception handling mechanisms are based on only a small number of primitives which allow all exceptions, both those detected by the user and by the system support, to be handled in a uniform way. A requirement for this is that the system support declares all exceptions

that it can raise. Also the any handler can be used to catch exceptions raised by the system.

- The use of exceptions enables the separation of concerns in the development of programs. The code for the normal processing of the program can, to a great degree, be developed separately from the code for the handling of exceptions. This leads to programs which are easier to develop, read and maintain.

A disadvantage of the exception handling mechanism is its greater complexity than the traditional ways of exception handling. The mechanism changes the flow of control of a program in an entirely different way than is done by the traditional programming methods.

Not all languages employ all five mentioned handler responses. The propagate response, however, is fundamental to the exception handling mechanism and is therefore available in all programming languages with advanced exception handling mechanisms. The return response is available in most languages. Raising an exception from within a handler should also be possible, since precondition errors can occur anywhere, including handlers themselves.

The retry response is not absolutely necessary, since its functionality can be approached using the return response together with programming constructs for repetition. This, however, leads to inelegant code which is less clear. The elegance of exception handling using the retry response will be shown in Section 7.3.

4.6.2 The return response as an inadequate default response

The return response is usually the default response of an exception handler. This is an inadequate choice because the accidental omission of a handler response can have disastrous consequences.

Exception handlers facilitate the creation of robust programs. Programming errors in complex systems cannot be completely avoided, however. An error which is easily made, and which cannot be detected by a compiler in the case that default responses are allowed, is the accidental omission of a response such as a propagate response. If such a propagate response is accidentally omitted, the program is allowed to continue in an incorrect state

which can lead to program crashes and, in the case of control systems, to catastrophic reactions of the controlled system.
If, on the other hand, the propagate response is chosen as the default response, then accidental omission of a response will automatically lead to the propagation of the exception to higher levels. In this way, all finalization obligations of higher levels will be fulfilled and the system, or a part of it, will terminate in a controlled way. Clearly it is much more likely that a response from a handler will be forgotten than that an incorrect response will be used.

It would also be possible not to allow a default response and to regard the omission of a response as an error, leading to a run-time exception. This exception would then discard the original exception, so the propagate response is preferred as a default response.

4.6.3 The resume response as an inadequate response in a sequential process

The resume response can be convenient in a small number of cases where the program can be continued right after the error, after correction of the error in an exception handler. One of these cases is the resumption from a debugger. This could, however, also be implemented without using the resume response, and in itself it does not justify the resume response. There are, in fact, severe drawbacks associated with the resume response. Consider the following part of a Pascal program:

```
i := 5;
b := a[i];
```

The value of `b` would be expected to be the value of the fifth element of the array `a`. If, however, array `a` were to have only 3 elements, then an exception would occur. The value of `i` could be set to 1 in an exception handler, which could be bound to the subprogram shown, whereafter a resume response could be given. As a result `b` would be assigned the first value of `a`.

This kind of behaviour makes programs hard to understand, since the result of an operation can be determined by an exception handler which is not bound to the operation itself but to the invoker of the operation, or to any other invoker in the call chain. This leads to similar problems as the use of the goto statement. It is also inconsistent with the use of procedure or

method abstractions. These abstractions hide the implementation aspects of lower layers in a system from the higher layers. When an exception is propagated to a higher level, the resume response enables correction at the higher level and consequent resumption at the lower level. Thus, in order to understand and develop programs with these kind of characteristics, it is necessary to study the implementation aspects of both the higher layers and the lower layers at the same time. And this is the very thing that the use of procedure and method abstractions aims to avoid.

Therefore, it is concluded that it is better not to use the resume response at all. In systems with parallel processes, there are additional problems with the resume response. These problems are treated in the next section and in Section 6.9.

4.6.4 Conflicts between the resume response and critical regions

A problem associated with the resume response is its use in a parallel system that uses critical regions to prevent simultaneous access to shared resources. A simple example of this is the use of a semaphore for mutual exclusion on which a wait or p-operation [Dijkstra, 1965] is performed when entering the critical region. When the region is left, the semaphore is signaled by executing a v-operation on it. If an error occurs in a critical region, an exception will be raised. If the exception cannot be handled within the critical region, it will be propagated to the invoker. Since propagating the exception means that the critical region is left, the semaphore should now normally be signaled in order not to keep the critical region locked and inaccessible to others.

If the resumption model is used, however, and an exception causes the critical region to be left and the semaphore to be signaled, then the exception can be caught by a handler in which a resume response is issued. This will cause continuation of the process within the critical region without, however, first executing a wait operation on the semaphore. In this way an inconsistent state is created such that two processes will always be allowed to enter the critical region instead of one. Therefore, the existence of the resume response prevents the semaphore from being signaled when the critical region is left by means of exception propagation. It may only be signaled when resumption in the critical region is no longer possible. Therefore, the critical region will remain locked until that time. This can be intolerable. It can even lead to deadlock if an exception handler, handling

the exception from the critical region, executes an operation which needs access to the critical region.

Chapter 5
The handling of constraint violations

Most exception handling mechanisms have been developed for the handling of internal exceptions in stand-alone sequential processes and have no special provisions for dealing with external exceptions in control systems. These mechanisms are meant to be used for exception occurrences in operations that are determined by the internal state of the operation itself. Exception occurrences are detected by having each operation explicitly test its own internal state.

In this chapter the concepts 'constraint' and 'constraint violation' will be defined. These concepts are essential in order to determine the requirements of a mechanism for the handling of external exceptions in control systems. It will be shown that a mechanism is required for the specification of constraints and the detection and handling of constraint violations. The known mechanisms for the handling of constraint violations are treated and shown to be inadequate. Finally, a new mechanism is presented. Although the material presented in this chapter is developed in the context of control systems, it is also of general relevance in multi-process environments.

5.1 Definition of terms

5.1.1 Constraints, constraint functions and constraint violations

The **constraint** of an operation is that part of its precondition which refers exclusively to the state of the environment of the process executing the operation and which is invariant over the operation: it has to be valid throughout the execution of the operation.

A constraint can be *compound*, in which case it consists of (sub-)constraints. A compound constraint is met if and only if all of its (sub-)constraints are met. In most situations we will not explicitly distinguish compound and sub-constraints, but simply use the term constraint. This makes it possible to

refer to the constraints (plural) of an operation, which is meant to include all of the sub-constraints of the operation's compound constraint.

Consider, for example, an operation that controls a cylinder which pushes a product upwards. The goal of the operation could be to make the cylinder extend to the upward position under the condition that the emergency button is not pressed. The emergency button which has to stay inactive would be a constraint of the operation. Another constraint could be that there must be adequate air under pressure available for the cylinder. Other examples of constraints are a temperature which has to stay within a certain range during the execution of an operation, or the external state of a process with which the operation wants to interact. To allow the interaction to take place, the operation's constraint will require that the interacting process is in such a state that it will eventually perform the desired interaction with the (process executing the) operation.

A constraint of an operation can be expressed by a boolean **constraint function** which is defined only during the execution of the operation. The constraint function returns true when the constraint is valid and false when the constraint is not valid.

A **constraint violation** is that part of the state of the environment of a process executing an operation which does not satisfy the operation's constraint.

A constraint violation is a precondition error, and it causes an external exception occurrence in the process executing the operation of which the constraint is violated.

The traditional exception handling mechanisms alone are not sufficient for the handling of external exceptions, because these mechanisms were developed for the handling of internal exceptions. Without additions to these mechanisms, the handling of most external exceptions becomes awkward.

The time-out mechanism has been added to virtually every language for the control of industrial systems in order to be able to handle external exceptions that are related to exceeding a time limit. Mechanisms for the handling of external exceptions due to constraint violations, however, are more complex and therefore not so common. This is due to the fact that constraints need to be valid during the complete execution of an operation. Constraint violations can occur at any point during the execution of an

operation, and they are usually determined by the execution of other processes.

5.1.2 The active constraints of a process

Many operation invocations can exist at the same time during the execution of a sequential process. The **active constraints** of a process are the collection of the constraints of all current operation invocations within the process. So, when an operation is invoked in a process, the operation's constraints are added to the active constraints of the process. When the operation is terminated, its constraints are removed from the process's active constraints.

The constraints of an operation in a process will be illustrated using the example of Figure 5.1.1. It is based on the transporter example of Section 2.3. This example is somewhat simplified, so that the fork-lifter is controlled by the CtrlTruck processor instead of the SlaveForkLifter processor.

The constraints of CtrlTruck >> body are that the emergency button is not activated and the operator does not send a stop command.

When two processes are mutually interactive, they both make assumptions about each other's external states. In the method CtrlTruck >> receiveStackFromTraverse, after returning from the statement self receiveFrom: 'sync-traverseAtFork', the CtrlTruck processor assumes that the CtrlTraverse processor has moved the traverse to the position of the fork, that the traverse will stay there, and that the CtrlTraverse processor will be waiting for CtrlTruck to put the fork up. CtrlTraverse expects to be informed of the successful completion of this operation by means of the self sendTo: 'sync-forkIsUp' statement in CtrlTruck. These expectations about the external states of interacting processes are constraints. Therefore, a constraint of CtrlTruck >> receiveStackFromTraverse in Figure 5.1.1 is that the external state of the CtrlTraverse processor corresponds to the state expected by CtrlTruck. The unactivated emergency button and the absence of a stop command from the operator are also constraints of this method.

Finally, the constraints of the method forkUp are that the emergency button is unactivated, that the state of the environment of the fork is such that it is safe for the fork to go up, and that the operating switch, used to switch off the fork lifter temporarily, is on.

```
CtrlTruck >> body
    "Constraints:
        - Emergency button not activated
        - No stop command from operator"
    self receiveStackFromTraverse.
    self transportStackToFurnace.
    self giveStackToFurnace.
    self goBackToTraverse

CtrlTruck >> receiveStackFromTraverse
    "Constraints:
        - Emergency button not activated
        - No stop command from operator
        - The external state of CtrlTraverse corresponds to the state expected
    by CtrlTruck"
    self sendTo: 'sync-truckResetAtTraverse'.
    self receiveFrom: 'sync-traverseAtFork'.
    self forkUp.
    self sendTo: 'sync-forkIsUp'

CtrlTruck >> forkUp
    "Constraints:
        - Emergency button not activated
        - The fork can safely go up without causing damage
        - Operating switch is on"
    self putOn: 'o-forkLifter-up'.
    self putOn: 'o-forkLifter-power'.
    self
        receive: true
        from: 'i-forkLifter-isUp'
        within: 6 seconds
        ifTimedOut: [KillSignal raise].
    self putOff: 'o-forkLifter-power'
```

Figure 5.1.1 *Constraints of some methods.*

The method forkUp is called from the method receiveStackFromTraverse which is in turn called by the body. Therefore, when the method forkUp is active, the other two methods will also be active, assuming that the forkUp method is not called by other methods. So when the forkUp method is invoked, the active constraints of the CtrlTruck process consist of at least the constraints of all three methods. This means that a violation of any of these constraints will result in an exception occurrence in CtrlTruck. If, on the other hand, CtrlTruck is waiting for an interaction to take place in the

receiveStackFromTraverse method, then the active constraints of CtrlTruck are the constraints of the methods receiveStackFromTraverse and body (plus the constraints of the method that invoked body and of the other methods that were already invoked).

5.1.3 Different kinds of invariant

Invariants define relationships within a sequential process or between several processes. We will define two special kinds of invariant: *internal* invariants and *external* invariants. Their definitions will facilitate the discussion of the handling of constraint violations in the following sections of this chapter.

An **internal invariant** is an invariant which is completely determined by the state of a sequential process or operation.

An **external invariant** is an invariant which is determined by the states of two or more processes.

Another important kind of invariant is the *general control invariant*, which essentially determines all constraints in control systems. All control systems are based on this invariant which specifies that the state of the controlling system corresponds to the state of the controlled system and the operator.
The controlling system changes from one state to another and drives the actuators in such a way that the state of the controlled system is made to correspond with its own state. When the controlling system is waiting for a change of state of the sensors, it is actually synchronizing its own state with the state changes of the controlled system. The state of the sensors can be tested from time to time to see whether the control invariant still holds, i.e. to see whether the state of the controlled system is still such as is expected by the controlling system. If this is not the case, the controlling system could request operator help to change the state of the controlled system in such a way that the control invariant holds again; or the controlling system can reset the controlled system to bring it into a defined state again.
A man-machine interface is used to keep the control invariant valid between the operator and the controlling system.

5.2 Constraints

5.2.1 The local specification of the constraints of an operation

The constraints of an operation are specific for the operation itself. Therefore, they should be specified locally, that is independently of the point in the program at which the operation is invoked. This enables parts of a program to be developed at different, largely independent, levels of abstraction. The local specification of the constraints of an operation is thus an important requirement for the creation of modular program units.

Referring to the example which has been presented in Figure 5.1.1, the constraints of the method CtrlTruck >> forkUp are that the emergency button is unactivated, that the state of the environment of the fork is such that it is safe for the fork to go up, and that the operating switch, used to switch off the fork lifter temporarily, is on. The constraint of the method CtrlTruck >> body which specifies that the operator does not send a stop command is not a constraint of forkUp, because forkUp could, for example, also be invoked in manual mode. In manual mode, the operator can choose several low level commands from the MMI (man-machine interface) to be executed, such as the command forkUp or forkDown. In this situation, there is no need for the operator to stop the production process, because control is automatically returned to the operator after execution of the chosen forkUp or forkDown command.

The local specification of constraints can lead to duplication of constraints. Consider, for example, the constraint that the emergency button is unactivated. This constraint is specified in all three methods. If the method forkUp would be a library routine, so that the callers (or senders) of forkUp cannot be determined in advance, then the duplication of the emergency button constraint is indeed necessary.

5.2.2 The specification of constraints common to many operations

Obviously the specification of the constraints of an operation should serve more purposes than just as a comment to the program: it would be useful to monitor the specified constraints during the execution of the operation and to signal constraint violations. The constraints of an operation must be valid throughout the execution of the operation, so that the operation's goal can be achieved. Therefore, they must also be valid during the execution of other

operations invoked by the original operation. So, a mechanism for the monitoring of the constraints of operations must work in such a way that all of the active constraints of the process executing the operations are monitored.

A consequence of such an approach is that unnecessary duplication of equivalent constraints can be avoided. Suppose, for instance, that the method forkUp of Figure 5.1.1 is only called by receiveStackFromTraverse and that the latter method is always called by body, then the constraint specifying that the emergency button is off can be removed from the methods forkUp and receiveStackFromTraverse. In this case, modularity of the methods forkUp and receiveStackFromTraverse is obviously not of primary concern, since it is known that they are only (indirectly) called by the method body. Avoiding the unnecessary duplication of constraints in such a way will lead to simpler and better programs, as long as it is evident that the constraints left out are under all circumstances already monitored when the operation is called.

In more general terms, it can be stated that the constraints which are common to many different operations can, in certain cases, be specified in a single operation. In certain control systems, for instance, all operations executing in a certain process are known to be (indirectly) called from a main program loop defining the process cycle. In such cases, constraints like the emergency button being off, can be specified in the main process cycle, so that they are guaranteed to be monitored for all operations executed in the main process cycle. The constraints need not be duplicated in the called operations.
In such a case, it is a better option to specify the constraints which are common to many operations in a single operation from which the other operations are called, rather than to re-specify the constraint in all called operations.
Note that in most imperative languages, such as Pascal, static scope rules can guarantee that certain operations are only called from certain other operations. In most object-oriented languages which use dynamic binding, such as Smalltalk, it is often necessary to analize the run-time behaviour of the program in order to determine such relationships.

5.3 Constraint violations

5.3.1 A traditional way to detect constraint violations

An important aspect of constraints is that they must be valid throughout the execution of an operation. Constraints may be violated while a process is blocked in an interaction. It is therefore generally insufficient to check the constraints at certain statements of the operation. The state of the environment of a fork-lift, for example, could be checked before the fork goes up, but there is no guarantee that the environment will remain in a safe state while the fork is going up.

Another complicating aspect of constraints is that, during the execution of an operation, many constraints must be checked which are not constraints of the executing operation itself but of other active operations on the call chain.

A possible way of detecting constraint violations is to let each operation explicitly detect all active constraints of the process in which it is executed. Thus, the constraints that an operation would need to detect would be the process's active constraints at the time of the call, plus the operation's own constraints.

In the example of Figure 5.1.1, the method forkUp could be extended to check all the active constraints of the process. In this case, instead of simply waiting for the fork to be up, the operation would have to wait for either the fork to be up or for any of the constraints to be violated. If a constraint is violated, an exception should be raised. This is illustrated in Figure 5.3.1 where every method tries to monitor the process's active constraints. (Error messages and exception handlers have been omitted for the sake of simplicity.) The reader should also note that, for the sake of clarity, not all constraints are specified.

The detection of constraint violations has been realized with a mechanism that enables a process to wait for one out of a set of interactions to occur. The way the program will proceed depends on the interaction that has taken place. In Process Calculus, this functionality is given by the possibility to receive an object from any of a set of ports, for example with the method Bubble >> receive:fromOneOf:do:.
The meaning of the message self receive: objectArray fromOneOf: portArray do: doBlock is that a receive action is specified which tries to receive one of the objects specified in objectArray from one of the ports specified in portArray.

```
CtrlTruck >> receiveStackFromTraverse
    "  ...  "
    self
        receive:
            #(nil nil true)
        fromOneOf:
            #('sync-traverseAtFork' 'mmi-operatorReset' 'i-emergencyStop')
        do:
            [:port :item | (port = 'sync-traverseAtFork') ifFalse: [KillSignal
raise]].
    self forkUp.
    "etc ..."

CtrlTruck >> forkUp
    self putOn: 'o-forkLifter-up'.
    self putOn: 'o-forkLifter-power'.
    self
        receive:
            #(true nil true false)
        fromOneOf:
            #('i-forkLifter-isUp' 'mmi-operatorReset' 'i-emergencyStop'
            'i-operatingSwitch')
        do:
            [:port :item | (port = 'i-forkLifter-isUp') ifFalse: [KillSignal raise]]
        within: 6 seconds
        ifTimedOut: [KillSignal raiseErrorString: 'time-out lift going up'].
    self putOff: 'o-forkLifter-power'
```

Figure 5.3.1 *Detection of constraint violations in every operation.*

The object to be received from a port in portArray is the object from objectArray with the same index as the port in portArray. The first object that can be received terminates the receive action and causes the doBlock to be executed with two arguments: the received object, and the port from which it was received. In Figure 5.3.1, the specification of the nil object in the object array means that any object can be received from the corresponding port in the port array.
This functionality is referred to as the select-interaction functionality. Such a functionality is also offered by other programming languages, such as Ada by means of the select statement (see the Ada Language Reference Manual [Ichbiah, 1983]), and Sequential Program Charts [IEC, 1988].

The method forkUp is now no longer reusable and it is also difficult to read. If, for example, the fork-lift were to be tested under direct control of the

operator, without any synchronization with other processes, the constraints would be entirely different, requiring a different forkUp method.

The conclusion is the same as stated in Section 5.2.1: namely, that each operation should only specify its own constraints. The constraints of an operation should not be unnecessarily duplicated in the operations called. It is not possible to achieve this with conventional mechanisms. The way in which constraints can be specified in operations such that they need not be repeated in the called operations will be dealt with in Chapter 6, where a new mechanism for the handling of constraint violations is introduced.

5.3.2 Constraint violations by controlling processes

In the examples given in Section 5.3.1, the active constraints of a controlling process were violated by processes in the controlled system. These constraint violations were detected by the controlling process itself. Constraints can also be violated by other controlling processes, which will then need to inform the controlling process of which an active constraint was violated. The action of informing another process of a violation of one of its active constraints will be referred to as **signaling a constraint violation to a process**.
This will be illustrated by a further elaboration of the example of the transport system from Section 2.3.

When the truck moves the stack to the furnace and reaches the point where it can turn the fork to the furnace, three actions will be performed in parallel: the fork will go down; it will turn 180 degrees towards the furnace; and, at the same time, the truck will continue to move towards the furnace. The three actions are controlled by the controllers SlaveForkLifter, SlaveForkTurner and CtrlTruck respectively. If the fork-lift is obstructed by another object while going down, it will be necessary to stop the fork-lift, the fork-turner and the truck in order to avoid damage. The SlaveForkLifter (and not the other two processors) should detect the obstruction of the lift. This is because the lift movement is initiated by SlaveForkLifter. When the error is detected by the SlaveForkLifter processor, the constraint violation will have to be signaled to the SlaveForkTurner and the CtrlTruck processors so that they can stop the fork turner and the truck. After correction of the error, by the operator for example, all three interrupted movements can be continued.

Constraint violations can also take place between processes that do not interact under normal processing circumstances. Consider a robot which accidentally drops a product on an assembly line, but which does not interact with the assembly line in any other way. The line will probably have to be stopped. In this case, the error will be detected by the robot controller. The process controlling the assembly line will have to be informed of this constraint violation in such a way that it can immediately stop the line. Once the dropped product is removed, the assembly line can continue normally.

The examples given above are examples of constraints which are imposed in order to prevent damage. Apart from such constraints, there will also be constraints imposed in order to maintain consistency in the synchronization between two processes. When two processes are interacting, they maintain certain assumptions about each other's external states. These assumptions should be correct in order to achieve correct synchronization between the processes and to prevent deadlock.

Consider the example given in Figures 5.3.2a-c, which is also based on the example of Section 2.3. The example is slightly modified so that CtrlTruck directly controls the fork-lift. The example describes the transfer of a stack from the traverse by the fork-lift truck. The methods invoked are indicated with an arrow. In the example the fork does not go up in time. Here this results in the immediate raising of the KillSignal.

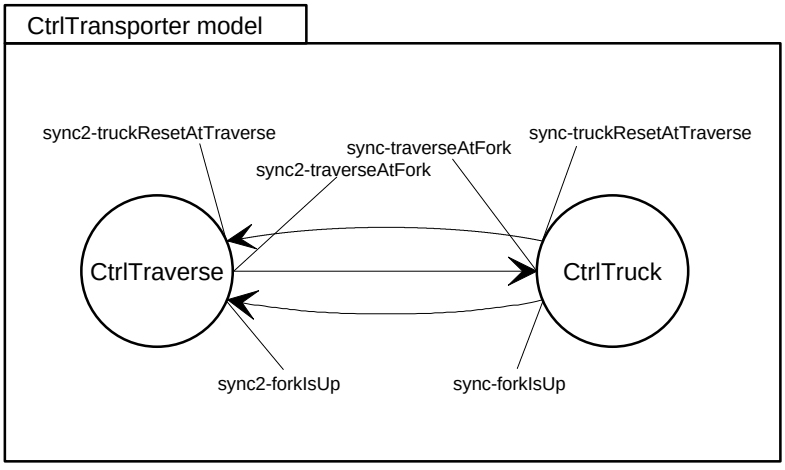

Figure 5.3.2a *Violation of a constraint in one process due to an exception in another A simplified model of CtrlTransporter.*

```
CtrlTraverse >> body
    stackSize >= self maxStackSize ifTrue: [self stackToTruck]. ←
    self stackTray

CtrlTraverse >> stackToTruck
    self receive: 'belowMiddle' from: 'sync1-pusherState'.
    self receiveFrom: 'sync2-truckResetAtTraverse'.
    self traverseToFork.
    self sendTo: 'sync2-traverseAtFork'.
    self receiveFrom: 'sync2-forkIsUp'.                    ←
    self traverseToPusher
```

Figure 5.3.2b *Violation of a constraint in one process due to an exception in another Process description of CtrlTraverse.*

```
CtrlTruck >> body
    self receiveStackFromTraverse.                         ←
    self transportStackToFurnace.
    self giveStackToFurnace.
    self goBackToTraverse

CtrlTruck >> receiveStackFromTraverse
    self sendTo: 'sync-truckResetAtTraverse'.
    self receiveFrom: 'sync-traverseAtFork'.
    self forkUp.                                           ←
    self sendTo: 'sync-forkIsUp'

CtrlTruck >> forkUp
    self putOn: 'o-forkLifter-up'.
    self putOn: 'o-forkLifter-power'.
    self
        receive: true
        from: 'i-forkLifter-isUp'
        within: 6 seconds
        ifTimedOut:
            [KillSignal raiseErrorString: 'time-out fork going up'.   ←
                "The raising of the KillSignal here implies a violation of one
                of CtrlTraverse's active constraints concerning the external
                state of CtrlTruck"].
    self putOff: 'o-forkLifter-power'
```

Figure 5.3.2c *Violation of a constraint in one process due to an exception in another Process description of CtrlTruck.*

In reality, the operator or the system itself would first try to correct the error and the KillSignal would only be raised if the error could not be corrected locally. The example has, however, been kept simple to show only the important aspects.

Due to the raising of the KillSignal, the methods forkUp, receiveStackFromTraverse and body from CtrlTruck will be terminated, leading eventually to the resetting of the truck (not shown in the example). This is a violation of a constraint of CtrlTraverse, since CtrlTraverse will now have an incorrect assumption about the external state of CtrlTruck. After correction of the error, the CtrlTruck processor will start the body again and become blocked in the statement self sendTo: 'sync-truckResetAtTraverse'. The CtrlTraverse processor will still be blocked in the statement self receiveFrom: 'sync2-forkIsUp'. Thus deadlock will result. To avoid this an exception must be raised in CtrlTraverse when CtrlTruck violates CtrlTraverse's constraint by breaking out of its synchronization with CtrlTraverse due to the raising of the KillSignal.

5.3.3 Some relationships between constraint violations, exceptions and the violation of invariants

In the preceding sections, it has been shown that a constraint violation causes an exception occurrence in the process of which an active constraint is violated. This exception occurrence should eventually result in the raising of an exception in the process. In this section, some different possibilities will be treated in order to answer the question of how constraint violations eventually can or should result in the raising of an exception.
The different possibilities found will be used in following sections to categorize and evaluate the known mechanisms for the handling of constraint violations, eventually leading to a new mechanism.

Consider two processes: a violator and a victim. The violator violates one of victim's active constraints, causing an (external) exception occurrence in the victim. If the victim has an exception handling mechanism which supports the raising of exceptions, then an exception needs to be raised in the victim.

There are three possible situations in regard to which process detects the constraint violation by the violator and which one raises the (external) exception in the victim:

- If the victim is the detector it makes no sense for the violator to be the raiser, so if the victim is the detector of the constraint violation, the victim will also be the raiser of the exception. An example of this is when the victim is a controlling process and the violator is a physical process in the controlled system. The victim detects a constraint violation in the controlled system and consequently raises an exception.

- If the violator is the detector, the victim could be the raiser. This means that the violator needs to inform the victim of the constraint violation by means of an interaction. Consequently the victim could raise the exception.

- In the third situation, the violator is the detector of the constraint violation and consequently raises an exception in the victim. This means that the violator raises an exception in another process.

The violator and the victim could also represent a set of processes, rather than a single process.

- If the victim is a set of processes, this leads to the raising of exceptions in several processes.

- If the violator is a set of processes, this indicates the possibility of concurrently occurring exceptions in the victim. Each exception occurrence should lead to the raising of an exception in the victim. Since it is not possible to raise exceptions concurrently in a single process, there must be a mechanism to selectively and sequentially raise one or more exceptions, and to buffer or discard any remaining exceptions.

Another point of interest is the point in the program at which the exception is actually raised in the victim. The simplest possibility is to place no restrictions on the point at which the exception is raised. Since the occurrence of the constraint violation by the violator generally is not synchronized with the process in which the exception should be raised, the exception can be raised at any point in the process, which could lead to an inconsistent state of the process such that its invariants no longer hold. This can be prevented by deferring the actual raising of the exception until the process is in such a state that its invariants are either valid or can be restored by an exception handler. This could, for example, be a state in which the process is blocked, waiting for a delay or for an interaction to take place.

When the exception is raised, the process should be unblocked in a well-defined way, taking account of other processes which possibly participate in the interaction.
The undesirability of allowing the raising of exceptions due to constraint violations at any point in a process is illustrated in Figures 5.3.3a-c.

```
ForkLifterCtrl >> forkUp
    AnySignal
        handle:
            [:exception |
            "If an exception due to a constraint violation would be raised at this
            point, then the power of the lift motor would not be switched off.
            This would mean a violation of the invariant that specifies that the
            lift motor is stopped when the forkUp method is terminated."
            self putOff: 'o-forkLifter-power'.
            exception reject]
        do:
            [self putOn: 'o-forkLifter-up'.
            self putOn: 'o-forkLifter-power'.
            self
                receive: true
                from: 'i-forkLifter-isUp'
                within: 6 seconds
                ifTimedOut: [KillSignal raise].
            self putOff: 'o-forkLifter-power']
```

Figure 5.3.3a *Violating invariants by raising exceptions due to constraint violations.*

```
LinkedList >> addLast: aLink
    "This example is taken from the Smalltalk-80 system.
    aLink is added to a linked list which is linked by pointers.
    firstLink and lastLink point to the beginning and end of the list
    respectively."
    self isEmpty
        ifTrue: [firstLink := aLink]
        ifFalse: [lastLink nextLink: aLink].
    "If an exception due to a violation of one of the process's active
    constraints would be raised here, the internal structure of the linked list
    would become incorrect."
    lastLink := aLink.
    ↑aLink
```

Figure 5.3.3b *Violating invariants by raising exceptions due to constraint violations.*

```
OrderController >> sendOrder
  "The invariant in this case is that the contents of orderBuffer plus order
  itself represent all orders which have not yet been sent.
  order is a temporary variable only used in this method"
    | order |
    AnySignal
        handle:
            [:exception |
            "Put the order back into the order buffer when the send action of
            the order has been terminated with an exception."
            order == nil ifFalse: [self putOrderBackIn: orderBuffer].
            exception reject]
        do:
            [order := self getOrderFrom: orderBuffer.
            self send: order to: 'out'.
            "If an exception due to a constraint violation would be raised at this
            point (after a successful send), then the exception handler would
            put the order back into the orderBuffer although the order had been
            successfully sent."
            order := nil]
```

Figure 5.3.3c *Violating invariants by raising exceptions due to constraint violations.*

In Figure 5.3.3a, the invariant specifying that the power of the lift motor is switched off when the method forkUp is terminated cannot be guaranteed, because all statements where the motor is switched off can be interrupted by the raising of an exception due to a constraint violation. In Figures 5.3.3b-c the violated invariants cannot be restored, because the exceptions could have been raised at any point in the program.

5.4 Requirements for a mechanism for the handling of constraint violations

It appears from the previous sections that there is a need for a mechanism to handle constraint violations. A constraint violation causes an exception occurrence in the process of which an active constraint is violated. In some cases, the process can detect the constraint violation itself and consequently raise the corresponding exception. In other cases, the constraint violation is detected by another process which must consequently inform the affected process about the constraint violation. This can be done either by raising an

exception in the affected process or by means of another kind of interaction which leads to the raising of an exception in the process.

The known mechanisms for the handling of constraint violations will be treated in Section 5.5, and a new mechanism will be introduced in Chapter 6. Before this, the requirements that should be fulfilled by any such mechanism will be given. Only the most important requirements are given. The existing and new mechanisms will be evaluated using these requirements, together with the requirements presented in Section 4.2. The terminology of violator and victim is taken from Section 5.3.3. The requirements are as follows:

1. The mechanism should be easy to use and to understand. Ideally, it should introduce only a small number of new elements that are orthogonal to the rest of the programming support. The mechanism should be compatible with the existing interaction and exception handling mechanisms.

2. The mechanism should make it possible to change the normal flow of control of a process when its active constraints are violated. The mechanism should enable the designer to specify this dependency between constraint violations and change of control flow in a controlling process in a precise, intuitive and natural way.

The above requirements are of a very general nature. The following requirements are more specific:

3. The mechanism should allow each operation to specify only its own constraints: the constraints of operations which have already been invoked should not need to be respecified.

4. The mechanism should be sufficiently flexible and precise so that, in the case of constraint violations, only those processes of which active constraints are violated will be affected.

5. The mechanism should defer the actual raising of the exception until the victim process is in such a state that its invariants are either valid or can be restored by an exception handler.

The last two requirements need some further explanation.

Requirement 4 calls for flexibility and precision of the mechanism. Many existing mechanisms severely restrict the way that exceptions can be raised in response to constraint violations. The mechanism may, for instance, be limited to the raising of the same exceptions in the parallel sections of a programming construct to express parallel actions. Or it may be limited to a static relationship, such that the occurrence of an exception in a process will always result in the raising of exceptions in a fixed group of other processes, independently of where in the original process the exception occurrence took place.

In reality, the determination of which processes should be affected by an exception occurrence in another process can very much depend on the point in the program where the exception occurred. This has already been illustrated by numerous examples in previous sections. When the truck controller from previous examples is synchronizing with the traverse controller, exceptions in the truck controller are likely to affect the traverse controller. If, on the other hand, the truck controller is synchronizing with the furnace controller, exceptions in the truck controller will affect the furnace controller.
The relationships can even extend beyond processes which interact. If a robot accidentally drops a product on a transporting belt which it needs to cross, an exception may need to be raised in the belt controller. In this case, the relationships could be due to the physical layout of the system. There need not be any other relationship between both controlling processes.

The last requirement has already been treated in Section 5.3.3. It is clearly undesirable to permit the raising of exceptions due to constraint violations at all times: this would mean that invariants of the process could be corrupted in such a way that they could not be restored by exception handlers. It is especially important to restrict the raising of pending exceptions in such a way that the internal invariants of a process are valid when pending exceptions are raised, because it is practically impossible to deal with the restoration of all internal invariants in exception handlers. To allow external exceptions to be raised at any time during the execution of a process will lead to time-dependent run-time errors. It is virtually impossible to detect such errors by program testing, and they constitute one of the most hazardous aspects of concurrent programming.

5.5 Known mechanisms for the handling of constraint violations

This section will discuss some known mechanisms for the handling of constraint violations. It will be shown that these mechanisms do not allow the general specification of constraints of operations and do not satisfy the requirements of Section 5.4. Not all mechanisms are treated but a selection is made of some representative mechanisms. Mechanisms that are only defined for the handling of exceptions which occur during the execution of an interaction, such as during a rendezvous, are language-specific and are therefore not treated.

5.5.1 The select-interaction functionality

This functionality has been treated in Section 5.3.1. The use of this functionality for the detection of constraint violations leads to unreliable and inelegant programs, whether used in Process Calculus, Ada, Sequential Function Charts, or any other language. This is because the interactions that are used to detect constraint violations need to be respecified in many operations. There is in fact a substantial analogy between this way of dealing with constraint violations and the use of returned values as exception codes in the handling of internal exceptions as treated in Section 4.3.1. In both cases, the creation of robust programs becomes difficult. Using the select-interaction functionality, the amount of code to check for *constraint violations* becomes enormous, leading to a great deal of code pollution. The constraints of operations which have already been invoked must be duplicated in all called operations. A constraint violation, such as the activation of the emergency button, should be catered for in every possibly blocking interaction. A single element of an interaction with many constraints is easily forgotten. Also, the process will not detect constraint violations between two blocking interactions.

An important disadvantage of the use of the select-interaction functionality for the detection of constraint violations is the fact that this way of programming does not reflect the way a designer thinks about the system. A more natural, but not yet precise, way to specify the relationship of the emergency button to a specific controlling process would be something like: 'If the emergency button is pressed while a certain unit is active, an exception should be raised in that unit'. Thus the specification is in terms of

coupling the raising of an exception to a certain unit and not to all individual interactions it contains.

5.5.2 Raising exceptions in other processes

The traditional ways of dealing with constraint violations are generally based on the possibility of raising an exception in another process in one way or another. This conforms to the situation described in Section 5.3.3, where the violator is the detector of the exception and consequently raises an exception in the victim. The terms violator and victim will also be used in this section.

The concept of raising exceptions in other processes can have a damaging effect on program modularity. This is especially the case when the raising of exceptions in other processes cannot be restricted.

In [Booch, 1991] modularity is defined as follows: 'Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules'. Modules serve to make the complexity of large systems manageable. The essence is that the complexity of a system can be made manageable by dividing the system into modules that can be understood and developed mainly independently, without the need to know the inner details of the other modules. The modules should be loosely coupled. Loosely coupled modules generally have relatively few interactions between them and they should not have access to each other's local data, nor to shared global data. The desirability of weak coupling and strong cohesion is described in many books on software engineering. See for example [Fairley, 1985] for a more detailed treatment.

The use of parallel processes to control inherently parallel physical systems is a way of dividing the complex controlling system into a set of more easily manageable modules. The desired loose coupling implies that each process should need to know as little as possible about the inner details of the other controlling processes. Therefore, if an exception occurs in a process (the violator), and other processes (the victims) may need to be interrupted as a result of this, the violator should convey the *intent* to raise an exception to the victims. The violator generally does not know, and does not wish to know, the exact state of the victims. The victims themselves should only raise the exception if they are in a state to handle the consequent exception. This makes the violator and the victims largely independent.

An example of the problematic consequences of permitting the unconditional raising of exceptions in other processes is the following. Consider three closely cooperating processes, for example the three processes involved in the control of the fork-lift truck of Section 2.3. If an error is detected in one of the three processes for which local error recovery is impossible, that process will have to raise an exception to terminate its control cycle and reset itself. As a result of the termination of its control cycle, the constraints of the other two processes will be violated and so exceptions should be raised in the other two processes to cause them to also terminate and reinitialize. Each of these two processes, however, will, as a result of their termination, in turn raise exceptions in the other two processes. This would mean that undesired exceptions are raised in the violators which have already terminated. To prevent this from occurring, the process that wants to raise the exception should know the state of the victim processes, leading to greater coupling and reduced modularity; or else provisions should be made in all processes to catch all undesirable exceptions. If, on the other hand, the processes are specified in such a way that a request for the raising of an exception is only honoured when this is appropriate, coupling will be reduced and modularity improved.

The mechanisms found in the literature will now be evaluated in relation to the requirements specified in Sections 5.4 and 4.2.

Ada

The Ada programming language [Ichbiah et al., 1983] offers no facilities for explicitly raising exceptions in other processes. The rendezvous mechanism is the main mechanism for synchronization and communication between processes. The use of global variables is also possible but is not advised.
The only occasion during which exceptions will be raised in other tasks is a rendezvous. If an exception is raised from within an accept statement, the exception is also raised in the other task participating in the rendezvous. This is done implicitly by the system. The programmer has no control over the raising of the exception.
In the preliminary version of Ada as described in [Ichbiah et al., 1979], there was the facility to raise a special FAILURE exception in another task.

VAXELN

VAXELN is a programming environment from Digital Equipment Corporation [Digital, 1986] which has provisions for parallel processing, synchronization and communication between processes, as well as facilities for the handling of internal and external exceptions. It is suitable for the development of real-time applications. VAXELN language definitions are provided in C and Pascal. Its exception handling facilities are based on user-defined functions of the predefined type EXCEPTION-HANDLER. Within a program, procedure, function or process block, only one function of this type can be established at a time as the exception handler for that block. This is done by calling the procedure 'establish' with the exception handler function as argument. An exception handler function established in this way is referred to as the handler for the block. This handler is then called on the occurrence of exceptions in that block's activation. The function receives the exception which has occurred as an argument. If no handler is established for the block in which the exception has occurred, then the stack of active blocks is searched for a handler. When a handler is found, it is executed with the exception as argument and, optionally, other arguments that were included when the exception was raised. Only two responses are allowed from the handler: the propagate and resume response. There is no return, or retry response. The handler must return a boolean. If true is returned, the process is resumed after the point where the exception was raised. This is the resume response. If false is returned, the exception is raised again and the system searches for a handler in an enclosing block. This is the propagate response. The GOTO statement can be used to jump to a label in a higher level block and in this way terminate active blocks. The stack can also be unwound explicitly by means of an unwind procedure.

A process can also raise exceptions in other processes. Exceptions that are raised in this way are termed asynchronous exceptions. They are the same as the external exceptions defined in section 4.1.2. A process can disable and enable asynchronous exceptions. When asynchronous exceptions are disabled, the raising of an asynchronous exception in the process by another process will have no effect.

The exception handling mechanism used in VAXELN has a number of qualities that should be present in an advanced exception handling mechanism. The mechanism is unstructured, however. The fact that the language does not provide structured exception handler responses is

compensated by offering unstructured sequencing constructs like the GOTO statement and the unwind procedure.

The mechanism for the handling of external exceptions offers only the bare functionality of raising exceptions in other processes. Constraint violations cannot be monitored by the process itself. The programming manual [Digital, 1986] does not specify that asynchronous exceptions are made pending, to be raised at a later time. If this is indeed not done, then invariants of a process can be corrupted in the case of asynchronous exceptions. This should be prevented by temporarily disabling asynchronous exceptions in all appropriate places of the user program and in all library routines that are called. Accidentally forgetting to disable asynchronous exceptions can result in subtle time-dependent run-time errors which are hard to detect, and occur only under very specific circumstances.

ROSKIT

ROSKIT [Rossingh and Rooda, 1985] is a small, real-time multitasking operating system designed especially for machine control. It is written in Modular Pascal [Bron and Dijkstra, 1987]. Process synchronization is achieved by means of semaphores. It includes the exception handling mechanism of Modular Pascal, which has already been treated in Section 4.4.

Three external exceptions are defined: the xKill, xAbort and xTimeOut exceptions. A process can raise these exceptions in another process. This is known as forcing an exception in another process. The actual raising of external exceptions is deferred until the process is blocked or when it may become blocked. External exceptions cannot be disabled.

Alarms are introduced to monitor constraint violations specifically caused by the state of the controlled system. For this purpose, an alarm can be bound to a boolean variable which is bound to the physical state of a binary sensor in the controlled system. Alarms are explicitly disabled and enabled: they cannot be bound to blocks. When enabled, they will signal a violation of the specified constraint when the actuator and the associated boolean variable take on the specified value. The constraint violation is signaled by creating a pending exception for the external exception xAbort. This pending exception will be raised when the process executes a possibly blocking operation.

The ROSKIT system has been successfully used for the control of complex production systems. Its exception handling mechanism, however, cannot be used for the specification of constraints of operations.

Szalas and Szczepanska's proposal

In [Szalas and Szczepanska, 1985] a proposal is given for the raising of exceptions, which these authors call signals, in another process. Only the most important issues in the proposal will be treated. Szalas and Szczepanska specify that the mechanism should satisfy five postulates, two of which are as follows:

- 'Signals can be received and handled if and only if the receiving process is active'.
- 'Receiving a signal consists in the immediate interruption of the execution of the receiving process and in a handler invocation (if a relevant handler exists)'.

These postulates are precisely the opposite of the requirements given in Section 5.4.
In the case of a constraint violation, an inactive process should immediately be activated to handle the constraint violation. If this were not necessary, then the signal should not have been sent in the first place.
Szalas and Szczepanska claim that delays in the raising of signals are unacceptable because 'the meaning carried by a given signal is strongly connected with the actual state of the system and its environment'. However, when the raising of a signal is delayed until the process executes an interaction (see the treatment of the new mechanism in Chapter 6), the delays will be determined solely by the speed of the controlling system, which should in any case be fast enough to meet the real-time requirements of the controlled system.

Real-time Euclid

Real-time Euclid is described in [Kligerman and Stoyenko, 1986]. Its exception handling mechanism is different from the usual mechanisms. The way that processing is continued after execution of an exception handler is not determined by the handler itself, but by the way the exception was

raised. The only way that a handler can affect the way processing is continued is by raising an exception itself.

Exceptions can be raised in three ways: by means of the kill, deactivate and except statement. All three statements take two arguments. The first argument is the identifier of the process where the exception is to be raised. This process can be the process which is currently executing. The second argument is the exception number. Exceptions raised with a kill statement will terminate the process after the execution of its handler. Exceptions raised with the deactivate statement will terminate the process's current frame. Frames are used to specify the period in which a process must complete its task. Periodic processes will automatically start a new frame after their period has expired. Exceptions raised with the except statement cause the handler of the process to execute, whereafter the process continues after the statement that raised the exception.
The exact functionality of the mechanism is not made clear in the article. It appears that the mechanism does not allow for handlers at different levels to perform finalization obligations when an exception is raised. Exceptions that are raised in other processes appear not to be made pending but are raised immediately, so that internal invariants can be violated. The mechanism essentially provides only the functionality of raising exceptions in arbitrary processes, whereby the exception handler response is determined by the type of statement used to raise the exception.

5.5.3 Handling the exception of one process in another process

Some proposals suggest that a handler for an exception can reside in a different process than the process in which the exception was raised. These mechanisms are related to the mechanisms described in the previous section. In the previous section, however, it was implicitly assumed that an exception is only raised in another process in the case of an exception occurrence (due to a constraint violation) in the other process. The proposals considered here suggest that, in the case of an exception occurrence in a certain process, say process A, the handler of that exception may be found in another process, say process B. The handler responses of the handler in process B are related to the continuation of process A after the handling of the exception by process B. However, in such a case there is no exception occurrence in process B. So this kind of communication between processes should be done by the normal interaction mechanisms provided by the language used. Exceptions should only be raised in a process if there is an

exception occurrence in that process. Therefore, the use of these mechanisms is considered to be in conflict with the definition and spirit of exceptions. The mechanisms also introduce unnecessary complexity.
These kinds of mechanism are found in [Levin , 1977; Antonelli, 1989; Atkins, 1985].

5.5.4 Dealing with exceptions in parallel constructs

One kind of mechanism is not evaluated in detail. This concerns mechanisms specifically designed to cope with the exception handling issues in languages that make use of parallel constructs, such as the parallel command in CSP [Hoare, 1978] as shown in Figure 5.5.1.

```
[P1 || P2 || .. || Pn]
```

Figure 5.5.1 *The parallel command in CSP.*

A parallel command specifies concurrent execution of its constituent processes (P1 .. Pn). They all start simultaneously and the parallel command terminates successfully only if and when the constituent processes have all successfully terminated. A similar construct exists in Sequential Function Charts [IEC 848, 1988], for instance. Double lines are used in SFCs to represent the beginning and end of simultaneous sequences.
The process that executes the parallel command is referred to as the father process. The processes represented by P1 .. Pn are referred to as the children processes. Difficulties arise when one or more of the children processes terminate with an exception. The questions in such a case are whether and how the other children should be automatically terminated and what exception should be raised in the father process. Problems like these occur in all systems that allow the dynamic creation and termination of processes, such as Ada, for example.
Industrial systems can easily be controlled with a fixed number of processes that are created once and are never deleted, unless the whole program is terminated. If provisions are desired in a language for the control of systems to dynamically create and delete processes, then the exception handling mechanism of such a language should fulfil all the requirements specified in Section 5.4, plus the additional requirements to cope with a premature termination of a child process due to an exception. Most languages with this kind of parallelism include only mechanisms to fulfil the additional requirements to cope with the exceptional termination of a child process. Because such mechanisms are language specific and cannot be used in a

general way to specify the constraints of an operation, they will not be treated in detail.

Some concrete limitations of these mechanisms, apart from the inability to specify the constraints of operations, are as follows:

- The mechanisms provide only means for dealing with the *termination* of a child process with an exception. In the treatment of the retry strategy in Section 7.6, it is shown that exceptions in a child process which are locally handled can also give rise to the necessity of raising exceptions in other processes.
- There is no way of propagating exceptions to parallel processes that are not included in the same parallel construct.

Examples of mechanisms that have been developed specifically to deal with exceptions occurring in parallel constructs are found in [Adamo, 1989, 1991; Lacoutre, 1991; Issarny, 1991].

5.5.5 Other mechanisms

Antonelli's dissertation

In his dissertation 'Exception handling in a multi-context environment', Antonelli (1989) proposes an enhancement to the Ada programming language. Exceptions can be exported from modules and imported into other modules. In this way, there is a static association between the violator where the exception is raised and the victims that import the exception. The exception will be raised in all modules that import the exception.

In the modules that import the exception, the exception must be handled by a special exception handler *task*. Each such task contains an entry associated with an imported exception. This proposed syntax is analogous to Ada interrupt handler tasks. The task should be suspended on an accept statement waiting for the imported exception to be raised by another task. When the exception is raised, the 'rendezvous' takes place and the accept statement is elaborated. The exception can only be caught in the other task when it is suspended at the accept statement. Therefore, the exception handler task must wait passively for the exception to occur and cannot do anything else.

In order to raise an exception in a victim task, the victim task must define an exception handler task to catch externally raised exceptions. But the problem still remains that an exception should be raised in the victim task itself. To effect this Antonelli uses global variables as a means of information transport between the victim task and its exception handler task. When an exception is caught in the exception handler task, a boolean is set. The victim task must *poll* the boolean and if it is set it must raise the exception itself.

This proposal is unsatisfactory for two reasons:

- The victim cannot poll when it is suspended in an accept or delay statement. So there is no means of interrupting the victim when it is blocked.
- The need for the victim to continuously poll leads to inelegant and unreliable code. A single poll is easily forgotten.

Lieber's dissertation and similar proposals

Lieber's dissertation 'Erweitertes CSP-Modell zur programmierung paralleler Prozesse' [Lieber, 1989] is an extension of Hoare's concept of Communicating Sequential Processes [Hoare, 1978]. The CSP model is extended with ports as a means of communication between processes.
Another of the proposed extensions deals with the addition of exception handling facilities. The essence is that a running process can be interrupted when a receive operation from a port can take place. This is realized by means of except statements, which consist of a statement list associated with an if-statement list by means of the interrupt or except operator. The if-statement list usually begins with a receive action from a port. See the part of a program taken from [Lieber, 1989] in Figure 5.5.2.

```
i := 1;
*[i < 1000 -> {skip}
    except [keyboard?x -> a[i] := x; i:= i + 1;]]
```

Figure 5.5.2 *An example of Lieber's except statement.*

The semantics of such a construct are such that the statement list is executed until the receive action from the if-statement list can take place. The statement list is then interrupted and execution continues with the if-statement list. When the if-statement list is finished, the continuation of

execution depends on the operator of the except statement. If the interrupt operator is used, execution continues after the interrupted statement from the statement list. This can be viewed as the resume response from a handler. In the case of the except operator, execution continues with the statement following the except statement, which resembles the return response. If the receive action from the except statement does not succeed at all, the statement list will finish normally and execution continues after the except statement.
Lieber considers send and receive actions to be atomic and does not allow the interruption of send or receive actions in the statement list of an except statement.

A positive aspect of this proposal is the fact that the mechanism uses the normal interaction mechanism as a means for exception handling between processes. Secondly, the effect that an exception in a process can have on another process is limited to except statements.

The proposal, however, does not satisfy the desired functionality of an exception handling mechanism for controlling systems for the following reasons:

- There is no means of interrupting a victim when it is blocked.
- There are no provisions for the handling of internal exceptions: exceptions cannot be declared, raised or propagated, for example.
- Finalization obligations cannot be locally specified nor be executed when an operation is terminated: if the statement list from an except statement is terminated because a receive action from the if-statement list receives from the specified port, all (nested) operations which are invoked from the statement list will simply be terminated.
- The process can be interrupted in the middle of executing statements. If the return response is chosen, this may lead to the violation of the program's internal invariants and to inconsistent data. The resume response has all the disadvantages indicated in Sections 4.6.3 and 4.6.4.

The exception handling facilities described in [Gerber and Lee, 1992] bear a great resemblance to Lieber's. Gerber and Lee describe their CSR (Communicating Shared Resources) Specification Language in their paper. The scope statement allows the specification of triggers to be associated with a statement S:

```
scope do
    S
interrupt recv(channel1) -> S1
interrupt send(channel2) -> S2
od
```

Figure 5.5.3 *An example of the scope statement.*

There are four kinds of trigger guards: send, recv, exec and timeout. The send and recv operators specify send and receive actions to or from communication links. Parallel processes interact by means of communication links. If, during execution of the statement S from a scope statement, the timeout expires, or one of the 'interrupts' can be executed (for instance when a send or receive action can take place), the S statement is terminated prematurely. Thus the interrupt specifications in Gerber and Lee's CSR language are analogous to the use of the except operator in the if-statement list of Lieber's except statement. The evaluation of this mechanism is therefore analogous to the evaluation of Lieber's mechanism.

C for Unix

C [Kernighan and Richie, 1978] for Unix does not provide advanced programming constructs for the handling of internal exceptions, apart from the setjmp and longjmp procedures. Setjmp saves the current context in a variable. When this variable is used as argument for the longjmp procedure the context is restored to the value at the time of the last call to setjmp, causing the process to continue right after this statement. This is a kind of goto-like mechanism. The main way to deal with internal exceptions is by using returned values as exception codes (see Section 4.3.1).

Unix provides signals as a means of exception handling between processes, see [Bell, 1983]. These signals are different from the raisable signals used in the present dissertation. Signals are sent to a process. They are used both to bring external exceptions and system detected internal exceptions to the attention of a process. A process can specify what to do when it receives a signal. It can choose to ignore the signal, terminate itself or to invoke a handler. Handlers are ordinary procedures. Handler procedures can be bound to signal occurrences by using the signal itself and the address of the signal handling procedure as arguments to the signal procedure. If a handler has been installed for a signal and the process receives a signal, the execution of the process is interrupted. The handler is executed, and after

that, execution of the process is continued from the point at which it was interrupted. If other responses are desired, these should be obtained by means of the setjmp, longjmp procedures.
Many other features of signals are described in [Sun, 1990]; they do not, however, change the main functionality described above.

The main problem with this way of handling exceptions is that C lacks a mechanism for the handling of internal exceptions, as described in Section 4.4.
Another problem is caused by the way signals are handled. The function of signal handlers is very similar to interrupt handlers. This leads to a limitation on the possible responses from such a handler: resumption or the drastic termination of the process, without the possibility of fulfilling local finalization obligations. If the resume response is chosen, the interrupted program will need some sort of polling mechanism to detect whether or not a signal interrupt has taken place.

Proposals by Issarny and Banâtre

These proposals are described in [Issarny, 1990; Issarny and Banâtre, 1990; Issarny, 1991]. The objective of the proposed mechanism in all three articles is restricted to deadlock avoidance in the presence of processes which terminate with an exception. In the context of these articles, the termination of a process means the termination of the current iteration step when the body of the process consists of a repetitive command.

Global exceptions are introduced in the first two articles. An exception is global when a handler for that exception is found in the handler list attached to a process body; otherwise the exception is local. If a process raises a global exception, the exception will automatically be raised in all processes where a handler for the exception is found in the handler list attached to the process body. Raising a global exception in a process implies the termination of that process.

The raising of exceptions in *all* processes that have a handler for the global exception can be too unrestrictive. If, for example, such a process has already finished its interactions with the process that originally raised the exception, deadlock will not result.

In the third article, the raising of global exceptions is restricted to processes that actually want to interact with a process that terminates by signaling a global exception. In this way, the mechanism bears a resemblance to the way that the tasking-error exception is raised in Ada when a process wants to execute a rendezvous with a terminated process. In this article, the mechanism is also adapted to take account of exceptions occurring in parallel constructs, as treated in Section 5.5.4.
The restrictions of the mechanism, which only allows for the raising of exceptions in other processes in the case of the termination of a process, make it unsuitable as a general mechanism for dealing with constraint violations.

Chapter 6
A new mechanism for the handling of constraint violations

This chapter describes a new mechanism for the handling of constraint violations. The new mechanism is first treated independently of any particular implementation. The implementation of the mechanism in Process Calculus is given in Section 6.10, which also includes some examples.

6.1 The specification of constraints with constraint monitors

6.1.1 Definition of terms

When an operation is executed, its constraints must be valid, since otherwise its goal cannot be achieved. A violation of one of its constraints is an exception occurrence which should eventually result in the raising of an exception. Therefore, each constraint should be linked to an exception which should be raised after a violation of the constraint. This results in the following definition of *constraint monitors*.

A **constraint monitor** consists of a constraint and an (external) exception; it is used to signal violations of the specified constraint.

The new mechanism for the handling of constraint violations is based on constraint monitors, which are used to detect and signal violations of the constraints of an operation. Since constraints are specified for operations, a constraint monitor can be bound to an operation. An operation is a logically related group of statements or expressions in a sequential process with a single entry (see Section 4.1.1). In many programming languages, blocks are available as programming construct. In these languages, the effect of binding a constraint monitor to an operation can be achieved by enclosing the operation in a block and subsequently binding a constraint monitor to the block. A block to which a constraint monitor is bound is said to be **protected** by the constraint monitor, and can be referred to as a **protected**

block. The operation enclosed by the block is likewise known as a **protected operation**. The constraint monitors bound to the protected block can be referred to as the block's constraint monitors.

During the activation of a protected block, the constraint monitors bound to the block will **monitor** the constraints specified in the constraint monitors, which means that the constraints will be continuously checked to see if they still hold. This monitoring takes place throughout the execution of the block, including during the time that other operations are called from within the block. When the monitor detects a constraint violation, monitoring is stopped, independently of the future termination of the block.

6.1.2 The binding of constraint monitors to blocks

The binding of a constraint monitor to a block can be done either statically or dynamically, meaning that the binding takes place at compile-time or at run-time.

In the case where a block has several (sub-)constraints, two approaches are possible. If only one constraint monitor can be bound to a block, all sub-constraints should be specified in this single constraint monitor using a compound constraint. The other approach is to allow multiple constraint monitors to be bound to a block. The last approach is chosen because in this way a separation of concerns is made possible: independent constraint monitors can be used for the specification of independent sub-constraints. It also facilitates reuse of constraint monitors.

A constraint monitor is **enabled** if and only if a protected block to which it is bound is executing. This is an important aspect of the functionality of constraint monitors, and one which fits in well with the concept of structured programming. If constraint monitors were not bound to blocks and were allowed to be explicitly enabled and disabled, then it would be more difficult to understand the operation of a program: in order to determine at which parts of the program a constraint monitor has an enabled or disabled status, it would be necessary to locate the statement where the constraint monitor was last enabled or disabled. This could be any statement executed by the program. If, on the other hand, constraint monitors are bound to blocks, then the status of a constraint monitor will only be enabled if it is bound to an invoked block. So, in order to determine the enabled

constraint monitors, only the active blocks – which are the blocks on the call chain of the process – need to be examined.
To keep things simple initially, it is assumed that blocks with the same constraints are not nested. The consequences of nesting blocks with the same constraints are treated in Section 6.7.

6.1.3 The language dependency of constraint monitors

The actual specification of constraints depends very much on the programming language used. Constraints always concern the environment of the process executing an operation. The interactions of the process with its environment are described by means of the interaction mechanism which is available in the programming language used. This interaction mechanism should be used as far as possible for the specification of constraints. This has the advantage of remaining compatible with the existing interaction mechanism, and at the same time being able to use its expressive power. Also, in this way only a few new concepts are introduced, keeping the new mechanism simple.

If, for instance, Sequential Function Charts [IEC, 1988] are used, constraints could be specified with those expressions that may occur in the specification of transition conditions. In message-based systems, on the other hand, constraints could be specified by means of messages. Specific messages could be defined to indicate constraint violations. The different possibilities for the specification of constraints in the different languages will not be dealt with in this thesis. The implementation chosen for Process Calculus will be treated in Section 6.10.

6.1.4 The binding of constraint monitors to identifiers

In this section it is shown that it should be possible to bind constraint monitors to identifiers.

If constraint monitors are integrated in a language in such a way that the binding of a constraint monitor to a block implies the definition of a new constraint monitor, then constraint monitors cannot be referred to in the program. This means that, in order to specify equivalent constraint monitors for different blocks, it would be necessary to create a new constraint monitor for every block. This is obviously a disadvantage if equivalent

constraint monitors are bound to many different blocks. In Chapter 7, this is shown to be quite a common situation. Note that two constraint monitors are equivalent when their constraints can be expressed with the same constraint function and their exceptions are the same.

If a constraint monitor may be created first and bound to a block later, then the mechanism should support the binding of a constraint monitor to an identifier. After the binding of a constraint monitor to an identifier, the identifier will denote the constraint monitor. The identifier can then be used every time that the constraint monitor should be bound to a different block. For an explanation of the concept binding, the reader is referred to [Tennent, 1981; Watt, 1990].

Note that some languages use assignment statements to 'bind' certain entities to identifiers, where other languages can produce true bindings. In Ada, for instance, there is a special exception declaration statement to bind an exception to an identifier. In Smalltalk, however, such 'bindings' between identifiers and signals can only be attained by means of assignment statements. First, an identifier is bound to a variable. The identifier can then be 'bound' to a signal by means of an assignment statement. It is the programmer's responsibility to initialize the variable with the required signal value, and to make sure that this value is not altered during program execution. In this way, the variable can be treated as a constant.

There is another reason why it is preferable to allow constraint monitors to be bound to identifiers. If a block is terminated with an exception, it may not be immediately obvious what caused its termination: it could have been the violation of a constraint, but it could also have been an internal exception occurrence. Also, the discarding of pending exceptions can make it possible for constraint violations to remain unnoticed. If the constraint monitor can be referred to with an identifier, the status of the constraint monitor can be examined to test whether its constraint has been violated. In this way constraint violations can always be detected.

6.2 Pending exceptions as a result of constraint violations

When a constraint monitor detects a constraint violation, the constraint monitor's exception will be raised. As has already been explained in Section 5.4, the exception generally cannot be raised immediately, since the constraint violation generally is not synchronized with the execution of the

process of which an active constraint is violated. Therefore, the raising of the constraint monitor's exception is generally deferred to a later time. To achieve this, a *pending exception* is created for the monitor's exception. A **pending exception** is an exception which is about to be raised. The actual raising of the exception is deferred until well-defined points in the program, so that the internal invariants of the process will not be invalidated.

When a constraint monitor detects a violation of its constraint, it will **signal** this constraint violation by creating a pending exception, after which it stops monitoring its constraint. This is because the pending exception should lead to the termination of the protected block with an exception. Leaving the constraint monitor to monitor its constraint could only cause more pending exceptions, equal to the already existing one, which would have no effect. However, the constraint monitor remains enabled. It is disabled when the executing protected block to which it is bound is terminated. A pending exception can be created for a constraint monitor only while it is enabled. When it is disabled, its pending exception (if present) is discarded (see Section 6.5).

The pending exception can be referred to as the constraint monitor's pending exception and the monitor can be referred to as the pending exception's constraint monitor. A pending exception is, in fact, an indication that the exception of a constraint monitor is about to be raised.

If the programming language used supports the raising of exceptions with arguments, these should also form part of the constraint monitor. In such a case, the pending exception will be raised with the appropriate arguments.

6.3 Raising pending exceptions

6.3.1 Instant and delayed response controlling systems

Instant response controlling systems

In instant response controlling systems, the points in time at which the external state of the controlled system – which is determined by the values of the actuators and sensors – changes from one state into another is not significantly affected by increasing the processing speed of the controlling system. Therefore, the abstraction can be made to consider the processing

speed of such controlling systems as infinite, which yields zero response times.
An infinitely fast execution of the controlling processes should have no effect on the correctness of the system; a properly designed controlling system should be independent of the relative and absolute execution speed of the different controlling processes, under the condition that the execution speed is above a certain minimum level in order to guarantee the real-time characteristics of the system.

Examples of such systems can be found in sequence control systems based on parallel processes. In these systems, several controlling processes are usually executed on a single physical processor. Each controlling process spends most of its time being blocked in an interaction, while waiting for a certain change in the state of the processes in the controlled or controlling system. When the external state of the controlled system changes, interactions will take place and the state of the controlling processes will be updated according to the new state of the controlled processes, whereafter the controlling processes will again each be blocked in an interaction. For each process in the controlling system, the time spent by computations between two subsequent suspended states is negligibly small in comparison with the time spent in each suspended state (blocked in an interaction).

There are also instant response controlling systems where the controlling processes are not normally blocked in an interaction. If, for example, there is only one controlling process executing on a dedicated physical processor, the process could be continuously polling the state of the controlled system, so that it can give a response when a change is detected.

Delayed response controlling systems

In delayed response controlling systems, the processing speed of the controlling system plays an essential part in the operation of the controlled system. Increasing the speed of the controlling system will significantly change the points in time at which the external state of the controlled system changes from one state into another. In these systems, there are processes that spend a significant amount of their time on the computations that are necessary in order to respond to changes in the state of the controlled system. An example of such a controlling system is a scheduler. The response time of such a scheduler can be a significant factor in the progress of the controlled system.

6.3.2 A strategy for raising pending exceptions in instant response controlling systems

In order to keep invariants – especially internal invariants – intact, pending exceptions should be raised at points where the invariants are either valid or where invalid invariants can be easily restored in an exception handler which can catch the pending exception that will be raised. Of these two options, raising pending exceptions at points where the invariants are valid appears to be preferable, because it avoids the detection and restoration of invalid invariants in exception handlers. Another requirement, however, is that the raising of pending exceptions should not be deferred for too long. It can therefore be necessary to raise pending exceptions at points where invariants are invalid. The question that is answered in this section is: how are the points determined where pending exceptions are raised?

One approach would be to have the programmer state, for every statement, whether or not it may be interrupted by the raising of a pending exception. This approach clearly places too great of a burden on the programmer and leads to a great deal of code pollution. It would be desirable that the run-time system or the compiler could determine where to raise pending exceptions, without explicit indications from the programmer. Therefore, a strategy for the raising of pending exceptions will be developed.

Raising pending exceptions at interaction points

First it is noted that, in instant response controlling systems, the raising of pending exceptions can be deferred until interaction points, which are statements that execute an interaction with another process. The argument for this is as follows.
When a constraint violation is detected, there are three kinds of delay involved in its handling. Firstly, there is the delay between the time of the actual constraint violation and the time of the detection of this violation by the controlling system. Secondly, there may be a delay between the time of the detection and the time of the notification of the affected process, caused for instance by the need to schedule concurrent processes on a single physical processor. Lastly, there is the delay between the time that the exception could in principle be raised in the affected process and the time that the next interaction point is reached. The first two delays are independent of the strategy chosen for raising pending exceptions: the strategy to defer the raising of pending exceptions until interaction points

introduces only the third delay. The maximum delay which can be introduced by this strategy is equal to the maximum amount of time that is needed for the computations between two interaction points. However, during the time that a process is performing computations between two interaction points, it cannot respond to changes in the controlled processes. Therefore this time must be limited in instant control systems, in order to be able to realize the desired 'instant' response of those systems, thereby yielding the same 'instant' response for constraint violations.

An important advantage of the approach of raising pending exceptions at interaction points is that interaction points are a natural place for internal invariants to be valid. Undesirable interactions after constraint violations are also prevented in this way, because interaction with other processes is only possible by means of interaction points. If global variables are used for the communication between processes, then the updating and reading of these globals are also referred to as interaction points.

Pending exceptions generally cannot be deferred during the execution of delay statements. Therefore, delay statements are also considered as interaction points. They can be viewed as interacting with a timing process.

Pending exceptions will be raised at interaction points, including delay statements, in the following way.
When there is already a pending exception prior to the execution of an interaction point, the pending exception will be raised, replacing the interaction point.
When the process is blocked in an interaction point and a pending exception is created, the process will be unblocked immediately and the pending exception will be raised, replacing the interaction point.
When a pending exception is created at the time the interaction actually takes place (and the process is therefore not blocked), the interaction will be allowed to terminate normally and the raising of the pending exception will be deferred until the next interaction point. In this way, it is guaranteed that an interaction either takes place successfully, or that it does not take place but is replaced by the raising of a pending exception. Raising a pending exception directly after a successful termination of an interaction could in fact cause internal invariants to be invalidated, for example when the statement following the interaction records the number of interactions that have taken place.

Another possible strategy is to defer the raising of exceptions until the process executes an interaction point that causes the process to be blocked. This approach has the advantage of restricting the number of points in the program where invariants can be violated due to the raising of pending exceptions. This could make certain exception handlers simpler. The extra delay introduced in this case can be ignored in instant response systems, because no blocking can take place in the interaction points where the pending exceptions are not raised. The disadvantage is that the execution of certain non-blocking interactions in the presence of pending interactions could be undesirable.
Clearly, this approach is only possible when the controlling processes become blocked when they wait for the controlled system to change state, and do not continuously poll the state of the controlled system. In systems that use polling processes, pending exceptions must be raised at all interaction points. Please note that each poll is an interaction point.

The strategy of raising pending exceptions at all interaction points is preferred to the strategy to raise pending exceptions at blocking interaction points only. In this way, undesirable interactions after the creation of pending exceptions are avoided. This strategy is also conceptually the simplest, and it can be used for all instant response controlling processes. The disadvantage of this strategy is that invariants can be invalidated more easily, because pending exceptions are raised at all interaction points. This is not considered to be a great problem, because the violated external invariants can be restored in exception handlers. If pending exceptions are raised at all interaction points, it is preferable to ensure that internal invariants are valid at all interaction points.

No raising of pending exceptions in exception handlers

The raising of pending exceptions is unacceptable in an exception handler. This is due to the fact that an important aim of exception handlers is to restore (external) invariants that have been invalidated by an exceptional termination of a program unit. If one were allowed to raise pending exceptions in exception handlers, then a handler could be terminated with an exception before it had been able to restore the violated invariants. An example of such a situation has already been given in Figure 5.3.3a. Therefore, pending exceptions are not raised in exception handlers; they are kept pending. As a result, exception handlers should not contain delay

statements or blocking interaction points the blocking of which depends on the state of the controlled system.

The monitoring of constraints

The monitoring of constraints can take place in two conceptually different ways.
In the first way, constraints are monitored throughout the execution of a protected block.
In the second way, monitoring of constraints is restricted to explicit points during the execution of a protected block. If pending exceptions are raised at interaction points only, then monitoring of constraints can be restricted to interaction points, so that the constraints are not monitored between interaction points. In that case, there is no need for pending exceptions, because the required external exception can be raised at the time that the constraint violation is detected.

Note that, in both cases, the constraints of a protected operation are not only monitored during the execution of the operation itself, but also during the execution of all operations (or during the execution of interaction points in these operations) which are called from the protected operation, and all operations called by them etc. Monitoring will stop only when the protected operation is terminated (apart from the temporary stopping of the monitoring between interaction points in the second case).

The disadvantage of restricting the monitoring of constraint violations to interaction points is that the indication of constraint violations in message based systems can, in this way, not be done by means of non-blocking synchronous send primitives. This disadvantage is obviously not relevant for systems in which such send primitives are not available.

When a non-blocking synchronous send primitive is executed, the transfer of the message will only take place if a process exists which can immediately receive the message. If there is no such process, the non-blocking send primitive is terminated immediately, and the message will remain in the sending process. An example of such a send primitive is the broadcast primitive. This primitive sends copies of a message only to those processes that are ready to receive the message. It does not block.

The non-blocking synchronous send primitives are very useful for the indication of constraint violations (see Section 6.10.8). The use of these send primitives, however, conflicts with the strategy to monitor constraints at interaction points alone.
Such conflicts can occur when two (or more) processes are interacting with each other during the execution of a certain program region in each process. Such program regions will be referred to as synchronization sections. Examples of such situations are given in Figures 5.3.2a-c, and Figures 7.5.5 (CtrlTraverse >> stackToTruck) and 7.5.6 (CtrlTruck >> receiveStackFromTraverse). If, in such a case, one of the two processes (the violator) prematurely leaves its synchronization section due to an exception occurrence, this will be a constraint violation which should be indicated to the other process (the victim). This is best done by means of the non-blocking synchronous send primitive. In each of the processes a constraint monitor, which monitors the state of the synchronizing process, will be bound to the synchronization section. The constraint monitor of the victim must receive the message which indicates the constraint violation by the violator. If, however, the victim monitors its constraint at interaction points only, then the non-blocking synchronous send primitive could fail to actually send the message. This will occur if the send primitive is executed at a point of time at which the victim is not monitoring its constraint, which can be any point of time at which it is proceeding from one interaction point onto the next. In such a case, the constraint monitor will not detect the constraint violation, which will lead to deadlock.

To prevent such errors in the case that constraints are monitored at interaction points only, constraint violations could be indicated to all affected processes by means of asynchronous or blocking synchronous send primitives. The asynchronous interaction mechanism has an undesirable buffering function, however, which could lead to the signaling of a constraint violation, due to a buffered message, at a time that the constraint indicated by the buffered message is no longer violated. The blocking synchronous interaction mechanism is also unsuitable because it will cause the sender (violator) to be blocked until the message, indicating the constraint violation, is received by the constraint monitor of the victim. If, however, the operation of which a constraint was assumed to be violated would also happen to terminate prematurely due to an exception occurring at the same time as the exception occurrence in the violator, then the constraint monitors bound to the operation could already be disabled. In such a case, the message would not be received, causing the sender (violator) to remain blocked, which would lead to deadlock.

The conclusion is that constraints should be monitored throughout the execution of a protected block in systems in which the non-blocking send primitive is available for the indication of constraint violations. This is the approach taken in this chapter. In systems where such primitives are not available, such as systems based on asynchronous interaction mechanisms, constraint violations will not remain unnoticed between interaction points. Therefore, the monitoring of constraint violations at interaction points alone is a legitimate option for those systems. This strategy, however, will not be dealt with in greater detail, because the behaviour of such systems regarding constraint violations can be deduced relatively easily from the behaviour of systems where constraints are monitored throughout the execution of a protected block.

The exit point of a protected block

The fact that the exit point of a protected block is not an interaction point can lead to the necessity to check the status of certain constraint monitors bound to the protected block after normal termination of the block.
The exit point of a block is the point at which the block is terminated normally, that is not with an exception. So the exit point of a block is the point just before the block's end identifier (or other symbol or identifier which closes the block), or the point where a return statement causes termination of the block.

Constraints concern the environment of the process executing an operation, and interaction points are the only way to interact with the environment. Therefore, constraints can only change after the execution of an interaction point. So in theory, the exit point of a protected block should coincide with an interaction point. This is not practical, however. One of the reasons for this is that the exit point of a protected block can depend on the execution of conditional programming constructs, such as if then else statements.

If constraints are monitored at interaction points only, they will no longer be monitored after execution of the interaction point executed last in the protected block. Therefore, in this case, the protected block can be considered to end at the interaction point executed last.
If constraints are monitored throughout the execution of a protected block, however, monitoring of constraints will not stop after execution of the interaction point executed last in the protected block; it will stop exactly at

the exit point of the protected block. In the situation discussed below, this leads to the necessity to check the status of a constraint monitor bound to a protected block after normal termination of the block.

In message based systems, a constraint monitor could monitor its constraint by trying to receive a specific message. This is elaborated on in greater detail in Section 6.10, where the implementation of constraint monitors in Process Calculus is treated. The receipt of such a message will indicate a constraint violation, which will normally result in the raising of an exception. This could either be the monitor's exception itself or, in the case of discarding (see Section 6.5), some other exception. The received message which indicated the constraint violation can normally be retrieved from the constraint monitor in the exception handler used to catch the pending exception which was raised. If, however, the constraint violation were to take place *after* the last interaction point in the protected block, but *before* the exit point of the protected block, the receipt by the constraint monitor of the message which indicated the constraint violation would *not* result in an exception being raised. The reason for this is that an interaction point is no longer encountered in the protected block, causing the constraint monitor's pending exception to be discarded at the exit point of the protected block (see Section 6.5).
In the case that the contents of the message indicating the constraint violation are needed, the constraint monitor must be checked both in the exception handler used to catch the raised pending exception, and in the statements dynamically following the exit point of the protected block. If the last test is not programmed, then the message indicating the constraint violation can go unnoticed in the special case of a constraint violation after the interaction point executed last in the protected block.

A practical example of this situation is a constraint monitor monitoring commands sent by the operator by means of the MMI (man-machine interface). Suppose that the operator can send the commands *manual* and *reset* to the controlling processes. The receipt of the command *reset* will cause the controlling system to interrupt the current activities of the controlled machines and return them to a predefined reset state (see Section 7.4). The receipt of the command *manual* will also cause the controlling system to interrupt the current activities of the controlled machines, after which the controlling system will await commands from the operator to sequentially operate selected parts of the controlled machines.
When the controlling system is executing a reset command, it will be continuously synchronizing with the controlled machines in order to bring

them into the reset position. A constraint for this reset operation is that the operator does not send a manual command, because in that case resetting the machine would have to be stopped and the controlling processes would have to wait for new commands from the operator. Therefore, a constraint monitor which monitors the operator to see whether a manual command can be received is bound to the protected block enclosing the reset operation.
If the monitor receives a manual command, the monitor's pending exception will be raised at the execution of the next interaction point, after which the received manual command can be retrieved from the constraint monitor in the exception handler which caught the raised exception.
If, however, the manual command were to be sent just *after* the execution of the last interaction point in the reset operation but *before* the exit point of the protected block enclosing the reset operation, the manual command would be received by the constraint monitor, but the pending exception thus created would be discarded upon termination of the protected block; and so the reset operation would terminate normally. Therefore, it is necessary to check the status of the constraint monitor after normal termination of the protected reset operation, so that the manual command can be retrieved from the constraint monitor. If the constraint monitor has not received a message, then the check of the constraint monitor can be followed by a normal receive action in order to wait for the (manual or other) command from the operator.

If pending exceptions would also be raised at the exit point of a protected block, the block would always terminate with an exception if a pending exception had been created due to a violation of a constraint of the block.
The problem with this approach, however, is that it may lead to the violation of internal invariants. Consider the example given in Figure 6.3.1:

```
["Start of protected block"
self send: item to: 'out'
"End of protected block"].
numberOfItemsSent := numberOfItemsSent + 1.
```

Figure 6.3.1 *Violation of an internal invariant when a pending exception is raised at the exit point of a protected block*

If a pending exception is raised at the exit point of the protected block, the variable numberOfItemsSent will not be incremented. This would be a violation of the internal invariant specifying that numberOfItemsSent equals the number of items that have been successfully sent. This invariant cannot be restored in an exception handler, because the handler cannot determine

whether the exception has been raised before (replacing the interaction point) or after (at the exit point of the protected block) the item has been sent. Therefore pending exception are not raised at the exit point of a protected block.

In conclusion, the status of a constraint monitor should be checked after normal termination of the block to which it is bound if the contents of the message received by the monitor are not only required after the exceptional termination of the protected block, but also after its normal termination. This observation is based on continuous monitoring of constraints. If constraints are monitored at interaction points only, the status of constraint monitors need only be checked in exception handlers.

Summary

In synchronous message based systems, the constraints of an operation are monitored throughout the execution of the operation. The actual raising of a pending exception in a process is deferred until no exception handler is executing and the process starts the execution of an interaction point. The pending exception is raised, replacing the interaction point. (Please note that interaction points are defined to also include delay statements.)
The raising of a pending exception is not deferred when a pending exception is created during the time that a process is blocked in an interaction point and no exception handler is executing. The process will be unblocked immediately, and the pending exception will be raised, replacing the interaction point.

6.3.3 Raising pending exceptions in delayed response controlling systems

Deferring the raising of pending exceptions until interaction points can be unacceptable in delayed response controlling systems, due to the fact that the processing time between interaction points is not negligibly small in these systems. An example of such a system is a scheduler which takes a relatively long time to calculate a new schedule. During this calculation constraints could be violated, which should cause the scheduler to end its calculation. Such a constraint violation could, for example, be the arrival of a new batch (forcing recalculation of the schedule), or a command from the

operator to terminate the calculation of the schedule. A timely response to such a constraint violation can be achieved in two ways.

First, extra interaction points can be introduced into the scheduler, thereby reducing the time between interaction points. These interaction points can be used to test the status of the environment to see whether constraints have been violated. The drawback of this approach is that interaction points are used in this way to test constraints explicitly, which contradicts the idea of using constraint monitors. Alternatively, 'dummy' interaction points, of which the sole purpose is to make the raising of pending exceptions possible, could be introduced into the scheduler.

Second, an extra primitive could be introduced which allows pending exceptions, if any, to be raised at points in the program specified by the programmer. When this routine is called, a pending exception, if available, will be selected to be raised at the point of the call to the routine. If there is no pending exception, the call to the routine has no effect. The routine could be called raisePendingException. The use of this routine gives the programmer maximum flexibility in his/her control of the raising of pending exceptions at appropriate points, without the need to introduce dummy interactions. It is a kind of polling, which is only necessary in the special case of delayed response controlling systems. Note, however, that the monitoring of the constraints by the enabled constraint monitors always takes place, independently of the polling calls on raisePendingException.

The first option keeps the mechanism conceptually simple, because pending exceptions are only raised at interaction points. The introduction of dummy interaction points or interaction points to test the constraints explicitly is, however, confusing. Therefore, it is preferred to add the raisePendingException routine as a primitive to the mechanism for raising pending exceptions, which has been treated in the previous section.

6.3.4 Overriding the default strategy for raising pending exceptions

The strategy for raising pending exceptions which has been treated in Sections 6.3.2 and 6.3.3 is not satisfactory in all cases. In some cases, it is desirable to be able to override the default strategy in such a way that pending exceptions are allowed to be raised at certain interaction points in exception handlers.

This is necessary when the retry strategy as treated in Sections 7.3 and 7.6 is used. The retry strategy is only possible in languages that support the retry response from an exception handler. It makes use of a blocking interaction in an exception handler, which is used to wait for the response from the operator indicating that the interrupted control action can be restarted. Such a blocking interaction, however, should be interruptible when a constraint is violated and a pending exception is created. Otherwise, the process would remain blocked in the exception handler until the operator response was received. If pending exceptions are never allowed to be raised in exception handlers, then the interaction used to wait for the response from the operator to continue cannot reside in the exception handler. In such a case, the retry response from an exception handler cannot be used. The return response will need to be used instead and the interaction used to wait for the response from the operator must be part of a while do statement, which can cause the control action, including the exception handler, to be re-executed. This will lead to less clear and less elegant code which cannot take advantage of the retry response from exception handlers, but it is not a problem for languages that do not support the retry response anyway.

Two main approaches can be distinguished for the specification of constructs that allow pending exceptions to be raised at certain interaction points in exception handlers. Firstly, this overriding of the default strategy can be explicitly specified for individual interaction points, because these are the only relevant places where the default strategy needs to be overridden. Secondly, a region of the program (a block, for instance), can be defined, where the default strategy is overridden. Such a block will be referred to as an 'override block' for easy reference.

The latter approach has two distinct disadvantages. Firstly, the designation of a block where pending exceptions are allowed to be raised in exception handlers does not make it clear that this is only relevant to interaction points contained in the block. Secondly, and even more important, this approach makes it conceptually more difficult to determine whether pending exceptions are allowed for a specific interaction point or not. This is due to the fact that exception handlers and override blocks can be mutually nested. If, for instance, an exception handler is nested in an 'override block', in which pending exceptions are allowed to be raised, then the exception handler would probably need to override the override block in such a way that pending exceptions in the nested handler are not allowed to be raised.

An advantage of the latter approach is its syntactic simplicity. The conceptual complexity is, however, considered to be more important. Therefore, it is preferred that the default strategy can only be overridden for explicitly and separately indicated interaction points.

The actual decision as to whether to allow the default strategy to be overridden depends on a number of aspects. Added complexity due to the added primitives should outweigh the increased programming power of the language. The possible presence of a retry response in the language is another aspect influencing this decision. In Process Calculus, special interaction primitives are available for use in exception handlers, so that pending exceptions can be raised during the execution of these primitives.

6.4 Handling exceptions resulting from constraint violations

When a constraint violation is signaled by a constraint monitor, the resulting pending exception will be raised somewhere in the protected block of which the constraint was violated. The handling of such exceptions is application-dependent, but there is one important common aspect. A constraint violation will make it impossible for the protected block to achieve its goal. Therefore, the exception that is signaled as a result of the constraint violation should result in the termination of the block with an exception. A result of this requirement is that handlers which are activated from within a protected block, that is during the execution of the protected block, must always propagate exceptions that indicate a violation of the protected block's constraints.

The observations discussed above can be summarized in the following obligation for the programmer regarding exception handling in the presence of exceptions from constraint monitors.

An exception handler that catches a constraint monitor's exception while the constraint monitor's block is not yet terminated must terminate by propagating the exception.

Programs should be developed in such a way that they adhere to this principle while at the same time making use of the concepts of abstraction and modularity. The use of these concepts implies that subprograms of a low hierarchical level need not know anything of the subprograms of a high

hierarchical level. Furthermore, subprograms should not need to know the inner details of the subprograms that are called by them.

In order to enforce the constraint monitor's exception handling principle (explained above), while at the same time maintaining abstraction and modularity, exceptions that are handled locally (and are thus not propagated) should be defined locally. This should be done in such a way that handlers bound to such locally-defined exceptions cannot catch exceptions from constraint monitors that are defined at a higher level.

Unfortunately, the local definition of all locally handled exceptions in a subprogram can be impractical: if many subprograms share the same constraints, all locally-defined exceptions and constraint monitors will need to be duplicated.

A lot of code duplication can be avoided by defining constraint monitors in one place and using them in several subprograms. In this case, their exceptions must be used in all of these subprograms, and must therefore be defined globally for those subprograms. The designer of the program must take care that the exceptions which are handled locally (and are not propagated to higher levels) are not used as exceptions for constraint monitors defined at higher levels.

6.5 Discarding pending exceptions

6.5.1 Dealing with multiple pending exceptions

When the state of a process is such that a pending exception can be raised, there could be more than one pending exception. Such a situation can arise if multiple constraint monitors are enabled simultaneously during the execution of a process. There are two reasons for this. Firstly, more than one constraint monitor can be bound to a block, so that each constraint monitor monitors a (sub-)constraint. Secondly, more than one block can be active at the same time, because a block can invoke other routines and other blocks. Therefore, when a process is not blocked and executing, several constraints can be violated by other processes, possibly leading to the signaling of constraint violations by more than one constraint monitor and therefore to multiple pending exceptions in the process.

When the state of a process is such that a pending exception can be raised, and there is just one pending exception, it will be raised. When there is more than one pending exception, two problems have to be resolved: which one of the pending exceptions will be raised, and what happens to the other pending exceptions. The question of which one of the pending exceptions will be raised will be treated in Section 6.6. Section 6.5 makes clear how the other pending exceptions are dealt with.

Upon termination of the protected block, the pending exceptions which belong to the protected block's constraint monitors, and which are not selected for raising, are discarded. The result of this is that, after termination of a protected block, either normally or with an exception, there will be no more pending exceptions belonging to the protected block's constraint monitors.

When a constraint monitor's pending exception is discarded due to the disabling of the constraint monitor when a protected block is terminated with an exception, the discarded pending exception is said to be **discarded** by the other exception. The other exception is referred to as the **discarding** exception.

6.5.2 The argument for discarding pending exceptions

The aim of the pending exception of a constraint monitor is to signal the exception occurrence and to cause termination of the protected block with an exception. If a pending exception is discarded by another exception, the protected block is already terminated with an exception. Therefore, the existence of the discarded exception is no longer needed to cause termination of the block with an exception. If, on the other hand, the pending exception is discarded while the protected block terminates normally, then the pending exception must have been created after the last interaction point of the protected block, otherwise it would already have been raised. The possibility of this situation is due to the fact that the exit point of a protected block does not coincide with an interaction point. In this case, the pending exception can be discarded after termination of the protected block, because the violation of the constraints of a protected block after the last executed interaction point is not relevant. From a conceptual point of view, the constraints of the protected block are only valid until the block's last interaction point. This has been explained in the subsection of Section 6.3.2: 'The exit point of a protected block'.

The only problem that remains is that a single exception cannot convey the information associated with two exceptions, and the handling of the discarding exception can be different from the handling of the discarded exception. It is, however, impossible to signal two exceptions at the same time, so a choice must be made. It is up to the programmer to take account of the possibility of discarding. This will usually imply that the handler actions for the discarding exception include all of the handler actions for the discarded exception. This is usually done by using 'any handlers' which catch all exceptions. The state of the relevant constraint monitors can be checked in the handler in order to determine if they have signaled a constraint violation. If the pending exception of a constraint monitor is not allowed to be discarded, then that constraint monitor should be placed in a separate process.

The reason for discarding the pending exception is found if one analizes the situation that could occur if the pending exception were not discarded, but retained for signaling at a later time. The pending exception would thus be signaled when the process encountered an interaction point, or in another state where pending exceptions can be raised. Such a state could occur at a point where the 'discarding' exception would already have been handled. After execution of the handler, the process could have been returned to a state where the goals of all active operations could be achieved. This would lead to a situation where there would be no exception occurrence, but there would still be a pending exception. This is intolerable since exceptions may only be raised in the case of an exception occurrence.

6.6 Selecting a pending exception for raising

When a pending exception can be raised and there is more than one pending exception available then a choice has to be made. Several selection criteria can be used:

- **Undefined**
  When the selection criterion is undefined, the user program may not rely on a specific algorithm. This is the simplest but also the least useful option.

- **User-assignable priorities**
  A selection based upon user-assignable priorities has the disadvantage of greater complexity, because the user would have to specify the priorities. Furthermore, if two constraint monitors are assigned the same priority, the same problem remains as to which one of their pending exceptions should be selected.

- **Constraint monitor's signaling time**
  Basing a selection criterion on the constraint monitor's signaling time is usually not what is desired by the programmer. By the monitor's signaling time, we mean the point of time at which the monitor signals the violation of its constraint. This is also the point of time at which the pending exception is created. The signaling time of a constraint monitor is determined by the processes in the environment of the process using the constraint monitor. So it is beyond the control of the programmer of the process. Usually, however, the programmer desires some control over the constraint monitor which will be selected. Consider, for example, the situation in a wafer processing unit where a real-time scheduler is used to determine a schedule. If a new batch is introduced into the system during the calculation of the schedule, then the calculation will need to be interrupted to take the new batch into account. To achieve this, a constraint monitor is used which monitors the arrival of new batches.
  Suppose that, either after or before the arrival of a new batch, an order has arrived to stop scheduling because the system needs to be reset. The arrival of such an order is also monitored by a constraint monitor. Therefore, this situation could lead to two pending exceptions. Clearly it is useless to restart calculating the schedule, taking account of the new batch, when the system has to be reset anyway. So the constraint monitor which monitors the reset command in this case has a higher priority, which is independent of the order of the signaling times of the two constraint monitors.

- **Constraint monitor's enabling time: last enabled or first enabled**
  If the first enabled option is chosen, then the constraint monitors that are enabled at a high (or outermost) level, so that they will be active in all operations called from that level, will have a high priority. This is due to the fact that the lower (or innermost, or deeply nested) levels are called by the higher levels, so that the constraint monitors enabled in the lower levels will be enabled later. Therefore, pending exceptions from constraint monitors which are enabled at a low level will be discarded by

exceptions from constraint monitors which are enabled at a higher level, which is the desired behaviour.
In the previous example, for instance, the constraint monitor which monitors the reset command will be bound to a block at a high level so that the constraint monitor will be active in all operations called from that level. The constraint monitor which monitors the arrival of a new batch will be enabled at a lower level, when the scheduling is actually started. This leads to the desired discarding of the 'new batch' constraint monitor's exception by the 'reset' constraint monitor's exception when their exceptions are both pending. If this kind of discarding needed to be detected, the constraint monitors bound to a block could be checked in the handler bound to that block to see if they had just signaled a constraint violation.

If the last enabled option is chosen, fewer discards will occur, since the pending exceptions of constraint monitors defined at a low level will be given precedence over the pending exceptions of the constraint monitors defined at a higher level. The exceptions from the constraint monitors from the higher level will remain pending, however, and will be raised after the handling of the exception from the lower level's constraint monitors, at a time when the process enters a state where the pending exceptions are checked again. The higher level constraint monitors would only be disabled, causing their pending exceptions (if any) to be discarded, in the case that the exception from the lower level's constraint monitor is propagated up to the higher level.

The first enabled option is preferred because the constraint monitors enabled at higher (or outermost) levels should have priority over the constraint monitors enabled at lower (or innermost) levels. So this form of discarding is actually desirable. This has been demonstrated by the example of the scheduler.

The choice of the first enabled selection criterion has an effect on the treatment of pending exceptions. It implies that it is no longer necessary to retain all pending exceptions of a process. Instead, it is sufficient to retain only one pending exception. Any time that a second pending exception is created, the selection criterion is used to determine which of the two has the highest priority. This one is retained, and the other one can be discarded immediately.

6.7 The nesting of blocks with the same constraints

Different operations can have the same constraints, as shown in Figure 5.1.1. This can lead to a situation where two blocks that have the same constraints are (dynamically) nested.

These constraints could be specified with separately defined constraint monitors. If the constraint is violated in such a case, both constraint monitors would signal this violation, creating two pending exceptions. In this case, the pending exception from the constraint monitor that was enabled first, and thus bound to the outermost block, will discard the other constraint monitor's pending exception.

If the constraints are specified by binding the same constraint monitor to both blocks, the desired effect of such a situation is not directly evident.
Firstly, the question arises if two pending exceptions should be created in the case of a constraint violation, or just one. The simplest approach is to create only one exception, belonging to the constraint monitor bound to the outermost block. The pending exception from the other constraint monitor would be discarded anyway, so its creation would serve no purpose. Note that if the pending exception would be handled by the handler bound to the innermost block (the block with the deepest nesting), then the exception would have to be propagated by the handler according to the programmer's obligation specified in Section 6.4.
Secondly, it can no longer be stated that the constraint monitor is disabled after a protected block to which it is bound is terminated. Instead, the entering or exiting of a block to which a constraint monitor is bound which was already enabled before the block was entered has no effect on the state of the constraint monitor or it's pending exception.
Therefore, binding a constraint monitor to a block which is invoked by another block to which the same constraint monitor is bound is essentially a non-operation.

An example of the nesting of blocks with the same constraints in Process Calculus is shown in Section 6.10.7.

6.8 The evaluation of constraint functions

The monitoring of constraints can be regarded as a continuous evaluation of the associated boolean constraint function. The constraint is violated when

this function returns false. The evaluation of the constraint function is not necessarily done by the constraint monitor alone.

In the case of a violation of an active constraint of a controlling process by a process in the controlled system, the constraints will refer to the state of the controlled system. The controlled system itself does not know when a constraint is violated, and therefore cannot evaluate constraint functions that are associated with constraint monitors in the controlling system. All it can do is to make its state available to the controlling processes. Constraint monitors in the controlling processes will monitor the relevant parts of the state of the controlled system. However, depending on how constraint monitors are integrated in a language, additional processes can be necessary for the monitoring of complex constraints, such as constraints which are defined with a combination of logical operators.
In Process Calculus, for instance, a constraint monitor can be defined to monitor a specific part of the state of the controlled system. It does this by being ready to receive a message by means of an interaction, which message is specific for the part of the controlled system's state which can be a constraint violation. The boolean constraint function which is evaluated in this way is a simple comparison of the relevant part of the state of the controlled system with a constant value. An and-function of such constraints can be realized by binding different constraint monitors to the same operation.

In the case of constraint violations between controlling processes, the constraints usually refer to a correct synchronization between the processes. When two processes are correctly synchronized, any of the two processes can break out of this synchronization, and by doing so will violate an active constraint of the other process. The reason for breaking out of the synchronization is usually an exception occurrence.
The two controlling processes will be called the violator and the victim, according to the terminology of Section 5.3.3. The violator will be aware of violating the constraint when it has to break out of the synchronization due to an exception occurrence, and it can thus signal the constraint violation to the victim. In this case, the violator has implicitly evaluated the constraint function, and it will signal the true-to-false transition of the constraint function. In this way, the constraint monitors of the victim can be kept simple, because they need not monitor the different states of the violating processes. A single constraint monitor is sufficient to register only the indication of the true-to-false transition of the constraint function, which is

signaled to it by the violator process. In Process Calculus, this signaling can be done by means of a broadcast send primitive (see Section 6.10.8).
The violator need not be absolutely sure that it has actually violated one of victim's constraints. Suppose, for instance, that an exception had already been raised in the victim, independently of the constraint violation by the violator. In that case, victim's constraint monitor which monitored the constraint would already be disabled, and the appropriate constraint function would no longer be defined. So, in order to achieve good modularity and low coupling between the processes, the violator signals an *assumed* constraint violation to the victim. It is up to the victim's enabled constraint monitors to treat the assumed constraint violation as an actual constraint violation.

6.9 Conflicts between the resume response and constraint monitors

The conflicts between the resume response and constraint monitors are similar to the conflicts between the resume response and critical sections, as discussed in Section 4.6.4.

The conflicts arise when an exception occurrence is detected in a part of a program protected by constraint monitors. When the corresponding exception is raised and the protected block is terminated with an exception, the block's constraint monitors could either be disabled or remain enabled.

If the constraint monitors are disabled and if thereafter the resume response is chosen in a handler, then execution will proceed from the point where the exception was originally raised, i.e. in the protected block. So execution is continued in the protected block, but all the protected block's constraint monitors will still be disabled. Therefore, the constraint monitors should be re-enabled before the resume response is chosen. The exception handling mechanism would become considerably more complicated if this were to be done automatically. If it is not done automatically, it can only be done with unstructured use of constraint monitors, because the constraint monitors will have to be re-enabled in an exception handler before issuing a resume response. This also implies precise knowledge of the state of all constraint monitors that may have been disabled.

Note that, in the case of a resumption model, it is strictly speaking not correct to use the phrase 'termination of a block with an exception', because

the block only terminates when either a return or a retry response is chosen, or another exception is raised. Before this is done, execution of the block can still be resumed by the resume response.

If the constraint monitors remain enabled, then the disabling of constraint monitors is postponed until the exception (which caused the block to terminate) is completely handled and the resume response can no longer cause resumption of the block. In such a case, pending exceptions may never be raised in exception handlers. If the raising of pending exceptions in handlers were possible, then, prior to giving a resume or other response, a pending exception could be raised which should have been discarded upon termination of its associated protected block.
Therefore, in systems where pending exceptions can be raised in exception handlers by overriding the default strategy treated in Section 6.3.2, constraint monitors must be disabled immediately after termination of their protected block with an exception.

Conflicts between the resume response and constraint monitors can only be prevented in systems where pending exceptions cannot be raised in exception handlers. In these systems, the disabling of constraint monitors of which a protected block has 'terminated' with an exception should be postponed until the exception (which caused the block to terminate) is completely handled and the resume response can no longer cause resumption of the block.

It is concluded that the conflicts between the resume response and constraint monitors in systems where pending exceptions can be raised in exception handlers are so severe that resume responses should not be used in such systems. If pending exceptions can never be raised in exception handlers, there need not be any conflicts between the resume response and constraint monitors.

6.10 The implementation of constraint monitors in Process Calculus

6.10.1 Monitoring constraints by executing receive actions

When a constraint monitor is enabled, it will try to receive an object by executing a receive action. This is the way that a constraint is monitored in Process Calculus models. The constraint is assumed to be violated when the

constraint monitor receives the object. As long as the desired object is not received, the process which enabled the constraint monitor is not blocked but can continue normally. The constraint is constantly monitored, even though the process is proceeding with other actions. The monitoring of the constraint is stopped when either the constraint monitor's receive action succeeds or when the block protected by the constraint monitor terminates.

6.10.2 The definition of constraint monitors

A constraint monitor is defined separately and can consequently be bound to different blocks. A constraint monitor is defined by sending it an initialization message. Such a message could be ConstraintMonitor class >> for: aProcessor receive: anObject from: portName thenRaise: anException. It is assumed that the monitor's constraint is violated when the object anObject can be received from the port named portName on the processor aProcessor. When the constraint monitor is enabled, it will try to receive anObject from the port named portName on the processor aProcessor. When this succeeds, the constraint monitor will signal the constraint violation by making anException pending in aProcessor and stop monitoring its constraint.

In reality, a slightly different method is chosen: ConstraintMonitor class >> for: aProcessor receive: anObject from: portName then: exceptionBlock. Hereby the constraint monitor is initialized with the block exceptionBlock instead of anException as in the previous method. In this case, the constraint monitor will signal the violation of its constraint by making its exceptionBlock pending in aProcessor, thereby creating a 'pending exception block' instead of a pending exception. When an interaction is executed in the protected block of which a constraint has been violated, the constraint monitor's pending exception block is executed. The pending exception block must terminate with an exception when it is executed. If it does not terminate with an exception, the supporting system will raise an exception after the termination of exceptionBlock, which exception is an indication of an illegal exception block. There are two reasons for this slightly different method. Firstly, it is more in the style of the task language of Process Calculus, where many methods follow the convention of executing a specified thenBlock when a receive action succeeds, as for example in receive: anObject from: portName within: interval then: thenBlock ifTimedOut: timeOutBlock. Secondly, this single initialization method is sufficient in all cases. Exceptions can be raised in many different ways in Smalltalk, for example with or without error messages and other arguments. All these

different ways can be accommodated by using the exception block. If the constraint monitor were initialized with the exception itself, different initialization methods would be needed for the different ways of raising the constraint monitor's exception.

The convention of only raising an exception in the exception block will be followed in this thesis, so that the creation of a pending exception block and the creation of a pending exception amount to the same thing. This is because raising a pending exception and evaluating the exception block both result in the raising of the constraint monitor's exception. To keep the explanation of the concepts as simple as possible, we will choose the terminology of creating and raising pending exceptions, rather than creating and evaluating pending exception blocks.

An example of the definition of a constraint monitor is shown in Figure 6.10.1.

```
ForkLifterCtrl >> initializeTasks
    emergencyMonitor := ConstraintMonitor
                        for: self
                        receive: true
                        from: 'i-emergency'
                        then: [KillSignal raise]
```

Figure 6.10.1 *Definition of a constraint monitor in Process Calculus.*

The KillSignal is a global signal known to all processors. It is raised in the case of external exception occurrences which cannot be handled locally, for example by locally restarting an action.

6.10.3 Binding constraint monitors to blocks

A constraint monitor can be bound to a block by sending it the message protect: with the block to be protected as argument. In Process Calculus, binding is dynamic. It is effected when the protect: block message is sent to a constraint monitor. This message will also cause the invocation of the protected block. Figure 6.10.2 shows the method ForkLifterCtrl >> forkUp once again. This time a constraint monitor is used to monitor the emergency button.

```
ForkLifterCtrl >> forkUp
    emergencyMonitor protect:
        [self putOn: 'o-forklifter-up'.
        self putOn: 'o-forklifter-power'.
        self
            receive: true
            from: 'i-forkLifter-isUp'
            within: 6 seconds
            ifTimedOut: [KillSignal raise].
        self putOff: 'o-forklifter-power']
```

Figure 6.10.2 *The use of a constraint monitor to protect a block.*

If the emergency button is pressed while the process is executing the protected block, the emergencyMonitor will signal this constraint violation; and the exception KillSignal, which was used to initialize the constraint monitor, will be raised, replacing the currently executing interaction.

6.10.4 The use of constraint monitors together with exception handlers

An exception will be raised when a constraint monitor detects a violation of a constraint. Consequently, there must be a handler for that exception. Finalization obligations can be performed in the handler in order to terminate the operation to which the handler is bound in a safe and consistent state. If necessary, the exception can then be propagated. In Figure 6.10.3, the method forkUp is extended with an exception handler.

The signal AnySignal is a global signal collection that is used to catch all exceptions. AnySignal contains the signal Object errorSignal amongst others. The interested reader is referred to [ParcPlace, 1989] for more information on signal collections.

```
ForkLifterCtrl >> forkUp
    AnySignal
        handle:
            [:exception |
            self putOff: 'o-forklifter-power'.
            exception reject]
        do:
            [emergencyMonitor protect:
                [self putOn: 'o-forklifter-up'.
                self putOn: 'o-forklifter-power'.
                self
                    receive: true
                    from: 'i-forkLifter-isUp'
                    within: 6 seconds
                    ifTimedOut: [KillSignal raise].
                self putOff: 'o-forklifter-power']]
```

Figure 6.10.3 *The use of a constraint monitor together with an exception handler.*

This use of a constraint monitor together with an exception handler is typical of the use of constraint monitors, because the exception raised by the constraint monitor is usually handled locally.

6.10.5 Specifying multiple constraint monitors

An operation can have several constraints. Often it is not possible to monitor different constraints with a single constraint monitor. Therefore, it should also be possible to bind several constraint monitors to the same block. This can be done by nesting protected blocks in the way shown in Figure 6.10.4.

```
ForkLifterCtrl >> forkUp
    emergencyMonitor protect:
        [operatingSwitchMonitor protect:
            [collisionMonitor protect:
                [self putOn: 'o-forkLifter-up'.
                self putOn: 'o-forkLifter-power'.
                self receive: true from: 'i-forkLifter-up'
                self putOff: 'o-forkLifter-power']]]
```

Figure 6.10.4 *Binding multiple constraint monitors to a block using nesting.*

The class ConstraintMonitorCollection is introduced in Process Calculus in order to avoid the syntactic ugliness of nested constraint monitors. This is a subclass of OrderedCollection so it inherits all OrderedCollection's messages and also the messages from the class Collection. In the class ConstraintMonitorCollection itself, only one method is defined namely protect:. In order to bind multiple constraint monitors to a single block, the constraint monitors can be added to a collection of constraint monitors which can then be bound to the block using the method protect:, in the same way as this method is used for a single constraint monitor. This construction is semantically identical to the construction where the constraint monitors are nested in the same order as they are added to the collection of constraint monitors.
In the method ForkLifterCtrl >> initializeTasks shown in Figure 6.10.5, a new ConstraintMonitorCollection is made by means of the method Collection >> with:with:with:. The collection is filled with the three constraint monitors that are used as arguments to the method. The method forkUp is defined using the collection of constraint monitors. The variable constraintMonitors is an instance variable of ForkLifterCtrl. The methods forkUp shown in Figures 6.10.4 and 6.10.5 are semantically identical.

```
ForkLifterCtrl >> initializeTasks
    constraintMonitors := ConstraintMonitorCollection
                with: emergencyMonitor
                with: operatingSwitchMonitor
                with: collisionMonitor

ForkLifterCtrl >> forkUp
    constraintMonitors protect:
        [self putOn: 'o-forkLifter-up'.
        self putOn: 'o-forkLifter-power'.
        self receive: true from: 'i-forkLifter-up'
        self putOff: 'o-forkLifter-power']
```

Figure 6.10.5 *Binding multiple constraint monitors to a block using a preinitialized collection of constraint monitors.*

6.10.6 Some additional functionality of constraint monitors

It can be useful to be able to check what object has been received by a constraint monitor, such as in the case that the pending exception from a constraint monitor can be discarded by other exceptions. By checking whether an object has been received by the constraint monitor, one can

determine whether a constraint monitor has signaled a constraint violation, even if its pending exception has been discarded. The object received can also be used itself if, for example, the operator sends a command to stop processing. Such a command could be received by a constraint monitor.

To make this possible, constraint monitors retain the object which they have received, and which indicates a violation of the monitored constraint. The object contained in the constraint monitor can be read at any time. It should also be possible to clear the object after having read it. To realize this functionality in Process Calculus, there are two messages that can be sent to constraint monitors:

- The method **ConstraintMonitor >> item** returns the item received by the constraint monitor. It returns nil if no message has been received since the constraint monitor was last cleared.
- The method **ConstraintMonitor >> clearItem** returns the value of the item received by the constraint monitor. It returns nil if no message has been received since the constraint monitor was last cleared. It ends by clearing the constraint monitor. The constraint monitor is also cleared when it is enabled.

6.10.7 Binding the same constraint monitor to nested blocks

If the same constraint monitor is bound to (dynamically) nested blocks, the binding of the constraint monitor to the innermost or deepest nested block can be regarded as a non-operation. This is illustrated in Figure 6.10.6 where the methods method1 and method2 are functionally equivalent.

```
TestProcessor >> method1
    constraintMonitor protect:
        [self someStartMethod.
        constraintMonitor protect: ["block2"].
        self someEndMethod]

TestProcessor >> method2
    "Is equivalent to method1"
    constraintMonitor protect:
        [self someStartMethod.
        ["block2"] value.
        self someEndMethod]
```

Figure 6.10.6 *Binding the same constraint monitor to nested blocks.*

6.10.8 The desired send primitives to signal constraint violations

The functionality of constraint monitors has been treated in previous sections. The constraint monitor's constraint is monitored by the execution of a receive action. When the receive action succeeds, the constraint monitor's constraint is supposed to be violated. For an interaction to take place, both a send action and a receive action are needed. This section will focus on the kind of send actions needed to support the detection of constraint violations.

Strictly speaking, any send action can be used, because the receive action executed by the constraint monitor neither assumes nor requires a specific send action. In practice, however, only two different send primitives are needed to support the detection of constraint violations.

The first send primitive needed is the method Bubble >> send:continuousTo:. This method is used to indicate a state, such as an emergency button which is pressed. It is used mainly for interfacing the sensors in the controlled system to the controlling system by means of a driver processor as explained in Section 2.2.3.

The second send primitive needed is the method Bubble >> broadcast: object to: portName, which will be fully explained at the end of this section. In some cases, the method Bubble >> send: object immediateTo: portName then: thenBlock else: elseBlock is sufficient. This method tries to send object to the port named portName. If this succeeds immediately then thenBlock is executed; if not elseBlock, so the method never blocks. For the indication of constraint violations the thenBlock and elseBlock are generally not needed, because no feedback information about the success or failure of the send action is required. The only aim of the send action is to indicate a constraint violation. It is up to the other processes to act on this if needed. This method can be used to indicate that the normal sequence of synchronization actions between two or more controlling processes is disrupted.

The synchronous send actions, which may be blocking, are not needed to indicate constraint violations, because it is never necessary to wait for another process to be able to receive the object indicating the constraint violation. This is due to the nature of constraint monitors: once they are enabled, they are always ready to receive, independently of the actions performed by the process that enabled the constraint monitor.

The asynchronous send actions are not needed, either. These interaction mechanisms have an undesirable buffering function which could lead to the signaling of a constraint violation, due to a buffered object, at a time that the constraint indicated by the object is no longer violated.

In many cases, the number of processors that need to be informed of a constraint violation exceeds one. If this is the case, and the method send:immediateTo:then:else: were used, a separate object would need to be sent to every processor concerned. This would be undesirable, since specifically sending an object to every processor concerned would lead to bad modularity. Therefore, the extra send primitive Bubble >> broadcast: object to: portName is introduced for the notification of constraint violations. This method tries to send copies of object to all the processors connected through an interaction path to the port named portName. If a certain send action cannot take place immediately, an attempt is made to send a copy of object to the next processor, so this method cannot block. It is functionally equivalent to separately trying to send the specified object to all processors concerned by means of the method send:immediateTo:then:else:, using an empty thenBlock and an empty elseBlock. The presence of the method broadcast:to: implies that the method send:immediateTo:then:else: is no longer needed for the indication of constraint violations.

Chapter 7
The specification of controlling systems illustrated by a case

Some examples of the use of the newly developed mechanism have already been given. This section gives some illustrative parts of a complete controlling system, with an emphasis on the way constraint monitors are used to handle exceptions. First, the desired functionality of the control system is explicitly stated. The controlling system is then modeled according to those requirements. A simple but effective strategy is developed for error recovery with the aid of exceptions and constraint monitors. This strategy is referred to as the retry strategy. It is demonstrated both in a single-process and in a multi-process environment.
The reader is referred to the appendices for more information about Smalltalk and the methods used in this chapter.

7.1 Requirements regarding the functionality of control systems

This section discusses some important requirements of exception handling regarding the interaction of the controlling and controlled system. Clearly these requirements are not applicable to all kinds of system since the requirements are usually a compromise between the functionality desired and the cost of implementing them. The controlling system of the transporter, as treated in this chapter, is modeled according to the requirements given in this section.

1. Exceptions should be handled as locally and as efficiently as possible. This means that as small as possible a part of the system should be affected by an exception. Consider, for example, the assembly of several parts. A failure in the assembly of a single part should not lead to the rejection of the complete subassembly, but rather should be correctable by the operator or automatically, whereafter assembly can continue.

2. Proper damage confinement strategies require that the effect of an exception should be minimized. This can mean that, as a result of an exception in one machine part, many other parts must be put in a safe state in order to prevent more errors from occurring.

3. If operator assistance is required, errors should be reported to the operator. If the operator would enter the machine in order to correct the error, the machine should be kept in a safe state and not suddenly start moving. The interrupted part of the production process may only continue when ordered to do so by the operator.

4. The controlling system should remain in a consistent state. Constraint violations should result in the raising of exceptions. This is especially important in parallel systems, where an exception in a process can cause a process to break off its current activities. If it was synchronizing with other processes, these processes could remain waiting for a synchronization which will no longer take place.

5. The operator should in principle be able to interrupt the production process at any time. For example to stop and reset the production process due to an error which was not detected by the controlling system.

7.2 Additional exception handling methods for controlling systems

The exception handling methods given in the previous chapter are of a general nature. The specific requirements for the programming of controlling systems make it advantageous to develop additional exception handling methods.
The first additional method is the method Bubble >> handle:do: shown in Figure 7.2.1. It is based on the method Signal >> handle:do:.

Due to the great variety of possible errors in the controlled system, the ways of recovering from these errors can be very system specific. Yet the stage of damage confinement can often be executed using the same concepts. Machine parts that may be affected by the error must be brought to a safe state. This should be done regardless of the type of exception. The machine should be brought to a safe state, even when no errors have occurred, but the production process is simply stopped by the operator. This can be done by

```
Bubble >> handle: exceptionHandler do: doBlock
    "The exceptionHandler will catch all exceptions from the doBlock.
    Before execution of exceptionHandler, an error message will be issued for
    the exception.
    After execution of exceptionHandler, the handled exception will be
    propagated (ex reject), unless another response is specified in
    exceptionHandler."
    AnySignal
        handle:
            [:ex |
            self errorMessageFor: ex.
            handler value: ex.
            ex reject]
        do: doBlock
```

Figure 7.2.1 *A new handle:do: method.*

using an exception handler that will catch all exceptions. Therefore, the handler set up by the method is a handler which catches all exceptions.

All errors should be presented to a man-machine interface (MMI) of the controlling system. This is done in the exception handler by means of the message self errorMessageFor: ex. The convention adopted in this thesis is that all controlling processors have two ports for communication with the MMI: the ports mmi-in and mmi-out. In the Smalltalk system, error messages reside in exception objects which are made available to exception handlers as argument. Therefore, the error message belonging to an exception object can be extracted from that object by sending it the message errorString. This feature enables error messages of exceptions raised by the user program code itself and by the system code to be treated in the same way. The error message is retracted from the exception object in the exception handler and sent to the MMI. To prevent the same error message from being sent to the MMI in different handlers that sequentially handle the same exception, the last exception for which an error message has been sent to the MMI is recorded in an instance variable which is available in every processor. An error message is only sent to the MMI if the value recorded in this instance variable is not equal to the exception which is being handled.

As has already been mentioned in Section 4.6.2, the default response of a handler should be the propagate response, which is also the response which occurs most frequently in controlling systems. The exception handler elaborated in Figure 7.2.1 terminates with ex reject. In this way, the exception caught by exceptionHandler, which is set up with the method

Bubble >> handle:do:, is propagated when no response is issued in exceptionHandler. In this way, the functionality of the default propagate response is created for this handler.

The second additional method is Bubble >> handle:constraintMonitors:do: shown in Figure 7.2.2. It is a combination of the method Bubble >> handle:do: and the method protect: in the class ConstraintMonitor (and ConstraintMonitorCollection). Its use leads to more easily readable code because the number of nested blocks is reduced. The only difference from the method Bubble >> handle:do: is that the doBlock is protected by the constraintMonitor argument monitorOrMonitors.

```
Bubble >> handle: exceptionHandler constraintMonitors:
monitorOrMonitors do: doBlock
    "The exceptionHandler will catch all exceptions from the doBlock.
    Before execution of exceptionHandler, an error message will be issued for
    the exception.
    After execution of exceptionHandler, the handled exception will be
    propagated (ex reject), unless another response is specified in
    exceptionHandler.
    During execution of the doBlock monitorOrMonitors will be enabled."
    AnySignal
        handle:
            [:ex |
            self errorMessageFor: ex.
            handler value: ex.
            ex reject]
        do: [monitorOrMonitors protect: doBlock]
```

Figure 7.2.2 *A method which binds an exception handler and constraint monitors to a block.*

External exceptions can be represented with the KillSignal and RetrySignal. They are global signals which are known to all processors. By convention, they are raised in a process in response to exception occurrences caused by the environment of the process. These exception occurrences can either be caused by the state of the controlled system, by other controlling processors, or by the operators. The KillSignal should be raised when local error recovery is not possible. The RetrySignal is raised when local error recovery using the retry strategy is possible in principle. The retry strategy is treated in the next section. Exceptions due to programming errors should be signaled by raising Object errorSignal or other specific signals.

7.3 The retry strategy in a sequential process

Many different strategies can be used in the recovery process. There is, however, one simple strategy that can be used to recover from many errors in a simple and efficient way.
This way of error recovery in a process is based on the observation that components of a machine are often controlled in the following way: in order to effect a change in the external state of the component, its actuators are made to change state. The controller consequently waits for a change of state of its sensors, indicating that the component has completed the desired external state change. This activation of actuators and consequent waiting for the desired state change of sensors will be referred to as a control action for the component.

Consider a cylinder, for example. The control action to make a two valve cylinder extend would be to open one valve and close the other one. Consequently the controller will wait for the limit sensor to be activated. If an error occurs in such a control action, the control action can often simply be re-executed after correction of the error. This will be referred to as restarting the control action. Restarting a control action can be elegantly implemented by means of the retry response of exception handlers. The type of exception caught will determine whether a retry response is given in the handler, or the exception is propagated to the invoker. Both types of exception, however, share the damage confinement and error reporting code in the handler.

This way of exception handling is illustrated in Figure 7.3.1 using the example of the fork-lift truck. The method ControlBubble >> restartMessage: sends the given restart message to the MMI and then remains blocked in a receive action to receive the response from the MMI. In this case, a special receive action is used, that can be interrupted by raising a pending exception, even when an exception handler is executing (see Section 6.3.2). If the operator can correct the error, he can continue by choosing the restart response in the MMI panel. This will cause the MMI processor to send a continue message to the processor from which the restart message was received. When this continue message is received, the invoked method ControlBubble >> restartMessage: will return and the exception will be restarted causing the do block to be re-executed.

```
SlaveForkLifter >> forkUp
    AnySignal
        handle:
            [:ex |
            self errorMessageFor: ex.
            self putOff: 'o-forkLifter-power'.
            ex signal == RetrySignal
                ifTrue:
                    [self restartMessage: 'fork lifter up'.
                    ex restart]
                ifFalse: [ex reject]]
        do:
            [self putOn: 'o-forkLifter-up'.
            self putOn: 'o-forkLifter-power'.
            self
                receive: true
                from: 'i-forkLifter-isUp'
                within: 6 seconds
                ifTimedOut:
                    [RetrySignal raiseErrorString: 'time-out on fork up'].
            self putOff: 'o-forkLifter-power']
```

Figure 7.3.1 *Exception handling with the retry strategy.*

Because this structure is often used, the method ControlBubble >> handle:restart:do: shown in Figure 7.3.2 offers a compact way of expressing it:

```
ControlBubble >> handle: handler restart: restartBlock do: doBlock
    AnySignal
        handle:
            [:ex |
            self errorMessageFor: ex.
            handler value: ex.
            ex signal == RetrySignal
                ifTrue:
                    [restartBlock value.
                    ex restart]
                ifFalse: [ex reject]]
        do: doBlock
```

Figure 7.3.2 *A simple method which can be used to express the retry strategy.*

The example of the fork-lift truck is thus reduced as follows:

```
SlaveForkLifter >> forkUp
    self
        handle: [:ex | self putOff: 'o-forkLifter-power']
        restart: [self restartMessage: 'fork lifter up']
        do:
            [self putOn: 'o-forkLifter-up'.
            "etc."
            self putOff: 'o-forkLifter-power']
```

Figure 7.3.3 *Exception handling with the retry strategy.*

The class ControlBubble is a subclass of the class Bubble. All processors used for control should belong to subclasses of ControlBubble in order that they inherit the functionality for the control of physical systems and so that they can interact with the MMI. The two additional messages are defined in the class ControlBubble instead of Bubble because they are less general. They use, for instance, the specific ports mmi-in and mmi-out for communication with the MMI.

In the case where a constraint monitor should be bound to the do block, one can use the method ControlBubble >> handle:restart:constraintMonitors:do:. The only difference from the method mentioned above is that, in the last method, the doBlock is bound to the constraint monitors specified in the message, just as it is done in the message Bubble >> handle:constraintMonitors:do:.

7.4 Different control modes

The control model of the transporter treated in Section 2.3 was based on an error-free production system. All machines were initially in a defined reset position.

Practical control systems must deal with machines that can be in an arbitrary initial position. For this purpose, many control programs are divided into at least two parts. The first part can bring the controlled system into a well-defined state. This is known as resetting the system. Resetting a system is not only necessary initially, but can also occur during the production process in order to recover from errors. When the controlled system is being reset by the controlling system, the controlling system is said to operate in *reset mode*. The second part controls the production process, which is usually

cyclic. In this situation, the controlling system is operating in *automatic mode*.

It is desirable to recover from errors as efficiently as possible. The production cycle can be temporarily suspended in order to correct the error, but after correction of the error it is desirable to continue the production cycle from where it was interrupted. To do this the retry strategy can be used, as treated in the previous section. It is, however, not always possible to use the retry strategy. For example, if the stack falls of the fork-lift and the products on the trays are damaged, then it is useless to continue the transportation of the damaged products to the furnace. In this case, the fork-lift truck should be reset to its initial position, waiting at the traverse for a new stack. Resetting a system can also be necessary if, for some reason, the controlling system is no longer synchronized with the controlled system. This happens if the controlled system is in a state which the controlling system does not expect. This kind of error should also be handled as efficiently as possible. It is, for example, not necessary to reset the processes that are stacking new trays onto the stack when the fork-lift truck needs to be reset. So resetting should be done as locally as possible.

Resetting a system could be implemented in such a way that the system always returns to the same defined state. It is often more efficient to define different reset positions for a given system. Consider the fork-lift truck, for example. If an error occurs when the fork-lift truck is depositing the stack at the furnace and a reset is necessary, it is a waste of time to make the fork-lift truck go back to the traverse with a full stack. It makes sense in this situation to define two reset positions: one with an empty fork at the traverse and a second position, with a full stack on the fork, at the furnace.

Resetting a system is usually done under the control of an operator. The operator may be necessary in order to remove damaged material from the machine. He may also be the person to decide that resetting is the only way to recover from an error.

The cyclic control of the production process is expressed in the simplest and clearest way if the cycle always begins with the controlled system in a unique initial state. Therefore, the automatic mode begins with an initialization stage in which the system is brought from a number of defined reset states to a unique, defined state. After this the main control loop can be entered.

When the reset or automatic mode is terminated by an exception, the controlling system enters the stand-by mode. In this mode, a command from the operator is awaited to reset the system. After the system has been reset, an operator command is awaited to enter automatic mode. In reality, the switching from one mode to another is more complex with, for example, an additional manual mode, but this does not lie within the scope of this thesis.

The methods required to implement the reset mode and automatic mode include the following methods:

ControlBubble >> switchToResetMode

```
"Sent when going to reset mode."
killFromMMIMonitor protect: [self resetMode].
```

ControlBubble >> switchToAutomaticMode

```
"Sent when going to automatic mode."
killFromMMIMonitor protect: [self automaticMode].
```

ControlBubble >> resetMode

```
"Sent when going to reset mode.
May be redefined by a subclass (copied to a subclass and then edited).
Any exception that is caught in the handler must be rejected. This is done
automatically so no response should be issued in the handler."
self
    handle: [:ex | "user defined exception handling"]
    constraintMonitors: killMonitors
    do: [self resetBody]
```

ControlBubble >> automaticMode

```
"Sent when going to automatic mode.
May be redefined by a subclass (copied to a subclass and then edited).
Any exception that is caught in the handler must be rejected. This is done
automatically so no response should be issued in the handler."
self
    handle: [:ex | "user defined exception handling"]
    constraintMonitors: killMonitors
    do:
        [self automaticInitialize.
        [self automaticBody] forever       "main control loop"]
```

The killFromMMIMonitor monitors whether a command is sent by the operator by means of the MMI interface to bring the system into a different mode. If such a command is received by the constraint monitor, it signals the

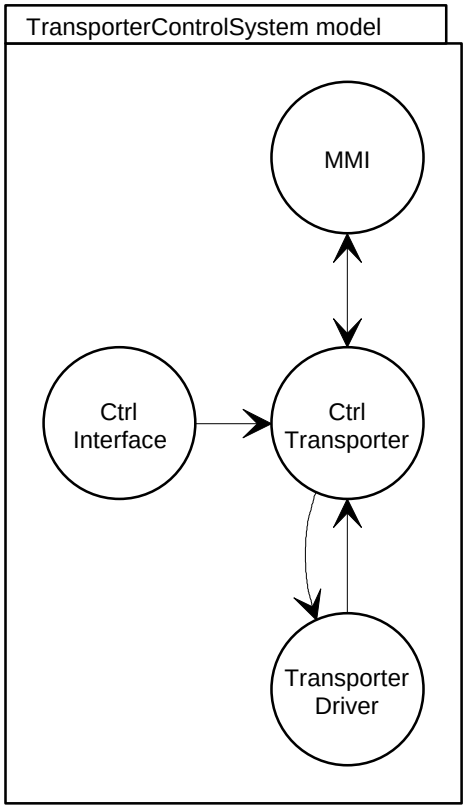

Figure 7.5.1a *Model of the controlling system of a transporter.*

constraint violation and the KillSignal is raised without an error message so that the reset or automatic mode that was active is terminated.

The instance variable killMonitors is a constraintMonitorCollection. User defined constraint monitors can be added to this collection to enable the reset or automatic mode to be terminated and stand-by mode to be entered due to constraint violations caused, for example, by the pressing of the emergency button.

When a controlling processor is required to respond to commands from the MMI to change its mode, its body should call the predefined method modeBody as follows:

```
body
    self modeBody
```

The method modeBody takes care of the synchronization and communication with the MMI and will eventually result in the methods resetMode or automaticMode being sent to the controlling processor (self). The application programmer only needs to define the methods resetBody, automaticInitialize and automaticBody or to redefine the methods resetMode and automaticMode.

7.5 The control model

7.5.1 The structure of the model

The model is shown in Figures 7.5.1a to c. The controlled system is the transport system treated in Section 2.3.
In Figure 7.5.1a the controlling system is shown. The CtrlTransporter

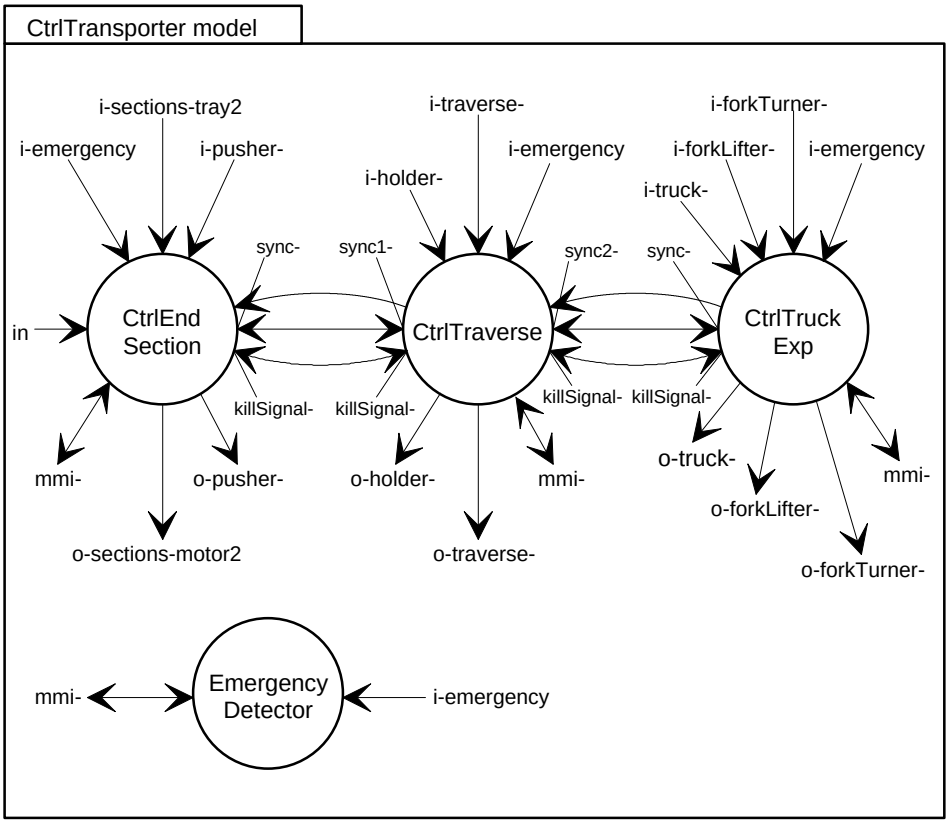

Figure 7.5.1b *Model of the CtrlTransporter processor.*

processor is expanded. Its model is shown in Figure 7.5.1b. It consists of the processors that control the transport machines by means of the actuators and sensors. The other processors of Figure 7.5.1a are interface processors. The MMI processor interfaces with the operator by means of the man-machine interface. The TranporterDriver interfaces with the physical actuators and sensors of the controlled machine and the CtrlInterface processor interfaces with controlling processors of the system's environment.

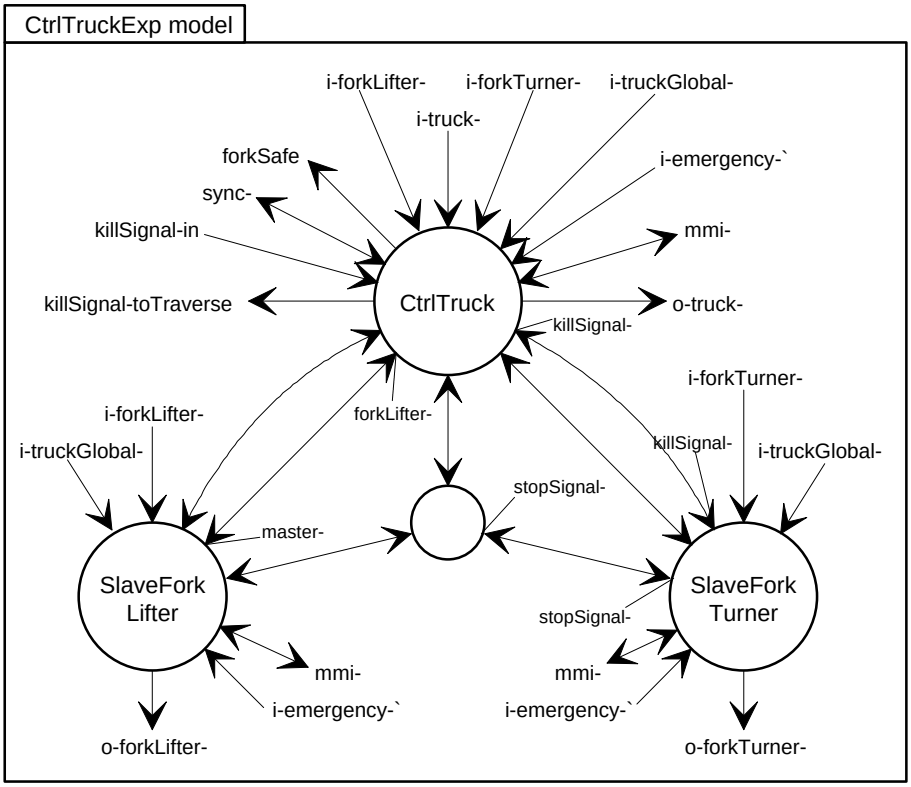

Figure 7.5.1c *Model of the CtrlTruckExp processor.*

The models shown in Figures 7.5.1b and 7.5.1c are similar to the models of Figures 2.3.2b and 2.3.2c. The difference is that interactions are added for the handling of constraint violations and for resetting the system.

7.5.2 The definition of the constraint monitors

The classes of all controlling processors in the model inherit from the class CtrlTransporterAbstract. This means that they are a subclass of this class itself, or of one of its subclasses. Therefore, all these processors share the same initialization and they also inherit the instance variables killMonitor and emergencyMonitor defined in the class CtrlTranporterAbstract. The definition of these constraint monitors is shown in Figure 7.5.2.

```
---- CtrlTransporterAbstract > initialization ----

initializeTasks
    super initializeTasks.          "initialize for superclasses"
    killMonitor := ConstraintMonitor
        for: self
        receiveFrom: 'killSignal-in'
        then: [ :item | KillSignal raise].
    emergencyMonitor := ConstraintMonitor
        for: self
        receive: true
        from: 'i-emergency'
        then: [ :item | KillSignal raise].
    killMonitors add: killMonitor.
    killMonitors add: emergencyMonitor
```

Figure 7.5.2 *Definition of the constraint monitors shared by the controlling processors.*

The killMonitor tries to receive the symbol #kill from the port killSignal-in. The convention used is that a process which violates an active constraint of another process sends the symbol #kill to the other process, which receives the symbol through the port killSignal-in.
The emergencyMonitor monitors the pressing of the emergency button.

7.5.3 Constraint violations between CtrlEndSection and CtrlTraverse

Figure 7.5.3 shows how the CtrlEndSection processor will receive a tray from the previous section. When it has received the tray, it will be stacked at the bottom of the stack. To control the stacking of the tray, CtrlEndSection synchronizes with the processor CtrlTraverse. This synchronization takes

```
-------- CtrlEndSection > process control --------

body
    self modeBody

automaticMode
1   self
2       handle:
3           [:ex |
4           self putOff: 'o-motor'.
5           self send: 'undefined' continuousTo: 'sync-pusherState']
6       constraintMonitors: killMonitors
7       do:
8           [self automaticInitialize.
9           [self automaticBody] forever]

automaticBody
1   self receiveTray.
2   self stackTray

automaticInitialize
    (self isOn: 'i-tray') ifTrue: [self stackTray]

stackTray
    "State upon entry: pusher down"
1   self receiveFrom: 'sync-traverseAtPusher'.
2   self
3       handle: [:ex | self broadcast: #kill to: 'killSignal-toTraverse']
4       do:
5           [self pusherUpAgainstStack.
6           self send: 'upAgainstStack' continuousTo: 'sync-pusherState'.
7           self receiveFrom: 'sync-holderOpened'.
8           self pusherMaximalUp.
9           self send: 'maximalUp' continuousTo: 'sync-pusherState'.
10          self receiveFrom: 'sync-holderClosed'.
11          self pusherDownStart.
12          self send: 'belowMiddle' continuousTo: 'sync-pusherState'].
13  self pusherDownWait.
14  self send: 'down' continuousTo: 'sync-pusherState'
```

Figure 7.5.3 *Process description of CtrlEndSection.*

place in the do block from lines 5 to 12 in the method CtrlEndSection >> stackTray.
This block could be prematurely terminated because of an exception occurrence. Such a premature termination would be a violation of a

constraint of the synchronizing CtrlTraverse processor. Therefore, the exception is handled in line 3 and the #kill symbol is sent to the CtrlTraverse processor, causing its killMonitor to signal, so that an exception will be raised in CtrlTraverse. If the cause of the exception occurrence in CtrlEndSection was, for instance, an error occurring during execution of the method pusherMaximalUp in line 8, then CtrlTraverse would have been waiting for the pusher to be maximally up in CtrlTraverse >> stackTray line 8 (Figure 7.5.4). At that point the exception due to the signaling of CtrlTraverse's killMonitor would be raised.

```
--------- CtrlTraverse > process control ---------

body
    self modeBody

automaticBody
    "State upon entry: pusher down, traverse retracted"
    (stackSize >= self maxStackSize) ifTrue: [self stackToTruck].
    self stackTray

stackTray
    "State upon entry: traverse at pusher"
1   self sendTo: 'sync1-traverseAtPusher'.
2   self
3       handle: [:ex | self broadcast: #kill to: 'killSignal-toEndSection']
4       do:
5           [self receive: 'upAgainstStack' from: 'sync1-pusherState'.
6           self holderOpen.
7           self sendTo: 'sync1-holderOpened'.
8           self receive: 'maximalUp' from: 'sync1-pusherState'.
9           self holderClose.
10          stackSize := stackSize + 1.
11          self sendTo: 'sync1-holderClosed'.
12          self receive: 'belowMiddle' from: 'sync1-pusherState']
```

Figure 7.5.4 *Process description of CtrlTraverse.*

7.5.4 Interaction mechanisms used for the synchronization between controlling processors

Two kinds of interaction mechanism are used for synchronization interactions between controlling processors: the *synchronous* mechanism and the *continuous* mechanism.

An example of the synchronous interaction mechanism is the interaction represented by `self receiveFrom: 'sync-holderOpened'` in CtrlEndSection >> stackTray (Figure 7.5.3, line 7) and `self sendTo: 'sync1-holderOpened'` in CtrlTraverse >> stackTray (Figure 7.5.4, line 7). Both processors must execute the send and receive actions for the interaction to take place.

An example of the continuous interaction mechanism is the interaction represented by self send: 'upAgainstStack' continuousTo: 'sync-pusherState' in CtrlEndSection >> stackTray (line 6) and self receive: 'upAgainstStack' from: 'sync1-pusherState' in CtrlTraverse >> stackTray (line 5). A second example is represented by self send: 'down' continuousTo: 'sync-pusherState' in CtrlEndSection >> stackTray (Figure 7.5.3, line 14) and self receive: 'down' from: 'sync1-pusherState' in CtrlTraverse >> stackToTruck (Figure 7.5.5, line 8). The traverse will only move back to the pusher when the pusher is completely down. If the pusher has gone down without errors, the receive action from CtrlTraverse >> stackToTruck (line 8) will receive immediately. A synchronous interaction mechanism is not appropriate in this case, because the receive action in CtrlTraverse could take place while a new tray is entering the end section (CtrlEndSection >> automaticBody – Figure 7.5.3, line 1), and so CtrlEndSection cannot execute a corresponding send action at that time. Therefore, the continuous mechanism is used.
In CtrlEndSection >> automaticMode (Figure 7.5.3, line 5), the state of the pusher is defined as 'undefined' when the methods automaticInitialize or automaticBody are terminated with an exception.

7.5.5 Synchronization between controlling processors without using sensors

As was mentioned in Section 2.3.2, the synchronization between the controlling processors is effected by means of interactions between them, rather than by synchronizing directly on the state of the sensors. This last option can lead to dangerous situations in the error recovery stage.

Consider, for example, the synchronization between CtrlEndSection >> stackTray (line 7) and CtrlTraverse >> stackTray (lines 6-7). The holder is opened by CtrlTraverse in line 6. In the following line, the processor CtrlTraverse is notified of the fully opened holder by means of an interaction. Instead of waiting for this interaction to take place in CtrlEndSection >> stackTray (line 7), the CtrlEndSection processor could also have waited for the sensor, indicating a fully opened holder, to be activated. This can lead to dangerous situations in the following way. Suppose that the pusher does not open properly due to an error. In order to correct the error, it could be necessary to open the pusher manually, thereby activating the sensor indicating an open pusher. This sensor could also be accidentally activated while trying to correct the error in another way. In both cases, the CtrlEndSection processor would immediately continue after having detected

the activation of the sensor. This would cause the stack to be pushed upward suddenly (line 8). This immediate activation of machine components during the correction of errors in machine parts can lead to unexpected and dangerous situations.

To prevent such situations, controlling processors synchronize directly with each other by means of interactions. Each controlling processor controls the sensors and actuators of certain machine components. These actuators and sensors are not used by any other processor. If an error occurs during the activation of a machine component, an error message is sent to the operator. If the processor in which the error was detected cannot automatically correct the error, it will wait for a signal from the operator, indicating that the controlling processor may continue. Therefore, the activation of the sensors of the machine part in question cannot cause a sudden activation of other components.

Figure 7.5.1c appears to be inconsistent with the above-mentioned convention of no more than one controlling processor for each component. For example, the sensors from the forkLifter (i-forkLifter-) are connected to the processors SlaveForkLifter and CtrlTruck. This means that the status of the fork lifter sensors is read by both processors. The sensors of the fork-lift truck, however, are only used by the CtrlTruck processor to determine the status of the truck while resetting it, and not as a means of synchronization between two controlling processors. The methods for resetting are not shown here because they are too specific and do not serve to illustrate the use of constraint monitors.

Checking the state of the controlled machines can also lead to situations where the sensors of a machine component are used by several controlling processors. This is done in order to check whether the state of the controlled machines conforms to the state which is expected by the controlling system. If the controlling system and the controlled machines would function correctly, this would not be necessary. Errors, however, are always possible. So, in order to prevent errors from causing serious damage, the state of the controlled machines can be checked.

7.5.6 Constraint violations between CtrlTraverse and CtrlTruck

In the methods CtrlTraverse >> stackToTruck and CtrlTruck >> receiveStackFromTraverse (Figures 7.5.5 and 7.5.6), the stack is transported from the traverse to the fork-lift.

```
--------- CtrlTraverse > process control ---------

stackToTruck
    "State upon entry: traverse at pusher"
1   self receiveFrom: 'sync2-truckResetAtTraverse'.
2   self
3       handle: [:ex | self broadcast: #kill to: 'killSignal-toTruck']
4       do:
5           [self traverseToFork.
6           self sendTo: 'sync2-traverseAtFork'.
7           self receiveFrom: 'sync2-forkIsUp'].
8   self receive: 'down' from: 'sync1-pusherState'.
9   self traverseToPusher
```

Figure 7.5.5 *The method stackToTruck of the processor CtrlTraverse.*

7.5.7 Constraint violations in the CtrlTruckExp model

The processors of the CtrlTruckExp model are hierarchically structured. The processor CtrlTruck receives the mode commands from the MMI. CtrlTruck coordinates the actions of the fork lifter, the fork turner and the horizontal movements of the truck itself. The two slave processors SlaveForkLifter and SlaveForkTurner are used by the CtrlTruck processor to realize parallelism in the movements of the fork-lift truck. The slave processors receive a command from their master CtrlTruck, execute it, and send an acknowledge back to the master when the command has been executed (see the method ControlBubble >> slaveBody in Figure 7.5.7). The processors are termed slaves because the master sends them commands which are executed by them. The slaves, in turn, cannot order the master to do anything. They can only inform the master of exception occurrences. The slaves SlaveForkTurner and SlaveForkLifter both inherit the method slaveBody from the class ControlBubble. A simplified body of slave processors has already been explained in Section 2.3.3. The complete version is shown in Figure 7.5.7 in ControlBubble >> slaveBody.

```
---------- CtrlTruck > process control ----------

body
    self modeBody

automaticMode
1   self
2       handle: [:ex | self broadcast: #kill to: 'killSignal-toSlaves']
3       constraintMonitors: killMonitors
4       do:
5           [self automaticInitialize.
6           [self automaticBody] forever]

automaticBody
    "State upon entry: truck at traverse, fork middle postion, turned to
    traverse"
    self receiveStackFromTraverse.
    self transportStackToFurnace.
    self giveStackToFurnace.
    self goBackToTraverse

receiveStackFromTraverse
    self sendTo: 'sync-truckResetAtTraverse'.
    self
        handle: [:ex | self broadcast: #kill to: 'killSignal-toTraverse']
        do:
            [self receiveFrom: 'sync-traverseAtFork'.
            self forkUp.
            self sendTo: 'sync-forkIsUp']
```

Figure 7.5.6 *Process description of CtrlTruck.*

If an exception occurs in the body of one of the slaves (ControlBubble >> slaveBody) in line 11 or 12, the do block (lines 11-12) will be terminated with an exception and the master can no longer receive the awaited acknowledge. This constraint violation should be signaled to the master. This is done by sending a #kill symbol to the master in line 6 so that its killMonitor will signal the constraint violation and an exception will be raised.

If an exception would cause the automaticBody (Figure 7.5.6 CtrlTruck >> automaticMode line 6) from the master to be terminated with an exception while the master was still waiting for an acknowledge from a slave (Figure 7.5.7 CtrlTruck >> forkUp line 2, for example), the acknowledge from the slave will no longer be received. The slave should be notified of this

```
---------- CtrlTruck > slave control ----------

forkUp
1   self send: #forkUp to: 'forkLifter-command'.
2   self receiveFrom: 'forkLifter-ack'


---------- ControlBubble > process control ----------

slaveBody
1   | command |
2   command := self receiveFrom: 'master-command'.
3   AnySignal
4       handle:
5           [ :ex |
6           self broadcast: #kill to: 'killSignal-toMaster'.
7           self errorMessageFor: ex.
8           ex return]
9       do:
10          [killMonitors protect:
11              [self perform: command.
12              self sendTo: 'master-ack']]


---------- SlaveForkLifter > process control ----------

body
    self slaveBody


---------- SlaveForkTurner > process control ----------

body
    self slaveBody
```

Figure 7.5.7 *The cooperation of the slaves with the master.*

constraint violation, otherwise it will remain blocked trying to send its acknowledge. This is done in CtrlTruck >> automaticMode in line 2.

In CtrlTruck >> forkUp, the CtrlTruck processor starts waiting for the acknowledge immediately after having sent the command. This need not always be the case. In the case of more parallelism, the processor could also send a command to a slave and after that continue with other actions, finally receiving the acknowledge at a completely different point in the program.

In CtrlTruck >> automaticMode (line 2), the #kill symbol is sent to the port killSignal-toSlaves. This port is connected with the ports killSignal-in on all slaves. Actually the master CtrlTruck should only send a #kill symbol to a slave of which a constraint has been violated. This would be a slave trying to send an acknowledge to the master while the master will no longer receive it. In this model the #kill symbol is sent to all the slaves, regardless of whether a constraint has been violated or not. This approach leads to much simpler code in the master because it need not know exactly which of the slave's constraints, if any, have been violated. When a slave has finished a command and is waiting for a new one, then its killMonitor will be disabled and it will simply not receive the #kill symbol.

7.6 The retry strategy in a multi-process environment

7.6.1 Exceptions in a group with a master and slaves

In the retry strategy treated in Section 7.3, a sequential process detects errors in the system components that it controls. An exception is raised if an error is detected in such a component. If the RetrySignal is raised, the exception will be handled by retrying the terminated control action. In a multi-process environment, exception occurrences in a process need not only be a result of errors in the system components controlled by the process itself. They can also be a result of errors in other components.
Consider the fork-lift truck, for example. When it transports a stack from the traverse to the furnace, three movements will take place in parallel when the truck passes the sensor canTurnToFurnace (Figure 2.3.1). The truck rides to the furnace, the fork-lift goes down and the fork turns to the furnace. Time-outs are used in the control of all three movements. If a time-out should occur in the control of any of these movements, the exact cause of the time-out cannot be determined by the control system because not enough sensors are available. Therefore, the control system should adopt a worst case scenario and stop all three movements in the case of a single time-out. The truck is also equipped with sensors to detect collisions. All three movements should also be stopped if a collision is detected. Note that systems also exist in which an error in a component controlled by a slave need not result in the stopping of all components controlled by the other slaves and the master.

The requirements of Section 7.1 imply that it should be possible to restart all interrupted control actions after correction of the error by the operator. The

fork-lift truck should only be reset in the case that the operator decides that it is useless to restart the interrupted actions.
It is evident that exceptions should only be raised in the processors that control a component which is actually moving at the time of the error. Suppose that the truck is moving and the fork is turning, but the fork-lift has already reached the stable middle position and is no longer going down at the time of the error. In this case, the exceptions need only be raised in the SlaveForkTurner en CtrlTruck processors in order to stop the movements of the truck and fork.

It is concluded that a RetrySignal should be raised in a processor controlling part of the fork-lift truck in any of the following three situations: first, when an error occurs in the part of the fork-lift truck controlled by the processor itself; second, when a collision is detected; and third, when an error occurs in a part of the fork-lift truck controlled by another processor.
These three situations are taken care of in the following way. In the first case, a time-out can be used to detect errors in the controlled component. The second case is taken care of by the bumperMonitor which is a constraint monitor present in all three processors. It monitors a collision of the truck. The third case is taken care of by the stopMonitor. This is a constraint monitor which is also used in all three processors. When enabled, it tries to receive the symbol #stop from the port stopSignal-in (see Figure 7.5.1c). The receipt of that object indicates a constraint violation which causes the RetrySignal to be raised. The #stop symbol is sent by any of the three processors in which a time-out is detected. It is sent to the other processors by means of a broadcaster processor which acts as an intermediate agent.

7.6.2 The BroadCaster processor

The reason for using the broadcaster is that this makes the structure of the model more elegant, especially when many slaves are involved. Without the broadcaster, each processor from the collection represented by the master and the slaves would have to be connected by means of interaction paths to all other processors from the collection. With the broadcaster, the processors from the collection need only be connected with the broadcaster.

The processor BroadCaster simply sends the received symbol to its port stopSignal-out by means of a continuous interaction (see Figure 7.6.1). The port stopSignal-out is connected to the ports stopSignal-in on the three controlling processors CtrlTruck, SlaveForkLifter and SlaveForkTurner. The

```
---------- BroadCaster > process control ----------

body
    | item |
    item := self receiveFrom: 'stopSignal-in'.
    self send: item continuousTo: 'stopSignal-out'
```

Figure 7.6.1 *Process description of BroadCaster.*

processor BroadCaster is shown in Figure 7.5.1c as a small circle without name and is connected with all three controlling processors in the same way. After the broadcaster has sent the #stop symbol to its stopSignal-out port, any enabled stopMonitors in the other two controlling processors will signal the constraint violation by creating a pending exception. Note that the #stop symbol will also be sent back to the controlling processor which originally sent it. The stopMonitor will be disabled in this processor, so that the #stop symbol will not be received and have no effect. The definition of the stopMonitor is shown in Figure 7.6.2.
The #stop symbol is sent by means of a continuous interaction mechanism, because a state is indicated to the other processors, and not an event: the presence of the detected error must not only cause components to stop which are actually moving, but must also prevent new movements from being started before the error is corrected.
It should be noted that in this example the #stop symbol is sent to all other controlling processors by means of the broadcaster. If the #stop symbol should not be sent to all other processors, this can of course be achieved by a different structure of the interaction paths used to connect the controlling processors and the broadcaster.

7.6.3 The definition of constraint monitors for the retry strategy

The definition of the constraint monitors used for the retry strategy in this example is shown in Figure 7.6.2. The classes of the three processors controlling the fork-lift truck all inherit from the class CtrlTruckAbstract.

```
------- CtrlTruckAbstract > initialization -------

initializeTasks
super initializeTasks.
retryMonitors := ConstraintMonitorCollection new.
stopMonitor := ConstraintMonitor
    for: self
    receive: #stop
    from: 'stopSignal-in'
    then: [ :item | RetrySignal raise].
bumperMonitor := ConstraintMonitor
    for: self
    receive: true
    from: 'i-truckGlobal-bumper'
    then: [ :item | RetrySignal raiseErrorString: 'activation of truck bumper'].
retryMonitors add: stopMonitor.
retryMonitors add: bumperMonitor
```

Figure 7.6.2 *The definition of constraint monitors for the retry strategy.*

7.6.4 An illustration of the retry strategy used in SlaveForkLifter

The method SlaveForkLifter >> forkUp is shown in Figure 7.6.3. It has been provided with additional monitors to implement the retry strategy in a multi-process environment.
In the case of a time-out, the #stop symbol is sent by the invocation of the method sendStop in line 15. After correction of the error, the operator will give a command via the MMI to restart the interrupted methods. Before an interrupted method can be restarted, the effect of the previously sent stop command must first be undone, otherwise the stopMonitor would immediately signal again when it was enabled. The stop command is undone by means of the invocation of the method clearStop in line 5. The definition of the methods sendStop and clearStop is given in Figure 7.6.4.
The stop command is cleared in the restart block in Figure 7.6.3, line 5. Under certain circumstances, however, this line will not be executed. Suppose that the operator does not want to restart the interrupted actions but instead wants to bring the processes to the stand-by mode. In that case, the stand-by command from the MMI will cause the killFromMMIMonitor to signal by creating a pending exception. This pending exception would be raised at the place of the blocking statement of line 4. Therefore, the automatic mode would be terminated before the stop command could be cleared. To make sure that the stop command is also cleared in such situations, the methods

```
---------- SlaveForkLifter > machine io ----------

forkUp
1  self
2      handle: [:ex | self putOff: 'o-forkLifter-power']
3      restart:
4          [self restartMessage: 'fork lifter up'.
5          self clearStop]
6      constraintMonitors: retryMonitors
7      do:
8          [self putOn: 'o-forkLifter-up'.
9          self putOn: 'o-forkLifter-power'.
10         self
11             receive: true
12             from: 'i-forkLifter-isUp'
13             within: 6 seconds
14             ifTimedOut:
15                 [self sendStop.
16                 RetrySignal raiseErrorString: 'time-out on fork up'].
17             self putOff: 'o-forkLifter-power']
```

Figure 7.6.3 *The method forkUp with additional constraint monitors for the retry strategy.*

ControlBubble >> slaveBody and CtrlTruck >> automaticMode are changed in such a way that the stop command is also cleared by the exception handlers of these methods. Figure 7.6.5 shows the new method CtrlTruck >> automaticMode.

```
------ CtrlTruckAbstract > process control -------

sendStop
    self send: #stop to: 'stopSignal-out'

clearStop
    (self receiveFrom: 'stopSignal-in') == #stop
        ifTrue:
            [self send: #ok to: 'stopSignal-out'.
            self receive: #ok from: 'stopSignal-in']
```

Figure 7.6.4 *Methods for stopping other processes and enabling their continuation.*

```
---------- CtrlTruck > process control ----------

automaticMode
    self
        handle:
            [:ex |
            self broadcast: #kill to: 'killSignal-toSlaves'.
            self clearStop]
        constraintMonitors: killMonitors
        do:
            [self automaticInitialize.
            [self automaticBody] forever]
```

Figure 7.6.5 *The clearing of the stop command when the automatic mode is terminated.*

7.6.5 An illustration of the retry strategy used in CtrlTruck

The use of the retry strategy for the control of the truck itself is somewhat more complicated than the retry strategy for the control of the fork-lift. This is due to the fact that the control of the truck, while it is going from the traverse to the furnace, is divided into two parts, as shown in Figure 7.6.6.
In the first part, the truck moves from the traverse to the turning point at the traverse. At that point, the fork-lift should start going down to the middle position and the fork should start to turn to the furnace. The commands to make the fork go down and turn are given in CtrlTruck >> transportStackToFurnace, lines 2-3. The commands are given in a block because they should be seen at the level of the method CtrlTruck >> transportStackToFurnace. At this level the coordination of the different movements of the fork-lift truck is determined. The actions should, however, be executed in the method CtrlTruck >> toTurnPointAtTraverseDo:, because

```
-------- CtrlTruck > process control --------

transportStackToFurnace
1  self toTurnPointAtTraverseDo:
2      [self send: #forkDownToMiddle immediateTo: 'forkLifter-command'.
3      self send: #turnToFurnace immediateTo: 'forkTurner-command'].
4  self continueToFurnace.
5  self receiveFrom: 'forkLifter-ack'.
6  self receiveFrom: 'forkTurner-ack'
```

Figure 7.6.6 *Transportation of the stack from the traverse to the furnace.*

errors occurring in the two send actions should result in the stopping of the truck and should therefore be caught by the handler specified in the method CtrlTruck >> toTurnPointAtTraverseDo:. The truck does not stop at the turning point under normal operation.
In the second part, the truck continues to go to the furnace.

Figures 7.6.7a-b show the two methods for the control of the truck movements. They make use of the monitorCollection retryMonitors specified in Figure 7.6.2. This collection contains the stopMonitor and the bumperMonitor. These two monitors will be referred to as the retry monitors.
Strictly speaking, the retry strategy should not be specified in the way shown in Figures 7.6.7a-b. The reason for this is that the motor of the truck remains switched on at the turning point, so the retry monitors should also remain enabled. In fact, the retry monitors should be enabled when the motor of the truck is on and be disabled when the motor is off. By separately binding the retry monitors in both methods, the retry monitors are temporarily disabled when the first method is terminated.
Nevertheless, the control of the truck is split into two methods, because in

```
-------- CtrlTruck > machine io --------

toTurnPointAtTraverseDo: endBlock
    "motor remains on when the do block is terminated normally"
    self
        handle: [:ex | self truckStop]
        restart:
            [self restartMessage: 'truck to turning position at traverse'.
            self clearStop]
        constraintMonitors: retryMonitors
        do:
            [self putOn: 'o-truck-toFurnace'.
            self putOn: 'o-truck-power'.
            self
                receive: true
                from: 'i-truck-canTurnToFurnace'
                within: 10 seconds
                ifTimedOut:
                    [self sendStop.
                    RetrySignal raiseErrorString:
                        'time-out moving to turning position at traverse'].
            endBlock value]
```

Figure 7.6.7a *The control of the truck using the retry strategy with constraint monitors.*

this way exceptions can be elegantly and locally handled in each method. The temporary disabling of the retry monitors has no undesirable effects.

```
-------- CtrlTruck > machine io --------

continueToFurnace
    "motor should be on upon activation of this method"
    self
        handle: [:ex | self truckStop]
        restart:
            [self restartMessage: 'truck to furnace'.
            self clearStop.
            self putOn: 'o-truck-power']
        constraintMonitors: retryMonitors
        do:
            [self
                receive: true
                from: 'i-truck-atFurnace'
                within: 20 seconds
                ifTimedOut:
                    [self sendStop.
                    RetrySignal raiseErrorString:
                        'time-out on truck-at-furnace detector'].
            self putOff: 'o-truck-power']
```

Figure 7.6.7b *The control of the truck using the retry strategy with constraint monitors.*

Chapter 8
Conclusions

8.1 Evaluation

Background and existing mechanisms

The handling of errors and exceptions is an important aspect in the development of industrial control systems. The amount of code needed for error handling is often several times greater than the amount needed for the control under error-free circumstances. Considerable progress has been made in the field of mechanisms for the handling of internal exceptions. We have shown that these mechanisms are important for the creation of robust programs. They are, however, not sufficient for controlling systems, because these systems require an additional mechanism for the handling of constraint violations. Several proposals and existing mechanisms for the handling of constraint violations are known from the literature. These mechanisms have been evaluated as either offering a functionality which is too restricted for controlling systems, as offering an incorrect or undesirable functionality, or as inadequate in other ways.

A clear definition of concepts

The inadequacy of the mechanisms which have been evaluated is ascribed to the absence of clearly defined concepts and the absence of a sound theory describing the essence of exception handling in controlling systems or, more generally, in multi-process environments. Many definitions found in the literature are imprecise or incorrect, or contain undesirable subjective elements. In order to arrive at a new theory and new concepts, the most important terms relating to errors and exceptions are accurately defined. The relationship between exceptions and errors has also been clarified. The definitions and relationships given here result in a better understanding of the terminology of errors, exceptions and the relationships between them.

New concepts to describe the essence of the handling of external exceptions in controlling systems

An important contribution of this study is the introduction of the new concepts 'constraint of an operation' and 'constraint violation', which are essential in order to determine the requirements of a mechanism for the handling of external exceptions in controlling systems. A constraint can be compound, in which case it consists of (sub-)constraints.

The constraints of an operation are specific for the operation itself and thus independent of the point in the program at which the operation is invoked. This point of view is essential for the development of modular subprograms.

The new concepts contribute to a better understanding of the way exceptions should be handled in controlling systems or multi-process environments. One of the aspects that is made clear is that there is a restriction for the handling of external exceptions caused by the violation of the constraints of an operation: a handler that catches such an exception, while the operation of which a constraint was violated is not yet terminated, must terminate by propagating the exception.

A new mechanism for the handling of constraint violations

The newly developed mechanism for the handling of constraint violations in controlling systems makes it possible to specify and monitor the constraints of each operation independently of other already invoked operations. This is a quality seldom found in programming languages or systems. At the same time the mechanism is well integrated with the advanced mechanisms for the handling of internal exceptions. The integration is achieved with the addition of only a single programming construct, namely a constraint monitor. This makes the resultant mechanism easy to use and to understand. The required binding of constraint monitors to operations or blocks enforces the use of constraint monitors in a structured way. A constraint monitor bound to a single operation can also be used to specify a constraint which is common to several operations, which will, in many cases, simplify programs.

Constraint violations will cause pending exceptions to be created. These pending exceptions will be raised at interaction points, where the internal invariants of the process can be expected to hold. The choice not to raise

pending exceptions in exception handlers makes it possible to safely restore invariants in exception handlers. Proposals that suggest the immediate raising of external exceptions will lead to time-dependent run-time errors due to violations of the internal invariants of a process. These errors are very dangerous because they are practically impossible to find by testing and can occur completely unexpectedly.

Several constraints can be violated at the same time by concurrently executing processes. This can result in more than one pending exception in a process. Several criteria for the selection of a pending exception have been evaluated. The choice is made to select the pending exception belonging to the constraint monitor which was enabled first. The other pending exceptions are discarded. This leads to the desirable discarding of an exception from a constraint monitor activated at a low (or innermost) level, by an exception from a constraint monitor activated at a high (or outermost) level. Constraint monitors can always be checked to determine whether their pending exception has been discarded.

The binding of a constraint monitor to a block which is invoked by another block bound to the same constraint monitor is essentially a non-operation. This design choice is conceptually simple and retains the desired functionality of constraint monitors.

Implementation of constraint monitors in Process Calculus

The implementation of the mechanism in Process Calculus is relatively straightforward. An important aspect that facilitates the integration of the mechanism is the powerful functionality of Smalltalk blocks.
A constraint monitor has been added to Process Calculus as a simple and relatively orthogonal primitive. Constraint monitors will try to receive an object from a port. This can be any port of a processor and there are no restrictions about the way objects are sent to the port. Constraint monitors are also well integrated with the existing Smalltalk exception handling mechanism. Any exception can, in principle, be raised by a constraint monitor. The mechanism is not orthogonal with respect to the fact that the receive action executed by a constraint monitor is specified slightly differently from the normal receive actions, because it is used to initialize the constraint monitor.

Although all send primitives can be used to indicate constraint violations, the broadcast primitive has been added to Process Calculus for the signaling of constraint violations to other processes. The use of this primitive is important in order to keep different processes, which interact by means of constraint violations, largely independent of each other and in order to achieve good modularity.

The resume response as an inadequate response

It is recommended that the resume response is not used, since its use easily leads to unstructured programs which are hard to understand. The use of the resume response is even more problematic in a multi-process environment. This is because the resume response can be used to enter critical regions containing semaphores, or to enter blocks bound to constraint monitors, without performing the necessary operations on the semaphores or constraint monitors.

A case and the retry strategy

The treatment of a case concerning the control of a transport system has shown the power and simplicity of constraint monitors for the handling of exceptions in control systems. The retry strategy has been developed as a simple strategy which can be used to deal locally with errors in an efficient and safe way. It usually implies the help of an operator. After correction of the error, the interrupted processes can continue by re-executing the interrupted control actions. In the case of errors that cannot be corrected locally, the use of constraint monitors makes it easy to keep communicating processes in a consistent state.

8.2 Recommendations for future research

The newly-developed mechanism has only been implemented in Process Calculus. The implementation in other programming languages should be studied. Also, more experience is needed with the mechanism and Process Calculus in practical complex control systems.

Two other fields for further research follow from the restrictions on the scope of this study as laid down in Section 1.2.

First, the specific characteristics of other programming languages in relation to the handling of exceptions in a multi-process environment should be investigated. An important aspect in this context is the exceptional termination of processes which are created dynamically in parallel constructs. Another aspect to be studied is the way in which exceptions should be handled if they occur during the execution of an interaction, such as during a rendezvous.
Second, the differences between continuous and discrete event systems in respect of exception handling need to be investigated.

References

Adamo, J.,
Exception handling in the C-NET parallel programming language,
Proc. North American Transputer Users Group, 1989, pp. 283-306, IOS, Amsterdam.

Adamo, J.,
Exception handling for a communicating-sequential-process-based extension of C++,
Concurrency, February 1991, pp. 15-41.

Antonelli, C.J.,
Exception handling in a multi-context environment,
Dissertation, University of Michigan, 1989.

Atkins, M.S.,
The role of exception mechanisms in software system design,
Dissertation, University of British Colombia, 1985.

Bell Telephone Laboratories,
Unix programmer's manual, second edition, volume 2,
Holt, Rinehart and Winston, New York, 1983, pp. 315-318.

Bendel, A. and Mellor, P., editors of
Software reliability: State of the art report,
Pergamon, Oxford, 1986.

Booch, G.,
Object oriented design,
Benjamin/Cummings, Redwood City, California, 1991.

Bron, C. and Dijkstra, E.J.,
Report on the programming language Modular Pascal,
Groningen University, Groningen, 1987a.

Bron, C. and Dijkstra, E.J.,
On the handling of exceptional situations in a multi-process environment,
Private communication, 1987b.

Bron, C. and Fokkinga, M.M.,
A proposal for dealing with abnormal termination of programs,
Report 150,
Twente University of Technology, Dept. Inf., Enschede, 1976.

Brynjolfsson, S. and Arnstrom, A.,
Error detection and recovery in flexible assembly systems,
Int. Journal of Advanced Manufacturing Technology, No. 5, 1990, pp. 112-125.

Christian, F.,
Exception handling and software fault tolerance,
IEEE Transactions on Computers, June 1982, pp. 531-539.

Christian, F.,
Correct and robust programs,
IEEE Transactions on Software Engineering, March 1984, pp. 163-174.

Cox, I.J. and Gehani, N.H.,
Exception handling in robotics,
Computer, March 1989, pp. 43-49.

Digital Equipment Corporation,
VAXELN Pascal language reference manual, part 2: programming,
Digital Equipment Corporation, Massachusetts, 1986.

Dijkstra, E.W.,
Cooperating Sequential Processes,
Technical Report EWD - 123,
Eindhoven University of Technology, 1965.
Reprinted in Genuys F. (ed.),
Programming languages,
Academic Services, New York, 1968.

Dony, C.,
Exception handling and object-oriented programming: towards a synthesis,
Proc. OOPSLA/ECOOP '90, pp. 322-330, ACM, New York.

Fairley, R.E.,
Software engineering concepts,
McGraw-Hill, New York, 1985.

Feder, C.,
Ausnahmebehandlung in objektorientierten Programmiersprachen,
Springer-Verlag, Berlin, 1990.

Gerber, R. and Lee, I.,
A layered approach to automating the verification of real-time systems,
IEEE Transactions on Software Engineering, September 1992, pp. 768-771.

Gini, M. and Smith, R.,
Reliable real-time robot operation employing intelligent forward recovery,
Technical Report TR 85-30,
University of Minnesota, 1985.

Goldberg, A. and Robson, D.,
Smalltalk-80, The language,
Addison Wesley, Reading MA, 1989.

Goodenough, J.B.,
Exception handling: Issues and a proposed notation,
Communications of the ACM, December 1975, pp. 683-696.

Hoare, C.A.R.,
An axiomatic basis for computer programming,
Communications of the ACM, October 1969, pp. 576-583.

Hoare, C.A.R.,
Communicating Sequential Processes,
Communications of the ACM, August 1978, pp. 666-677.

Horning, J.J. et al.,
A program structure for error detection and recovery,
Lecture Notes in Computer Science 16, pp 171-187,
Springer Verlag, Berlin, 1974.

Horowitz, E.,
Fundamentals of programming languages,
Springer-Verlag, Berlin-Heidelberg, 1983.

IEC 50.
International Electrotechnical Vocabulary (IEV),
CEI IEC, 1975, Chapter 351.

IEC 848,
International Standard IEC 848: Preparation of function charts for control systems,
CEI IEC, Geneva, 1988.

Ichbiah, J.D. et al.,
Preliminary Ada reference manual,
SIGPLAN Notices, June 1979.

Ichbiah, J.D. et al.,
Reference manual for the Ada programming language,
ANSI/MIL-STD-1815A, 1983.

Issarny, V.,
Design and implementation of an exception handling mechanism for communicating sequential processes,
Proc. CONPAR 90-VAPP IV, 1990, pp. 604-615.

Issarny, V. and Banâtre, J.P.,
An exception handling mechanism for communicating sequential processes and its verification rules,
Proc. Computer Systems and Software Engineering (COMPEURO) 1990, pp. 550-551, IEEE, Los Alamitos.

Issarny, V.,
An exception handling model for parallel programming and its verification,
Sofware Engineering Notes, Proc. ACM SIGSOFT '91 Conference on Software for Critical Systems, December 1991, pp. 92-100.

Kernighan, B.W. and Richie, D.M.,
The C programming language,
Prentice Hall, Englewood Cliffs, 1978.

Kilgerman, E. and Stoyenko, D.,
Real-time Euclid: A language for reliable real-time systems,
IEEE Transactions on Software Engineering, September 1986, pp. 941-949.

Knudsen, J.L.,
Better exception handling in block structured systems,
IEEE Software, May 1987, pp. 40-49.

Laprie, J.C.,
Dependability: a unifying concept for reliable computing and fault tolerance,
in Dependability of Resilient computers ed. T. Anderson,
BSP Professional Books, Oxford, 1989.

Laprie, J.C. (ed),
Dependability: basic concepts and terminology,
Springer-Verlag, Vienna, 1992.

Lacoutre, S.,
Exceptions in Guide, an object oriented language for distributed applications,
Proc. ECOOP 1991, pp. 268-287, Springer-Verlag, Berlin.

Lee, P.A. and Anderson, T.,
Fault tolerance: principles and practice,
Springer-Verlag, Vienna, 1990.

Levin, R.,
Program structures for exceptional condition handling,
Ph.D. thesis, Carnegie-Mellon University, June 1977.

Lieber, G.,
Erweitertes CSP-Modell zur Programmierung Paralleler Prozesse (in German),
(Extended CSP-model for the programming of parallel processes),
Dissertation, Technische Universitat Wien, 1989.

Liskov, B.H. and Snyder, A.,
Exception handling in CLU,
IEEE Transactions on Software Engineering, November 1979, pp. 546-558.

Melliar-Smith, P.M. and Randell, B.,
Software reliability: the role of programmed exception handling,
Proc. ACM Conf. on Language Design for Reliable Software,
March 1977, pp. 95-100.

Meyer, B.,
Object oriented software construction,
Prentice Hall, New York, 1989a.

Meyer, G.R. and Hertzberger, L.O.,
Off-line programming of exception handling strategies,
Proc. IFAC Robot Control '88 (SYROCO '88), pp. 431-436, Pergamon Press Oxford.

Meyer, G.R. and Hertzberger, L.O.,
Exception handling system for autonomous robots based on PES,
Proc. Intelligent Autonomous Systems 2, 1989b, Vol. 1, pp. 65-77.

Overwater, R.,
Processes and interactions, an approach to the modelling of industrial systems,
Dissertation, Eindhoven University of Technology, 1987.

ParcPlace,
Objectworks, advanced user's guide, Smalltalk-80 Version 2.5,
ParcPlace Systems, Mountain View, California, 1990.

Randell, B.,
System structure for software fault tolerance,
Current Trends in Programming Methodology, pp. 195-219,
Prentice Hall, Englewood Cliffs, 1977.

Randell, B., Lee, P.A. and Treleaven, P.C.,
Reliability issues in computing system design,
Computing Surveys, June 1978, pp. 123-165.

Redford, A.H.,
Error recovery in assembly by robot,
Advanced Manufacturing Engineering, Vol. 1, January 1989, pp 109-112.

Rossingh, T.J. and Rooda, J.E.,
ROSKIT, a real-time operating system kit,
Research report, Technical University Twente, 1985.

Rooda, J.E.,
Discrete event simulation for the design and operation of logistics systems,
International Logistics Congress, San Fransisco, 1981.

Rooda, J.E.,
Procescalculus: systemen, modellen en formele talen (in Dutch),
I^2 Werktuigbouwkunde, August 1991, pp. 36-39.

Rooda, J.E.,
Procescalculus: definities en begrippen (in Dutch),
I^2 Werktuigbouwkunde, October 1991, pp. 35-40.

Srinivas, S.,
Error recovery in robots through failure reason analysis,
Proc. National Computer Conference, 1978, pp. 275-283.

Sun Microsystems,
Sun OS reference manual, Volume 2,
Sun Microsystems, Mountain View, 1990.

Szalas, A. and Szczepanska, D.,
Exception handling in parallel computations,
Sigplan Notices, October 1985, pp. 95-104.

Tennent, R.D.,
Principles of programming languages,
Prentice Hall International, Englewood Cliffs, 1981.

Watt, D.A.,
Programming language concepts and paradigms,
Prentice Hall, London, 1990.

Wirth, N.,
Programming in Modula-2,
3rd edition, Springer-Verlag, Berlin, 1985.

Wortmann, A.M.,
Modelling and simulation of industrial systems,
Dissertation, Eindhoven University of Technology, 1991.

Young, S.J.,
Real time languages: design and development,
Ellis Horwood Ltd., Chichester, 1982.

Appendix A
An introduction to Smalltalk-80

This appendix gives an introduction to the aspects of the Smalltalk-80 programming environment used in the example programs in this thesis. The text is not meant to be a general introduction to Smalltalk-80. For this the reader is referred to one of the many textbooks on Smalltalk, such as [Goldberg and Robson, 1989]. This introduction concentrates on the Smalltalk programming language; the interactive programming environment is not treated.

Classes and Instances

The Smalltalk-80 programming language is a true object-oriented programming language. Every object is an instance of a certain class. A class is comparable to a module implementing an abstract data type, as in Modula-2 [Wirth, 1985] or other Pascal-like languages that support modules. A class defines an abstract data type, together with some allowed operations on that data type.

The instance variables of the class define the internal representation, or private memory, of the data type. The definition of the instance variables of a class is comparable with a record type definition of an abstract data type in a module of Modula-2-like languages. All instances of a class have the same instance variables. The values of the instance variables, however, are private and usually differ between instances.

The operations that are defined in the class are called the methods of the class. They can be performed on the instances of the class and are comparable to the procedures implementing the functionality of an abstract data type of Modula-2 like languages.

Messages and Methods

Each object can access only its *own* instance variables and the instance variables of its superclasses. The object interfaces with the outside world by means of messages that can be sent to the object. Messages are sent in message expressions. A message expression consists of a receiver of the message and the message itself.

An example is self receive: true from: 'i-forkLifter-isUp'. In this expression, self is the receiver. The receiver is the object to which the message receive: true from: 'i-forkLifter-isUp' is sent. The message can have zero or more arguments. Arguments in messages are placed immediately after a colon (:). In this case, there are two actual arguments: the boolean object true and the string 'i-forkLifter-isUp'. The message selector is the message without the arguments. In this case receive:from:. The message selector, together with the receiver of the message, defines which method is invoked as a result of the evaluation of the message expression. The method that will be invoked is the method in the class of the receiver that has the same message selector. If, in the example, the class of the receiver self is the class Bubble, then the method receive: object from: portName as defined in the class Bubble would be invoked. The formal arguments object and portName would be set to the value of the actual arguments true and 'i-forkLifter-isUp'.

Another example is the message expression KillSignal raise. In this case the receiver is KillSignal. The message is raise and the message selector is also raise, because there are no arguments. KillSignal is a signal object. It is an instance of class Signal. Therefore, the message expression will result in the invocation of the method raise in the class Signal.

The variable self is a pseudo-variable. It can only be used in method definitions. No values can be assigned to pseudo-variables in assignment statements. When a certain method is executing, self refers to the receiver of the message which resulted in the execution of the method. So, the message expression KillSignal raise will cause the method raise in the class Signal to be executed, while the value of the pseudo-variable self, which can be used in the method raise, will refer to KillSignal.

Inheritance

Every class (except Object) has one (direct) superclass. A class inherits the instance variables and the methods from its superclasses. The superclasses of a class are the class's direct superclass, together with the superclasses of the superclass's direct superclass. A class can have several subclasses. In this way, a tree-like class structure is created where the class Object is at the root of the tree.

A superclass contains the instance variables and methods that are common to all of its subclasses. This approach makes it easy to reuse code. Apart from inheriting already defined methods from superclasses, methods can also be redefined.

When a message is sent to an object, the search for the method to be invoked starts in the object's class. If a corresponding method cannot be found there, the search is continued in the class's superclass and recursively in all of the other superclasses. If the method is not found in any of the superclasses an error results. In this case, the object is said not to understand the message.

The pseudo-variable super in a method refers to the receiver of the message which resulted in the invocation of the method. This is analogous to the meaning of the pseudo-variable self. The difference is that, when the pseudo-variable super is used as the receiver of a message, the search for the method to be invoked starts in the superclass of the class of the receiver of the message. An example of this is when the message expression super initializeTasks is found in a method initializeTasks. In this case, self cannot be used because this would result in endless recursion.

Variables

Five kinds of variable have been used in this thesis: globals, class variables, instance variables, arguments, and temporary variables.

Global variables are accessible throughout the system. They are written with a capital initial letter. All classes can be referred to by means of global variables.

Class variables are a kind of global variables, but they have a more restricted scope. Class variables also have an initial capital letter. The signals KillSignal and AnySignal that are used in this thesis are class variables.

There are two kinds of argument: method arguments and block arguments. Method arguments have already been treated in this appendix. Block arguments are treated in the next section.

Temporary variables are declared between bars, as in | temp1 temp2 |. They are usually declared at the beginning of a method and exist only during the invocation of the method.

Smalltalk is an untyped language. Therefore variables are untyped: all variables can refer to objects of an arbitrary class.

Blocks

A block represents a deferred sequence of actions. A block expression consists of a sequence of expressions separated by periods and delimited by square brackets. Block expressions can be seen as in-line function definitions. Blocks can be assigned to variables. Their expressions will only be executed when the message value is sent to the block.

A block can have one or more arguments. The formal arguments of a block are listed immediately after the opening bracket of the block and are prefixed by a colon. The argument declaration is terminated with the symbol '|'. Actual arguments are provided to a block by sending a block the message value: actualArgument or value: argument1 value: argument2 etc.

Consider the example given in Figure A.1. The time-out block acts as last argument to the method receive:from:within:ifTimedOut:. If a time-out occurs, then the time-out block will be executed by sending it the message value. This is done by the implementation of the method receive:from:within:ifTimedOut:, so it is not seen here. The exception handler is also a block. The exception handler block will only be executed when an exception is caught. The Smalltalk exception handling mechanism will set the formal argument exc of the exception handler block to the exception object which was created when the KillSignal was raised in the time-out block.

```
ForkLifterCtrl >> forkUp
    AnySignal
        handle:
            [:exc |
            self putOff: 'o-forklifter-power'.
            exc reject]
        do:
            [self putOn: 'o-forklifter-up'.
            self putOn: 'o-forklifter-power'.
            self
                receive: true
                from: 'i-forkLifter-isUp'
                within: 6 seconds
                ifTimedOut: [KillSignal raise].
            self putOff: 'o-forklifter-power']
```

Figure A.1 An example of the use of blocks.

Control Structures

Control structures are not a part of the language definition in Smalltalk-80. Selection and iteration are implemented using the classes Boolean with subclasses True and False and a class the instances of which are blocks. The predefined pseudo-variables true and false are the only instances of the respective classes True and False. Boolean expressions yield either the true or false object. Selection is achieved by means of the methods ifTrue:, ifFalse:, and ifTrue:ifFalse:. The arguments of these methods are blocks that are only executed in the case of the corresponding boolean receiver. For instance the message

```
(3 > 2) ifFalse: [...]
```

is analogous to

```
true ifFalse: [...].
```

Therefore, the block will not be executed. The message

```
(1 == 2) ifTrue: ["block A"] ifFalse: ["block B"]
```

will result in the execution of block B.

Some final syntactic issues

- Every Smalltalk method returns an object, even when no explicit return statement is specified. In this case, the method returns the receiver (self) of the message expression which resulted in the method invocation. Frequently, the returned object is not used. An explicit return statement is constructed by prefixing an expression with an up arrow: '↑'. For example, ↑'error'.
- The expression or statement separator in Smalltalk is the period.
- Strings are created by enclosing a sequence of characters by single quotes as in: 'an example string' or 'i-forkLifter-isUp'.
- Symbols can be constructed by prefixing identifiers with the character # as in #forkUp. Symbols are unique objects in the systems, strings are not.
- The two most important parsing rules are the following:
  Parsing is normally done from left to right. For example self maxStackSize negated evaluates as (self maxStackSize) negated.
  Messages without arguments have precedence over messages with arguments. For example self send: 'ok' to: self errorPort is evaluated as self send: 'ok' to: (self errorPort).

Appendix B
The semantics of the Smalltalk methods used in the program examples

This appendix gives a description of the most important Smalltalk methods that are used in the programs contained in this thesis. The methods from the Smalltalk system itself are given in Section B.1. Additional methods that support modeling using Process Calculus are given in Section B.2, and additions for exception handling in a Process Calculus environment are given in Section B.3. The methods that are used and explained in only one place in this thesis are not given here. For the methods that have already been properly defined in this thesis only a cross-reference to the section of the definition is given.

B.1 Methods from the Smalltalk system

------- Signal > handling -------

handle: handlerBlock do: doBlock
"Establish handlerBlock as an exception handler for the doBlock which will catch the exceptions represented by the signal (the receiver of the message). Explained in Section 4.5."

raise
"Raise the exception represented by the signal (the receiver). The result of this is the creation of an exception object (an instance of class Exception). A corresponding handler will be sought to catch the exception."

raiseErrorString: errorString
"Raise the exception represented by the signal. The errorString will be available in the created exception object (see Exception >> errorString)."

------- SignalCollection > handling -------

handle: handlerBlock do: doBlock
"Establish handlerBlock as an exception handler for the doBlock which will catch the exceptions represented by the signals in the signal collection (the receiver)."

------- Exception > handler responses -------

"The messages in this protocol are messages that can be sent to the exception object which acts as argument of the handler block of a handle:do: expression."

reject
"The current exception is propagated to the invoker of the handle:do: expression."

restart
"Restart the handle:do: expression."

return
"This is the default response from an exception handler. The handle:do: is terminated. Execution continues with the expression following the handle:do: statement."

errorString
"Return the error string that was given as argument to the signal which was raised to create the exception object. The signal could have been raised with the raiseErrorString: message."

------- OrderedCollection > adding -------

add: newObject
"Add newObject as the last element to the ordered collection represented by the receiver."

B.2 Methods for the modeling with Process Calculus

A number of the methods given here come from [Wortmann, 1991], these methods are indicated with [W] in the comment.

------- Bubble > process control -------

initializeTasks
"[W]. This method is called before any processor executes initialActions or body. It should not contain any send or receive actions, as the processes are not running yet. It is mainly intended to initialize instance variables."

initialActions
"[W]. This method is called once before the first execution of body."

body
"[W]. This method is called repeatedly during the execution of the process description of the model. It must be redefined by all subclasses."

------- Bubble > receiving objects -------

receiveFrom: portName
"[W]. The most basic receive action. Receive from the specified port. Block until some sender is available for communication. Return the item received."

receive: object from: portName
"Receive from the specified port if the item to be received equals object (using the = message). Block until object can be received.
If object is nil then any item will be received.
Return the item received."

receive: object from: portName within: interval ifTimedOut: timeOutBlock
"Receive from the specified port if the item to be received equals object (see receive:from:).
If an item is not received within interval, timeOutBlock is evaluated (no arguments)."

------- Bubble > sending objects -------

send: object to: portName
"[W]. The most basic send action. Send object synchronously to the port specified by portName. The process blocks until a matching receive is performed by another processor."

sendTo: portName
"Used for the purpose of synchronization only. Behaves just like send:to:, only sends an arbitrary object to portName."

send: object to: portName within: interval then: thenBlock else: elseBlock
"Try to send object to portName within interval. If that succeeds, evaluate the thenBlock, if it does not succeed, evaluate elseBlock."

send: object immediateTo: portName then: thenBlock else: elseBlock
"[W]. Try to send object to portName at this moment. If this succeeds, evaluate the thenBlock, if it does not succeed, evaluate elseBlock. So this send cannot block."

send: object immediateTo: portName
"[W]. Try to send object to portName at this moment. If this is not immediately possible, raise an exception. So this send cannot block."

broadcast: object to: portName
"Send copies of object to all receivers that are able to receive it at this moment. No special action is taken for those receivers that are connected to portName by means of an interaction path but could not receive the object. This method cannot block."

send: object continuousTo: portName
"[W]. Send object to portName. Copies of the object will be available for an unlimited number of receivers until the object is replaced by a new call to this method.
Never blocks."

------- Bubble > actuator interfacing -------

putOn: oActuatorPortName
"Activate the binary actuator represented by oActuatorPortName"
self send: true to: oActuatorPortName

putOff: oActuatorPortName
"Deactivate the binary actuator represented by oActuatorPortName"
self send: false to: oActuatorPortName

B.3 Methods for exception handling in Process Calculus

There are two global signals available for all processors. These are the KillSignal and the RetrySignal. They are explained in Section 7.2.
The class ControlBubble is a subclass of class Bubble. It contains the functionality for the control of processes that interface with operators by means of a specific Man Machine Interface (MMI).

------- ControlBubble > mmi interfacing -------

errorMessageFor: exceptionObject
"Send the error message contained in the exceptionObject (exceptionObject errorString) to the MMI."

restartMessage: restartString
"Send the given restartString to the operator by means of the MMI. Then wait for the command from the operator to continue."

------- Bubble > exceptions handlers -------

handle: exceptionHandler do: doBlock
"See Section 7.2. The exceptionHandler will catch all exceptions from the doBlock. Before execution of exceptionHandler, an error message will be issued for the exception. After execution of exceptionHandler, the handled exception will be propagated (ex reject), unless another response is specified in exceptionHandler."

handle: exceptionHandler constraintMonitors: monitorOrMonitors do: doBlock
"See Section 7.2. Just like the above given method. The difference is that the doBlock is protected by monitorOrMonitors."

------- ControlBubble > exceptions handlers -------

handle: exceptionHandler restart: restartBlock do: doBlock
"See Section 7.3 (and 7.5). Used to implement the retry strategy."

handle: exceptionHandler restart: restartBlock constraintMonitors: monitorOrMonitors do: doBlock
"See Section 7.3 (and 7.5). Used to implement the retry strategy. The doBlock is protected by monitorOrMonitors."

------- ControlBubble > process control -------

modeBody
"See Section 7.4. Processors that want to use the mode control (automatic mode, reset mode etc.) provided by the MMI interface, should define the method body in such a way that it only calls this method."

slaveBody
"See Section 7.4. Processors that act like slaves should define the method body in such a way that it only calls this method. A slave receives a command from another processor (the master), executes it, and sends an acknowledge back to the master."

automaticMode
"See Section 7.4. This method is executed due to a command from the operator causing the processor (the receiver of the automaticMode message) to change to automatic mode.
May be redefined by a subclass (copied to a subclass and then edited)."

automaticBody
"See Section 7.4. Must be redefined by a subclass to define the cyclic control sequence for the machine when in automatic mode."

------- ConstraintMonitor class > instance creation -------

for: processor receive: object from: portName then: thenBlock
"See Section 6.10.2. Define a constraint monitor. The constraint violation that can be monitored by the constraint monitor is defined to be the receipt of object from the port named portName on processor. The constraint violation will be signaled by making the thenBlock pending in processor. This amounts to creating a pending exception. The thenBlock must terminate by raising an exception. A pending thenBlock will be selected for execution when the processor executes an interaction or a delay, or when the method Bubble >> raisePendingException is called."

for: processor receiveFrom: aPortName then: thenBlock
"Similar to the previous method, only for this constraint monitor the constraint violation is defined to be the receipt of any object."

------- ConstraintMonitor > controlling -------

protect: block
"See Section 6.10.3. The argument block is bound to the constraint monitor. The monitor will be enabled during execution of the block."

------- ConstraintMonitor > accessing -------

item
"See Section 6.10.6. Return the item received by the constraint monitor (the receiver). Return nil if no item has been received since the constraint monitor was last enabled or cleared. A constraint monitor is cleared by the method ConstraintMonitor >> clearItem."

clearItem
"See Section 6.10.6. Similar to the previous method. Returns the item received by the constraint monitor, but also clears the constraint monitor."

------- ConstraintMonitorCollection > controlling -------

protect: block
"See Section 6.10.5. The argument block is bound to the collection of constraint monitors included in the collection represented by the receiver. The monitors will be enabled during execution of the block."

Index

Curriculum Vitae

Bert van Beek was born in Eindhoven, The Netherlands on the 25th. March 1958. He attended the Protestant Lyceum in Eindhoven where he gained his Atheneum-B certificate in 1976. In 1977, he was accepted into the Eindhoven University of Technology to study Electronic Engineering. Following a software project on the Graphical Kernel System, he graduated with distinction as a Master of Science in 1985. Soon thereafter, he was appointed as a Lecturer within the Faculty of Mechanical Engineering at the same University. Current activities include teaching and post-graduate research in languages, methods and techniques for the control of manufacturing systems.