A compositional semantics for fault-tolerant real-time systems

Eindhoven University of Technology

Department of Mathematics and Computing Science

A Compositional Semantics for Fault-Tolerant Real-Time Systems

by

J. Coenen and J. Hooman

92/02

Computing Science Note 92/02
Eindhoven, January 1992

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

Editors: prof.dr.M.Rem
prof.dr.K.M. van Hee

A Compositional Semantics for Fault-Tolerant Real-Time Systems *

J. Coenen † J. Hooman ‡

Dept. of Math. and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands

Abstract

Motivated by the close relation between real-time and fault-tolerance, we investigate the foundations of a formal framework to specify and verify real-time distributed systems that incorporate fault-tolerance techniques. Therefore a denotational semantics is presented to describe the real-time behaviour of distributed programs in which concurrent processes communicate by synchronous message passing. New is that in this semantics we allow the occurrence of failures, due to faults of the underlying execution mechanism, and we describe the effect of these failures on the real-time behaviour of programs. Whenever appropriate we give alternative choices for the definition of the semantics. The main idea is that making only very weak assumptions about faults and their effect upon the behaviour of a program in the semantics, any hypothesis about faults must be made explicit in the correctness proof of a program.

1 Introduction

The development of distributed systems with real-time and fault-tolerance requirements is a difficult task, which may result in complicated and opaque designs. This, and the fact that such systems are often embedded in environments where a small error can have serious consequences, calls for formal methods to specify the requirements and verify the development steps during the design process.

Unfortunately most methods that have been proposed up to the present deal either with fault-tolerance requirements, e.g. [ScSc83, Cristian85, JMS87], or with real-time requirements, e.g. [ShLa87, HoWi89, Ostroff89], but not with both simultaneously. This can be a problem, because fault tolerance is obtained by some form of redundancy. For example, a backward recovery mechanism introduces not only information redundancy

*To appear in *Proc. Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems.*

†Supported by NWO/SION Project 612-316-022: "Fault Tolerance: Paradigms, Models, Logics, Construction". E-mail: wsinjosc@win.tue.nl

‡Supported by ESPRIT-BRA Project 3096: "Formal Methods and Tools for the Development of Distributed Real-Time Systems (SPEC)". E-mail: wsinjh@win.tue.nl

and modular redundancy, but also time redundancy. Hence, it is possible to obtain a higher degree of fault-tolerance by introducing more checkpoints, i.e by introducing more time redundancy. This is the main reason why program transformations that are used to transform a program into a functionally equivalent fault tolerant program, e.g. by superimposition of an agreement algorithm, may transform a real-time program into one that doesn't meet its deadlines.

The trade-off between reliability and timeliness extends to one between reliability, timeliness and functionality. An elegant way of exploiting this trade-off can be observed in graceful degrading systems. For example, if a fault occurs a system may temporary sacrifice a service in order to ensure that more important deadlines are met.

Motivated by the close relation between the reliability, timeliness and functionality of a system, we would like to reason about these properties simultaneously. Related research on the integration of these three aspects of real-time programs within one framework can be found in [HaJo89]. In that paper a probabilistic (quantitative) approach is presented, whereas we are mainly concerned with the qualitative aspects of fault-tolerance.

To illustrate how we would like to reason over fault-tolerant real-time systems, consider the Triple Modular Redundancy (TMR) system in figure 1. The TMR system consists of four components. Each of the components P_i computes for an input x on its channel c_i the same function $f(x)$ and outputs the result on its channel d_i five time units later. We consider synchronous communication over directed channels. Because communication is synchronous, a process may have to wait for its communication partner to become available.

The components P_i may use different programs (i.e. algorithms) S_i to compute $f(x)$. We will use $\langle P \Leftarrow S \rangle$ to denote that component P executes program S. The component V waits until it receives the same input at the same time on two different input channels, say d_i and d_j, and outputs this value on channel r two time units later. The informal specifications above can be formalized in, for example, a small extension of first-order predicate logic. A specification is of the form S **sat** φ, where S is a program and φ a sentence of the assertion language. The assertion language includes predicates with the following meaning

$\mathbf{fail}(P)\,\mathbf{at}\,t$	:	at time t a fault occurs in component P during the execution of its program;
$c.v\,\mathbf{at}\,t$	:	at time t the process is communicating v over channel c;
$c!v\,\mathbf{at}\,t$	:	at time t the process tries to send v over channel c.

A formula $\langle P \Leftarrow S \rangle$ **sat** φ expresses that every possible execution of the program S by component P satisfies assertion φ.

The specification of a component $\langle P_i \Leftarrow S_i \rangle$ might be $(i = 1, \ldots, n)$

$$\langle P_i \Leftarrow S_i \rangle \ \mathbf{sat}$$
$$\forall_{t':t \le t' \le t+5}(\neg \mathbf{fail}(P_i)\,\mathbf{at}\,t') \rightarrow (c_i.v\,\mathbf{at}\,t \rightarrow d_i!f(v)\,\mathbf{at}\,t+5)\ .$$

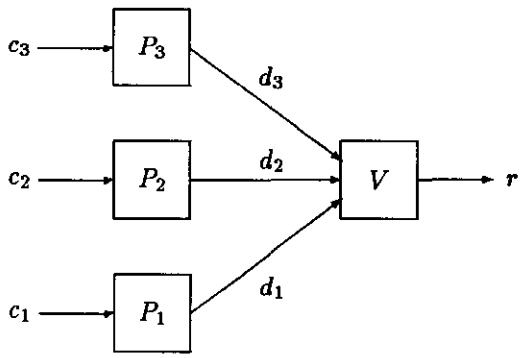

Figure 1: *TMR system*

The voter $\langle V \Leftarrow \rangle S_0$ might be specified by

$$\langle V \Leftarrow S_0 \rangle \ \mathbf{sat}$$
$$\forall_{t':t \le t' \le t+2}(\neg \mathbf{fail}(V)\,\mathbf{at}\,t') \rightarrow (d_i.u\,\mathbf{at}\,t \wedge d_j.u\,\mathbf{at}\,t \wedge i \neq j \rightarrow r!u\,\mathbf{at}\,t+2)\ .$$

The following proof rule for parallel composition ($\|$) holds under certain syntactic restrictions on the assertions

$$\frac{N_1\ \mathbf{sat}\ \varphi_1,\ N_2\ \mathbf{sat}\ \varphi_2}{N_1 \parallel N_2\ \mathbf{sat}\ \varphi_1 \wedge \varphi_2} \quad \text{(Parallel)}.$$

This proof rule is the same as the one used in the proof system of [HoWi89] for real-time programs without considering the possible occurrence of faults.

If we assume that V is always ready to communicate on the channels c_1, c_2, and c_3 — this can be added formally to the specification of V — then $d_i!v\,\mathbf{at}\,t$ corresponds to $d_i.v\,\mathbf{at}\,t$. Then, by repeated application of the parallel composition rule and some predicate calculus, the following specification for the complete TMR system can be derived.

$$\langle P_1 \Leftarrow S_1 \rangle \parallel \langle P_2 \Leftarrow S_2 \rangle \parallel \langle P_3 \Leftarrow S_3 \rangle \parallel \langle V \Leftarrow S_0 \rangle\ \mathbf{sat}$$
$$(\forall_{t':t \le t' \le t+5}(\neg \mathbf{fail}(P_i)\,\mathbf{at}\,t' \wedge \neg \mathbf{fail}(P_j)\,\mathbf{at}\,t' \wedge i \neq j)$$
$$\wedge \forall_{t':t+5 \le t' \le t+7}(\neg \mathbf{fail}(V)\,\mathbf{at}\,t')) \rightarrow (c_i.v\,\mathbf{at}\,t \wedge c_j.v\,\mathbf{at}\,t \rightarrow r!f(v)\,\mathbf{at}\,t+7)$$

Since the rule for parallel composition is compositional, this can be done *without* information about the implementation of the processes S_i $(i = 0, \ldots, 3)$.

Notice that a specification typically is of the format N **sat** $(FH \rightarrow \varphi)$. The antecedent FH in the assertion is called the *fault hypothesis*. Because FH is assumed for a particular process it is called a *local* fault hypothesis, as opposed to a *global* fault hypothesis which hold for all processes. A global fault hypothesis is an axiom of the proof system, provided it is expressible in the assertion language.

A fault hypothesis characterizes faults by (c.f. [RLT78])

- Duration, i.e the time when faults occur, how long will the fault be present, etc.
- Location, i.e. the place where a fault occurs, in which processes, etc.
- Effect, i.e. the effect of the fault on the behaviour of a process, on program variables, etc.

For instance, the following fault hypothesis asserts that faults are transient

$$\mathbf{fail}(P)\,\mathbf{at}\,t \rightarrow \exists_{t' \geq t}(\neg\mathbf{fail}(P)\,\mathbf{at}\,t')\ ,$$

and another example is the following which relates the occurrence of faults in two processors (a fault P_1 will propagate within five time units to P_2)

$$\mathbf{fail}(P_1)\,\mathbf{at}\,t \rightarrow \exists_{t':t \leq t' \leq t+5}(\mathbf{fail}(P_2)\,\mathbf{at}\,t')\ .$$

In this report we take a first step towards a formal method for designing real-time systems with fault-tolerance requirements. Our aim is a *compositional* proof system, i.e. is proof system in which the specification of a compound program can be inferred from the specifications of the constituent components without referring to the internal structure of these components. Compositionality is a desirable property, because it enables one to decompose a large specification of a system into smaller specifications for the subsystems. As basis for such a proof system we define a denotational (and therefore compositional) semantics, i.e. a semantics in which the semantics of a compound program is defined by the semantics of the components independently from the structure of these components.

From the discussion in the preceding paragraphs it is clear that we need semantics that simultaneously describes the following views of a system:

- Functional behaviour. The functional behaviour defines the relation between initial and final states of a program and its communication behaviour.
- Timed behaviour. For real-time systems the time at which a process terminates and the time that it communicates is of interest.
- Fault behaviour. The behaviour of a process in the presence of faults may deviate considerably from its behaviour in absence of faults. Therefore we want to distinguish the fault behaviour from the correct behaviour.

It is inevitable to make some assumptions about the fault behaviour of a process when defining a semantics. However, by making only very weak assumptions we enforce that the assumptions used when dealing with software fault-tolerance — and indeed many of the assumptions for hardware fault-tolerance — have to be made explicit by a fault hypothesis (cf. [Cristian85, BGH86, Bernstein88, Pow+88, TaWi89, CDD90]).

The remainder of this report is organized as follows. In section 2, a programming language is defined, inspired by OCCAM [OCCAM88]. We also give an informal explanation of the language constructs under the assumption that faults don't occur. Faults are taken into consideration in section 3, where we give a semantics of the language defined in section 2. Whenever appropriate we discuss alternative choices for the assumptions that are implicit in the semantics. Conclusions are present in section 4, where we also discuss some future work.

2 Programming Language

To describe real-time systems we use an OCCAM-like programming language, named RT. An RT program is a network of sequential processes that communicate over synchronous channels. Each channel is directed and connects exactly two processes. Processes can only access local variables, i.e. variables are not shared between parallel processes. Processes have unique names.

We assume that the following disjunct sets are defined:

- ($x \in$) *VAR*, the set of program variables;
- ($e \in$) *EXP*, the set of (integer) expressions with free occurrences of program variables only;
- ($b \in$) *BOOL*, the set of boolean expressions with free occurrences of program variables only;
- ($c \in$) *CHAN*, a set of channel names;
- ($P \in$) *PID*, a set of process names.

The formal syntax of an RT program N is defined by

Statement	S	::=	$\mathbf{skip} \mid \mathbf{delay}\, e \mid x := e \mid c!e$
			$\mid c?x \mid S_1; S_2 \mid ALT \mid *ALT$
Alternative	ALT	::=	$[]_{i=1}^{n}\, b_i \rightarrow S_i]$
			$\mid [[]_{i=1}^{n}\, b_i;\, c_i?x_i \rightarrow S_i \,[]\, b_0;\, \mathbf{delay}\, e \rightarrow S_0]$
Network	N	::=	$\langle P \Leftarrow S \rangle \mid N_1 \parallel N_2$

If we forget about faults for the moment, and concentrate on the functional and timed behaviour of programs only, we obtain the following intended meaning for the programming language constructs above.

Primitive Constructs

- **skip** causes no state changes and terminates immediately. Hence, it consumes no time.
- **delay** e takes exactly $K_d + e$ time units to be executed if $e \geq 0$ and $K_d \geq 0$ time units otherwise, but has no other effect. The constant K_d is the minimal amount of time needed to execute a **delay**-statement.
- $x := e$ assigns the value of the expression e to the variable x. Its execution takes $K_a \geq 0$ time units.

- Communication takes place by synchronous message passing over directed channels. Because communication is synchronous a process may have to wait until its communication partner is ready to communicate. There are two primitives for communication:
  - The output statement $c!e$ is used to send the value of e on channel c. It causes the process to wait until the communication partner is prepared to receive a value on channel c.
  - The input statement $c?x$ is similar to the output statement, except that the process waits to receive a value on channel x. If communication takes place the received value is assigned to x.

  The actual communication itself, i.e. without the waiting period, takes exactly $K_c > 0$ time-units.

Instead of using a fixed amount of time for the execution of, for example, the assignment statement we could have chosen an interval of time or a function that assigns an amount of time to an assignment. These options, however, lead to a more difficult to understand semantics, with essentially the same properties.

Compound Constructs

- $S_1; S_2$ denotes the sequential composition of the statements S_1 and S_2. First S_1 is executed, then S_2. The total amount of time needed for execution, is the sum of the execution times of S_1 and S_2. Thus, sequential composition itself takes zero time.
- The alternative statement comes in two formats:
  - $[]_{i=1}^{n}\, b_i \rightarrow S_i]$
    First the boolean expressions b_i are evaluated, which takes $K_g > 0$ time. If all the b_i evaluate to false, the statement terminates immediately after the evaluation of the guards. Otherwise, nondeterministically one of the b_i that evaluated to true is chosen and the corresponding alternative S_i is executed.
  - $[]_{i=1}^{n}\, b_i; c_i?x_i \rightarrow S_i \,[]\, b_0;$ **delay** $e \rightarrow S_0]$
    If all the boolean guards evaluate to false execution of this statement takes exactly $K_g > 0$ time units. Otherwise, if b_0 evaluates to false, the process waits until one of communications $c_i?x_i$ for which b_i $(i \neq 0)$ evaluated to true, is completed. After this communication, the process continues with the execution of the corresponding alternative S_i. If b_0 evaluated to true, the execution is as in the previous case, except that the process waits at most e time units for a communication. If, after evaluation of the guards, e time units have elapsed without starting a communication, the statement S_0 is executed. In this case, the process has consumed $K_g + e$ time before S_0 is executed.
- $*ALT$ denotes the iteration of an alternative statement ALT until all the boolean expressions in the guards evaluate to false. Because, the evaluation of the boolean expressions takes positive time $(K_g > 0)$ only a finite number of iterations is possible in finite time.

- $\langle P \Leftarrow S \rangle$ associates the process identifier P with process S. It is not a statement that is actually executed or implemented, but it is included to enable us to reason over processes by referring to their names. Consequently, this statement consumes no time.
- $N_1 \parallel N_2$ denotes parallel composition. We assume maximal parallelism, which means that each process has its own processor. This ensures maximal progress, i.e. minimal waiting.

3 Denotational Semantics

We will define a denotational semantics that formalizes the informal description of the previous section and extends it with fault behaviour. First, we define the model and give an informal explanation. Second we define and explain the semantics of RT programs. We will distinguish between the correct behaviour of a program (defined by its normal semantics) and the behaviour of a program in the presence of faults (defined by its fault semantics).

Preliminary Definitions

The functional behaviour of a program is partially defined by the initial and final states of a program. A state $s \in STATE$ assigns to each program variable a value. Thus $STATE$ is the set of mappings $VAR \rightarrow VAL$, where VAL is the set of possible values of program variables. We use $s(e)$ to denote the value of expression e in state s, even if e is not a variable. The variant $(s|x \mapsto v)$ of a state s is defined by ($\doteq$ denotes syntactic equality):

$$(s|x \mapsto v)(y) = \begin{cases} v & , x \doteq y \\ s(y) & , \text{otherwise.} \end{cases}$$

The communication behaviour, timed behaviour and fault behaviour of a computation is described by a mapping σ over a time domain $TIME$. The time domain is dense and $t \geq 0$ for all $t \in TIME$. Furthermore, $TIME$ is linearly ordered and closed under addition and multiplication. $TIME$ includes the values of constants K_a, K_d etc. and VAL. For simplicity we assume that $TIME$ is the set of nonnegative rational numbers and that program variables are of type integer. The special symbol ∞ ($\infty \notin TIME$) denotes infinity with the usual properties.

Let Σ be the set of mappings σ of type $[0,t) \rightarrow (\mathcal{P}(CHAN \times (VAL \cup \{!,?\})) \times \mathcal{P}(PID \cup \{X\}))$, where $t \in TIME \cup \{\infty\}$. Thus for all $t \in [0,t')$, $\sigma(t)$ is a pair $(comm, fail)$ with $comm \subseteq CHAN \times (VAL \cup \{!,?\})$ and $fail \subseteq PID \cup \{X\}$. We use $\sigma(t).comm$ and $\sigma(t).fail$ to refer to respectively the first and the second field of $\sigma(t)$.

- $comm \subseteq CHAN \times (VAL \cup \{!,?\})$ defines the communication and timed behaviour. The intended meaning of $comm$ at time $t \in [0,t')$ is as follows.
  - If $(c,v) \in \sigma(t).comm$ then the value v is being communicated on channel c at time t.
  - If $(c,!) \in \sigma(t).comm$ then a process is waiting to send a value on channel c at time t.

– If $(c, ?) \in \sigma(t).comm$ then a process is waiting to receive a value on channel c at time t.

The waiting for a communication is included in the model to obtain a ***compositional*** semantics.

- $fail \subseteq PID \cup \{X\}$, $X \notin PID$. If $P \in \sigma(t).fail$ then process P is behaving according to its fault semantics. Otherwise, P is behaving correctly, i.e. according to its normal semantics. For programs S to which a name has not yet been assigned by a $\langle P \Leftarrow S \rangle$ construct, X is used as a place holder. The *fail*-field enables one to distinguish between normal behaviour (whenever $\sigma(t).fail = \emptyset$) and fault behaviour (whenever $\sigma(t).fail \neq \emptyset$).

The length $|\sigma|$ of a mapping σ with domain $[0, t)$ is defined as t.

The meaning of an RT program is denoted by a set M of triples ($M \subseteq \Delta$), where Δ is the Cartesian product $STATE \times \Sigma \times STATE$. In a triple (s^0, σ, s), s^0 denotes the initial program state and s the final program state.

We define the initial part of length t of σ for $t \in [0, |\sigma|]$, notation $\sigma \downarrow t$, as

$$\begin{array}{lcl} |\sigma \downarrow t| & \hat{=} & t \\ (\sigma \downarrow t)(t') & \hat{=} & \sigma(t') \ , \ t' \in [0, t) \ . \end{array}$$

If $t > |\sigma|$ then $\sigma \downarrow t$ is undefined. The semantics of a RT program is typically defined in two steps. First, we define the normal semantics of the program as described in the previous section, i.e. the semantics when faults do not occur. This is done by defining the interpretation function $\mathcal{M}[\![.]\!] : \mathrm{RT} \rightarrow \mathcal{P}(\Delta)$. Second, we define the interpretation function $\mathcal{M}^{\dagger}[\![.]\!] : \mathrm{RT} \rightarrow \mathcal{P}(\Delta)$ which defines the general semantics when faults are taken into account. The normal behaviour is considered to be a special case of the general behaviour, therefore for all RT programs it is guaranteed that $\mathcal{M}[\![S]\!] \subseteq \mathcal{M}^{\dagger}[\![S]\!]$, or more precisely

$$\mathcal{M}[\![S]\!] = \{(s^0, \sigma, s) \in \mathcal{M}^{\dagger}[\![S]\!] \mid \sigma(t).fail = \emptyset, \text{ for all } t \in [0, |\sigma|)\} \ .$$

The general behaviour can be partitioned into the normal behaviour and the fault behaviour that describes the behaviour if a fault occurs. This is best illustrated by the definition of the semantics of the assignment statement. First we define the normal semantics $\mathcal{M}[\![x := e]\!]$. Then we apply a function $FAIL : \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta)$ to $\mathcal{M}[\![x := e]\!]$, which transforms the normal behaviour into the fault behaviour. Finally we define the general semantics $\mathcal{M}^{\dagger}[\![x := e]\!]$ as the union of the normal behaviour and the fault behaviour.

Let $M \subseteq \Delta$, then $FAIL$ is defined as follows

$$\begin{array}{l} FAIL(M) \hat{=} \\ \quad \{(s^0, \sigma, s) \mid \text{ there exist } (s^0, \sigma', s') \in M \text{ and } t \in [0, \min(|\sigma|, |\sigma'|)) \\ \qquad\qquad \text{such that } \sigma \downarrow t = \sigma' \downarrow t \text{ and for all } t' \in [t, |\sigma|) : \ \sigma(t').fail = \{X\}\} \end{array}$$

For a program S, $FAIL(\mathcal{M}[\![S]\!])$ defines the same behaviour as $\mathcal{M}[\![S]\!]$ up to a point in time where a fault occurs and after that the program may exhibit arbitrary behaviour. For instance it may never terminate.

Proposition 1

(*a*) $FAIL(M) = \emptyset \Leftrightarrow$ for all $(s^0, \sigma, s) \in M$: $|\sigma| = 0$.
(*b*) for all $(s^0, \sigma, s) \in FAIL(M)$ there exists a $t \in [0, |\sigma|)$ such that for all $t' \in [t, |\sigma|)$: $\sigma(t').fail \neq \emptyset$.

■

Part (*a*) of proposition 1 expresses that if, and only if, the executions in M don't consume time they cannot fail and therefore $FAIL(M)$ is empty. Part (*b*) expresses that the mappings σ of all executions in $FAIL(M)$ have a non-empty suffix — because the time domain is dense — during which the *fail*-field is continuously non-empty. As a consequence all computations in $FAIL(M)$ take time.

Skip, Delay, and Assignment

The semantics of the **skip**-statement is:

$$\mathcal{M}[\![\mathbf{skip}]\!] \quad \hat{=} \quad \{(s^0, \sigma, s^0) \mid |\sigma| = 0\}$$

Because **skip** takes no time its execution can't fail. Therefore $FAIL(\mathcal{M}[\![\mathbf{skip}]\!])$ is empty and thus the general semantics is equal to the normal semantics.

$$\begin{aligned} \mathcal{M}^{\dagger}[\![\mathbf{skip}]\!] &\hat{=} \mathcal{M}[\![\mathbf{skip}]\!] \cup FAIL(\mathcal{M}[\![\mathbf{skip}]\!]) \\ &= \mathcal{M}[\![\mathbf{skip}]\!] \end{aligned}$$

The definition of the semantics of the **delay**-statement and the assignment statement should cause no trouble after the previous discussion.

$$\begin{aligned} &\mathcal{M}[\![\mathbf{delay}\, e]\!] \hat{=} \\ &\quad \{(s^0, \sigma, s^0) \mid |\sigma| = K_d + \max(s^0(e), 0) \\ &\qquad \text{and for all } t \in [0, |\sigma|) : \sigma(t).comm = \emptyset \wedge \sigma(t).fail = \emptyset\} \end{aligned}$$

$$\mathcal{M}^{\dagger}[\![\mathbf{delay}\, e]\!] \quad \hat{=} \quad \mathcal{M}[\![\mathbf{delay}\, e]\!] \cup FAIL(\mathcal{M}[\![\mathbf{delay}\, e]\!])$$

$$\begin{aligned} &\mathcal{M}[\![x := e]\!] \hat{=} \\ &\quad \{(s^0, \sigma, s) \mid |\sigma| = K_a \wedge s = (s^0 | x \mapsto s^0(e)) \\ &\qquad \text{and for all } t \in [0, |\sigma|) : \sigma(t).comm = \emptyset \wedge \sigma(t).fail = \emptyset\} \end{aligned}$$

$$\mathcal{M}^{\dagger}[\![x := e]\!] \quad \hat{=} \quad \mathcal{M}[\![x := e]\!] \cup FAIL(\mathcal{M}[\![x := e]\!])$$

Recall from the previous section that communication is synchronous and therefore the behaviour of, for example, a send statement can be split into two parts. During the first part, the process executing the send statement waits until the communication partner is available. If the communication partner eventually is available, which is not always guaranteed, the process will continue with the second part, i.e. the communication itself. Thus a communication statement can be seen as a sequential composition of two smaller processes. Therefore, we first define sequential composition before proceeding with the communication statements.

Sequential Composition

The concatenation $\sigma_0\sigma_1$ of two mappings σ_0 and σ_1 is defined by

$$\begin{array}{lcl} |\sigma_0\sigma_1| & \hat{=} & |\sigma_0| + |\sigma_1| \\ (\sigma_0\sigma_1)(t) & \hat{=} & \begin{cases} \sigma_0(t) & \text{, if } t \in [0, |\sigma_0|); \\ \sigma_1(t - |\sigma_0|) & \text{, if } t \in [|\sigma_0|, |\sigma_0\sigma_1|). \end{cases} \end{array}$$

Sequential composition $SEQ(M_0, M_1)$ of two models $M_0, M_1 \subseteq \Delta$ is defined as follows.

$$\begin{array}{l} SEQ(M_0, M_1) \hat{=} \\ \quad \{(s^0, \sigma_0, s) \in M_0 \mid |\sigma_0| = \infty\} \\ \cup \quad \{(s^0, \sigma_0\sigma_1, s) \mid \text{there exists } s' \text{ such that :} \\ \qquad\qquad (s^0, \sigma_0, s') \in M_0 \wedge |\sigma_0| \neq \infty \wedge (s', \sigma_1, s) \in M_1\} \end{array}$$

The SEQ operator is associative, i.e.

Proposition 2

$$SEQ(SEQ(M_0, M_1), M_2) = SEQ(M_0, SEQ(M_1, M_2)) .$$

■

The normal semantics of sequential composition of two program fragments is

$$\mathcal{M}[\![S_0; S_1]\!] \quad \hat{=} \quad SEQ(\mathcal{M}[\![S_0]\!], \mathcal{M}[\![S_1]\!]) .$$

Observe that sequential composition itself doesn't consume time. Hence, faults occur in the component statements only.

A possible way to define the general semantics of sequential composition is to use the *FAIL* function as we did for **delay**-statement, but there are reasonable alternatives to consider.

1. Using the *FAIL* function in the same manner as in the definition of the assignment statement leads to the following definition.

$$\begin{array}{lcl} \mathcal{M}_1^{\dagger}[\![S_0; S_1]\!] & \hat{=} & \mathcal{M}[\![S_0; S_1]\!] \cup FAIL(\mathcal{M}[\![S_0; S_1]\!]) \\ & = & FAIL(\mathcal{M}[\![S_0]\!]) \cup SEQ(\mathcal{M}[\![S_0]\!], \mathcal{M}^{\dagger}[\![S_1]\!]) . \end{array}$$

   This alternative implies that once a process fails it remains failed. Note that the definition only depends on the normal semantics of the components.

2. It is also possible to assume that if a failing process terminates it will continue with the next statement:

$$\mathcal{M}_2^{\dagger}[\![S_0; S_1]\!] \hat{=} SEQ(\mathcal{M}_2^{\dagger}[\![S_0]\!], \mathcal{M}_2^{\dagger}[\![S_1]\!]) .$$

3. Another option is to assume absolutely nothing about the behaviour of a program once it has failed. This choice even allows the code of a program to be affected by a fault. Suppose S_0 fails and terminates some time units later. The process continues with an arbitrary behaviour, which is considered normal (because the *fail*-field of σ is empty at this time). In this case the behaviour of the program is considered to be correct, i.e. as if its code has been modified.

$$\mathcal{M}_3^{\dagger}[\![S_0; S_1]\!] \hat{=} SEQ(\mathcal{M}[\![S_0]\!], \mathcal{M}_3^{\dagger}[\![S_1]\!]) \cup SEQ(FAIL(\mathcal{M}[\![S_0]\!]), \Delta)$$

Notice that each of these definitions results in a compositional semantics, because $\mathcal{M}[\![S]\!]$ can be defined in terms of $\mathcal{M}^{\dagger}[\![S]\!]$ for all statements S in RT.

Each of the three alternatives ensures that sequential composition is associative.

Proposition 3

$$\mathcal{M}_i^{\dagger}[\![(S_0; S_1); S_3]\!] = \mathcal{M}_i^{\dagger}[\![S_0; (S_1; S_3)]\!],\ i = 1, 2, 3\,.$$

■

The following proposition relates the behaviors defined by these alternatives for a given program fragment S.

Proposition 4

$$\mathcal{M}_i^{\dagger}[\![S]\!] \subseteq \mathcal{M}_{i+1}^{\dagger}[\![S]\!],\ i = 1, 2\,.$$

■

Although, the third alternative defines the less restrictive behaviour we prefer to use the second definition. The reason is that in case of the third alternative a process may exhibit a behaviour that is considered to be correct (i.e. the *fail*-field is empty) even if this behaviour doesn't correspond with an RT-program.

<u>Communication</u>

The normal semantics of the receive statement is defined as the concatenation of two models. The first model denotes the behaviour of the process while it is waiting for its communication partner ($c \in CHAN$):

$$\begin{aligned} &WaitRec(c) \mathrel{\hat{=}} \\ &\quad \{(s^0, \sigma, s) \mid \ (|\sigma| < \infty \rightarrow s^0 = s) \\ &\qquad\qquad \text{and for all } t \in [0, |\sigma|) :\ \sigma(t).comm = \{(c, ?)\} \wedge \sigma(t).fail = \emptyset\}\,. \end{aligned}$$

The second model denotes the behaviour of the process while the actual communication is taking place:

$$\begin{aligned} &CommRec(c, x) \mathrel{\hat{=}} \\ &\quad \{(s^0, \sigma, s) \mid \ |\sigma| = K_c \wedge \text{ there exists a } v \text{ such that } s = (s^0 | x \mapsto v) \\ &\qquad\qquad \text{and for all } t \in [0, |\sigma|) :\ \sigma(t).comm = \{(c, v)\} \wedge \sigma(t).fail = \emptyset\}\,. \end{aligned}$$

So, the complete normal behaviour of the receive statement is

$$\mathcal{M}[\![c?x]\!] \mathrel{\hat{=}} SEQ(WaitRec(c), CommRec(c, x))\,.$$

For the general semantics we have similar options as in case of sequential composition. We give three reasonable alternatives.

1. The first alternative is our standard approach for the primitive constructs.

$$\mathcal{M}_1^{\dagger}[\![c?x]\!] \mathrel{\hat{=}} \mathcal{M}[\![c?x]\!] \cup FAIL(\mathcal{M}[\![c?x]\!])$$

If the process fails during the waiting period and eventually terminates, it skips the communication part. Observe that while the process is still failing it may attempt to communicate because we don't want to make assumptions about the behaviour of a failing process.

2. Alternatively, it is possible to assume that if the process fails while waiting, it remains failed until communication succeeds. This models an execution mechanisms with a reliable communication channel.

$$\mathcal{M}_2^{\dagger}[\![c?x]\!] \triangleq \mathcal{M}[\![c?x]\!] \cup SEQ(FAIL(WaitRec(c)), CommRec(c,x))$$

3. If one does not assume a reliable communication channel then a process that fails while waiting but does not remain failed, may thereafter attempt to communicate. Thus a successful communication is not guaranteed. The possibility of failing or not failing during the waiting period and the actual communication is modelled by $WaitRec^{\dagger}(c)$ and $CommRec^{\dagger}(c,x)$ respectively.

$$\begin{array}{lcl} WaitRec^{\dagger}(c) & \triangleq & WaitRec(c) \cup FAIL(WaitRec(c)) \,, \\ CommRec^{\dagger}(c,x) & \triangleq & CommRec(c,x) \cup FAIL(CommRec(c,x)) \,. \end{array}$$

The general behaviour of the receive statement is in this case

$$\mathcal{M}_3^{\dagger}[\![c?x]\!] \triangleq SEQ(WaitRec^{\dagger}(c), CommRec^{\dagger}(c,x)) \,.$$

We prefer to use the third alternative for two reasons. One reason is that we don't want to assume a reliable communication channel. The other reason is that third alternative defines the less restrictive behaviour in case of a fault.

The send statement is defined in a similar way as the receive statement. First the behaviour of the process while it is waiting is defined. Second, the behaviour during the communication itself is defined. Finally, we define the normal behaviour as the concatenation of these behaviors.

$$\begin{array}{l} WaitSend(c) \triangleq \\ \quad \{(s^0,\sigma,s) \mid \; (|\sigma| < \infty \rightarrow s^0 = s) \\ \qquad\qquad \text{and for all } t \in [0,|\sigma|) : \; \sigma(t).comm = \{(c,!)\} \wedge \sigma(t).fail = \emptyset\} \end{array}$$

$$\begin{array}{l} CommSend(c,e) \triangleq \\ \quad \{(s^0,\sigma,s) \mid \; |\sigma| = K_c \\ \qquad\qquad \text{and for all } t \in [0,|\sigma|) : \; \sigma(t).comm = \{(c,s^0(e))\} \wedge \sigma(t).fail = \emptyset\} \,. \end{array}$$

$$\mathcal{M}[\![c!e]\!] \triangleq SEQ(WaitSend(c), CommSend(c,e))$$

For the same reasons as in case of the receive statement we define the general behaviour of the send statement by

$$\mathcal{M}^{\dagger}[\![c!e]\!] \triangleq SEQ(WaitSend^{\dagger}(c), CommSend^{\dagger}(c,e)) \,,$$

where $WaitSend^{\dagger}(c)$ and $CommSend^{\dagger}(c,e)$ are defined as follows.

$$\begin{array}{lcl} WaitSend^{\dagger}(c) & \triangleq & WaitSend(c) \cup FAIL(WaitSend(c)) \,, \\ CommSend^{\dagger}(c,e) & \triangleq & CommSend(c,e) \cup FAIL(CommSend(c,e)) \,. \end{array}$$

Guarded Statements

The alternative statement $ALT \doteq [[]_{i=1}^{n}\ b_i \rightarrow S_i]$ is is executed as follows. First the boolean guard are evaluated, and if one of the guards evaluated to true, the appropriate alternative is executed. The evaluation of the guards takes K_g time units, but has no other effect.

$$\begin{aligned} &\mathit{Guard}(ALT) \hat{=} \\ &\quad \{(s^0, \sigma, s^0) \mid \ |\sigma| = K_g \\ &\qquad\qquad \text{and for all } t \in [0, |\sigma|) :\ \sigma(t).\mathit{comm} = \emptyset \wedge \sigma(t).\mathit{fail} = \emptyset\} \end{aligned}$$

If all the guards evaluated to false the remainder of the statement is skipped. Otherwise nondeterministically an appropriate alternative is chosen, and executed.

$$\begin{aligned} &\mathit{Select}(ALT) \hat{=} \\ &\qquad \{(s^0, \sigma, s) \mid \text{there exists an } i \in \{1, \ldots, n\} \text{ s.t. } s^0(b_i) \wedge (s^0, \sigma, s) \in \mathcal{M}[\![S_i]\!]\} \\ &\cup \quad \{(s^0, \sigma, s^0) \mid |\sigma| = 0 \wedge \bigvee_{i=1}^{n} \neg s^0(b_i)\} \end{aligned}$$

The complete normal behaviour of the simple alternative statement is thus defined by

$$\mathcal{M}[\![ALT]\!] \hat{=} \mathit{SEQ}(\mathit{Guard}(ALT), \mathit{Select}(ALT)) .$$

We consider two possible definitions of the general semantics of the simple alternative statement.

1. The first possible definition is obtained by simply applying the *FAIL* function.

   $$\mathcal{M}_1^{\dagger}[\![ALT]\!] \hat{=} \mathcal{M}[\![ALT]\!] \cup \mathit{FAIL}(\mathcal{M}[\![ALT]\!]) .$$

   The disadvantage of this definition is that it does not discriminate between the occurrence of a fault during the evaluation of the guards and the occurrence of a fault in one of the constituent statements: both faults cause the failure of the whole alternative statement.

2. The second possibility is

   $$\begin{aligned} &\mathcal{M}_2^{\dagger}[\![ALT]\!] \hat{=} \\ &\qquad \mathcal{M}[\![ALT]\!] \cup \mathit{FAIL}(\mathit{Guard}(ALT)) \\ &\cup \quad \mathit{SEQ}(\mathit{Guard}(ALT), \mathit{FAIL}(\mathit{Select}(ALT))) \\ &\cup \quad \bigcup_{i=1}^{n} \mathit{SEQ}(\mathit{FAIL}(\mathit{Guard}(ALT)), \mathcal{M}^{\dagger}[\![S_i]\!]) \end{aligned}$$

   Where $\mathcal{M}^{\dagger}[\![S]\!] = \mathcal{M}_2^{\dagger}[\![S]\!]$ in case $S \doteq ALT$. This definition doesn't have the disadvantage of the previous one.

Because $\mathcal{M}_1^{\dagger}[\![ALT]\!] \subseteq \mathcal{M}_2^{\dagger}[\![ALT]\!]$ we prefer the second definition.

If $ALT \doteq [[]_{i=1}^{n}\ b_i; c_i?x_i \rightarrow S_i [] b_0; \mathbf{delay}\, e \rightarrow S_0]$ there are three possible ways the process may continue after evaluation of the guards.

1. If all the guards are false the remainder of the ALT statement is skipped.
2. If one of the b_i ($i \neq 0$) is true the process waits for an input on one of the c_i for which b_i is true. If b_0 is true communication has to begin within e time units. After the input is received the process continues with the corresponding alternative.
3. If b_0 is true and the process has not received an input within e time units after the guards were evaluated it continues with the execution of S_0.

The first behaviour is defined by

$$\{(s^0, \sigma, s) \in \mathit{Guard}(ALT) \mid \bigwedge_{i=0}^{n} \neg s^0(b_i)\}$$

The second behaviour is defined as the concatenation of three behaviors

$$\mathit{SEQ}(\mathit{Guard}(ALT), \mathit{Wait}(ALT), \mathit{Comm}(ALT)) ,$$

where $\mathit{Guard}(ALT)$ is defined as before and $\mathit{Wait}(ALT)$ and $\mathit{Comm}(ALT)$ are defined as follows.

$$\mathit{Wait}(ALT) \hat{=}$$
$$\{(s^0, \sigma, s) \mid (\bigvee_{j=0}^{n} s^0(b_j)) \wedge (s^0(b_0) \rightarrow |\sigma| < \min(s^0(e), 0)) \wedge (|\sigma| < \infty \rightarrow s^0 = s)$$
$$\text{and for all } t \in [0, |\sigma|) : \ \sigma(t).\mathit{comm} = \{(c_i, ?) \mid s^0(b_i)\}\}$$

$$\mathit{Comm}(ALT) \hat{=}$$
$$\{(s^0, \sigma, s) \mid \text{there exists an } i \in \{1, \ldots, n\} \text{ such that}$$
$$s^0(b_i) \wedge (s^0, \sigma, s) \in \mathit{SEQ}(\mathit{CommRec}(c_i, x_i), \mathcal{M}[\![S_i]\!])\}$$

The third behaviour is also defined as the concatenation of three behaviors

$$\mathit{SEQ}(\mathit{Guard}(ALT), \mathit{TimeOut}(ALT), \mathcal{M}[\![S_0]\!]) ,$$

where $\mathit{TimeOut}(ALT)$ is defined as follows.

$$\mathit{TimeOut}(ALT) \hat{=} \{(s^0, \sigma, s) \in \mathit{Wait}(ALT) \mid s^0(b) \wedge |\sigma| = \min(s^0(e), 0)\}$$

The complete normal behaviour of this ALT statement is the union of the three behaviors described above.

$$\mathcal{M}[\![ALT]\!] \hat{=}$$
$$\{(s^0, \sigma, s) \in \mathit{Guard}(ALT) \mid \bigwedge_{i=0}^{n} \neg s^0(b_i)\}$$
$$\cup \ \mathit{SEQ}(\mathit{Guard}(ALT), \mathit{Wait}(ALT), \mathit{Comm}(ALT))$$
$$\cup \ \mathit{SEQ}(\mathit{Guard}(ALT), \mathit{TimeOut}(ALT), \mathcal{M}[\![S_0]\!])$$

To understand the definition of the general semantics below, one must consider the places where a fault may occur while executing the ALT statement. We start near the end of the statement.

I Suppose a fault does not occur until the execution of one of the alternatives. Or a fault occurs while the process is communicating. If the fault behaviour is finite the process may skip the remainder of the ALT statement or continue with the execution of one of the alternatives which of course may also result in a fault. This possibility is captured in the following definition.

$$\begin{aligned} & SEQ(Guard(ALT), Wait(ALT), Comm^{\dagger}(ALT)) \\ \cup\ & SEQ(Guard(ALT), TimeOut(ALT), \mathcal{M}^{\dagger}[\![S_0]\!]) \end{aligned}$$

Where $Comm^{\dagger}(ALT)$ is defined as follows.

$$\begin{aligned} & Comm^{\dagger}(ALT) \triangleq \\ & \{(s^0, \sigma, s) \mid \text{there exists an } i \in \{1, \ldots, n\} \text{ such that} \\ & \qquad s^0(b_i) \wedge (s^0, \sigma, s) \in SEQ(CommRec^{\dagger}(c_i, x_i), \mathcal{M}^{\dagger}[\![S_i]\!])\} \end{aligned}$$

II Suppose a fault occurs while the process is waiting to communicate. If the fault behaviour if finite the process may continue with any of the communications or alternatives for which it was waiting (i.e. those for which the guard evaluated to true). Of course each of these continuations may again lead to a fault. So we get

$$SEQ(Guard(ALT), Wait^{\dagger}(ALT)) ,$$

where $Wait^{\dagger}(ALT)$ is defined by

$$\begin{aligned} & Wait^{\dagger}(ALT) \triangleq \\ & \{(s^0, \sigma, s) \mid \text{there exist } s', \sigma_0, \text{ and } \sigma_1 \text{ such that} \\ & \qquad \sigma = \sigma_0\sigma_1 \wedge (s^0, \sigma_0, s') \in FAIL(Wait(ALT)) \\ & \qquad \wedge((s^0(b_0) \wedge (s', \sigma_1, s) \in \mathcal{M}^{\dagger}[\![S_0]\!]) \\ & \qquad\quad \vee(\text{there exists an } i \in \{1, \ldots, n\} \text{ such that} \\ & \qquad\qquad s^0(b_i) \wedge (s', \sigma_1, s) \in CommRec^{\dagger}(ALT)))\} . \end{aligned}$$

III Suppose the fault occurs during the evaluation of the boolean part of the guards. In this case the may wait for an arbitrary communication for an arbitrary period of time, or it may exit the alternative statement immediately. This results in the following behaviour.

$$\begin{aligned} & SEQ(FAIL(Guard), Wait(ALT), Comm^{\dagger}(ALT)) \\ \cup\ & SEQ(FAIL(Guard), TimeOut(ALT), \mathcal{M}^{\dagger}[\![S_0]\!]) \\ \cup\ & SEQ(FAIL(Guard), Wait^{\dagger}(ALT)) \\ \cup\ & \{(s^0, \sigma, s) \in FAIL(Guard) \mid \bigwedge_{i=0}^{n} \neg s(b_i)\} \end{aligned}$$

The general semantics of the ALT statement is the union of the normal semantics and the semantics given in I–III above.

Iteration

We define BB as $\bigvee_{i=1}^{n} b_i$ in case ALT is the simple alternative statement and as $\bigvee_{i=0}^{n} b_i$ otherwise. The semantics of the iteration is defined as a greatest fixed-point:

$$\begin{aligned}&\mathcal{M}[\![*ALT]\!] \hat{=} \\ &\quad \nu Y.(\quad \{(s^0,\sigma,s) \mid \neg s^0(BB) \wedge (s^0,\sigma,s) \in \mathcal{M}[\![ALT]\!]\} \\ &\qquad \cup\{(s^0,\sigma,s) \mid s^0(BB) \wedge (s^0,\sigma,s) \in SEQ(\mathcal{M}[\![ALT]\!], Y)\})\end{aligned}$$

Because evaluation of the boolean guards takes $K_g > 0$ time greatest fixed-point exists and is not empty (cf. [Hooman91]).

We consider two possible definitions of the general semantics.

1. Using the $FAIL$ function gives the simplest definition.

$$\mathcal{M}_1^{\dagger}[\![*ALT]\!] \hat{=} \mathcal{M}[\![*ALT]\!] \cup FAIL(\mathcal{M}[\![*ALT]\!])$$

   If a fault occurs the process will remain failed until the complete statement terminates. However, we want a definition that discriminates between, for example, a single fault in one pass of the iteration and two consecutive passes with a fault.

2. A definition that does discriminate between the above mentioned cases, and also between the place where a fault occurs is

$$\begin{aligned}&\mathcal{M}_2^{\dagger}[\![*ALT]\!] \hat{=} \\ &\quad \nu Y.(\quad \{(s^0,\sigma,s) \mid \neg s^0(BB) \wedge (s^0,\sigma,s) \in \mathcal{M}[\![ALT]\!]\} \\ &\qquad \cup\{(s^0,\sigma,s) \in SEQ(\mathcal{M}^{\dagger}[\![ALT]\!], Y) \mid s^0(BB)\} \\ &\qquad \cup FAIL(Guard(ALT)))\end{aligned}$$

   Where $\mathcal{M}^{\dagger}[\![S]\!] = \mathcal{M}_2^{\dagger}[\![S]\!]$ in case $S \doteq *ALT$. This definition allows a process to continue or exit the loop due to a failure. The existence of the greatest fixed-point follows from the fact that failing processes consume time (see proposition 1).

For the reasons mentioned above, we prefer to use the second definition.

Networks

As explained in section 2, the naming construct is not executed or implemented, but only included to facilitate reasoning over programs. Therefore it doesn't introduce new faults.

$$\begin{aligned}&\mathcal{M}^{\dagger}[\![(P \Leftarrow S)]\!] \hat{=} \\ &\quad \{(s^0,\sigma,s) \mid \quad \text{there exists } (s^0,\sigma',s) \in \mathcal{M}^{\dagger}[\![S]\!] \text{ such that } |\sigma| = |\sigma'| \\ &\qquad\qquad \text{and for all } t \in [0,|\sigma|) : \ \sigma(t).comm = \sigma'(t).comm \\ &\qquad\qquad \wedge \sigma(t).fail = \emptyset \leftrightarrow \sigma'(t).fail = \emptyset \wedge \sigma(t).fail = \{P\} \leftrightarrow \sigma'(t).fail \neq \emptyset\}\end{aligned}$$

The parallel composition operator doesn't consume time. Hence, it cannot introduce faults that were not already present in the component processes. We use $var(N)$ and

$chan(N)$ to denote the set of program variables in N and the set of channels incident with N respectively. Recall that variables are not shared and channels connect exactly two processes.

$$
\begin{aligned}
&\mathcal{M}^{\dagger}[\![N_1 \parallel N_2]\!] \hat{=} \\
&\quad \{(s^0, \sigma, s) \mid \text{ there exists } (s_i^0, \sigma_i, s_i) \in \mathcal{M}^{\dagger}[\![N_i]\!] \text{ such that} \\
&\qquad |\sigma| = \max(|\sigma_1|, |\sigma_2|) \wedge (x \in var(N_i) \rightarrow s^0(x) = s_i^0(x)) \\
&\qquad \wedge (x \in var(N_i) \rightarrow s(x) = s_i(x)) \\
&\qquad \wedge (x \notin var(N_1, N_2) \rightarrow s(x) = s^0(x)) \\
&\qquad \text{and for all } t \in [0, |\sigma|),\ c \in CHAN, \text{ and } v \in VAL: \\
&\qquad \sigma(t).comm = \sigma_1(t).comm \cup \sigma_2(t).comm \\
&\qquad \wedge \sigma(t).fail = \sigma_1(t).fail \cup \sigma_2(t).fail \\
&\qquad \wedge |\sigma(t).comm \cap \{(c,?),(c,!),(c,v)\}| \leq 1 \qquad (1) \\
&\qquad \wedge \left\{ \begin{array}{l} \text{if } c \in chan(N_1) \cap chan(N_2) \\ \text{then } (c,v) \in \sigma_1.comm \leftrightarrow (c,v) \in \sigma_2.comm \end{array} \right. \qquad (2) \\
&\quad \}
\end{aligned}
$$

It easily seen that parallel composition is commutative. Associativity follows from the fact that channels connect exactly two processes. Hence, the following proposition.

Proposition 5

$$
\begin{aligned}
\mathcal{M}^{\dagger}[\![N_1 \parallel N_2]\!] &= \mathcal{M}^{\dagger}[\![N_2 \parallel N_1]\!] \\
\mathcal{M}^{\dagger}[\![(N_1 \parallel N_2) \parallel N_3]\!] &= \mathcal{M}^{\dagger}[\![N_1 \parallel (N_2 \parallel N_3)]\!]
\end{aligned}
$$

■

Note that (1) is the maximal progress assumption and (2) models regular communication. The assumption that (1) and (2) hold can be weakened for failing processes, by replacing them with

$$\sigma(t).fail = \emptyset \rightarrow (1) \wedge (2) .$$

This transformation affects commutativity nor associativity of the parallel composition operator. The weaker version has our preference.

4 Conclusions

We have taken a first step towards a formal method for specifying and verifying real-time systems in the presence of faults. A compositional semantics has been defined together with many alternative definitions. The semantics is defined such that only very weak assumptions about faults and their effect upon the behaviour of a program are made. In this way it is ensured that a proof system that takes this semantics as a basis for its soundness will include few hidden assumptions. Therefore, if one uses such a proof system to verify a real-time system, almost all assumptions about faults will have to be made explicit.

The semantics is compositional which eases the development of a compositional proof system, thereby making the verification of larger systems possible. In section 1 we discussed a small example to illustrate what a proof system might look like. Based upon the semantics defined in this report, we are currently developing a compositional proof system using a real-time version of temporal logic. Future work also includes the design of a proof system that is more like the conventional Hoare-style proof system with pre- and postconditions for sequential programs.

In our semantic definition, faults may affect any channel or local variable. For instance, a fault in a processor may affect any channel in the network, including those that are not connected to the failing processor. This is justified by our philosophy that we want to make only very few (and weak) assumptions about the effect of fault within the model itself. A first study, however, shows that it is possible to parameterize the semantics by function that restrict the set of variables and channels that might be affected by a fault during the execution of a statement.

Acknowledgment. We would like to thank the members of the NWO project "Fault Tolerance: Paradigms, Models, Logics, Construction," in particular Thijs Krol, for their remarks when this work was presented to them in the context of this project.

References

[Bernstein88] P.A. Bernstein. *Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing.* IEEE Computer pp. 37–46, February 1988.

[BGH86] J. Bartlett, J Gray & B. Horst. *Fault Tolerance in Tandem Computer Systems.* Symp. on the Evolution of Fault-Tolerant Computing, Baden, Austria, 1986.

[CDD90] F. Cristian, B. Dancey & J. Dehn. *Fault-Tolerance in the Advanced Automation System.* In "20th Annual Symp. on Fault-Tolerant Computing", 1990.

[Cristian85] F. Cristian. *A Rigorous Approach to Fault-Tolerant Programming.* IEEE Trans. on Softw. Engin. ; SE-11(1):23–31, 1985.

[HaJo89] H. Hansson & B. Jonsson. *A Framework for Reasoning About Time and Reliability.* Proc. 10th IEEE Real-Time Systems Symposium, pp. 101–111, 1989..

[Hooman91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* LNCS 558, Springer-Verlag, 1991.

[HoWi89] J. Hooman & J. Widom. *A Temporal-Logic Based Compositional Proof System for Real-Time Message Passing.* Proc. PARLE '89 Vol. II:424–441; LNCS 366, 1989.

[JMS87] M. Joseph, A. Moitra & N. Soundararajan. *Proof Rules for Fault Tolerant Distributed Programs.* Science of Comp. Prog. ; 8:43–67, 1987.

[KLS86] N. Kronenberg, H. Levy & W. Strecker. *VAXclusters: A Closely-Coupled Distributed System.* ACM Trans. on Computer Systems, 4:130–146, 1986.

[OCCAM88] INMOS Ltd. *OCCAM 2 Reference Manual.* Prentice-Hall, 1988.

[Ostroff89] J. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press, 1989.

[Pow+88] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck & D. Seaton. *The Delta-4 Approach to Dependability in Open Distributed Computing Systems.* Proc. FTCS-18, IEEE Computer Society Press, 1988.

[RLT78] B. Randell, P.A. Lee & P.C. Treleaven. *Reliability Issues in Computing System Design.* ACM Computing Surveys, 10:123–165, 1978.

[ScSc83] R.D. Schlichting & F.B. Schneider. *Fail-stop processors: an approach to designing fault-tolerant computing systems.* ACM Trans. on Comp. Sys. ; 1(3):222–238, 1983.

[ShLa87] A.U. Shankar & S.S. Lam. *Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties.* Distributed Computing; 2:61–79, 1987.

[TaWi89] D. Taylor & G. Wilson. *Stratus.* In "Dependability of Resilient Computers", T. Anderson Ed., Blackwell Scientific Publications, 1989.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.

91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.