European Journal of Control, nr. 2, 1996, pp. 88-100.

A New Mechanism for Exception Handling in Concurrent Control Systems

D.A. van Beek* and J.E. Rooda
Eindhoven University of Technology
Department of Mechanical Engineering
P.O. Box 513, 5600 MB Eindhoven
The Netherlands

Abstract

The most difficult aspect of concurrent discrete-event control is the handling of errors. Most present day languages for concurrent control system specification do not provide adequate mechanisms for exception handling, which is a major limitation on their effectiveness. In this paper, a new mechanism for exception handling in concurrently executing discrete-event control processes is treated, which simplifies the complex task of robust control system specification. The mechanism is based on constraint monitors, and can be used in conjunction with known mechanisms for exception handling in sequential programs. Constraints and constraint monitors are new concepts which are essential for dealing with exceptions in control systems. The constraints of a statement are conditions which must be valid throughout the execution of the statement. Constraint monitors are used to specify the constraints of a statement in a structured way, leading to programs in which the code for normal operation is separated from the code for exception handling. During the execution of the statement, the specified constraints are monitored at all encountered interaction points. If a constraint violation is detected, an exception is raised. In this way, the invariants of a process remain valid, finalization obligations of statements are executed, deadlock in the case of exception occurrences is prevented, and exceptions are not raised in processes in which no constraints have been violated. Constraint monitors are explained using a CSP-like language to which exception handling constructs have been added. The constructs have been chosen in such a way, that the resulting syntax and semantics are simple and especially suitable for the specification of robust control systems. The mechanism is finally illustrated by an example of the specification of a control system.

1. Introduction

An important aspect in the design of industrial control systems is the handling of errors in the controlled systems. Controlled physical systems undergo deterioration due to wear and aging; components have tolerances and robots suffer from imprecise positioning. Such characteristics lead to errors. The amount of code required for the recovery of such errors is usually many times greater than the amount needed to control the system under error-free circumstances ([1]).

An important concept which facilitates the handling of errors in a structured way is the concept of exception. Most mechanisms available in programming languages, or proposed in articles, are restricted to exceptions within a sequential process (e.g. [2], [3]). In this paper, we describe the functionality desired for the handling of exceptions in concurrently executing discrete-event control processes, and define a new mechanism which provides this functionality.

The concurrent programming language χ we use to specify control processes is based on the real-time CSP-like ([4]) language described in [5]. The language χ, and its application to the modelling of industrial systems, are described in [6]. In our paper, exception handling constructs have been added to the language. The resulting syntax and an informal explanation of the semantics of the language is given in Section 2. Section 3 deals with the basics of exception handling. In Section 4, we introduce the new concepts of constraint and constraint violation. In Section 5, the new mechanism, which is based on constraint monitors, is introduced, and the syntax and semantics of the mechanism is explained. This mechanism is primarily intended to deal with errors in the controlled system, and not with incorrect programs. In Section 6, the use of the mechanism is illustrated by an example. Finally, in Section 7, we describe the basic functionality of some representative known mechanisms for the handling of exceptions in a multi-process environment.

2. The language χ

The most important elements of the language χ are defined below, using an extended BNF notation. Non-terminals are represented in italics. Terminals are either symbols or identifiers in normal sans-serif type-face. Some non-terminals are not further defined: *c*, *v*, *exc*, and *pr* denote identifiers representing a channel, a variable, an exception, and a process, respectively; *exp* denotes an expression, *re* denotes a real expression, *b* denotes a boolean expression, *D* denotes a declaration, and *T* denotes a type. The braces in $\{X\}$ are BNF symbols indicating zero (empty) or more repetitions of the construct X. The notation $\{X\}^{0/1}$ indicates zero or one

* E-mail: vanbeek@wtb.tue.nl; phone: +31-40-2472892.
E-mail: rooda@wtb.tue.nl; phone: +31-40-2474553.
Fax: +31-40-2452505.

occurrence of X. The brackets [ ] are terminals.

$$
\begin{array}{lll}
I & ::= & c?v \mid c\tilde{} \\
Y & ::= & I \mid c!exp \mid \Delta re \\
GY & ::= & b; Y \rightarrow S \ \{[]\ b; Y \rightarrow S\} \\
GB & ::= & b \rightarrow S \ \{[]\ b \rightarrow S\} \\
S & ::= & \mathsf{skip} \mid v := exp \mid \triangleright exc \mid Y \\
 & & \mid [GY] \mid *[GY] \mid [GB] \mid *[GB] \\
 & & \mid [\![S \quad EH]\!] \\
 & & \mid [\![S \quad CM]\!] \\
 & & \mid S; S
\end{array}
\tag{2.1}
$$

$$
\begin{array}{lll}
EH & ::= & \mathsf{skip} \mid v := exp \mid \triangleright exc \mid Y \\
 & & \mid [GY_{EH}] \mid *[GY_{EH}] \mid [GB_{EH}] \mid *[GB_{EH}] \\
 & & \mid \curvearrowright \mid \curvearrowleft \mid \Diamond exc \\
 & & \mid EH; EH \\
GY_{EH} & ::= & b; Y \rightarrow EH \ \{[]\ b; Y \rightarrow EH\ \} \\
GB_{EH} & ::= & b \rightarrow EH \ \{[]\ b \rightarrow EH\}
\end{array}
$$

$$
\begin{array}{lll}
CM & ::= & b; I \rightarrow S_{CM} \ \{[]\ b; I \rightarrow S_{CM}\} \\
S_{CM} & ::= & \mathsf{skip} \mid v := exp \mid \triangleright exc \mid Y \\
 & & \mid [GY_{CM}] \mid *[GY_{CM}] \mid [GB_{CM}] \mid *[GB_{CM}] \\
 & & \mid S_{CM}; S_{CM} \\
GY_{CM} & ::= & b; Y \rightarrow S_{CM} \ \{[]\ b; Y \rightarrow S_{CM}\ \} \\
GB_{CM} & ::= & b \rightarrow S_{CM} \ \{[]\ b \rightarrow S_{CM}\ \}
\end{array}
\tag{2.2}
$$

$$
\begin{array}{lll}
cl & ::= & c\{, c\} : T \\
PR & ::= & \mathsf{proc}\ pr\ (cl\{, cl\}) = [\![\{D\}^{0/1}\ S]\!]
\end{array}
$$

In the sequel, the non-terminals b, c, exc, exp, v, CM, EH, GB, GY, I, S, Y (with or without subscripts) are also used to represent (the set of) language elements that can be constructed using the non-terminal as the start symbol.

The statements $[\![S \quad EH]\!]$ and $[\![S \quad CM]\!]$ may not be used inside exception handlers EH and/or constraint monitors CM, nor in any of the sub-statements occurring in handlers or constraint monitors. This requires the additional production rules GY_{EH}, GB_{EH}, S_{CM}, GY_{CM}, and GB_{CM}. They are very similar to the production rules S, GY, and GB. In fact the following relations exist: $S_{CM} \subset S$, $GY_{CM} \subset GY$, $GB_{CM} \subset GB$, and also $CM \subset GY$.

Some abbreviations are defined in order to achieve a more compact notation. By $S_1 :\equiv S_2$ we mean that S_1 may be abbreviated to S_2. The following abbreviations are used:

- $*[\text{true} \rightarrow S] \ :\equiv \ *[S]$.
- $\text{true}; Y \rightarrow S \ :\equiv \ Y \rightarrow S$.
- $Y \rightarrow \mathsf{skip} \ :\equiv \ Y$.
- $[\![[\![X]\!]]\!] \ :\equiv \ [\![X]\!]$, for $X \in \{S \quad EH, S \quad CM\}$.
- $[\![D\ [\![X]\!]]\!] \ :\equiv \ [\![D\ X]\!]$, for $X \in \{S \quad EH, S \quad CM\}$.

We refer to $[\![X]\!]$, where $X \in \{S, S \quad EH, S \quad CM\}$, as a block. We define $[]_{i=1}^{n}\ (X_i)$ as $X_1\ []\ X_2\ []\ \ \ []\ X_n$, for $n \geq 1$.

The semantics of many language constructs depends on whether the language is used as an abstract specification language, for instance for simulation purposes, or for actual real-time control. We treat (in an informal way) the semantics of the language χ from the viewpoint of actual real-time control.

Processes have local variables only; all interactions between processes take place by means of channels. A channel connects two processes or systems. Synchronization channels are symmetrical. Communication channels are used for output in one process and input in the other. Channels are declared in systems. In this paper, system definitions are represented graphically. Channels are also declared as process parameters, in which case the usage of the channel is declared as either output (e.g. c : !real), input (e.g. c : ?real), or synchronization (c : ˜void). A communication channel is represented graphically by an arrow, a synchronization channel by a line; processes are represented by circles.

The statements $c!exp$ and $c?v$, where c is a channel connecting two processes, denote communication by means of synchronous message passing. Execution of $c!exp$ in one process causes the process to be blocked until $c?v$ is executed in the other process, and vice versa. Subsequently the value of expression exp is assigned to variable v.

The statement $c\tilde{}$ denotes synchronization between two processes. Execution of $c\tilde{}$ in one process causes the process to be blocked until $c\tilde{}$ is executed in the other process.

The statement Δt denotes delays or time-outs. A process executing Δt is blocked for a duration of t seconds.

The statement $[GY]$, or $[]_{i=1}^{n}\ (b_i; Y_i \rightarrow S_i)]$ denotes selective waiting. The boolean expression b_i denotes a guard, which is open if b_i evaluates to true and is otherwise closed. An event statement Y_i which is prefixed by a guard b_i ($b_i; Y_i$) is enabled if the guard is open and the communication or synchronization action specified in Y_i can actually take place. If all guards are closed, the selective waiting statement terminates after evaluation of the guards. Otherwise, the process executing $[GY]$ remains blocked until at least one event statement is enabled. Then, one of the enabled event statements Y_i is chosen for execution, followed by execution of the corresponding S_i. Time-outs are event statements of the form Δt that are prefixed by a guard ($b; \Delta t$). A process cannot be blocked in a selective waiting statement with time-outs for longer than the smallest time-out time t_s (provided the associated time-out guard is open). If no other event statements are enabled within that period, a statement S associated with an enabled time-out of time t_s is executed.

The statement $[GB]$, or $[]_{i=1}^{n}\ (b_i \rightarrow S_i)]$, denotes selection. If all guards are closed, the statement terminates after evaluation of the guards. Otherwise, a statement S_i associated with an open guard b_i is executed.

The iteration statements $*[GB]$ and $*[GY]$ denote repeated execution of $[GB]$ and $[GY]$, respectively. The iteration terminates when all guards are closed.

Exceptions are denoted by identifiers and are declared in the following way: $ED ::= \mathsf{exception}\ exc\ \{, exc\}$. They are raised by means of the statement $\triangleright exc$. An exception handler is denoted by EH. The statement $[\![S \quad EH]\!]$ defines an exception handler EH for the statement S. The construct EH can also be referred to as the exception handler of the block $[\![S \quad EH]\!]$. The operator $\Diamond$ on exceptions is used to determine the exception raised last in a process. The retry statement $\curvearrowleft$ and the return statement $\curvearrowright$ are statements which cause an exception handler to terminate. Exception handling constructs are explained in more detail in Section 3.

The non-terminal CM denotes one or more constraint monitors. Constraint monitors are explained in Section 5. The statement $[\![S \quad CM]\!]$ defines constraint monitors CM for the statement S. A single constraint monitor is specified as $b; I \rightarrow S_{CM}$. There is an important restriction in the specification of the statement part S_{CM} of a constraint monitor: the execution of S_{CM} must terminate by raising an exception. An example of a valid constraint monitor is: $emg\tilde{} \rightarrow$

mmi!"*emergency*"; ▷*kill*, or $c^{\sim} \rightarrow [b_1 \rightarrow \triangleright e_1 \; [] \; b_2 \rightarrow \triangleright e_2]$. An example of an invalid (albeit syntactically correct) constraint monitor is: $emg^{\sim} \rightarrow$ *mmi*!"*emergency*".

Although most statements are allowed in exception handlers EH and in the statement parts S_{CM} of constraint monitors $b; I \rightarrow S_{CM}$, it is good programming practice to keep them simple. In fact, many constraint monitors are of the form true; $c^{\sim} \rightarrow \triangleright exc$, which may be abbreviated to $c^{\sim} \rightarrow \triangleright exc$.

Processes are defined by PR. All channels of a process must be declared as formal parameters in the parameter list. Declarations are separated from subsequent statements by the symbol | .

3. Basics of exception handling

We have chosen the exception handling constructs defined in Section 2 in such a way that the resulting syntax and semantics are simple, and especially suitable for the specification of robust control systems. In particular, our choice of constructs makes it easy to specify the *finalization obligations* of statements. Finalization obligations are actions which need to be executed when a statement is terminated with an exception, independently of the type of exception (for an example, see Section 4.2). Our exception handling mechanism is similar to that of Eiffel ([7]). Two differences are that Eiffel does not have a return response, and that where we use the ◊ operator to determine the exception raised last in a process, Eiffel imports the variable exception from class EXCEPTIONS.

An exception occurs when a program-unit in execution cannot achieve its goal. We define an *exception occurrence* to be a state of a sequential process, possibly combined with the state of the environment of the process, such that some program-unit which is being executed by the process cannot achieve its goal. In this definition, the environment of a process consists of all other processes and systems with which the process interacts. An *exception* is associated with a class of exception occurrences ([3]). If we consider correct programs only, then exception occurrences in a process are always caused by interaction of the process with other processes. An important difference between errors and exceptions is that the latter are used only in programs, whereas errors occur in both programs and physical systems. Errors in controlled machines usually cause exception occurrences in the control processes.

An example of an exception occurrence is the state of a process which is attempting to compute the square root of a negative number. Another example deals with a program-unit, the goal of which is to make a cylinder extend within 5 seconds. The state of the control process executing the program-unit, combined with the state of the controlled system, is an exception occurrence if the cylinder is blocked while extending, if there is no air pressure, or if the cylinder is not extended within 5 seconds for any other reason.

The semantics of our exception handling constructs is treated in an informal way. If an exception occurrence is detected, a corresponding exception should be raised. This causes a change in the normal control flow. The statement which is executed after the raising of an exception *exc* (▷*exc*), is the exception handler (EH) of the smallest block, enclosing the statement ▷*exc*, which contains a handler ($[\![S \quad EH]\!]$). In this case, the exception handler is said to have *caught* the raised exception *exc*. An exception handler may terminate in four different ways: with the propagate response (default); the return response (execution of ↷); the retry response (execution of ↶); or by raising an exception (execution of ▷*exc*). We have adopted the *termination model*, so that in all four cases the termination of the handler EH implies the simultaneous termination of $[\![S \quad EH]\!]$. The termination model is much simpler than the *resumption model*, in which the resume response from a handler causes execution to resume right after the statement ▷*exc* which raised the exception caught. We consider the resume response to be undesirable, because the exception handlers of *all* enclosing blocks may cause resumption at the statement which syntactically follows ▷*exc*. In order to be able to determine whether the resume response is acceptable, the handlers must be aware of the statements which follow ▷*exc*. This conflicts with the concepts of modularity and abstraction. Examples of programming languages which have adopted the termination model are Modular Pascal ([8]), and Ada ([9]).

We continue with the responses from a handler. Consider the program fragment $[\![S_1 \quad EH]\!]; S_2$. The handler EH terminates with the return response when the ↷ statement is executed in EH. The next statement to be executed is S_2. The handler terminates with the retry response when the ↶ statement is executed in EH. The result of this is that the statement $[\![S_1 \quad EH]\!]$ is first terminated, and then re-executed. Raising an exception *exc*' in the handler EH (execution of ▷*exc*'), causes the exception handler of the smallest block enclosing the statement $[\![S_1 \quad EH]\!]$ to be executed. It amounts to replacing the terminated statement $[\![S_1 \quad EH]\!]$ by the execution of ▷*exc*'. If neither the ↷, ↶, nor ▷*exc* statement is executed in EH, the handler terminates with the propagate response after termination of its last statement. The propagate response is therefore the default response of exception handlers. It causes the exception handler of the smallest enclosing block of $[\![S_1 \quad EH]\!]$ to be executed. The propagate response amounts to replacing the terminated statement $[\![S_1 \quad EH]\!]$ by execution of ▷*exc*, where *exc* is the exception caught by the handler. It is chosen to be the default response, because in this way, accidental omission of a response causes the exception to be propagated, leading to a controlled termination of statements. If, however, the return response were to be issued in this case, the program would be allowed to continue normally in an incorrect state with the likelihood of catastrophic consequences.

In order to be able to determine in a handler which exception has been raised, the operation ◊ is introduced. The boolean expression ◊*exc* evaluates to true if *exc* is the exception raised last in the process in which ◊*exc* is evaluated, and to false otherwise.

4. Constraints

4.1 Definitions

In this section, the concepts constraint and constraint violation are defined. These concepts are essential in order to determine the requirements of a mechanism for the handling of exceptions in control systems. Definitions of the two terms are given below.

A *constraint* of a program-unit is a condition over the state of the environment of the process executing the unit, which condition is invariant over the unit; it must be valid through-

out the execution of the unit in order that the unit may achieve its goal. The environment of a process consists of all other processes and systems with which the process interacts. In the rest of this paper, we use the term statement instead of the more general term program-unit.

A *constraint violation* is a state of the environment of the process executing a statement which state does not satisfy the statement s constraint.

An example of a constraint is that an emergency button must remain deactivated, or that a sensor detecting the entry of a person into a hazardous production area must remain deactivated during the execution of statements which activate the production processes in the production area.

The examples given above deal with constraint violations in a controlled system. A second type of constraint (violation) relates to the raising of exceptions in interacting control processes. Raising an exception in a process which interacts with another process causes deadlock if the other process remains blocked in a synchronization or communication action which will no longer succeed (due to the raised exception). In such a case, constraints of statements in both interacting processes can specify that certain statements in the other process must not be terminated prematurely with an exception. This kind of constraint (violation) is illustrated by the example in Section 6.

A constraint violation causes an exception occurrence in the process executing the statement of which a constraint is violated, because the constraint violation makes it impossible to achieve the goal of the statement. Therefore, an exception must be raised in a process when a constraint of an executing statement is violated.

4.2 Where to raise exceptions

In the case of a constraint violation, an exception must be raised in the process as soon as possible. It is undesirable, however, to raise such an exception immediately, regardless of the state of the process. Doing so could easily lead to errors in the state of the process, either because finalization obligations are not executed, or because invariants cannot be restored to a valid state. Consider, as an example of the first case, the following statement M:

$$\mathsf{M} = \quad [\![\, m!\text{true};\ [endPos^{\sim}\ []\ \Delta 10 \rightarrow \triangleright timeout];\ m!\text{false} \quad (4.1)$$
$$\qquad\quad m!\text{false}$$
$$\qquad [\!]$$

A vehicle is driven by a motor m to an end position indicated by a sensor *endPos*, where the motor is switched off. If an exception occurs, the motor is also switched off. Therefore, this action is a finalization obligation of the statement M (see Section 3). Suppose, that the vehicle does not reach the end position within 10 seconds, so that the *timeout* exception is raised. This will cause the statement m!false in the exception handler to be executed. Suppose, that the emergency button is pressed at the moment that execution of m!false is about to start in the exception handler. If this constraint violation would immediately result in the raising of an exception, the statement m!false would not be executed, causing M to terminate with the motor still running.

A second example deals with a statement S counting the number of times a synchronization action takes place:

$$\mathsf{S} = \quad count := 0;\ *[in^{\sim};\ count := count + 1] \quad (4.2)$$

In this case, the loop invariant specifies that *count* equals the number of times the synchronization action has taken place. An invariant of a statement is valid before and after execution of the statement. During the execution of the statement it can temporarily be invalid. If a constraint violation occurs while the invariant of a statement is temporarily invalid, an exception which causes the statement to be terminated may only be raised when the invariant can be made valid again in an exception handler. If, in the example, an exception due to a constraint violation would be raised just before the assignment in the loop, the loop invariant would be invalid, but it would be impossible to return the invariant to a valid state in an exception handler. The reason for this is that it is impossible to determine in a handler whether the exception due to the constraint violation was raised just before or after the assignment statement.

The need to raise exceptions caused by constraint violations as soon as possible, without causing errors, leads to a solution in which such exceptions are raised at interaction points, but not in exception handlers. An interaction point in our language is a statement of type Y or $[GY]$. Interaction points are chosen, because they are the only points at which processes can influence each other s state. Also, experience suggests that invariants either hold at interaction points, or else can reliably be restored in exception handlers. This is elaborated below.

Invariants that are related to the internal state of a process normally hold at interaction points. The loop invariant in (4.2) is an example of such an invariant. Another example is a process which repeatedly receives an object at an interaction point (e.g. $c?x$), and then adds the received object to a linked list. During the addition to the linked list, pointers need to be readjusted and invariants will temporarily be invalid. After addition to the list, the invariants will be valid again and other interactions may be executed.

Invariants that are related to the state of controlled processes need not hold at interaction points. In (4.1) the invariant specifies that the motor is off before and after execution of M. After execution of m!true in M, the invariant will be invalid temporarily. Such an invariant can be reliably restored in an exception handler. In the example the motor is simply switched off.

The computational delay introduced by raising the exceptions at interaction points only, is negligible for control systems where response times are crucial. This delay can never be greater than the time required for computations that take place from the termination of an interaction point until the initiation of the next interaction point. During these computations, the control process executing the computations cannot respond to changes in the controlled process, since interactions are the only means to synchronize with the controlled process. Therefore, in order to achieve adequate response times, real-time control programs should be designed in such a way that the computational delay is considerably smaller than the required response time of the control system.

5. Constraint monitors

Our new mechanism is based on constraint monitors. In a

constraint monitor $I \rightarrow S_{CM}$, the synchronization or input action I ($c\tilde{}$ or $c?v$) is used for the specification of the constraint. A constraint violation is detected when action I succeeds. As a result of this, the statement S_{CM} will be executed, terminating with an exception. Constraint monitors make it possible to specify the constraints of a statement (which may consist of a complex sequence of many other statements) only once. The specification of a constraint by means of a constraint monitor ensures that violations of the specified constraint are monitored at each interaction point that is executed during the execution of the statement, and that the corresponding exception will be raised at an interaction point when a constraint violation is detected.

A constraint monitor $b; I \rightarrow S_{CM}$ is defined as being active (or activated) in a process, when that process is executing a statement $[\![S \quad b; I \rightarrow S_{CM}]\!]$, and b evaluates to true. Consider the program fragment

$$S_3; [\![[\![S_4; S_5 \quad b_1; I_1 \rightarrow S_{CM1}]\!]; S_6 \quad b_2; I_2 \rightarrow S_{CM2}]\!].$$

If a process A is executing the statement S_4 or S_5, and b_1 and b_2 are true, then the constraint monitors $b_1; I_1 \rightarrow S_{CM1}$, and $b_2; I_2 \rightarrow S_{CM2}$, are both active in A.

Consider a channel c connecting the processes A and B. If a constraint monitor $c\tilde{} \rightarrow \triangleright exc$ is active in A, execution of $c\tilde{}$ in B will cause the exception exc in A to be raised when the next interaction point in A is executed (assuming that the constraint monitor $c\tilde{} \rightarrow \triangleright exc$ is still active in A at that time, and that the interaction point is not in an exception handler).

5.1 Semantics

The semantics of a block with constraint monitors $[\![S \quad CM]\!]$ is defined by means of a set of rewrite rules, which enable us to rewrite the block $[\![S \quad CM]\!]$ in terms of other statements which no longer contain blocks with constraint monitors. For this purpose, the language is extended with a prioritized selective waiting statement $[\succ_{i=1}^{n} (GY_i)]$, where

$$\succ_{i=1}^{n} (GY_i) = GY_1 \succ GY_2 \succ \quad \succ GY_n.$$

The statement $[\succ_{i=1}^{n} (GY_i)]$ is intended to be used only in the rewrite rules for the definition of the semantics of $[\![S \quad CM]\!]$. It is not intended to be used in programs. The statement is a selective waiting statement with priority. It makes use of a new operator $\succ$. The semantics of the statement is similar to that of a normal selective waiting statement, which is obtained by replacing the $\succ$ operator in

$$[GY_1 \succ GY_2 \succ \quad \succ GY_n] \tag{5.1}$$

by the $[\!]$ operator:

$$[GY_1 \,[\!]\, GY_2 \,[\!] \quad [\!]\, GY_n]. \tag{5.2}$$

The only semantic difference between (5.1) and (5.2) is that in (5.1), the order of GY_1 to GY_n is relevant: the priority decreases from left to right. Therefore, in (5.1) the priority of GY_x is higher than the priority of GY_y, if $x < y$. If there is none or only one GY (where GY can be any one of GY_1 to GY_n) with enabled event statements, the statements in (5.1) and (5.2) are semantically equivalent. If there is more than one GY with enabled event statements in (5.1), the GY which is chosen from these, is the one with the highest priority. If the GY chosen has itself more than one enabled event statement, then one of these is chosen in the same way as for a normal selective waiting statement $[GY]$.

A second extension of the language for the purpose of rewriting blocks with constraint monitors is realized by redefining the production rule (2.2) for constraint monitors:

$$CM \ ::= \ b; I \rightarrow S_{CM} \,\{[\!]\, b; I \rightarrow S_{CM}\,\} \mid CM \succ CM. \tag{5.3}$$

In this way the constraint monitors defined by production rule (2.2) are supplemented by constraint monitors of the form

$$CM_1 \succ CM_2 \succ \quad \succ CM_n.$$

This kind of constraint monitor occurs only as a result of applying rewrite rule (5.10). It cannot be used directly in χ programs.

A block $[\![S \quad CM]\!]$ can be rewritten in a form in which the constraint monitors CM appear only in selective waiting statements of the form $[\succ_{i=1}^{n} (GY_i)]$ by means of the rewrite rules (5.4) to (5.10). The statements of type S (S, S_1, S_2, S_i, and S'), that are used in these rules, can be any statement that can be constructed with production rule (2.1). The constraint monitors CM represent the constraint monitors defined by production rule (5.3). The rewrite rules are described below.

Assignments and the raising of exceptions are not affected by constraint monitors:

$$[\![X \quad CM]\!] = X \ \text{ for } \ X \in \{\mathsf{skip}, v := exp, \ \triangleright exc\}. \tag{5.4}$$

Synchronization statements Y with constraint monitors are transformed into prioritized selective waiting statements:

$$[\![Y \quad CM]\!] = [CM \succ \text{true}; Y \rightarrow \mathsf{skip}] = [CM \succ Y]. \tag{5.5}$$

Selective waiting statements $[GY]$ with constraint monitors are rewritten as follows (likewise for $*[\,]$):

$$[\![[[\!]_{i=1}^{n} (b_i; Y_i \rightarrow S_i)] \quad CM]\!] =$$
$$[CM \succ [\!]_{i=1}^{n} (b_i; Y_i \rightarrow [\![S_i \quad CM]\!])]. \tag{5.6}$$

Constraint monitors affect only the statement parts of selection statements $[GB]$ (likewise for $*[\,]$):

$$[\![[[\!]_{i=1}^{n} (b_i \rightarrow S_i)] \quad CM]\!] = [[\!]_{i=1}^{n} (b_i \rightarrow [\![S_i \quad CM]\!])] \tag{5.7}$$

Exception handlers are not affected by constraint monitors:

$$[\![[\![S \quad EH]\!] \quad CM]\!] = [\![[\![S \quad CM]\!] \quad EH]\!]. \tag{5.8}$$

The reason for leaving the exception handler EH unaffected by the constraint monitors CM, is that an important aim of an exception handler is to perform finalization obligations. Many such actions can be executed only by means of an interaction, such as switching off a motor. Such an interaction must be executed under all circumstances. Therefore, a constraint monitor may never cause an exception to be raised in the place of the execution of such interactions in exception handlers.

A sequential composition of statements with constraint monitors is rewritten as follows:

$$[\![S_1 \,; S_2 \quad CM]\!] = [\![S_1 \quad CM]\!] \,; [\![S_2 \quad CM]\!]. \tag{5.9}$$

When nesting constraint monitors, the enclosed constraint monitors (CM_1 in (5.10)) are not affected by the enclosing constraint monitors (CM_2 in (5.10)). The enclosing monitors take priority, and are therefore inserted before the enclosed monitors:

$$[\![[\![S \quad CM_1]\!] \quad CM_2]\!] = [\![S \quad CM_2 \succ CM_1]\!]. \tag{5.10}$$

The reason why enclosed constraint monitors are not affected by enclosing constraint monitors is similar to the reason why exception handlers are not affected by enclosing constraint monitors. We wish to be able to guarantee that the execution of the statement S_{CM} of a constraint monitor $b; I \rightarrow S_{CM}$ cannot be aborted prematurely due to an enclosing constraint monitor. The effect of rules (5.8) and (5.10) is that exceptions due to violations of constraints specified in enclosing constraint monitors are not raised in enclosed exception handlers, nor in the statement parts of enclosed constraint monitors.

Although in some cases there need not be any difference in priority between enclosing and enclosed constraint monitors, there are many cases where the priority is important. This is explained by means of an example. Consider the statement $[\![S \quad CM_2]\!]$, where

$$S = S_a; [\![[\![Y \quad CM_1]\!] \quad EH]\!]; S_b. \tag{5.11}$$

The enclosing constraint monitor is CM_2; the enclosed constraint monitor CM_1. By substitution, the statement is rewritten as

$$[\![S_a; [\![[\![Y \quad CM_1]\!] \quad EH]\!]; S_b \quad CM_2]\!].$$

Using (5.9) this is rewritten as

$$[\![S_a \quad CM_2]\!]; [\![[\![[\![Y \quad CM_1]\!] \quad EH]\!] \quad CM_2]\!]; [\![S_b \quad CM_2]\!].$$

By applying (5.8), the middle statement

$$[\![[\![[\![Y \quad CM_1]\!] \quad EH]\!] \quad CM_2]\!]$$

is rewritten as

$$[\![[\![[\![Y \quad CM_1]\!] \quad CM_2]\!] \quad EH]\!].$$

Using (5.10), followed by (5.5) yields

$$[\![[CM_2 \succ CM_1 \succ Y] \quad EH]\!]. \tag{5.12}$$

In (5.12), the enclosing constraint monitor CM_2 has a higher priority than CM_1, because it is listed first. The reason why CM_2 should have a higher priority than CM_1 is described below.

Suppose that, during execution of Y, the actions I_1 and I_2 of both constraint monitors can succeed (let $CM_1 = I_1 \rightarrow \triangleright e_1$, and $CM_2 = I_2 \rightarrow \triangleright e_2$). This means that both constraints are violated. If I_1 is chosen, the exception e_1 will be raised. This exception would not necessarily have to be propagated by EH; EH could issue a return or retry response. Such a return or retry response would be undesirable, because the constraint violation detected by CM_2 requires that the statement S in (5.11) is terminated as soon as possible with exception e_2. Therefore, the action I_2 of CM_2 should be chosen, causing exception e_2 to be raised. The exception handler EH must propagate this exception (after executing the finalization obligations), because the exception must cause the termination of S.

Using the rewrite rules (5.4) to (5.10), any syntactically correct block $[\![S \quad CM]\!]$ can be rewritten in terms of statements which no longer contain blocks with constraint monitors. This is shown below.

All statements S are constructed using the production rule S, which is defined in (2.1) and is repeated below.

$$\begin{aligned} S \quad ::= \quad & \mathsf{skip} \mid v := exp \mid \triangleright exc \mid Y \\ & \mid [GY] \mid *[GY] \mid [GB] \mid *[GB] \\ & \mid [\![S \quad EH]\!] \mid S; S \mid [\![S \quad CM]\!] \end{aligned}$$

For the first three alternatives ($\mathsf{skip} \mid v := exp \mid \triangleright exc$), application of rewrite rule (5.4) on $[\![S \quad CM]\!]$ yields S. For the subsequent alternatives, the rewrite rules (5.5) to (5.10) are used. The result of rewriting $[\![S \quad CM]\!]$ after application of one rewrite rule, is that the constraint monitor CM is removed (rule (5.4)), appears in a prioritized selective waiting statement (rules (5.5) and (5.6)), or appears in one or more blocks $[\![S' \quad CM']\!]$ (rules (5.6) to (5.10)), where S' represents a sub-statement used in S. The constraint monitor CM' is different from CM after application of rule (5.10), and equal to CM in all other cases. By recursively rewriting the blocks $[\![S' \quad CM']\!]$, the resulting statement will eventually no longer contain blocks with constraint monitors. The constraint monitors then occur in prioritized selective waiting statements only. Production rules (2.2) and (5.3) for constraint monitors ensure that blocks with constraint monitors cannot be used in the statement part S_{CM} of constraint monitors.

5.2 Implementation

Based on the rewrite rules described above, an implementation of constraint monitors in χ can be realized. All active constraint monitors are stored in a list L. When the execution of a statement of type $[\![S \quad CM]\!]$ is initiated, the constraint monitor CM is added to the head of list L. Subsequently the statement S is executed. When $[\![S \quad CM]\!]$ is terminated, CM is removed from L. If the list L is not empty, the execution of statements of type Y and $[GY]$ is changed in the way described below. The execution of all other statements is not affected by the constraint monitors of L.

A statement of type Y is executed as a prioritized selective waiting statement

$$[CM_1 \succ CM_2 \succ \quad \succ CM_n \succ Y], \tag{5.13}$$

where n is the size of list L. The constraint monitors CM_1 to CM_n are the constraint monitors occurring in list L in that order, CM_1 being the head of the list.

A statement of type $[GY]$ is likewise executed as a prioritized selective waiting statement

$$[CM_1 \succ CM_2 \succ \quad \succ CM_n \succ GY]. \tag{5.14}$$

The construct GY may also be written as $[]_{i=1}^{n} (b_i; Y_i \rightarrow S_i)$. If execution of (5.14) results in execution of one of the statements S_i occurring in GY, such a statement S_i is also executed using the constraint monitors in list L.

Rule (5.8) specifies that statements executed in exception handlers are not affected by constraint monitors. Therefore the execution of statements of type Y and $[GY]$ is not affected by the constraint monitors of L if the statements are being executed in exception handlers.

6. An example

In this section, the use of constraint monitors is illustrated using the control system of a simple robot that transports products from workstation A to B. The specification of the ideal control system without errors is given first, followed by the specification of the control system with exception handling.

The interface with sensors in the physical system is realized

by means of channels in the model. If a channel has the same name as a sensor, a synchronization action on the channel succeeds when the sensor is on (activated). If the channel name is prefixed with *not* (such as in *notvac*) a synchronization action succeeds when the sensor is off (not activated). Channels which are associated with actuators have the same names as the corresponding actuators.

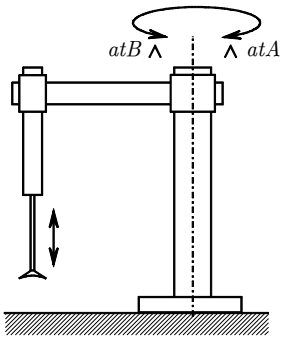

Figure 1

The robot is shown in Figure 1. A turning cylinder with actuators *aToA* and *aToB* turns the robot from position A to B. The cylinder is freely moveable when its actuators are not activated. A product can be picked up by means of a suction cup, which is activated by actuator aV. The sensor *vac* detects the presence of vacuum in the suction cup. The product can be lifted by means of a cylinder. Initially, the robot is assumed to be waiting at workstation A with the suction cup up, and no vacuum applied.

An informal graphical representation of the structure of the control system is shown in Figure 2. Circles denote processes; lines and arrows denote channels. Only the channels required for interaction between the control processes are shown. The following conventions are used in the channel names: *req* for request and *done* for done; *com* for command and *ack* for acknowledge; the subscripts A, B, V and T for *WSA*, *WSB*, *Vacuum* and *Transport*, respectively.

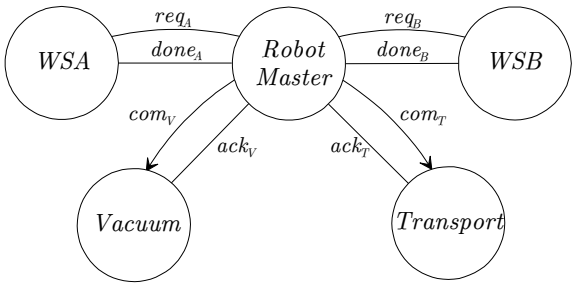

Figure 2

6.1 No exception handling

The specification of the control processes without exception handling is given below. The specification is not complete; only the most relevant parts are shown.

Each of the processes *WSA*, *RobotMaster* and *WSB* has its own start button which is pressed by the operator in order to start the process. The types vCom and tCom are enumerated types. They are defined as type vCom = ON | OFF, and type tCom = UP | DOWN | TOA | TOB. The type bool denotes a boolean.

proc *WSA* ($start_A$, req_A, $done_A$: ˜void) =
$[\![start_A\tilde{}; *[\mathsf{receiveParts}; \mathsf{assembleProduct}; req_A\tilde{}; done_A\tilde{}]]\!]$

proc *RobotMaster*
($start_R$, req_A, $done_A$, req_B, $done_B$, ack_V, ack_T : ˜void,
com_V : !vCom, com_T : !tCom) =

$[\![$ $start_R\tilde{}$;
$*[$ $req_A\tilde{}$; pick; $done_A\tilde{}$;
com_T!TOB; $ack_T\tilde{}$;
$req_B\tilde{}$; place; $done_B\tilde{}$;
com_T!TOA; $ack_T\tilde{}$
$]$
$]\!]$,

where pick = com_T!DOWN; $ack_T\tilde{}$; com_V!ON; $ack_V\tilde{}$;
com_T!UP; $ack_T\tilde{}$,

and place = com_T!DOWN; $ack_T\tilde{}$; com_V!OFF; $ack_V\tilde{}$;
com_T!UP; $ack_T\tilde{}$.

The processes *Vacuum* and *Transport* are slaves. Slaves receive a command from their master (*RobotMaster*), execute the command by operating actuators and reading sensors, and consequently signal an acknowledge to the master. The specification of the process *Vacuum* is as follows:

proc *Vacuum*
(com_V : ?vCom, ack_V, *vac*, *notvac* : ˜void, aV : !bool) =
$[\![x : vCom \mid *[com_V?x; \mathsf{doX}; ack_V\tilde{}]]\!]$,

where doX =
$[x = \text{ON} \rightarrow aV!\text{true}; vac\tilde{} \;[]\; x = \text{OFF} \rightarrow aV!\text{false}; notvac\tilde{}]$.

6.2 Exception handling using constraint monitors

In Figure 3, the channels necessary for exception handling are shown. The complete model is a combination of the models shown in Figures 2 and 3. Each of the processes *WSA*, *RobotMaster* and *WSB* has its own emergency button. The channels emg_A, emg_R, and emg_B are associated with the respective emergency buttons. The following additional conventions are used in the channel names: e_{RA} for exception from *RobotMaster* to *WSA*; e_{AR} for exception from *WSA* to *RobotMaster*; and likewise for the other channels.

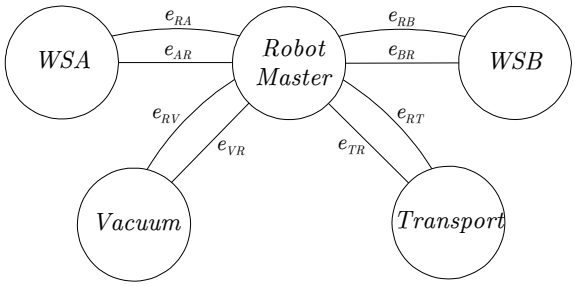

Figure 3

If an exception occurs in a process, the process will probably no longer be able to perform the interactions that other processes expect of it. This could easily lead to deadlock. Therefore, for every control process, say A, an exception handler is defined at the outermost level. In this handler, A signals the other processes (using $[e\tilde{} \;[]\; \Delta t_{handle}]$ or similar statements such as $*[e_a\tilde{} \;[]\; e_b\tilde{} \;[]\; e_c\tilde{} \;[]\; \Delta t_{handle} \rightarrow \curvearrowleft]$) with which it interacts, that an exception has occurred. For every channel e which is used in such a $[e\tilde{} \;[]\; \Delta t_{handle}]$ or similar statement, there is a corresponding constraint monitor in another process, say B, which uses the same channel (e.g. $e\tilde{} \rightarrow \triangleright exc$). If the synchronization actions $e\tilde{}$ succeed in both processes, the exception *exc* will be raised in B. If, however, the constraint monitor $e\tilde{} \rightarrow \triangleright exc$ is not active in process B at

the time of execution of $[e^{\sim} \,[]\, \Delta t_{handle}]$ in A, the time-out Δt_{handle} will occur. In such a case, the exception occurrence in process A did not cause a constraint violation in B. The time-out Δt_{handle} prevents A from remaining blocked in $e^{\sim}$ when the corresponding constraint monitor (e.g. $e^{\sim} \rightarrow \triangleright exc$) is not active in B. The time t_{handle} should be larger than the maximum computational delay between interaction points (see Section 4.2), and larger than the maximum amount of time that a process may need for exception handling in the statement parts of constraint monitors and in exception handlers. The reason for this is that constraints are not monitored during that time. Exception handlers and statement parts of constraint monitors should be simple and not execute synchronization statements which may block. This will ensure that exception occurrences are handled instantly, and that the time t_{handle} can be small (infinitely small in the case of infinitely fast control processes).

For exception handling, the exceptions *error* and *kill* are declared as follows: exception *error*, *kill*. These exceptions are declared globally, so that they are available in all processes. The exception *error* is raised due to an error in the process itself, or in the controlled components. The exception *kill* is raised due to an exception in another control process, in order to prevent incorrect synchronization. If an error is detected, an error message is sent to the man-machine interface. For this purpose, all control processes use a different channel mmi_X. These channels that connect the processes to the man-machine interface process are not shown in the system. In the following specification, only the elements that have changed in comparison to the specification without exception handling are shown.

Reset actions of the control processes after an error are not specified. It is assumed that the operator resets the controlled system prior to pushing the different start buttons of the control processes.

In the exception handler of *WSA* (see (6.1)), *RobotMaster* is notified of the exception in *WSA*, after which a retry response is given.

proc *WSA*
($start_A$, emg_A, req_A, $done_A$, e_{RA}, e_{AR} : ˜void, mmi_A : string)
=
⟦ $start_A{}^{\sim}$;
⟦ ∗[body]
$emg_A{}^{\sim} \rightarrow mmi_A$!"*emergency in WSA*"; ▷*error*
⟧
$[e_{AR}{}^{\sim} \,[]\, \Delta t_{handle}]$; ↶ (6.1)
⟧,

where body =
receiveParts; assemble; $req_A{}^{\sim}$; ⟦$done_A{}^{\sim}$ $e_{RA}{}^{\sim} \rightarrow \triangleright kill$⟧. (6.2)

In order to prevent deadlock, the constraint monitor $e_{RA}{}^{\sim} \rightarrow \triangleright kill$ in *WSA* (see (6.2)) raises the exception *kill* if, while waiting for the acknowledge $done_A{}^{\sim}$ in (6.2), an exception occurs in *RobotMaster* (e.g. due to the detection of an error in process *Vacuum* or *Transport* when a product is picked up). The synchronization $req_A{}^{\sim}$ in (6.2) cannot lead to deadlock and is therefore not included in the block with the constraint monitor.

In *RobotMaster*, two constraint monitors (see (6.4) and (6.5)) are specified, which monitor exception occurrences in the slaves *Vacuum* and *Transport*. In the exception handler, all processes with which *RobotMaster* communicates are notified of the exception occurrence in *RobotMaster* (see (6.6)). The state of every process which is notified determines whether an exception will actually be raised in the process. If, at the time of execution of the exception handler in (6.6), a constraint monitor $e_{RX}{}^{\sim} \rightarrow \triangleright exc$ is active in such a process (X equals either A, B, V or T), success of the synchronization action $e_{RX}{}^{\sim}$ will cause the exception $\triangleright exc$ to be raised.

In order to prevent deadlock, the constraint monitor $e_{AR}{}^{\sim} \rightarrow \triangleright kill$ in *RobotMaster* (see (6.7)) raises the exception *kill* if, during the execution of pick; $done_A{}^{\sim}$, an exception occurs in *WSA* (e.g. due to pressing emergency button emg_A).

proc *RobotMaster* (6.3)
($start_R$, emg_R, req_A, $done_A$, req_B, $done_B$, ack_V, ack_T,
e_{AR}, e_{BR}, e_{VR}, e_{TR}, e_{RA}, e_{RB}, e_{RV}, e_{RT} : ˜void,
com_V : !vCom, com_T : !tCom, mmi_R : !string) =
⟦ $start_R{}^{\sim}$;
⟦ ∗[body]
$emg_R{}^{\sim} \rightarrow mmi_R$!"*emergency in robot*"; ▷*error*
[] $e_{VR}{}^{\sim} \rightarrow \triangleright kill$ (6.4)
[] $e_{TR}{}^{\sim} \rightarrow \triangleright kill$ (6.5)
⟧
∗$[e_{RA}{}^{\sim} \,[]\, e_{RB}{}^{\sim} \,[]\, e_{RV}{}^{\sim} \,[]\, e_{RT}{}^{\sim} \,[]\, \Delta t_{handle} \rightarrow \curvearrowleft]$ (6.6)
⟧,

where body =
$req_A{}^{\sim}$; ⟦pick; $done_A{}^{\sim}$ $e_{AR}{}^{\sim} \rightarrow \triangleright kill$⟧; (6.7)
com_T!*toB*; $ack_T{}^{\sim}$;
$req_B{}^{\sim}$; ⟦place; $done_B{}^{\sim}$ $e_{BR}{}^{\sim} \rightarrow \triangleright kill$⟧;
com_T!*toA*; $ack_T{}^{\sim}$,

and pick = com_T!DOWN; $ack_T{}^{\sim}$; com_V!ON; $ack_V{}^{\sim}$;
com_T!UP; $ack_T{}^{\sim}$.

In the process *Vacuum*, the status of the vacuum in the suction cup is kept in the variable *vacOn*. Initially, and after an exception occurrence, the vacuum is assumed to be off. The operator must see to it that these assumptions are satisfied. Alternatively, the assumptions could be tested in the process *Vacuum*. In the handler (see (6.8)), the master (*RobotMaster*) is notified of an exception occurrence only if the exception caught is not a *kill*, because a *kill* exception occurrence in *Vacuum* is caused by an exception occurrence in the master itself (see the constraint monitor $e_{RV}{}^{\sim} \rightarrow \triangleright kill$ in (6.11)).

proc *Vacuum*
(com_V : ?vCom, ack_V, *vac*, *notvac*, e_{RV}, e_{VR} : ˜void,
aV : !bool, mmi_V : !string) =
⟦ x : *vCom*; *vacOn*: bool
| *vacOn* := false; ∗[body]
$[\neg\Diamond kill \rightarrow [e_{VR}{}^{\sim} \,[]\, \Delta t_{handle}]]$; ↶ (6.8)
⟧,

where body =
⟦$com_V?x$ *vacOn*; *notvac*$^{\sim} \rightarrow$ vacError⟧; (6.9)
⟦ [x = ON → doVacOn [] x = OFF → doVacOff];
⟦$ack_V{}^{\sim}$ *vacOn*; *notvac*$^{\sim} \rightarrow$ vacError⟧ (6.10)
$e_{RV}{}^{\sim} \rightarrow \triangleright kill$ (6.11)
⟧,

and vacError =
aV!false; *vacOn* := false; mmi_V!"*no vacuum*"; ▷*error*, (6.12)

and doVacOn =
aV!true; *vacOn* := true;
[*vac*$^{\sim}$ [] Δ0.5 → mmi_V!"*time-out vacuum on*"; ▷*error*].

The constraint monitors $vacOn; notvac^{\sim} \rightarrow$ vacError in (6.9) and (6.10) monitor the presence of the vacuum in the case of the vacuum being applied ($vacOn$ = true) and the suction cup holding a product. If, during transportation of the product from *WSA* to *WSB*, the product is dropped and the vacuum is no longer present, vacError is executed in (6.9) or (6.10), resulting in the raising of the exception *error* in (6.12). This exception will be signaled to *RobotMaster* by $e_{VR}{}^{\sim}$ in (6.8). This will cause the constraint monitor $e_{VR}{}^{\sim} \rightarrow \triangleright kill$ of *RobotMaster* in (6.4) to raise the exception *kill*. The slave *Transport* is notified of the exception by $e_{RT}{}^{\sim}$ in (6.6), causing the constraint monitor $e_{RT}{}^{\sim} \rightarrow \triangleright kill$ in *Transport* (see (6.14)) to raise an exception and consequently stop any movements of the robot arm (for example in handler (6.15)). The structure of the specification of *Transport* is similar to the specification of *Vacuum* because both processes are slaves.

proc *Transport*( ) =
⟦ $x : tCom$
| ∗[body]
$[\neg\Diamond kill \rightarrow [e_{TR}{}^{\sim} \,[]\, \Delta t_{handle}]]; \curvearrowleft$ (6.13)
⟧,

where body =
$com_T?x$;
⟦ $[x = \text{UP} \rightarrow$ doUp $[]$ $[]$ $x = \text{TOB} \rightarrow$ gotoB];
$ack_T{}^{\sim}$
$e_{RT}{}^{\sim} \rightarrow \triangleright kill$ (6.14)
⟧,

and gotoB =
⟦ $aToA$!false; $aToB$!true;
$[\![atB^{\sim} \,[]\, \Delta 4 \rightarrow mmi!\text{"}time\text{-}out\ going\ to\ WSB\text{"}; \triangleright error]\!]$
$aToA$!false; $aToB$!false (6.15)
⟧.

6.3 Exception handling without constraint monitors

In this section the control process *RobotMaster* (see (6.3)) is specified without the use of constraint monitors. Prioritized selective waiting statements are used to monitor the constraints. The specification illustrates how the code for normal operation and the code for exception handling get severely mixed up when constraint monitors are not used. This is due to the fact that the constraints need now be specified separately for each interaction. In order to save space, only the first part of body is specified.

proc *RobotMaster*( ) =
⟦ $start_R{}^{\sim}$; ∗[body]
$*[e_{RA}{}^{\sim} \,[]\, e_{RB}{}^{\sim} \,[]\, e_{RV}{}^{\sim} \,[]\, e_{RT}{}^{\sim} \,[]\, \Delta t_{handle} \rightarrow \curvearrowleft]$
⟧,

where body =
$[emg_R{}^{\sim} \rightarrow$ em $\succ req_A{}^{\sim}]$;
pick;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{VR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \succ done_A{}^{\sim}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{VR}{}^{\sim} \rightarrow \triangleright kill \succ com_T!\text{TOB}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{VR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{TR}{}^{\sim} \rightarrow \triangleright kill \succ ack_T{}^{\sim}]$;
,

and pick =
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \succ com_T!\text{DOWN}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{TR}{}^{\sim} \rightarrow \triangleright kill \succ ack_T{}^{\sim}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \succ com_V!\text{ON}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{VR}{}^{\sim} \rightarrow \triangleright kill \succ ack_V{}^{\sim}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{VR}{}^{\sim} \rightarrow \triangleright kill \succ com_T!\text{UP}]$;
$[emg_R{}^{\sim} \rightarrow$ em $[]\, e_{AR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{VR}{}^{\sim} \rightarrow \triangleright kill \,[]\, e_{TR}{}^{\sim} \rightarrow \triangleright kill \succ ack_T{}^{\sim}]$,

and em = $mmi_R!\text{"}emergency\ in\ robot\text{"}; \triangleright error$.

The constraints which need to be monitored must be respecified at each interaction point. Compare this with the specification of the *RobotMaster* process in (6.3), where each constraint monitor needs to be specified only once. In the latter specification, some constraint monitors such as $e_{TR}{}^{\sim} \rightarrow \triangleright kill$ (see (6.5)) will be active during the complete execution of body. In the specification above, it is clear that this constraint monitor needs to be active only when a product is actually being transported by the *Transport* control process. The specification of this constraint monitor in one place (6.5) only, however, simplifies the specification considerably. The constraint monitor can only raise the exception *kill* when transportation of a product causes an exception occurrence to be signalled to *RobotMaster* in (6.13).

7. Related work

Programming languages such as Real-time Euclid ([10]), and operating systems such as VAXELN ([11]), and ROSKIT ([12]), support the explicit raising of exceptions in other processes. The problem with this approach is that it is impossible, and in view of modularity undesirable, to know the exact state of other processes. Thus, the use of such a mechanism may lead to exceptions being raised in other processes even though no constraints are actually violated in those processes. In [13] and [14], mechanisms are proposed in which the handler for an exception can reside in a different process to that in which the exception was raised. We consider the use of these mechanisms to be in conflict with the definition and spirit of exceptions, because the mechanisms will cause exceptions to be handled in processes in which there is no exception occurrence. The mechanisms also introduce unnecessary complexity. In [15], [16], and [17], proposals are treated which deal with exceptions in parallel constructs, such as the parallel statement in CSP ([4]). These mechanisms deal only with the *termination* of a child process with an exception, and there is no way of propagating exceptions to processes that are not included in the same parallel construct. The mechanisms mentioned above, and many others, are elaborated in [18], which also includes a more detailed treatment of the concepts of error, exception, and related terms. An implementation in Smalltalk-80 of constraint monitors with a functionality close to that described in this paper, is also treated in [18]. This implementation is based on the modelling tool described in [19].

8. Conclusions

A proposal for the handling of exceptions in concurrently executing control processes has been treated. It consists of a mechanism for the raising and handling of exceptions in a sequential process, and constraint monitors for dealing with constraint violations. The syntax and semantics for the specification of exception handlers is mainly based on principles well covered in the literature. The syntax and semantics

chosen are simple and especially suitable for dealing with exceptions in control systems. Constraints and constraint monitors have been introduced as new concepts which are essential for dealing with exceptions in control systems. Constraint monitors are used to specify the constraints of a statement in a structured way, leading to programs in which the code for the normal operation of the program is separated from the code for exception handling. A constraint monitor for a statement needs to be specified only once, but the constraints will be monitored at all interaction points encountered during the execution of the statement. If a constraint violation is detected, an exception is raised. In this way, the invariants of a process remain valid, finalization obligations of statements are executed, deadlock in the case of exception occurrences is prevented, the controlled system is monitored for constraint violations only if necessary, and exceptions are not raised in processes in which no constraints have been violated. Therefore, the use of constraint monitors simplifies the complex task of robust control system specification.

Acknowledgments

We would like to thank C. Bron for advising us to restrict constraint monitoring to interaction points, and other helpful suggestions. Thanks are also due to J.M. van de Mortel-Fronczak, M. Rem, and the anonymous reviewers for their helpful comments on the drafts of this paper.

References

[1] Gini, M. and Smith, R., *Reliable real-time robot operation employing intelligent forward recovery*, Technical Report TR 85-30, University of Minnesota, 1985.

[2] Goodenough, J.B., Exception handling: Issues and a proposed notation, *Communications of the ACM*, December 1975, pp. 683-696.

[3] Knudsen, J.L., Better exception handling in block structured systems, *IEEE Software*, May 1987, pp. 40-49.

[4] Hoare, C.A.R., Communicating Sequential Processes, *Communications of the ACM*, August 1978, pp. 666-677.

[5] Hooman, J., *Specification and compositional verification of real-time systems*, Springer-Verlag, 1991.

[6] Mortel-Fronczak, J.M. van de, Rooda, J.E., and Nieuwelaar, N.J.M. van den, Specification of a flexible manufacturing system using concurrent programming, *Concurrent Engineering Research and Applications*, September 1995, pp. 187-194.

[7] Meyer, B., *Eiffel the language*, Prentice Hall, New York, 1992.

[8] Bron, C. and Dijkstra, E.J., *Report on the programming language Modular Pascal*, Groningen University, Groningen, 1987.

[9] ISO/IEC 8652, *Information Technology - Programming Languages - Ada*, 1995.

[10] Kligerman, E. and Stoyenko, A. D., Real-time Euclid: A language for reliable real-time systems, *IEEE Transactions on Software Engineering*, September 1986, pp. 941-949.

[11] Digital Equipment Corporation, *VAXELN Pascal language reference manual, part 2: programming*, Digital Equipment Corporation, Massachusetts, 1986.

[12] Rossingh, T.J. and Rooda, J.E., *ROSKIT, a real-time operating system kit*, Research report, Technical University Twente, 1985.

[13] Levin, R., *Program structures for exceptional condition handling*, Ph.D. dissertation, Carnegie-Mellon University, June 1977.

[14] Antonelli, C.J., *Exception handling in a multi-context environment*, Ph.D. dissertation, University of Michigan, 1989.

[15] Adamo, J., Exception handling for a communicating-sequential-process-based extension of C++, *Concurrency*, February 1991, pp. 15-41.

[16] Lacoutre, S., Exceptions in Guide, an object oriented language for distributed applications, *Proc. ECOOP 1991*, pp. 268-287, Springer-Verlag, Berlin.

[17] Issarny, V., An exception handling model for parallel programming and its verification, *Software Engineering Notes, Proc. ACM SIGSOFT '91 Conference on Software for Critical Systems*, December 1991, pp. 92-100.

[18] Beek, D.A. van, *Exception handling in control systems*, Ph.D. dissertation, Eindhoven University of Technology, Eindhoven, 1993.

[19] Wortmann, A.M., *Modelling and simulation of industrial systems*, Ph.D. dissertation, Eindhoven University of Technology, Eindhoven, 1991.