Formalisms for program reification and fault tolerance

Formalisms for Program Reification and Fault Tolerance

Jos Coenen

Formalisms for Program Reification and Fault Tolerance

Proefschrift

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

woensdag 18 mei 1994 om 16.00 uur

door

Joseph Albert Adriaan Coenen

Geboren te Eindhoven

Dit proefschrift is goedgekeurd
door de promotoren
prof.dr. W.-P. de Roever
prof.dr. J.C.M. Baeten

en de copromotor
dr. J.J.M. Hooman

Acknowledgements

The research on which this thesis is based was carried out while the author was employed by NWO on the combined SION/STW project [1] "Fault Tolerance: Paradigms, Models, Logics, Construction." I'm grateful for the very generous financial support received from NWO. In particular I thank drs. R. Kellermann-Deibel for sharing his administrative expertise and wisdom.

I thank Willem-Paul de Roever and Jozef Hooman for their guidance and stimulation during the last four years. I'm indebted to my co-authors and (former) colleagues at the computing science department of the Eindhoven University of Technology, and the participants of the afore mentioned project for their collaboration. In particular, I thank Frank de Boer, Tijn Borghuis, Wim Koole, and Henk Schepers.

I thank Rob Nederpelt for giving me sound advice when it was most needed.

[1] SION project numbers 612-316-022 and 612-316-103.

Contents

Chapter 1

Introduction

Introduction

Nowadays, computers are used for numerous applications. We may think of simple batch systems for information processing for administrative applications, but also of highly complex systems for process control for industrial applications. For each of these applications the issue of correctness is an important one. Incorrect software or hardware may have undesirable consequences. To ensure correctness of the system some formal method may be used to specify, design, and verify the system. A formal method should therefore include a high-level specification language — to describe the requirements of the desired system — which allows to abstract from implementation details, and a low-level language to describe an actual implementation. It is not required that these languages are separated. The method should also provide a means to verify whether an implementation, i.e. the description of an actual implementation, satisfies its specification, i.e. whether it provides the requested services. Ideally the method provides the designer with guidelines how to structure the development of the desired system.

Examples of formal methods are Hoare's logic [Hoare69, Apt81] — or more appropriate Floyd-Hoare logic) — and VDM [Jones90]. Hoare's logic and VDM are examples of model oriented methods. In chapter 2 a relational framework which unifies Hoare's logic and VDM is presented. The importance of such unifying framework lies in the fact that it allows to go forth and back between these two formalisms. It serves to illustrate that despite the apparent differences formalisms do not differ in an essential way, i.e. they focus on the same aspects of system development. Other examples of model oriented methods are Z [Spivey88, Spivey92], and refinement calculi [Back80, Morgan90]. As opposed to model oriented methods there exist various algebraic methods, most notably CCS [Milner80, Milner89], CSP [Hoare85], and ACP [BeKl84, BaWe90].

Formal methods such as Hoare's logic and VDM are only concerned with the functional correctness of a system. Some applications, however, require that a system is not only functionally correct, but also correct with respect to the timing of actions. A system whose correctness depends on the timing of actions is called a real-time system. Several formal techniques to verify timeliness and functional correctness of systems have been developed. Examples of model oriented methods can be found in [Ostroff89, Hooman91], and examples of algebraic methods are [NiRiSiVo90] and [BaBe91]. Real-time systems are often used for critical applications. For example, avionic systems for flight control and control systems for power plants. Systems for critical applications are required to have additional fault tolerant properties.

Of course a system can not be fault tolerant with respect to all possible faults. For a system to be fault tolerant it is therefore required only to tolerate a specified class of faults. Faults can be classified by location (i.e. where the fault occurs), duration (i.e. when and how long the fault occurred), and effect (how it influences the system behaviour). A formal method for fault tolerance must therefore also provide a way to formally define classes of faults.

Algebraic theories that consider fault tolerance properties are presented in e.g. [Prasad87] and [HeHo87]. Model oriented methods can be found in e.g. [ScSc83], [Cristian85], and [JoMoSo87]. Significantly less has been achieved in developing theories which combine functionality, timeliness, and fault tolerance. This

can cause problems because fault tolerance is typically obtained by some form of redundancy. For example, a backward recovery mechanism for databases introduces information redundancy (replicated data), modular redundancy (duplication of information carriers), and time redundancy (back-up and restore operations). Adding time redundancy may transform a correct real-time program into an incorrect one. Graceful degrading systems may sacrifice certain services, thereby changing the functional behaviour, in order for more important services to meet their deadlines.

An algebraic theory for reasoning about timeliness and reliability is outlined in [HaJo89]. In chapter 3 the foundations of a formal framework for the specification and verification of fault tolerant distributed real-time systems with synchronous message passing are investigated. It presents a denotational semantics for a model oriented theory which combines functionality, timeliness, and reliability. In this semantics the occurrence of faults, due to a malfunctioning of the underlying execution mechanism, and their effects upon the behaviour of real-time systems are considered. The main idea is that by making very weak assumptions in the semantics about faults and their effects, any hypothesis about fault must be made explicit in the correctness proof of a program.

Chapter 4 provides an algebraic method for specifying and verifying fault tolerant systems. There is an important difference between the ideas behind the definitions in chapter 3 and chapter 4. The semantics defined in chapter 3 adopts the principle that a system is incorrect (i.e. error prone) unless specified otherwise, whereas the theory in chapter 4 adapts the principle that a system is correct (i.e. error free) unless specified otherwise. A proof method based on the semantics of chapter 4 will therefore require that all assumptions about the occurrence, duration, and effect of faults must be made explicit in the specification of a system. The theory in chapter 4 requires that faults are inserted in the system by explicitly defining the fault hypothesis as an additional process in the specification. The theory of chapter 4 is more general than that of [Prasad87] and [HeHo87], in the sense that a more general class of processes is considered. The class of programs considered in [Prasad87] and [HeHo87], i.e. restartable systems, is a subclass of the systems definable in the theory of chapter 4.

Finally, chapter 5 discusses the problems encountered when attempting to construct a formal method for designing fault tolerant systems which supports top-down development. It appears that we need to distinguish between preferred and less preferred behaviours of a system. Formal methods such as Hoare's logic and VDM do not distinguish between those possible behaviours, and there seems to be no obvious way to adapt these methods so that they will distinguish such behaviours. As pointed out in chapter 5, one needs a more expressive assertion language [1]. A possible candidate is dyadic deontic logic. Although modal logics in general (e.g. [Harel79] and [BaKuPn84]) and deontic logic in particular (e.g. [Khosla88]) have been proposed and developed for specifying (fault tolerant) systems, it is not immediate how to obtain a theory with the properties described in chapter 5. These are interesting cliff hangers for the future.

[1] Besides the obvious need to distinguish between preferred and less preferred behaviours in the semantics.

References

[Apt81] Apt, K.R.: *Ten Years of Hoare's Logic: A Survey — Part I.* ACM TOPLAS **4**, pp.:431–483, 1981.

[Back80] Back, R.J.R.: *Correctness Preserving Program Refinements: Proof Theory and Applications.* Mathematical Centre Tracts **131**, CWI Amsterdam 1980.

[BaWe90] Baeten, J.C.M. & Weijland, W.P.: *Process Algebra.* Cambridge Tracts in Theor. Comp. Sci. **18**, Cambridge University Press 1990.

[BaBe91] Baeten, J.C.M. & Bergstra, J.A.: *Real-Time Process Algebra.* Formal Aspects of Computing **3**, pp.:142–188, 1991.

[BaKuPn84] Barringer, H., Kuiper, R. & Pnueli, A.: *Now You May Compose Temporal Logic Specifications.* Proc. 16th ACM Symp. on Theory of Computing, pp.:51–63, 1984.

[BeKl84] Bergstra, J.A. & Klop, J.W.: *Process Algebra for Synchronous Communication.* Information and Control **60**, pp.:109–137, 1984.

[Cristian85] Cristian, F.: *A Rigorous Approach to Fault Tolerant Programming.* IEEE Trans. on Softw. Engin. **11**, pp.:23–31, 1985.

[HeHo87] He Jifeng & Hoare, C.A.R.:. *Algebraic Specification and Proof of a Distributed Recovery Algorithm.* Distributed Computing **2**, pp.:1–12, 1987.

[JoMoSo87] Joseph, M., Moitra, A. & Soundararajan, N.: *Proof Rules for Fault Tolerant Distributed Programs.* Science of Comp. Prog. **8**, pp.:43–67, 1987.

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

[Harel79] Harel, D.: *First-Order Dynamic Logic.* LNCS **68**, Springer-Verlag 1979.

[Hoare69] Hoare, C.A.R.: *An Axiomatic Basis for Computer Programming.* Communications of the ACM **12**, pp.:576–580, 1969.

[Hoare85] Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice-Hall 1985.

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

[Jones90] Jones, C.B.: *Systematics Software Development Using VDM* (Second edition). Prentice-Hall 1990.

[Khosla88] Khosla, S.: *System Specification: a Deontic Approach.* PhD Thesis University of London 1988.

[Milner80] Milner, R.: *A Calculus of Communicating Systems.* LNCS **92**, Springer-Verlag 1980.

[Milner89] Milner, R.: *Communication and Concurrency.* Prentice-Hall 1989.

[Morgan90] Morgan, C.: *Programming from Specifications.* Prentice-Hall 1990.

[NiRiSiVo90] Nicollin, X., Richier, J.-L., Sifakis, J. & Voiron, J.: *ATP: an Algebra for Timed Processes.* In "Programming Concepts and Methods" (M. Broy & C.B. Jones eds.), pp.:415–442, 1990.

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

[Prasad87] Prasad, K.V.S.: *Combinators and Bisimulation Proofs for Restartable Systems.* PhD Thesis University of Edinburgh, 1987.

[ScSc83] Schlichting, R.D. & Schneider, F.B.: *Fail-Stop Processors: an Approach to Designing Fault Tolerant Computing Systems.* ACM Trans. on Comp. Sys. **1**, pp.:222–238, 1983.

[Spivey88] Spivey, J.M.: *Understanding Z : a Specification Language and Its Formal Semantics.* Cambridge Tracts in Theor. Comp. Sci. **3**, Cambridge University Press 1988.

[Spivey92] Spivey, J.M.: *The Z Notation : a Reference Manual* (Second edition). Prentice-Hall 1992.

Chapter 2

Hoare's Logic and VDM

This chapter is a revised version of:

J. COENEN.
Hoare's Logic and VDM.
To appear in 'Formal Aspects of Computing.'

Hoare's Logic and VDM

J. Coenen[1]

Dept. of Math. and Computing Science, Eindhoven University of Technology.

Keywords: Hoare's logic; Program specification; Realizability; VDM.

Abstract. A relational framework which unifies Hoare's logic and VDM is presented. Within this framework a partial correctness version of VDM is defined. It is argued that this partial correctness version of VDM is intuitive and consistent with the original total correctness version. Furthermore it is shown how both partial and total correctness formulae and specifications can be translated from Hoare's logic into VDM and vice versa. VDM's satisfiability requirement is briefly discussed, and a similar condition for Hoare's logic is defined.

1 Introduction

In nineteen sixty-nine Hoare published a number of axioms and proof rules for proving assertions about programs [Hoa69], which is usually referred to as Floyd-Hoare logic, or simply Hoare's logic. Hoare's logic has been thoroughly investigated by various researchers, which led to some interesting extensions (see eg. [Hoa71, Gor75, Bak80, Apt81]). We will consider Hoare's logic as it appeared in [Apt81], because the proof system presented there includes a complete set of proof rules for *logical variables*.

Although logical variables were originally introduced in Hoare's logic in order to obtain a complete proof system for recursive procedures [Gor75], we're mainly interested in them because of their use in specifications. For example, if we want to specify a program that increases the value of program variable x by one for an arbitrary initial value we have to use a logical variable. The intended meaning of the specification (the superscript $\mathcal{H}$ is used for Hoare-style specifications and correctness formulae)

$$\{x = x_0\}\ inc1\ \{x = x_0 + 1\}^{\mathcal{H}} \tag{1}$$

is that if x has initially the same value as logical variable x_0 then $inc1$ should establish that afterwards x has the value of $x_0 + 1$. Because logical variables do not occur in

Correspondence and offprint requests to: J. Coenen, Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: wsinjosc@win.tue.nl.

[1]Supported by NWO/SION project 612-316-103: 'Fault Tolerance: Paradigms, Models, Logics, Construction.'

programs, the above specification indeed specifies that the value of x should increased by one. Notice how the logical variable x_0 is used to carry information from the initial state, characterized by $x = x_0$, to the final state, characterized by $x = x_0 + 1$.

In VDM [Jon90] logical variables are not needed, because the relationship between the initial program state and the final program state is established by 'hooked' variables $\overleftarrow{x}$. The operation *inc*1 can be specified as follows (VDM-style specifications and correctness formulae are decorated with a superscript $\mathcal{J}$).

$$\{\text{true}\}\ inc1\ \{x = \overleftarrow{x} + 1\}^{\mathcal{J}} \tag{2}$$

The variable $\overleftarrow{x}$ in the postcondition $x = \overleftarrow{x} + 1$ refers to the value of x in the initial program state, and the variable x in the postcondition refers to the value of x in the final program state.

Another difference between the above specifications is the use of the precondition. To make this difference clearer we need to distinguish between *partial* correctness and *total* correctness. A partial correctness specification such as (1) has the intended meaning that if initially the precondition is satisfied and if *inc*1 terminates then the final program state must satisfy the postcondition. Hence, a program that does not terminate satisfies the specification (1). A total correctness specification for *inc*1 is (notice the subscript)

$$\{x = x_0\}\ inc1\ \{x = x_0 + 1\}^{\mathcal{T}}_{\perp} \tag{3}$$

The intended meaning of (3) is that if initially $x = x_0$ is satisfied then *inc*1 must terminate in a state satisfying $x = x_0 + 1$. VDM is, as opposed to Hoare's original system, a total correctness formalism. In VDM it is possible to separate the issue of termination from the issue of functionality. The precondition in a VDM specification specifies a set of initial states for which the program must terminate; the postcondition specifies the functional behaviour of the program. In Hoare's logic (for total correctness) the precondition is used both for specifying the initial states for which the program must terminate and, together with the postcondition, for specifying the functional behaviour.

Despite these apparent differences, Hoare's logic and VDM are interchangeable, and can be unified within a simple relational framework (see section 2). In section 3 we show how to translate Hoare-style partial correctness specifications into VDM-style partial correctness specifications, and vice versa. Similar results for total correctness are obtained in section 4. Section 4 also contains a brief explanation of VDM's notion of 'satisfiability' and formulates an equivalent condition for Hoare's logic. Finally, conclusions can be found in section 5.

2 A Unified Framework for Hoare's Logic and VDM

The relational framework we use to capture both Hoare's logic and VDM is extracted from the one presented in [CRZ91]. First we define the syntax and semantics of a general class of assertions. Preconditions and postconditions of specifications in Hoare's logic are identified as a subset of this class. VDM postconditions — which characterize a relation — are in a different subset, and so are the VDM preconditions. Second, we define the syntax of correctness formulae and specifications for both VDM and Hoare's logic.

2.1 Syntax and Semantics of Assertions

We assume a syntactic class $\mathcal{E}expr$ of expressions with occurrences of program variables $x \in \mathcal{V}ar$, a disjoint set of 'hooked' program variables $\overleftarrow{x}$, and another set, disjoint with the previous ones, of logical variables $x_0 \in \mathcal{L}var$. We use Σ for the set of (program) states $\sigma : \mathcal{V}ar \to \mathcal{V}al$ and Γ for the set of logical states $\gamma : \mathcal{L}var \to \mathcal{V}al$. Furthermore, we assume that an interpretation function $\mathcal{E}[\![.]\!] : \mathcal{E}expr \to (\Gamma \to ((\Sigma \times \Sigma) \to \mathcal{V}al))$ is defined such that

$$\begin{aligned}
\mathcal{E}[\![x]\!]\gamma(\sigma,\tau) &\triangleq \tau(x)\\
\mathcal{E}[\![\overleftarrow{x}]\!]\gamma(\sigma,\tau) &\triangleq \sigma(x)\\
\mathcal{E}[\![x_0]\!]\gamma(\sigma,\tau) &\triangleq \gamma(x_0)
\end{aligned}$$

The syntactic class $\mathcal{A}ssn$ of assertions, with typical element χ, is defined as follows.

Definition 2.1. *(Syntax of assertions)*
Let $e_1, e_2 \in \mathcal{E}expr$.

$$\chi ::= \mathsf{true} \mid e_1 = e_2 \mid e_1 < e_2 \mid \neg\chi \mid \chi_1 \to \chi_2 \mid \exists_{x_0}(\chi)$$

Abbreviations such as $\chi_1 \vee \chi_2$ are also included. We also allow syntactic substitutions in assertions. Let $e \in \mathcal{E}expr$ then $\chi[e/z]$ is the assertion χ with all free occurrences of z replaced by e, and a renaming of the bounded variables in χ to avoid bindings of the variables in e.

Definition 2.2. *(Semantics of assertions)*
Assertions $\chi \in \mathcal{A}ssn$ are interpreted by a truth-valued function $\mathcal{T}[\![.]\!] : \mathcal{A}ssn \to (\Gamma \to ((\Sigma \times \Sigma) \to \{tt, ff\}))$.

$$\begin{aligned}
\mathcal{T}[\![\mathsf{true}]\!]\gamma(\sigma,\tau) &\triangleq tt\\
\mathcal{T}[\![e_1 = e_2]\!]\gamma(\sigma,\tau) &\triangleq \mathcal{E}[\![e_1]\!]\gamma(\sigma,\tau) = \mathcal{E}[\![e_2]\!]\gamma(\sigma,\tau)\\
\mathcal{T}[\![e_1 < e_2]\!]\gamma(\sigma,\tau) &\triangleq \mathcal{E}[\![e_1]\!]\gamma(\sigma,\tau) < \mathcal{E}[\![e_2]\!]\gamma(\sigma,\tau)\\
\mathcal{T}[\![\neg\chi]\!]\gamma(\sigma,\tau) &\triangleq \neg\mathcal{T}[\![\chi]\!]\gamma(\sigma,\tau)\\
\mathcal{T}[\![\chi_1 \to \chi_2]\!]\gamma(\sigma,\tau) &\triangleq \mathcal{T}[\![\chi_1]\!]\gamma(\sigma,\tau) \Rightarrow \mathcal{T}[\![\chi_2]\!]\gamma(\sigma,\tau)\\
\mathcal{T}[\![\exists_{x_0}(\chi)]\!]\gamma(\sigma,\tau) &\triangleq \begin{cases} tt & \text{, if there exists a } v \in \mathcal{V}al \text{ such} \\ & \quad \text{that } \mathcal{T}[\![\chi]\!](\gamma|x_0 : v)(\sigma,\tau) \\ ff & \text{, otherwise.} \end{cases}
\end{aligned}$$

With

$$(\gamma|x_0 : v)(y_0) \triangleq \begin{cases} v & \text{, if } x_0 = y_0 \\ \gamma(y_0) & \text{, otherwise.} \end{cases}$$

We distinguish three kinds of assertions in $\mathcal{A}ssn$.

- $\mathcal{A}ssn^{\mathcal{H}}$, with typical elements φ and ψ:
  assertions in which 'hooked' variables $\overleftarrow{x}$ do not occur. For example preconditions and postconditions of Hoare-style correctness formulae and specifications.

- $\mathcal{A}ssn^{\mathcal{R}}$, with typical element ρ:
  assertions without free occurrences of logical variables x_0. For example 'postconditions' of VDM-style correctness formulae and specifications.

- $\mathcal{A}ssn^{\mathcal{J}}$, with typical element π:
  assertions in $\mathcal{A}ssn^{\mathcal{H}} \cap \mathcal{A}ssn^{\mathcal{R}}$, i.e. assertions without 'hooked' variables $\overleftarrow{x}$ and without free occurrences of logical variables x_0. For example 'preconditions' of VDM-style correctness formulae and specifications.

For these assertions the following alternative interpretation functions are defined.

$$[\![.]\!]^{\mathcal{H}} : \mathcal{A}ssn^{\mathcal{H}} \to (\Gamma \to \mathcal{P}(\Sigma))$$
$$[\![.]\!]^{\mathcal{R}} : \mathcal{A}ssn^{\mathcal{R}} \to \mathcal{P}(\Sigma \times \Sigma)$$
$$[\![.]\!]^{\mathcal{J}} : \mathcal{A}ssn^{\mathcal{J}} \to \mathcal{P}(\Sigma)$$

Definition 2.3. *(Interpretation of assertions as sets)*
Let $\varphi \in \mathcal{A}ssn^{\mathcal{H}}$, $\rho \in \mathcal{A}ssn^{\mathcal{R}}$, and $\pi \in \mathcal{A}ssn^{\mathcal{J}}$.

$$[\![\varphi]\!]^{\mathcal{H}}\gamma \triangleq \bigcap_{\sigma}\{\tau \mid \mathcal{T}[\![\varphi]\!]\gamma(\sigma,\tau)\}$$
$$[\![\rho]\!]^{\mathcal{R}} \triangleq \bigcap_{\gamma}\{(\sigma,\tau) \mid \mathcal{T}[\![\rho]\!]\gamma(\sigma,\tau)\}$$
$$[\![\pi]\!]^{\mathcal{J}} \triangleq \bigcap_{\gamma,\sigma}\{\tau \mid \mathcal{T}[\![\pi]\!]\gamma(\sigma,\tau)\}$$

If no confusion can arise we drop the superscripts.

2.2 Specifications and Correctness Formulae

We introduce two more syntactic classes. The class $\mathcal{F}orm$ of correctness formulae and the class $\mathcal{S}pec$ of specifications. A correctness formula consists of a precondition, a postcondition, and a program segment. The precondition and postcondition are elements of $\mathcal{A}ssn$. The program segment is an element, typically S, of a syntactic class $\mathcal{P}rog$. We assume $\mathcal{P}rog$ defines a relational programming language, i.e. a programming language with a relational semantics. More precisely we assume the existence of an interpretation function $\mathcal{R}[\![.]\!] : \mathcal{P}rog \to \mathcal{P}(\Sigma \times \Sigma)$.

A specification consists of a precondition, a postcondition, and an operation identifier. An operation identifier is an element of a set of names called $\mathcal{N}ame$.

Definition 2.4. *(Syntax of correctness formulae)*
Let $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, $\rho \in \mathcal{A}ssn^{\mathcal{R}}$, and $S \in \mathcal{P}rog$. The syntactic class $\mathcal{F}orm$ of correctness formulae consists of the following elements.

- $(\varphi)\ S\ (\psi)^{\mathcal{H}}$: A Hoare-style partial correctness formula.
- $(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp}$: A Hoare-style total correctness formula.
- $(\pi)\ S\ (\rho)^{\mathcal{J}}$: A VDM-style partial correctness formula.
- $(\pi)\ S\ (\rho)^{\mathcal{J}}_{\perp}$: A VDM-style total correctness formula.

Correctness formulae are interpreted by a truth-valued function $\mathcal{F}[\![.]\!] : \mathcal{F}orm \to \{tt, ff\}$, whose definition is postponed until the next sections.

Definition 2.5. *(Syntax of specifications)*
Let $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, $\rho \in \mathcal{A}ssn^{\mathcal{R}}$, and $op \in \mathcal{N}ame$. The syntactic class $\mathcal{S}pec$ of specifications consists of the following elements.

- $\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}$: A Hoare-style partial correctness specification.
- $\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}_{\perp}$: A Hoare-style total correctness specification.
- $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}$: A VDM-style partial correctness specification.
- $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}_{\perp}$: A VDM-style total correctness specification.

Specifications are interpreted as relations by $[\![.]\!] : \mathcal{S}pec \to \mathcal{P}(\Sigma \times \Sigma)$, whose definition is also postponed until the next sections.

When defining the meaning of correctness formulae and specifications we use the following notation for relations.

Definition 2.6. *(Notation for relations)*
Let $p, q \in \mathcal{P}(\Sigma)$ and $r, s \in \mathcal{P}(\Sigma \times \Sigma)$.

$$\begin{aligned}
\|p\| &\triangleq \{(\sigma,\sigma) \mid \sigma \in p\} \\
p \rightsquigarrow q &\triangleq \{(\sigma,\tau) \mid \sigma \in p \to \tau \in q\} \\
p \rightsquigarrow r &\triangleq \{(\sigma,\tau) \mid \sigma \in p \to (\sigma,\tau) \in r\} \\
r;\ s &\triangleq \{(\sigma,\tau) \mid \exists_{\zeta}((\sigma,\zeta) \in r \wedge (\zeta,\tau) \in s)\}
\end{aligned}$$

In the section on total correctness we introduce a special state $\perp$, $\perp \notin \Sigma$, to denote divergence. It has the property that $\mathcal{T}[\![\chi]\!]\gamma(\perp,\tau) = ff$ and $\mathcal{T}[\![\chi]\!]\gamma(\sigma,\perp) = ff$ for all $\gamma \in \Gamma$ and $\chi \in \mathcal{A}ssn$. We abbreviate $\Sigma \cup \{\perp\}$ as $\Sigma_{\perp}$, and write $p \rightsquigarrow_{\perp} q$ and $p \rightsquigarrow_{\perp} r$ whenever $p, q \in \mathcal{P}(\Sigma_{\perp})$ and $r \in \mathcal{P}(\Sigma_{\perp} \times \Sigma_{\perp})$.

3 Partial Correctness

Partial correctness formalisms abstract from the issue of termination, and focus on the functional behaviour. We define what partial correctness means in Hoare's logic, define partial correctness for VDM, and show how these formalisms are related.

3.1 Hoare's Logic

The intended meaning of Hoare-style partial correctness formula $(\varphi)\ S\ (\psi)^{\mathcal{H}}$ is that if the initial program state satisfies the precondition φ and if the program S terminates then the final program state must satisfy the postcondition ψ. This is captured in the following definition.

Definition 3.1. *(Hoare partial correctness formula)*
Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$\mathcal{F}[\![(\varphi)\ S\ (\psi)^{\mathcal{H}}]\!] \triangleq \forall_{\gamma}(\mathcal{R}[\![S]\!]([\![\varphi]\!]\gamma) \subseteq [\![\psi]\!]\gamma)$$

Notice that we sometimes treat $\mathcal{R}[\![.]\!]$ as a function, i.e. it is considered to be of type $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. Hence,

$$\mathcal{R}[\![S]\!]([\![\varphi]\!]\gamma) \triangleq \{\tau \mid \exists_{\sigma}(\sigma \in [\![\varphi]\!]\gamma \wedge (\sigma,\tau) \in \mathcal{R}[\![S]\!])\}$$

A partial correctness specification should specify the largest relation that satisfies the corresponding correctness formula. Or, in other words, if a program S satisfies the specification $\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}$ then $(\varphi)\ S\ (\psi)^{\mathcal{H}}$ should hold.

Definition 3.2. *(Hoare partial correctness specification)*
Let $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$ and $op \in \mathcal{N}ame$.

$$[\![\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}]\!] \triangleq \bigcap_{\gamma}([\![\varphi]\!]\gamma \rightsquigarrow [\![\psi]\!]\gamma)$$

Lemma 3.3 states that definitions 3.1 and 3.2 are consistent in the sense that they satisfy the condition mentioned above.

Lemma 3.3. Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$\forall_\gamma(\mathcal{R}[\![S]\!]([\![\varphi]\!]\gamma) \subseteq [\![\psi]\!]\gamma) \Leftrightarrow \mathcal{R}[\![S]\!] \subseteq \bigcap_{\gamma}([\![\varphi]\!] \rightsquigarrow [\![\psi]\!])$$

Proof.

$$\begin{aligned}
& \forall_\gamma(\tau \in \mathcal{R}[\![S]\!]([\![\varphi]\!]\gamma) \rightarrow \tau \in [\![\psi]\!]\gamma) \\
\Leftrightarrow\ & \forall_\gamma\forall_\sigma((\sigma \in [\![\varphi]\!]\gamma \wedge (\sigma,\tau) \in \mathcal{R}[\![S]\!]) \rightarrow \tau \in [\![\psi]\!]\gamma) \\
\Leftrightarrow\ & \forall_\gamma\forall_\sigma((\sigma,\tau) \in \mathcal{R}[\![S]\!] \rightarrow (\sigma \in [\![\varphi]\!]\gamma \rightarrow \tau \in [\![\psi]\!]\gamma)) \\
\Leftrightarrow\ & \forall_\sigma((\sigma,\tau) \in \mathcal{R}[\![S]\!] \rightarrow \forall_\gamma(\sigma \in [\![\varphi]\!]\gamma \rightarrow \tau \in [\![\psi]\!]\gamma)) \\
\Leftrightarrow\ & (\sigma,\tau) \in \mathcal{R}[\![S]\!] \rightarrow (\sigma,\tau) \in \bigcap_{\gamma}([\![\varphi]\!]\gamma \rightsquigarrow [\![\psi]\!]\gamma)
\end{aligned}$$

□

3.2 VDM

Although VDM is a total correctness formalism, it makes sense to define a partial correctness version and to compare it with Hoare's partial correctness logic. Such a partial correctness version should be consistent with total correctness VDM, i.e. if we abstract from the issue of termination both versions of VDM should specify the same operations. From definitions 4.5 and 4.7 in section 4.2 it is immediate that the following definitions of VDM-style partial correctness formulae and specifications are indeed sensible.

Definition 3.4. *(VDM partial correctness formula)*
Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$\mathcal{F}[\![(\pi)\ S\ (\rho)^{\mathcal{J}}]\!] \triangleq \|[\![\pi]\!]\|;\ \mathcal{R}[\![S]\!] \subseteq [\![\rho]\!]$$

Thus if we restrict the behaviour of program S to those starting in an initial state for which the precondition π holds, then we only observe behaviours allowed by the postcondition ρ.

Definition 3.5. *(VDM partial correctness specification)*
Let $op \in \mathcal{N}ame$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$[\![\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}]\!] \triangleq [\![\pi]\!] \rightsquigarrow [\![\rho]\!]$$

Analogous to lemma 3.3 for Hoare's logic we have a lemma to ensure that the above two definitions are consistent with each other.

Lemma 3.6. Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$\|[\![\pi]\!]\|;\ \mathcal{R}[\![S]\!] \subseteq [\![\rho]\!] \Leftrightarrow \mathcal{R}[\![S]\!] \subseteq [\![\pi]\!] \rightsquigarrow [\![\rho]\!]$$

Proof.

$$\begin{aligned}
& (\sigma,\tau) \in \|[\![\pi]\!]\|;\ \mathcal{R}[\![S]\!] \rightarrow (\sigma,\tau) \in [\![\rho]\!] \\
\Leftrightarrow\ & (\sigma \in [\![\pi]\!] \wedge (\sigma,\tau) \in \mathcal{R}[\![S]\!]) \rightarrow (\sigma,\tau) \in [\![\rho]\!] \\
\Leftrightarrow\ & (\sigma,\tau) \in \mathcal{R}[\![S]\!] \rightarrow (\sigma \in [\![\pi]\!] \rightarrow (\sigma,\tau) \in [\![\rho]\!]) \\
\Leftrightarrow\ & (\sigma,\tau) \in \mathcal{R}[\![S]\!] \rightarrow (\sigma,\tau) \in [\![\pi]\!] \rightsquigarrow [\![\rho]\!]
\end{aligned}$$

□

3.3 Translation of Partial Correctness Formulae

We adopt the convention that x, x_0, and $\overleftarrow{x}$ denote lists of variables rather than single variables. For example, in theorem 3.7 we use x_0 to denote the list of *all* free logical variables in the precondition of the Hoare formula.

The claim that VDM allows the separation of functional behaviour — defined by the postcondition — from termination — defined by the precondition — is justified by the fact that we have the assertion true as the precondition of the VDM formula in the following theorem.

Theorem 3.7. *(Translation of Hoare formulae into VDM formulae)*
Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$(\varphi)\ S\ (\psi)^{\mathcal{H}} \Leftrightarrow (\mathsf{true})\ S\ (\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi))^{\mathcal{J}}$$

Proof. Using lemma 3.3 and lemma 3.6 it suffices to prove the following equality.

$$\begin{aligned}
& \bigcap_{\gamma}([\![\varphi]\!]\gamma \rightsquigarrow [\![\psi]\!]\gamma) \\
=\ & \bigcap_{\gamma}(\{(\sigma,\tau) \mid \sigma \in [\![\varphi]\!]\gamma \rightarrow \tau \in [\![\psi]\!]\gamma\}) \\
=\ & \{(\sigma,\tau) \mid \forall_{\gamma}(\sigma \in [\![\varphi]\!]\gamma \rightarrow \tau \in [\![\psi]\!]\gamma)\} \\
=\ & [\![\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)]\!]\ \text{, with } \forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi) \in \mathcal{A}ssn^{\mathcal{R}} \\
=\ & \{(\sigma,\tau) \mid \sigma \in [\![\mathsf{true}]\!] \rightarrow (\sigma,\tau) \in [\![\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)]\!]\}\ , \\
& \text{with } \mathsf{true} \in \mathcal{A}ssn^{\mathcal{J}} \\
=\ & [\![\mathsf{true}]\!] \rightsquigarrow [\![\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)]\!]
\end{aligned}$$

□

From this theorem and lemma's 3.3 and 3.6 we immediately obtain a similar result for Hoare-style and VDM-style partial correctness specifications.

Corollary 3.8. The Hoare-style specification $\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}$ and the VDM-style specification $\{\mathsf{true}\}\ op\ \{\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)\}^{\mathcal{J}}$ are correctly implemented by the same operations.

The following theorem and corollary describe how a VDM partial correctness formula or specification can be directly translated into an equivalent Hoare-style partial correctness formula or specification. Hence, we can go from Hoare's logic to VDM and vice versa if preserving partial correctness is our only concern.

We use $x = x_0$ as an abbreviation of $\bigwedge_{i=1}^{n} x^i = x_0^i$, where $x^1, ..., x^n$ is the list of program variables in the precondition π of the VDM formula. Notice how the logical variables are used to copy the values of the program variables from the initial program state to the final program state.

Theorem 3.9. *(Translation of VDM formulae into Hoare formulae)*
Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$(\pi)\ S\ (\rho)^{\mathcal{J}} \Leftrightarrow (x = x_0 \wedge \pi)\ S\ (\rho[x_0/\overleftarrow{x}])^{\mathcal{H}}$$

Proof. Using lemma 3.3 and lemma 3.6 it suffices to prove the following equality.

$$\begin{aligned}
& [\![\pi]\!] \rightsquigarrow [\![\rho]\!] \\
= \ & \{(\sigma,\tau) \mid \sigma \in [\![\pi]\!] \rightarrow (\sigma,\tau) \in [\![\rho]\!]\} \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma((\gamma(x_0) = \sigma(x) \wedge \sigma \in [\![\pi]\!]) \rightarrow \tau \in [\![\rho[x_0/\overleftarrow{x}]]\!]\gamma)\} \\
& \text{with } \rho[x_0/\overleftarrow{x}] \in \mathcal{A}ssn^{\mathcal{H}} \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma(\sigma \in [\![\pi \wedge x = x_0]\!]\gamma \rightarrow \tau \in [\![\rho[x_0/\overleftarrow{x}]]\!]\gamma)\} \\
& \text{with } (\pi \wedge x = x_0) \in \mathcal{A}ssn^{\mathcal{H}} \\
= \ & \bigcap_\gamma \{(\sigma,\tau) \mid \sigma \in [\![\pi \wedge x = x_0]\!]\gamma \rightarrow \tau \in [\![\rho[x_0/\overleftarrow{x}]]\!]\gamma\} \\
= \ & \bigcap_\gamma ([\![\pi \wedge x = x_0]\!]\gamma \rightsquigarrow [\![\rho[x_0/\overleftarrow{x}]]\!]\gamma)
\end{aligned}$$

□

Corollary 3.10. The VDM-style specification $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}$ and the Hoare-style specification $\{x = x_0 \wedge \pi\}\ op\ \{\rho[x_0/\overleftarrow{x}]\}^{\mathcal{H}}$ are correctly implemented by the same operations.

4 Total Correctness

In order to deal adequately with nontermination (divergence) we assume a special state $\perp$ with the properties mentioned at the end of section 2, and adopt the proposed notation. As a consequence of definition 2.3 we have that $\perp \notin [\![\varphi]\!]^{\mathcal{H}}\gamma$, $(\sigma,\perp) \notin [\![\rho]\!]^{\mathcal{R}}$, $(\perp,\sigma) \notin [\![\rho]\!]^{\mathcal{R}}$, and $\perp \notin [\![\pi]\!]^{\mathcal{J}}$ (σ and τ are supposed to be elements of $\Sigma_\perp$).

We also assume a relational semantics $\mathcal{R}[\![.]\!]_\perp : \mathcal{P}rog \rightarrow \mathcal{P}(\Sigma_\perp \times \Sigma_\perp)$, with the following properties.

$$\forall_\sigma(\exists_\tau((\sigma,\tau) \in \mathcal{R}[\![S]\!]_\perp)) \tag{4}$$

$$(\sigma,\perp) \in \mathcal{R}[\![S]\!]_\perp \Leftrightarrow \forall_\tau((\sigma,\tau) \in \mathcal{R}[\![S]\!]_\perp) \tag{5}$$

Thus $\mathcal{R}[\![S]\!]_\perp$ is a total relation (property (4)). Property (5) is typical for Smyth-semantics [Smy78], which adopts the principle that if, given an initial state, a program might not terminate then it will not terminate. The reason that we choose a Smyth-semantics is that in [Jon87] Jones defines such a semantics for VDM. Furthermore we require that $\mathcal{R}[\![.]\!]_\perp$ is a *conservative* extension of $\mathcal{R}[\![.]\!]$, i.e. we require that for all $p \subseteq \Sigma$

$$\perp \notin \mathcal{R}[\![S]\!]_\perp(p) \Rightarrow \mathcal{R}[\![S]\!](p) = \mathcal{R}[\![S]\!]_\perp(p)\ . \tag{6}$$

Because $\perp \notin \mathcal{R}[\![S]\!](p)$, it follows immediately that $\mathcal{R}[\![S]\!](p) \neq \mathcal{R}[\![S]\!]_\perp(p)$ if, and only if, $\perp \in \mathcal{R}[\![S]\!]_\perp(p)$.

4.1 Hoare's Logic

The intended meaning of Hoare-style total correctness formula $(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp}$ is that if the initial program state satisfies the precondition φ then the program S terminates in a final program state which satisfies the postcondition ψ. This is captured in definition 4.1.

Definition 4.1. *(Hoare total correctness formula)*
Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$\mathcal{F}[\![(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp}]\!] \triangleq \forall_\gamma(\mathcal{R}[\![S]\!]_\perp([\![\varphi]\!]\gamma) \subseteq [\![\psi]\!]\gamma)$$

We prove the following folk theorem, which relates partial and total correctness formulae.

Lemma 4.2. Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp} \Leftrightarrow ((\varphi)\ S\ (\psi)^{\mathcal{H}} \wedge (\varphi)\ S\ (\mathsf{true})^{\mathcal{H}}_{\perp})$$

Proof. We distinguish two cases.

1. $\perp \in \mathcal{R}[\![S]\!]_\perp([\![\varphi]\!]\gamma)$.
   Because $\perp \notin [\![\psi]\!]\gamma$ and $\perp \notin [\![\mathsf{true}]\!]\gamma$ both $(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp}$ and $(\varphi)\ S\ (\mathsf{true})^{\mathcal{H}}_{\perp}$ are false.

2. $\perp \notin \mathcal{R}[\![S]\!]_\perp([\![\varphi]\!]\gamma)$.
   Because $[\![\mathsf{true}]\!]\gamma = \Sigma$ and in this case $\mathcal{R}[\![S]\!]_\perp([\![\varphi]\!]\gamma) \subseteq \Sigma$, it follows that $(\varphi)\ S\ (\mathsf{true})^{\mathcal{H}}_{\perp}$ is true. From requirement (6) it follows that $\mathcal{R}[\![S]\!]_\perp([\![\varphi]\!]\gamma) = \mathcal{R}[\![S]\!]([\![\varphi]\!]\gamma)$, which means that $(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp}$ holds if, and only if, $(\varphi)\ S\ (\psi)^{\mathcal{H}}$ holds.

□

A total correctness specification specifies the largest Smyth-relation — i.e. the largest relation on $\Sigma_\perp \times \Sigma_\perp$ satisfying requirements (4) and (5) — which satisfies the corresponding correctness formula.

Definition 4.3. *(Hoare total correctness specification)*
Let $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$ and $op \in \mathcal{N}ame$.

$$[\![\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}_{\perp}]\!] \triangleq \bigcap_\gamma([\![\varphi]\!]\gamma \rightsquigarrow_\perp [\![\psi]\!]\gamma)$$

Lemma 4.4 ensures that definitions 4.1 and 4.3 are consistent.

Lemma 4.4. Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$\forall_\gamma(\mathcal{R}[\![S]\!]_\perp([\![\varphi]\!]\gamma) \subseteq [\![\psi]\!]\gamma) \Leftrightarrow \mathcal{R}[\![S]\!]_\perp \subseteq \bigcap_\gamma([\![\varphi]\!] \rightsquigarrow_\perp [\![\psi]\!])$$

Proof. Analogous to the proof of lemma 3.3. □

4.2 VDM

In [Jon87, Jon90] Jones defines what it means for an operation S to be a correct implementation of a specification $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}_{\perp}$. Within the relational framework of section 2, Jones' requirement becomes (recall that $\mathcal{R}[\![S]\!]_{\perp}$ is a total relation and $[\![\rho]\!]^{\mathcal{R}} \subseteq \Sigma \times \Sigma$)

$$\forall_{\sigma,\tau}(\sigma \in [\![\pi]\!]^{\mathcal{J}} \rightarrow ((\sigma,\tau) \in \mathcal{R}[\![S]\!]_{\perp} \rightarrow (\sigma,\tau) \in [\![\rho]\!]^{\mathcal{R}})) \tag{7}$$

Because (7) is equivalent with

$$\forall_{\sigma,\tau}(((\sigma,\sigma) \in \|[\![\pi]\!]^{\mathcal{J}}\| \wedge (\sigma,\tau) \in \mathcal{R}[\![S]\!]_{\perp}) \rightarrow (\sigma,\tau) \in [\![\rho]\!]^{\mathcal{R}})$$

it is easily seen that definition 4.5 is indeed equivalent with the VDM condition for implementation correctness.

Definition 4.5. *(VDM total correctness formula)*
Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$\mathcal{F}[\![(\pi)\ S\ (\rho)^{\mathcal{J}}_{\perp}]\!] \triangleq \|[\![\pi]\!]\|;\ \mathcal{R}[\![S]\!]_{\perp} \subseteq [\![\rho]\!]$$

For VDM total correctness formulae we have an analogous result to lemma 4.2 for Hoare's logic.

Lemma 4.6. Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$(\pi)\ S\ (\rho)^{\mathcal{J}}_{\perp} \Leftrightarrow ((\pi)\ S\ (\rho)^{\mathcal{J}} \wedge (\pi)\ S\ (\mathsf{true})^{\mathcal{J}}_{\perp})$$

Proof.

$$\begin{aligned}
& \|[\![\pi]\!]^{\mathcal{J}}\|;\ \mathcal{R}[\![S]\!]_{\perp} \subseteq [\![\rho]\!]^{\mathcal{R}} \\
\Leftrightarrow\ & \forall_{\sigma,\tau}((\sigma \in [\![\pi]\!]^{\mathcal{J}} \wedge (\sigma,\tau) \in \mathcal{R}[\![S]\!]_{\perp}) \rightarrow (\sigma,\tau) \in [\![\rho]\!]^{\mathcal{R}}) \\
\Leftrightarrow\ & \forall_{\sigma,\tau}(\tau \in \mathcal{R}[\![S]\!]_{\perp}(\{\sigma\}) \rightarrow (\sigma \in [\![\pi]\!]^{\mathcal{J}} \rightarrow (\sigma,\tau) \in [\![\rho]\!]^{\mathcal{R}}))
\end{aligned}$$

In case the initial program state $\sigma \notin [\![\pi]\!]$ the above theorem clearly holds. Therefore assume that $\sigma \in [\![\pi]\!]$. We consider two cases of the last formula.

1. $\perp \in \mathcal{R}[\![S]\!]_{\perp}(\{\sigma\})$.
   Because $(\sigma,\perp) \notin [\![\rho]\!]^{\mathcal{R}}$ the correctness formula $(\pi)\ S\ (\rho)^{\mathcal{J}}_{\perp}$ is false. Likewise, because $(\sigma,\perp) \notin [\![\mathsf{true}]\!]^{\mathcal{R}}$, the correctness formulae $(\pi)\ S\ (\mathsf{true})^{\mathcal{J}}_{\perp}$ is false.
2. $\perp \notin \mathcal{R}[\![S]\!]_{\perp}(\{\sigma\})$.
   Because for all $\tau \neq \perp$ we have that $(\sigma,\tau) \in [\![\mathsf{true}]\!]^{\mathcal{R}}$, the correctness formula $(\pi)\ S\ (\mathsf{true})^{\mathcal{J}}_{\perp}$ holds. From requirement (6) it immediately follows that $(\pi)\ S\ (\rho)^{\mathcal{J}}_{\perp}$ holds if, and only if, $(\pi)\ S\ (\rho)^{\mathcal{J}}$ holds.

□

Lemma 4.8 guarantees that the interpretation of VDM total correctness specifications as defined in definition 4.7 is consistent with the definition of VDM total correctness formulae.

Definition 4.7. *(VDM total correctness specification)*
Let $op \in \mathcal{N}ame$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$[\![\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}_{\perp}]\!] \triangleq [\![\pi]\!] \rightsquigarrow_{\perp} [\![\rho]\!]$$

Lemma 4.8. Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$\|[\![\pi]\!]\|;\ \mathcal{R}[\![S]\!]_{\perp} \subseteq [\![\rho]\!] \Leftrightarrow \mathcal{R}[\![S]\!]_{\perp} \subseteq [\![\pi]\!] \rightsquigarrow_{\perp} [\![\rho]\!]$$

Proof. Analogous to the proof of lemma 3.6. □

4.3 Translation of Total Correctness Formulae

In this section we give theorems for total correctness similar to the ones for partial correctness in section 3. First, we show how to translate a Hoare-style total correctness formula into an equivalent VDM-style total correctness formula.

Theorem 4.9. *(Translation of Hoare formulae into VDM formulae)*
Let $S \in \mathcal{P}rog$ and $\varphi, \psi \in \mathcal{A}ssn^{\mathcal{H}}$.

$$(\varphi)\ S\ (\psi)^{\mathcal{H}}_{\perp} \Leftrightarrow (\exists_{x_0}(\varphi))\ S\ (\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi))^{\mathcal{J}}_{\perp}$$

Proof. First we apply lemma 4.2, and then proceed in two major steps.

1. $(\varphi)\ S\ (\psi)^{\mathcal{H}} \Leftrightarrow (\exists_{x_0}(\varphi))\ S\ (\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi))^{\mathcal{J}}$. Following the proof of theorem 3.7 we proceed as follows.

$$\begin{aligned}
& \bigcap_{\gamma}(\llbracket\varphi\rrbracket\gamma \rightsquigarrow \llbracket\psi\rrbracket\gamma) \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma \rightarrow \tau \in \llbracket\psi\rrbracket\gamma)\} \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma \rightarrow \forall_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma \rightarrow \tau \in \llbracket\psi\rrbracket\gamma))\} \\
= \ & \{(\sigma,\tau) \mid \exists_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma) \rightarrow (\sigma,\tau) \in \llbracket\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)\rrbracket\} \\
= \ & \llbracket\exists_{x_0}(\varphi)\rrbracket \rightsquigarrow \llbracket\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)\rrbracket \text{ , with } \exists_{x_0}(\varphi) \in \mathcal{A}ssn^{\mathcal{J}}
\end{aligned}$$

2. $(\varphi)\ S\ (\mathsf{true})^{\mathcal{H}}_{\perp} \Leftrightarrow (\exists_{x_0}(\varphi))\ S\ (\mathsf{true})^{\mathcal{J}}_{\perp}$. This is proved as follows.

$$\begin{aligned}
& \bigcap_{\gamma}(\llbracket\varphi\rrbracket\gamma \rightsquigarrow_{\perp} \llbracket\mathsf{true}\rrbracket\gamma) \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma \rightarrow \tau \in \llbracket\mathsf{true}\rrbracket\gamma)\} \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma \rightarrow (\sigma,\tau) \in \llbracket\mathsf{true}\rrbracket)\}, \text{ with } \mathsf{true} \in \mathcal{A}ssn^{\mathcal{R}} \\
= \ & \{(\sigma,\tau) \mid \exists_\gamma(\sigma \in \llbracket\varphi\rrbracket\gamma) \rightarrow (\sigma,\tau) \in \llbracket\mathsf{true}\rrbracket\} \\
= \ & \llbracket\exists_{x_0}(\varphi)\rrbracket \rightsquigarrow_{\perp} \llbracket\mathsf{true}\rrbracket, \text{ with } \exists_{x_0}(\varphi) \in \mathcal{A}ssn^{\mathcal{J}}
\end{aligned}$$

An application of lemma 4.6 concludes the proof. □

Corollary 4.10. The Hoare-style specification $\{\varphi\}\ op\ \{\psi\}^{\mathcal{H}}_{\perp}$ and the VDM-style specification $\{\exists_{x_0}(\varphi)\}\ op\ \{\forall_{x_0}(\varphi[\overleftarrow{x}/x] \rightarrow \psi)\}^{\mathcal{J}}_{\perp}$ are correctly implemented by the same operations.

The following theorem shows how to translate VDM-style total correctness formulae into Hoare-style total correctness formulae. Hence, we can go back and forth between VDM and Hoare's logic.

Theorem 4.11. *(Translation of VDM formulae into Hoare formulae)*
Let $S \in \mathcal{P}rog$, $\pi \in \mathcal{A}ssn^{\mathcal{J}}$, and $\rho \in \mathcal{A}ssn^{\mathcal{R}}$.

$$(\pi)\ S\ (\rho)^{\mathcal{J}}_{\perp} \Leftrightarrow (\pi \wedge x = x_0)\ S\ (\rho[x_0/\overleftarrow{x}])^{\mathcal{H}}_{\perp}$$

Proof. First we apply lemma 4.6, and then proceed in two steps.

1. $(\pi)\ S\ (\rho)^{\mathcal{J}} \Leftrightarrow (\pi \wedge x = x_0)\ S\ (\rho[x_0/\overleftarrow{x}])^{\mathcal{H}}$. See theorem 3.9.

2. $(\pi)\ S\ (\mathsf{true})^{\mathcal{J}}_{\perp} \Leftrightarrow (\pi \wedge x = x_0)\ S\ (\mathsf{true})^{\mathcal{J}}_{\perp}$. This is proved as follows.

$$
\begin{aligned}
& [\![\pi]\!] \rightsquigarrow_{\perp} [\![\mathsf{true}]\!] \\
= \ & \{(\sigma,\tau) \mid \sigma \in [\![\pi]\!] \rightarrow (\sigma,\tau) \in [\![\mathsf{true}]\!]\} \\
= \ & \{(\sigma,\tau) \mid \forall_\gamma(\sigma \in [\![\pi]\!]\gamma \rightarrow \tau \in [\![\mathsf{true}]\!]\gamma\}, \text{ with } \pi, \mathsf{true} \in \mathcal{A}ssn^{\mathcal{H}} \\
= \ & \bigcap_\gamma(\{(\sigma,\tau) \mid \exists_\gamma(\sigma \in [\![\pi]\!]\gamma) \rightarrow \tau \in [\![\mathsf{true}]\!]\gamma\}) \\
& \text{there are no free logical variables in true} \\
= \ & \bigcap_\gamma(\{(\sigma,\tau) \mid \sigma \in [\![\exists_{x_0}(\pi \wedge x = x_0)]\!]\gamma \rightarrow \tau \in [\![\mathsf{true}]\!]\gamma)\}) \\
& \text{there are no free logical variables in } \pi \\
= \ & \bigcap_\gamma([\![\exists_{x_0}(\pi \wedge x = x_0)]\!]\gamma \rightsquigarrow_{\perp} [\![\mathsf{true}]\!]\gamma)
\end{aligned}
$$

One application of lemma 4.2 concludes the proof. □

Corollary 4.12. The VDM-style specification $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}_{\perp}$ and the Hoare-style specification $\{\pi \wedge x = x_0\}\ op\ \{\rho[x_0/\overleftarrow{x}]\}^{\mathcal{H}}_{\perp}$ are correctly implemented by the same operations.

4.4 Realizability

Consider the following VDM-style specification.

$$\{\mathsf{true}\}\ op\ \{\mathsf{false}\}^{\mathcal{J}} \tag{8}$$

According to definition 3.5 this specification denotes the empty relation. If abort is a (nonterminating) program with partial correctness semantics $\mathcal{R}[\![\mathrm{abort}]\!] = \emptyset$ then (8) is correctly implemented by abort. Admittedly abort doesn't seem to be a very useful implementation, but the point is that there *exists* a program which satisfies (8). In this sense (8) is *realizable*. The total correctness specification

$$\{\mathsf{true}\}\ op\ \{\mathsf{false}\}^{\mathcal{J}}_{\perp}\ , \tag{9}$$

however, is *unrealizable*. The reason for this is that (9) specifies a partial relation, viz. $\{\perp\} \times \Sigma_\perp$, but $\mathcal{R}[\![S]\!]_\perp$ defines a total relation for all S (see (4)).

In VDM [Jon90] realizable and unrealizable specifications are separated by the *satisfiability* requirement. A specification $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}_{\perp}$ is satisfiable if, and only if,

$$\forall_\sigma(\sigma \in [\![\pi]\!] \rightarrow \exists_\tau((\sigma,\tau) \in [\![\rho]\!])) \tag{10}$$

Thus $\{\pi\}\ op\ \{\rho\}^{\mathcal{J}}_{\perp}$ is satisfiable if $[\![\rho]\!]$ is total on $[\![\pi]\!]$. Condition (10) is expressed by $\pi[\overleftarrow{x}/x] \rightarrow \exists_x(\rho)$.

Hoare's logic has the same problem. The specification $\{\mathsf{true}\}\ op\ \{\mathsf{false}\}^{\mathcal{H}}_{\perp}$ is not realizable, because it specifies the same partial relation as (9). Following Jones, we define a condition for Hoare's logic similar to requirement (10) for VDM:

$$\forall_{\sigma,\gamma}(\sigma \in [\![\varphi]\!]\gamma \rightarrow \exists_\tau(\tau \in [\![\psi]\!]\gamma)) \tag{11}$$

This requirement is expressed by $\varphi \rightarrow \exists_y(\psi[y/x])$, where variables in the list y don't occur in ψ. However, for Hoare's logic the issue of realizability is more complicated. Consider the following specification.

$$\{\mathsf{true}\}\ op\ \{x = x_0\}^{\mathcal{H}}_{\perp}$$

This specification satisfies requirement (11), but is nevertheless unrealizable because it is denoted by *partial* relation:

$$\bigcap_{\gamma}(\Sigma \rightsquigarrow_{\perp} [\![x = x_0]\!]\gamma) = \bigcap_{\gamma}\{(\sigma,\tau)|\sigma \neq \perp \rightarrow \tau(x) = \gamma(x_0)\} = \{\perp\} \times \Sigma_{\perp} .$$

Hence, this specification is not satisfiable because programs are denoted by *total* relations (requirement (4)). This is explained by the fact that logical variables do not occur in programs, and therefore a correct implementation must nondeterministically guess the value of x_0. Requirement (11) is sufficient (and necessary) in case logical variables don't appear in the specification. If logical variables are present we still need to check whether the specification denotes a total relation.

5 Conclusions

We presented a unified framework for Hoare's logic and VDM, and showed how to translate correctness formulae and specifications from one formalism into the other. An interesting observation is that for VDM there exists a simple requirement which is necessary and sufficient to guarantee realizability of a specification, whereas for Hoare's logic a similar requirement is not so easily found.

In case we have an *adaptation complete* proof system for Hoare's logic such as the one in [Apt81] theorems 3.7 and 4.9 are not needed. A proof system is adaptation complete whenever if $(\varphi)\ S\ (\psi)$ implies $(\varphi')\ S\ (\psi')$ then $(\varphi')\ S\ (\psi')$ is derivable from $(\varphi)\ S\ (\psi)$ (cf. [Old83]). [2] This can be seen as follows. Suppose we want to translate the Hoare formula $(\varphi)\ S\ (\psi)^{\mathcal{H}}$ using only theorem 3.9. In order to apply theorem 3.9 the assertions in the Hoare formula have to be of a particular format. Because we assume that the proof system is adaptation complete, it is sufficient to show that $(\varphi)\ S\ (\psi)^{\mathcal{H}}$ implies $(\varphi')\ S\ (\psi')^{\mathcal{H}}$ with φ' and ψ' in the specific format needed in theorem 3.9. According to theorem 3.7 an equivalent VDM formula exists, say $(\pi)\ S\ (\rho)^{\mathcal{J}}$. Theorem 3.9 provides us with another equivalent Hoare formula $(\varphi'')\ S\ (\psi'')^{\mathcal{H}}$ which obviously has the format required by this theorem. Hence, $(\varphi'')\ S\ (\psi'')^{\mathcal{H}}$ is the formula $(\varphi')\ S\ (\psi')^{\mathcal{H}}$ we were looking for.

Acknowledgement

The author would like to thank Kai Engelhardt and Willem-Paul de Roever for many stimulating discussions and their helpful comments. Willem-Paul de Roever suggested the topic of this paper, which builds upon joint research.

References

[Apt81] Apt, K. R.: Ten Years of Hoare's Logic: A Survey — Part I. *ACM TOPLAS*, 4, 431–483 (1981).

[Bak80] Bakker, J. W. de: *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[2] Adaptation completeness is proved using the assumption that the logic of the assertion language is complete. Thus adaptation completeness is relative w.r.t. the underlying proof system of the assertion language.

[CRZ91] Coenen, J., Roever, W.-P. de, Zwiers, J.: Assertional Data Reification Proofs: Survey and Perspective. *Proc. 4th BCS-FACS Refinement Workshop*, pp. 97–114, Workshops in Computing, Springer-Verlag 1991.

[Gor75] Gorelick, G. A.: A Complete Axiomatic System for Proving Assertions about Recursive and Non-Recursive Programs. Tech. Report No. 75, Dept. of Computer Science, University of Toronto, 1975.

[Hoa69] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *C. ACM*, **12**, 576–580 (1969).

[Hoa71] Hoare, C. A. R.: Procedures and Parameters: An Axiomatic Approach. *Symp. on Semantics of Algorithmic Languages*, pp. 102-116, LNM 188, Springer-Verlag 1971.

[Jon87] Jones, C. B.: VDM Proof Obligations and their Justification. *Proc. VDM-Europe Symposium*, pp. 260–286, LNCS 252, Springer-Verlag 1987.

[Jon90] Jones, C. B.: *Systematic Software Development Using VDM* (second edition). Prentice-Hall, 1990.

[Old83] Olderog, E.-R.: On the Notion of Expressiveness and the Rule of Adaptation. *Theor. Comp. Sci.*, **24**, 337–347 (1983).

[Smy78] Smyth, M. B.: Power Domains. *J. Comp. & Sys. Sci.*, **16**, 23–36 (1978).

Chapter 3

Parameterized Semantics for Fault Tolerant Real-Time Systems

This chapter is a revised version of:

J. COENEN AND J. HOOMAN.
Parameterized Semantics for Fault Tolerant Real-Time Systems.
Formal Techniques in Real-Time and Fault Tolerant Systems,
(J. Vytopil ed.), pp. 51–78, Kluwer Academic Press 1993.

Parameterized Semantics for Fault Tolerant Real-Time Systems

J. Coenen[1] & J. Hooman[2]

Department of Mathematics 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. In this semantics we allow the occurrence of faults, due to faults of the underlying execution mechanism, and we describe the effect of these faults 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. Next we introduce two parameters in the semantics that restrict the way in which variables and communication channels can be affected by faults. These parameters provide an easy way to incorporate some interesting fault hypotheses within the semantics.

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. [15, 4, 10], or with real-time requirements, e.g. [16, 8, 12], 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 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.

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

[2]Supported by ESPRIT-BRA project 3096: "Formal Methods for the Development of Distributed Real-Time Systems." E-mail: wsinjh@win.tue.nl

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 [6]. In that paper a probabilistic (quantitative) approach is presented, whereas we are mainly concerned with the qualitative aspects of fault tolerance.

To motivate our semantic model we describe, by means of an example, how we would like to reason about fault-tolerant real-time systems. We consider concurrent systems in which parallel processes communicate by message passing along unidirectional channels. Communication is synchronous, i.e., both sender and receiver have to wait until a corresponding partner is available. We illustrate our approach using specifications of the form S **sat** φ, where S is a program and φ a sentence in a first-order predicate logic. Informally, S **sat** φ is valid if φ holds in any execution of S. To express the timed communication behaviour of programs, the logic includes the following predicates.

- (c, v) **at** t to denote the start of a communication along channel c with value v at time t.
- **await** $c?$ **at** t to express that a process starts waiting to receive a value along channel c at time t until the communication takes place.
- **await** $(c!, v)$ **at** t to express that a process starts waiting to send the value v along channel c at time t until the communication takes place.

Let $[t_1, t_2]$ denote a closed interval of time points. For a predicate P **at** t we define the following abbreviations.

- P **at** $[t_1, t_2] \triangleq \forall t, t_1 \leq t \leq t_2 : P$ **at** t
- P **in** $[t_1, t_2] \triangleq \exists t, t_1 \leq t \leq t_2 : P$ **at** t

In this paper we assume maximal progress which means that a process only waits if it tries to communicate and no communication partner is available. Communication takes place as soon as possible, i.e., as soon as both partners are ready to communicate. This assumption leads to the following proposition.

Proposition 1.1

$$\textbf{await } (c!, exp) \textbf{ in } [t_1, t_2] \wedge \textbf{await } c? \textbf{ in } [t_1, t_3] \rightarrow (c, exp) \textbf{ in } [t_1, \min(t_2, t_3)]$$

■

As an example, we design a program P such that if P receives input v along channel *in* then it will be ready to send the value $f(v)$ along channel *out* in less than T time units. Formally, P **sat** $\varphi(t)$, where

$$\varphi(t) \triangleq (in, v) \textbf{ at } t \rightarrow \textbf{await } (out!, f(v)) \textbf{ in } [t, t + T]$$

Free variables, such as v and t in the specification above, are implicitly universally quantified. Using a formal method for real-time systems (see, e.g., [12, 7]) we could now derive a program S satisfying this specification for suitable values of T. In such a verification method there is usually an implicit assumption that the underlying execution mechanism of programs is correct. In this paper, however, we want to take these faults into account and make assumptions about faults explicit. To refer to programs we use the naming construct $\langle P \Leftarrow S \rangle$ which assigns the name P to the program S. Then the occurrence of faults is expressed in the logic by the predicate

- **fail**(P) **at** t to denote the failure of a process with name P at time t.

The main aim of this paper is to give a semantics for programs which does not only describe the normal executions of the program, as in traditional semantic models, but also all possible executions in which the program fails. Then $\langle P \Leftarrow S \rangle$ **sat** ψ is valid if ψ holds in any execution of S, including those in which there are faults. Because the behaviour of a program that fails can be arbitrary, the assertion ψ will in general select a subset of all possible executions by means of a fault hypothesis. Hence in ψ we have to express explicitly what is assumed about faults. For instance in our example we can use the fault hypothesis $(\neg$**fail**$(P))$ **at** $[t - T_P, t + T_P]$, for some parameter T_P, and obtain the specification

$$\langle P \Leftarrow S \rangle \textbf{ sat } (\neg\textbf{fail}(P)) \textbf{ at } [t - T_P, t + T_P] \rightarrow \varphi(t)$$

Clearly $\varphi(t)$ need not hold if a fault occurs in the interval $[t - T_P, t + T_P]$. Therefore we will derive a program that can tolerate one fault. This can be achieved using a Triple Modular Redundancy (TMR) system. Instead of a single process S we take three copies, S_1, S_2 and S_3, of S, where S_i is obtained from S by replacing in by in_i and out by out_i, for $i = 1, 2, 3$. Then the TMR system consists of five processes, as depicted in figure 1. In the first place there is a distribution node D with program

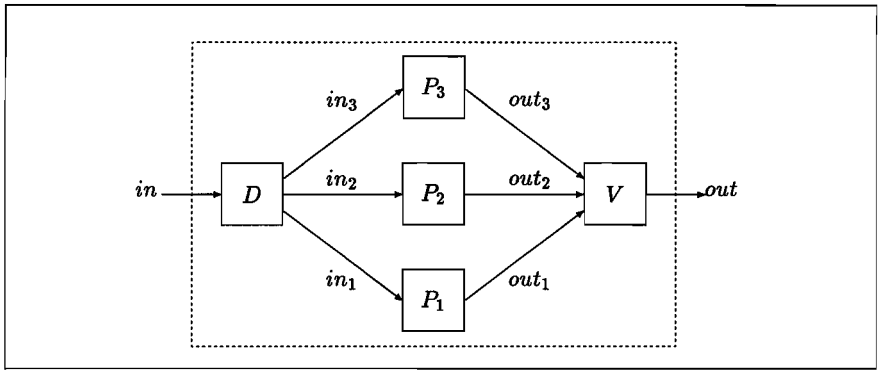

Figure 1: *TMR system*

S_0 which copies the input of channel in on three channels in_1, in_2, and in_3 provided there is no fault during a certain period. Using parameter T_D this leads to

$$\langle D \Leftarrow S_0 \rangle \textbf{ sat } (\neg\textbf{fail}(D)) \textbf{ at } [t - T_D, t + T_D] \rightarrow \varphi_D(t)$$

where $\varphi_D(t) \triangleq (in, v)$ **at** $t \rightarrow \bigwedge_{i=1}^{3}$ **await** $(in_i!, v)$ **at** $(t + T_D)$. Process $\langle P_i \Leftarrow S_i \rangle$, for $i = 1, 2, 3$, is ready to receive a message on channel in_i at least once every T_P time

units. If a value v is received, it offers $f(v)$ on channel out_i in less than T_f time units, again using a suitable fault hypothesis. Thus we have

$$\langle P_i \Leftarrow S_i \rangle \textbf{ sat } (\neg \textbf{fail}(P_i)) \textbf{ at } [t - T_P, t + T_P] \rightarrow \varphi_i(t)$$

where

$$\begin{array}{l} \varphi_i(t) \triangleq \\ \quad (\textbf{await } in_i? \textbf{ in } [t, t + T_P]) \\ \quad \wedge ((in_i, v) \textbf{ at } t \rightarrow \textbf{await } (out_i!, f(v)) \textbf{ in } [t, t + T_f]) \ . \end{array}$$

The voter V is implemented by a program S_4. Given a suitable fault hypothesis, it is ready to receive a value on each of the channels out_1, out_2, and out_3 at least once every T_V time units. If it receives the same input on two different channels during a period of at most τ_1 time units, then it offers this value on channel out in less than τ_2 time units. Formally, the voter is specified by

$$\langle V \Leftarrow S_4 \rangle \textbf{ sat } (\neg \textbf{fail}(V)) \textbf{ at } [t - T_V, t + T_V] \rightarrow \varphi_V(t)$$

where $\varphi_V(t)$ is defined as follows.

$$\begin{array}{l} \varphi_V(t) \triangleq \bigwedge_{i=1}^{3} \textbf{await } out_i? \textbf{ in } [t, t + T_V]) \\ \quad \wedge (\exists i, j, i \neq j : \ (out_i, u) \textbf{ in } [t, t + \tau_1] \wedge (out_j, u) \textbf{ in } [t, t + \tau_1] \\ \qquad\qquad \rightarrow \textbf{await } (out!, u) \textbf{ in } [t + \tau_1, t + \tau_1 + \tau_2]) \end{array}$$

Observe that for each process the specification only refers to the process name and the channels of the process itself. Then we can take the conjunction of the specifications for the parallel composition of these processes. This leads to

$$\begin{array}{l} \langle D \Leftarrow S_0 \rangle \| \langle P_1 \Leftarrow S_1 \rangle \| \langle P_2 \Leftarrow S_2 \rangle \| \langle P_3 \Leftarrow S_3 \rangle \| \langle V \Leftarrow S_4 \rangle \\ \quad \textbf{sat} \quad (\neg \textbf{fail}(D) \textbf{ at } [t - T_D, t + T_D] \rightarrow \varphi_D(t)) \\ \qquad \wedge (\bigwedge_{i=1}^{3} \neg \textbf{fail}(P_i) \textbf{ at } [t - T_P, t + T_P] \rightarrow \varphi_i(t)) \\ \qquad \wedge (\neg \textbf{fail}(V) \textbf{ at } [t - T_V, t + T_V] \rightarrow \varphi_V(t)) \end{array}$$

To derive $\varphi(t)$ we consider the following fault hypothesis.

$$\begin{array}{l} FH(t) \triangleq (\neg \textbf{fail}(D)) \textbf{ at } [t - T_D, t + T_D] \\ \quad \wedge (\exists i, j, i \neq j : \ (\neg \textbf{fail}(P_i)) \textbf{ at } [t + T_D - T_P, t + T_D + 2T_P] \\ \qquad\qquad \wedge (\neg \textbf{fail}(P_j)) \textbf{ at } [t + T_D - T_P, t + T_D + 2T_P]) \\ \qquad \wedge (\neg \textbf{fail}(V)) \textbf{ at } [t + T_D - T_V, t + T_D + T_P + T_f + T_V] \end{array}$$

From the previous specification we can then derive

$$\begin{array}{l} \langle D \Leftarrow S_0 \rangle \| \langle P_1 \Leftarrow S_1 \rangle \| \langle P_2 \Leftarrow S_2 \rangle \| \langle P_3 \Leftarrow S_3 \rangle \| \langle V \Leftarrow S_4 \rangle \\ \textbf{sat } FH(t) \rightarrow \\ \quad \varphi_D(t) \wedge (\forall t_1 \in [t + T_D, t + T_D + T_P + T_f] : \varphi_V(t_1)) \\ \quad \wedge (\exists i, j, i \neq j : \ (\forall t_0 \in [t + T_D, t + T_D + T_P] : \varphi_i(t_0) \wedge \varphi_j(t_0))) \end{array}$$

To obtain φ we use the following proposition.

Proposition 1.2
If $\tau_1 \geq T_P + T_f + T_V$ and $T_D + \tau_1 + \tau_2 \leq T$, then

$$\begin{array}{ll} & ((\exists i, j, i \neq j : (\forall t_0 \in [t + T_D, t + T_D + T_P] : \varphi_i(t_0) \wedge \varphi_j(t_0))) \\ & \wedge \varphi_D(t) \wedge (\forall t_1 \in [t + T_D, t + T_D + T_P + T] : \varphi_V(t_1))) \\ \rightarrow & \varphi(t) \end{array}$$

■

Hence by proposition 1.2 we obtain

$$\langle D \Leftarrow S_0 \rangle \| \langle P_1 \Leftarrow S_1 \rangle \| \langle P_2 \Leftarrow S_2 \rangle \| \langle P_3 \Leftarrow S_3 \rangle \| \langle V \Leftarrow S_4 \rangle \\ \textbf{sat } FH(t) \rightarrow \varphi(t)$$

provided $\tau_1 \geq T_P + T_f + T_V$ and $T_D + \tau_1 + \tau_2 \leq T$.

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. [14])

- 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

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

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)

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

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 a 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 a 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. [4, 2, 1, 13, 17, 5]).

The remainder of this report is organized as follows. In section 2, a programming language is defined, inspired by OCCAM [9]. We also give an informal explanation of the language constructs under the assumption that faults don't occur. In section 3 we introduce the computational model, and in section 4 we define the semantics of the programming language under assumption that faults do not occur. Faults are taken into consideration in section 5, where we define the general semantics of the programming language. This semantics is essentially the one presented in [3]. Whenever appropriate we discuss alternative choices for the assumptions that are implicit in the semantics. In section 6 we parameterize the semantics in such a way that it includes the semantics of section 5 as a special case by selecting the right parameters. Conclusions are present in section 7, 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
$::= \quad \mathbf{skip} \mid \mathbf{delay}\, e \mid x := e \mid c!e \mid c?x \mid S_1; S_2 \mid ALT \mid *ALT$

Alternative ALT
$::= \quad [[]_{i=1}^{n} b_i \rightarrow S_i] \mid [[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i [] b_0; \mathbf{delay}\, e \rightarrow S_0]$

Network N
$::= \quad \langle P \Leftarrow S \rangle \mid N_1 \parallel N_2$

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.

2.1 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.

2.2 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 Computational Model

We define explain the computational model that is used in the remainder to define the semantics of RT programs.

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_c,

K_d, K_f, K_g, 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 denotes the final program state. In case the program does not terminate s is undefined.

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

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

If $t > |\sigma|$ then $\sigma \downarrow t$ is undefined.

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

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

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

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

The SEQ operator is associative, i.e.

Proposition 3.1

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

■

4 Normal Semantics

The semantics of an RT program is typically defined in two steps. First, we define the normal semantics of the programming language as described in section 2, i.e. the semantics when faults do not occur. This is done by defining the interpretation function $\mathcal{M}[\![\,.\,]\!] : \mathrm{RT} \to \mathcal{P}(\Delta)$. Second, in section 5 we define the interpretation function $\mathcal{M}^\dagger[\![\,.\,]\!] : \mathrm{RT} \to \mathcal{P}(\Delta)$ 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, i.e.

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

Hence, for all RT programs S it is guaranteed that $\mathcal{M}[\![S]\!] \subseteq \mathcal{M}^\dagger[\![S]\!]$.

4.1 Skip, Delay, and Assignment

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

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

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

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

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

4.2 Communication

Recall from section 2 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.

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 ($c \in CHAN$):

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

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

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

So, the complete normal behaviour of the receive statement is

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

The normal behaviour of a receive statement is pictured in figure 2.

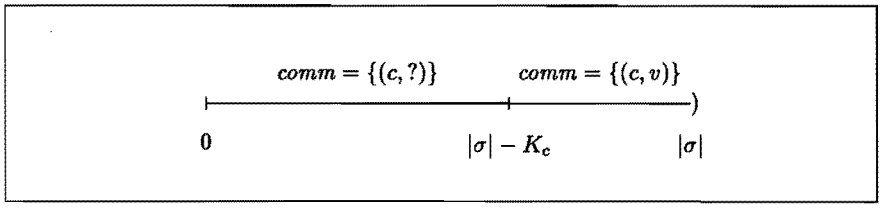

Figure 2: *A normal sequence σ of a receive statement $c?x$*

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.

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

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

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

4.3 Sequential Composition

The normal semantics of sequential composition of two program fragments is simply defined as follows.

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

Observe that sequential composition itself doesn't consume time. Hence, faults occur in the component statements only. As a consequence of proposition 3.1, we may conclude that sequential composition is associative.

Proposition 4.1

$$\mathcal{M}[\![(S_0;\ S_1);\ S_2]\!] = \mathcal{M}[\![S_0;\ (S_1;\ S_2)]\!]$$

■

4.4 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.

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

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

$$\begin{array}{l} \mathit{Select}(ALT) \triangleq \\ \qquad \{(s^0, \sigma, s) \mid \text{there exists an } i \in \{1, \ldots, n\} \text{ such that} \\ \qquad\qquad\qquad\qquad s^0(b_i) \wedge (s^0, \sigma, s) \in \mathcal{M}[\![S_i]\!]\} \\ \cup\ \ \{(s^0, \sigma, s^0) \mid |\sigma| = 0 \wedge \bigvee_{i=1}^{n} \neg s^0(b_i)\} \end{array}$$

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

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

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

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

The second behaviour is defined as the concatenation of three behaviors

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

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

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

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

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

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

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

$$\begin{aligned}
&\mathit{TimeOut}(\mathit{ALT}) \triangleq \\
&\quad \{(s^0, \sigma, s) \in \mathit{Wait}(\mathit{ALT}) \mid s^0(b) \wedge |\sigma| = \min(s^0(e), 0)\}
\end{aligned}$$

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

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

4.5 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:

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

Because evaluation of the boolean guards takes $K_g > 0$ time greatest fixed-point exists and is not empty (cf. [7]).

4.6 Networks

As explained in previous sections, the naming construct is not executed or implemented, but only included to facilitate reasoning over programs. Consequently, it does not affect the communication behaviour or the program states. Because in the normal semantics the *fail*-field will always be empty, it follows that the naming construct has no effect at all with respect to the normal semantics. In section 5 we shall see that naming does have an affect to fault semantics, and therefore its introduction is justified.

$$\mathcal{M}[\![\langle P \Leftarrow S \rangle]\!] \triangleq \mathcal{M}[\![S]\!]$$

The parallel composition operator doesn't consume time. 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.

$$\begin{aligned}
&\mathcal{M}[\![N_1 \parallel N_2]\!] \triangleq \\
&\{(s^0, \sigma, s) \mid \text{ there exists } (s_i^0, \sigma_i, s_i) \in \mathcal{M}^\dagger[\![N_i]\!] \text{ such that} \\
&\qquad |\sigma| = \max(|\sigma_1|, |\sigma_2|) \\
&\qquad \wedge (x \in var(N_i) \rightarrow (s^0(x) = s_i^0(x) \wedge s(x) = s_i(x))) \\
&\qquad \wedge (x \notin var(N_1, N_2) \rightarrow s(x) = s^0(x)) \qquad (1) \\
&\qquad \text{and for all } t \in [0, |\sigma|),\ c \in CHAN, \text{ and } v \in VAL: \\
&\qquad \sigma(t).comm = \sigma_1(t).comm \cup \sigma_2(t).comm \\
&\qquad \wedge \sigma(t).fail = \sigma_1(t).fail \cup \sigma_2(t).fail \\
&\qquad \wedge |\sigma(t).comm \cap \{(c, ?), (c, !), (c, v)\}| \leq 1 \qquad (2) \\
&\qquad \wedge \left\{ \begin{array}{l} \text{if } c \in chan(N_1) \cap chan(N_2) \\ \text{then } (c, v) \in \sigma_1.comm \leftrightarrow (c, v) \in \sigma_2.comm \end{array} \right. \qquad (3) \\
&\}
\end{aligned}$$

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

Proposition 4.2

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

■

Notice that (1) ensures that a process can affect only its local variables and that (2) is the maximal progress assumption. Condition (3) corresponds with regular communication.

5 General Semantics

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

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

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 (see also figure 3). The definition ensures that there is a fixed lower bound K_f on the period of time during which a process fails. We will motivate this decision when we discuss the the semantics of the iteration statement.

Proposition 5.1

(*a*) $FAIL(M) = \emptyset \Leftrightarrow$ for all $(s^0, \sigma, s) \in M$: $|\sigma| = 0$.
(*b*) for all $(s^0, \sigma, s) \in FAIL(M)$: $|\sigma| > K_f$

■

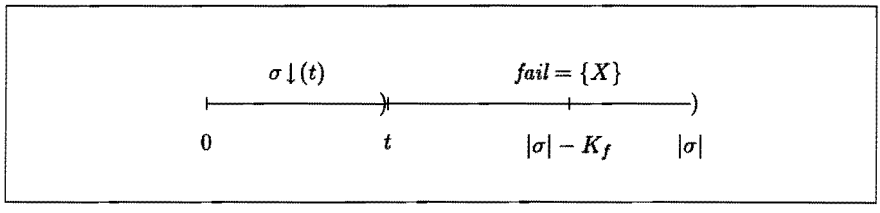

Figure 3: *A sequence σ of a failing computation*

Part (*a*) of proposition 5.1 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 minimal length of the mappings of all executions in $FAIL(M)$ is at least K_f . As a consequence all computations in $FAIL(M)$ take at least K_f time.

5.1 Skip, Delay, and Assignment

Because executing a **skip**-statement takes no time, its execution can not fail. Therefore $FAIL(\mathcal{M}[\![\textbf{skip}]\!])$ is empty (see proposition 5.1). Hence, the general semantics is equal to the normal semantics.

$$
\begin{array}{rcl}
\mathcal{M}^{\dagger}[\![\textbf{skip}]\!] & \triangleq & \mathcal{M}[\![\textbf{skip}]\!] \cup FAIL(\mathcal{M}[\![\textbf{skip}]\!]) \\
& = & \mathcal{M}[\![\textbf{skip}]\!]
\end{array}
$$

The definition of the **delay** and the assignment statement are according the pattern described in the introduction of this section.

$$\mathcal{M}^{\dagger}[\![\textbf{delay}\, e]\!] \triangleq \mathcal{M}[\![\textbf{delay}\, e]\!] \cup FAIL(\mathcal{M}[\![\textbf{delay}\, e]\!])$$

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

5.2 Sequential Composition

Sequential composition itself doesn't consume time. Therefore, 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.

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

   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:

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

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 alternatives ensures that sequential composition is associative.

Proposition 5.2

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

■

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

Proposition 5.3

$$\mathcal{M}_1^\dagger[\![S]\!] \subseteq \mathcal{M}_2^\dagger[\![S]\!]$$

■

5.3 Communication

For the general semantics of the communication statements 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.

$$\mathcal{M}_1^\dagger[\![c?x]\!] \triangleq \mathcal{M}[\![c?x]\!] \cup \mathit{FAIL}(\mathcal{M}[\![c?x]\!])$$

   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.

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

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.

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.

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

5.4 Guarded Statements

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.

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

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 fault of the whole alternative statement.

2. The second possibility is

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

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.

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 either 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.

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

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 process 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.

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

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

5.5 Iteration

We consider two possible definitions for the general semantics of the iteration construct.

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

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

   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

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

   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 fault. The existence of the greatest fixed-point follows from the fact that there is a lower bound $\min(K_f, K_g)$ on the amount of time a failing processes must consume (proposition 5.1).

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

5.6 Networks

As explained in section 4 the naming construct itself doesn't introduce new faults. However it does have an affect on the fault behaviour of a process, and consequently on the general semantics of a process.

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

The definition of the general semantics of a network is almost the same as for the normal semantics.

$$
\begin{aligned}
&\mathcal{M}^{\dagger}[\![N_1 \parallel N_2]\!] \triangleq \\
&\quad \{(s^0, \sigma, s) \mid \text{ there exists } (s_i^0, \sigma_i, s_i) \in \mathcal{M}^{\dagger}[\![N_i]\!] \text{ such that} \\
&\qquad\qquad |\sigma| = \max(|\sigma_1|, |\sigma_2|) \\
&\qquad\qquad \wedge (x \in \mathit{var}(N_i) \rightarrow (s^0(x) = s_i^0(x) \wedge s(x) = s_i(x))) \\
&\qquad\qquad \wedge (x \notin \mathit{var}(N_1, N_2) \rightarrow s(x) = s^0(x)) \qquad (4) \\
&\qquad\qquad \text{and for all } t \in [0, |\sigma|),\ c \in \mathit{CHAN}, \text{ and } v \in \mathit{VAL}: \\
&\qquad\qquad \sigma(t).\mathit{comm} = \sigma_1(t).\mathit{comm} \cup \sigma_2(t).\mathit{comm} \\
&\qquad\qquad \wedge \sigma(t).\mathit{fail} = \sigma_1(t).\mathit{fail} \cup \sigma_2(t).\mathit{fail} \\
&\qquad\qquad \wedge |\sigma(t).\mathit{comm} \cap \{(c, ?), (c, !), (c, v)\}| \leq 1 \qquad (5) \\
&\qquad\qquad \wedge \left\{ \begin{array}{l} \text{if } c \in \mathit{chan}(N_1) \cap \mathit{chan}(N_2) \\ \text{then } (c, v) \in \sigma_1.\mathit{comm} \leftrightarrow (c, v) \in \sigma_2.\mathit{comm} \end{array} \right. \qquad (6) \\
&\quad \}
\end{aligned}
$$

The assumptions (4) (a process can only affect its local variables), (5) (maximal progress), and (6) (regular communication) can be weakened for failing processes, simply by replacing them with

$$\sigma(t).\mathit{fail} = \emptyset \rightarrow (4) \wedge (5) \wedge (6)\ .$$

This transformation affects commutativity nor associativity of the parallel composition operator.

6 Parameterization of the Semantics

In this section we reconsider the definition of the $FAIL$ function that was introduced in section 5. We define a new function $PFAIL$ that is similar to the $FAIL$ function, except that it has two parameters. In this way we obtain a parameterized semantics in which the previously defined semantics is included. The parameters provide an easy way of adapting the semantics to a large class of fault hypothesis.

Consider the partitioned network of three processes in figure 4. The network consists of two processes P_1 and P_2 which are connected by the channel c, and a single (stand-alone) process P_3. Suppose each process only executes a single **delay**-statement. Now, if a fault occurs in P_3 it is possible that the communications between P_1 and P_2 are affected by this fault. Although such a situation may arise in practice, it is a correlation of faults one may want to exclude in the fault hypothesis (e.g. when dealing with software fault tolerance).

A simple way of incorporating fault hypothesis about which channels and variables can not be affected by a fault during the execution of a statement is provided by including two parameters in the semantics. The two parameters appear only in the definition of $PFAIL$:

$$\mathit{PFAIL} : (\mathcal{P}(\Delta) \times \mathcal{P}(\mathit{VAR}) \times \mathcal{P}(\mathit{CHAN})) \rightarrow \mathcal{P}(\Delta)$$

$$\mathit{PFAIL}(M, V, C) \triangleq$$

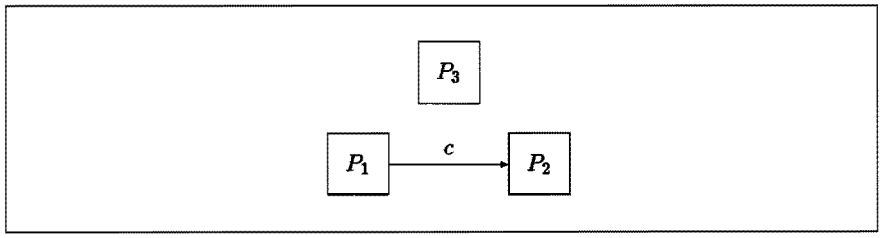

Figure 4: *A partitioned network*

$$
\begin{aligned}
\{(s^0, \sigma, s) \mid\ & \text{there exists } (s^0, \sigma', s') \in M \text{ and} \\
& t \in [0, \min(|\sigma| - K_f, |\sigma'|)) \text{ such that} \\
& \sigma \downarrow t = \sigma' \downarrow t \text{ and for all } x \in V:\ s^0(x) = s(x) \\
& \text{and for all } t' \in [t, |\sigma|): \\
& \quad \sigma(t').\mathit{fail} = \{X\} \\
& \quad \text{and for all } c \in C \text{ and } v \in \mathit{VAL}: \\
& \quad \sigma(t').\mathit{comm} \cap \{(c,!),(c,?),(c,v)\} \\
& \quad = \sigma'(t').\mathit{comm} \cap \{(c,!),(c,?),(c,v)\} \\
\}&
\end{aligned}
$$

Thus V defines the set of protected variables that can not be affected by a fault. Similarly, C defines the set of protected channels that can not be affected by a fault, i.e. the communication behaviour is the same until the execution terminates. However, this does not guarantee that a communication statement is always successful, because the statement may be aborted before communication is completed.

- The *FAIL* function can be defined in terms of the *PFAIL* function

$$\mathit{FAIL}(M) = \mathit{PFAIL}(M, \emptyset, \emptyset)\ ;$$

- It is possible to define statements $\lhd S \rhd$ that are executed successfully, or behave correctly until a fault occurs in which case the original values of the variables are restored when a fault occurs

$$\mathcal{M}^{\dagger}[\![\lhd S \rhd]\!] \triangleq \mathcal{M}[\![S]\!] \cup \mathit{PFAIL}(\mathcal{M}[\![S]\!], \mathit{VAR}, \mathit{CHAN})\ ;$$

- One may choose to use different parameters, depending on the statement for which the semantics are defined, e.g. a fault while executing the statement S can only affect variables and channels that occur in S. For instance, the assignment statement can be defined by $(\mathit{chan}(x := e) = \emptyset)$

$$
\begin{aligned}
&\mathcal{M}^{\dagger}[\![x := e]\!] \triangleq \mathcal{M}[\![x := e]\!] \\
&\quad \cup\ \mathit{PFAIL}(\mathcal{M}[\![x := e]\!], \mathit{VAR} - \mathit{var}(x := e), \\
&\qquad\qquad \mathit{CHAN} - \mathit{chan}(x := e))
\end{aligned}
$$

The use of parameters imposes a condition on the assertion language. It is required that corresponding fault hypothesis is expressible.

7 Discussion

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.

7.1 Acknowledgment

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

References

[1] BERNSTEIN PA. *Sequoia: A Fault Tolerant Tightly Coupled Multiprocessor for Transaction Processing.* IEEE Computer pp. 37–46, February 1988.

[2] BARTLETT J, GRAY J & HORST B. *Fault Tolerance in Tandem Computer Systems.* Symp. on the Evolution of Fault Tolerant Computing, Baden, Austria, 1986.

[3] COENEN J & HOOMAN J. *A Compositional Semantics for Fault-Tolerant Real-Time Systems.* Proc. 2nd Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems pp. 33–51, LNCS 571, Springer-Verlag 1992.

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

[5] CRISTIAN F, DANCEY B & DEHN J. *Fault Tolerance in the Advanced Automation System.* In "20th Annual Symp. on Fault Tolerant Computing", 1990.

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

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

[8] HOOMAN J & WIDOM J. *A Temporal-Logic Based Compositional Proof System for Real-Time Message Passing.* Proc. PARLE '89 Vol. II:424–441, LNCS 366, Springer-Verlag 1989.

[9] INMOS LTD. *OCCAM 2 Reference Manual.* Prentice-Hall, 1988.

[10] JOSEPH M, MOITRA A & SOUNDARARAJAN N. *Proof Rules for Fault Tolerant Distributed Programs.* Science of Comp. Prog. ; 8:43–67, 1987.

[11] KRONENBERG N, LEVY H & STRECKER W. *VAXclusters: A Closely-Coupled Distributed System.* ACM Trans. on Computer Systems, 4:130–146, 1986.

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

[13] POWELL D, VERISSIMO P, BONN G, WAESELYNCK F & SEATON. D. *The Delta-4 Approach to Dependability in Open Distributed Computing Systems.* Proc. FTCS-18, IEEE Computer Society Press, 1988.

[14] RANDELL B, LEE PA & TRELEAVEN PC. *Reliability Issues in Computing System Design.* ACM Computing Surveys, 10:123–165, 1978.

[15] SCHLICHTING RD & SCHNEIDER FB. *Fail-stop processors: an approach to designing fault tolerant computing systems.* ACM Trans. on Comp. Sys. ; 1(3):222–238, 1983.

[16] SHANKAR AU & LAM SS. *Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties.* Distributed Computing; 2:61–79, 1987.

[17] TAYLOR D & WILSON G. *Stratus.* In "Dependability of Resilient Computers", T. Anderson Ed., Blackwell Scientific Publications, 1989.

Chapter 4

Exception Handling in Process Algebra

This chapter is a revised version of:

F.S. DE BOER, J. COENEN, AND R. GERTH.
Exception Handling in Process Algebra.
Proc. of the First North-American Process Algebra Workshop
(S. Purushothaman & A. Zwarico, eds.),
pp. 86–100, Workshops in Computing, Springer-Verlag 1993.

Exception Handling in Process Algebra

F.S. de Boer * J. Coenen † R. Gerth‡
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: {wsinfdb, wsinjosc, robg}@win.tue.nl

Abstract

We study exception handling as it occurs e.g. in ADA, aiming at an algebraic characterization. We take Bergstra and Klop's Algebra of Communicating Processes (ACP) as our starting point and equationally define strong bisimulation for ACP extended with exception handling primitives. This theory is then applied to showing fault tolerance under an explicitly stated fault hypothesis of a system that is made more fault resilient by applying dynamic redundancy.

1 Introduction

Exception handling has received scant algebraic treatment. In fact, [HH87] and [Dix83] are the only papers that we are aware of that touch on this topic. In [HH87], the *interrupt* construct of [Hoa85], $P \hat{} Q$, is utilized to express recovery from errors or exceptions. In [Dix83] the term exception is used for Hoare's interrupt construct. The construct satisfies the following SOS rules: $P \hat{} Q \xrightarrow{a} P' \hat{} Q$ provided $P \xrightarrow{a} P'$ and $P \hat{} Q \xrightarrow{a} Q'$ just in case $Q \xrightarrow{a} Q'$. I.e., execution of P can always be interrupted by the first action of Q. If $\dagger$ is a symbol standing for an error, then associating an *error handler* Q for this error to process P is done by $P \hat{} (\dagger \rightarrow Q)$: if one assumes that P does not generate $\dagger$, then in a process $\dagger \parallel (P \hat{} (\dagger \rightarrow Q))$, in which the leftmost process specifies the *error hypothesis* that at most one error may occur while P or Q is executing, the handler Q can interrupt P only if the error actually occurs (Hoare's parallel operator imposes synchronization on common actions.)

Admirably though this approach fits their purpose, we feel there is room for improvement. Let us write $P \hookrightarrow Q$ for associating an exception handler Q to a process P. Now, what should this construct satisfy? Authoritative answers to this question can be found in [Ada83, Cri85, LS90]:

1. In a process $(P \hookrightarrow Q) \hookrightarrow Q$ only the inner handler Q should be activated by an exception during execution of P so that a second exception occurring while Q is active can still be caught by the outer handler.

2. If the process P in $P \hookrightarrow Q$ raises an exception, then Q ought to handle this exception if it can.

*NWO/SION project "Research and Education in Computer Science (REX)."
†NWO/SION project "Fault Tolerance: Paradigms, Models, Logics, Construction."
‡ESPRIT project: "Building Correct Reactive Systems (REACT)."

Now, $P^\frown(\dagger \to Q)$ as an implementation of $P \hookrightarrow Q$ fails on both counts: exception handlers need not be invoked innermost first; if P raises the error $\dagger$ in $\dagger \parallel (P^\frown(\dagger \to Q))$ then it can not be handled by Q, and therefore it ought to be handled by a parallel process.

Another issue that we address is the algebraic treatment of *failure* due to unrecoverable errors. Such failures we want to be visible.

In this paper we propose to extend aprACP [BK84][1] to aprACP$_\mathsf{E}$ with an exception handling construct that does satisfy the three earlier conditions. Furthermore, a recovery operator is introduced, which interprets the occurrence of certain actions as errors. Recovery from errors is described in terms of the synchronization merge of aprACP, i.e., the occurrence of an error requires synchronization with a corresponding action, a so-called handler, otherwise a failure occurs which gives rise to uncontrolled behaviour of the system. One of the main difficulties of a proper algebraic treatment of failure is the asymmetry between errors and their corresponding handlers: an error is an autonomous action whereas handlers are only activated when an error occurs. This asymmetry is analogous to the one between asynchronous send and receive actions, and between synchronous put and get actions [BW90]. However there is an important difference between the occurrence of an error, on the one hand, and synchronous put and asynchronous send actions, on the other hand: an error *has* to synchronize with a corresponding handler, otherwise a failure occurs, whereas a synchronous put or asynchronous send action is completely autonomous.

The language and an axiomatization of strong bisimulation is presented in Section 2. In Section 3 we turn to an example due to Peleska [Pel91]. A simple transformational process P is made more fault resilient by putting two copies of P in an arbitration protocol. The fault resilient version equals P under the fault hypothesis that the time interval between errors is large enough. Some conclusions are presented in Section 4.

2 aprACP$_\mathsf{E}$ and its Axiomatization

2.1 Language and SOS

Let A be an alphabet of actions. We have $a, b, \ldots \in A$. We assume two more actions disjoint from A: δ and $\bot$; the former denoting inaction and the latter indicating the occurrence of an unrecoverable exception. Elements of $A \cup \{\delta, \bot\}$ are denoted by $\alpha, \beta, \ldots$.

The grammar in Table 1 specifies the syntax of process terms of aprACP$_\mathsf{E}$ (we treat the left-merge $\cdot \between \cdot$ and the communication merge $\cdot | \cdot$ of aprACP as auxiliary operators.) By convention, prefixing $(\alpha\cdot)$ binds strongest, then comes $\hookrightarrow$, then $+$ and finally $\parallel$. The behaviour of aprACP$_\mathsf{E}$-terms is described in Table 2. Some of the SOS rules have negative premises, so there is a question of well-definedness. However, the rules are all in GSOS format, hence stratifiable, so that they define a proper transition relation on the process terms [Gro90].

We have the following definition of bisimulation:

2–1 DEFINITION (Bisimulation). *Two process expressions x and y are bisimilar, notation: $x \simeq y$, if and only if there exists a relation R on process expressions such that $x\ R\ y$ and whenever $x'\ R\ y'$ then for every $a \in A$ if $x' \xrightarrow{a} x''$ then $y' \xrightarrow{a} y''$ for some*

[1] I.e., ACP with action prefixing instead of sequential composition.

x	$:=$	$\delta \mid \bot \mid \alpha{\cdot}x$	$(\alpha \in A \cup \{\delta, \bot\})$
	$\mid$	$x + y \mid x \parallel y$	
	$\mid$	$x \hookrightarrow y$	
	$\mid$	$\partial_H(x)$	$(H \subseteq A)$
	$\mid$	$\mathcal{R}_H(x)$	$(H \subseteq A)$

Table 1: Process terms

y'' such that x'' R y'' and vice versa; and also if $x' \xrightarrow{\bot} x''$ then $y' \xrightarrow{\bot} y''$ for some y'' and vice versa.

It should be noted that in the above definition in case of the occurrence of a failure the resulting processes are *not* required to be bisimilar. As a consequence the behaviour of a process becomes uncontrollable after the occurrence of a failure.

Since the SOS rules are obviously well-founded, we obtain that bisimulation is a congruence for the operators in Table 1 (Theorem 4.4 in [Gro91].)

The rules for $\bot$, $\alpha{\cdot}x$ and $+$ are as should be expected. In aprACP, the parallel operator is modelled as interleaving plus synchronization, where synchronization is described in terms of a communication function $\cdot|\cdot \in Act \cup \{\delta, \bot\} \times Act \cup \{\delta, \bot\} \to Act \cup \{\delta, \bot\}$. The encapsulation $\partial_H(\cdot)$ prohibits any action in H to occur and, hence, is similar to the CCS restriction [Mil89]. The process $x \hookrightarrow y$ resembles Hoare's $x \mathbin{\hat{}} y$ in that y may interrupt x anytime, but is dissimilar w.r.t. one essential point: control may only transfer to y through executing an initial action of y that x *cannot* perform. E.g., the process $a{\cdot}\delta \hookrightarrow a{\cdot}b{\cdot}\delta$ admits only one sequence of transitions:

$$(a{\cdot}\delta \hookrightarrow a{\cdot}b{\cdot}\delta) \xrightarrow{a} (\delta \hookrightarrow a{\cdot}b{\cdot}\delta) \xrightarrow{a} b{\cdot}\delta \xrightarrow{b} \delta \ .$$

We stress that in $x \hookrightarrow y$, activation of y is not subject to any other constraints. At the end of this subsection we shall see how to enforce that exception handlers can be activated by the occurrence of an error only. Given an action a the set of handlers of a, i.e., those actions b such that $a|b \in A$, is denoted by $a^{\downarrow}$. We assume that $a^{\downarrow} \cap b^{\downarrow} = \emptyset$ if $a \neq b$, i.e. a recovery action b can recover a particular type of exception actions only (viz. the unique action a such that $b \in a^{\downarrow}$.) The recovery operator $\mathcal{R}_H(\cdot)$ interprets the execution of an action $a \in H$ as the occurrence of an error which raises an exception handled by an action in $a^{\downarrow}$. The result of the recovery of an error generated by an action a by a handler b for a is indicated by $a|b$. Unrecovered actions are indicated by $\bot$. As an example consider the process $\mathcal{R}_{\{a\}}(x)$, where $x = (a{\cdot}\delta \hookrightarrow b{\cdot}y) \parallel b{\cdot}z$, with b a handler of a. The occurrence of a is interpreted as an error, which can be handled by either $b{\cdot}y$ or $b{\cdot}z$: we have both

$$\mathcal{R}_{\{a\}}(x) \xrightarrow{a|b} y \parallel b{\cdot}z$$

and

$$\mathcal{R}_{\{a\}}(x) \xrightarrow{a|b} (\delta \hookrightarrow b{\cdot}y) \parallel z \ .$$

Thus we see that an error generated by a process is broadcasted so that it may raise an exception in any (but only one) process of the system. In this way, fault hypotheses,

$\alpha \neq \delta$ and $\alpha \mid \beta \neq \delta$		+ and $\parallel$ are commutative
Fail	$\bot \xrightarrow{\bot} \delta$	$\bot \cdot x \xrightarrow{\bot} \delta$
Prefixing	$a \cdot x \xrightarrow{a} x$	
Choice	$\dfrac{x \xrightarrow{\alpha} z}{x + y \xrightarrow{\alpha} z}$	
Merge	$\dfrac{x \xrightarrow{\alpha} z}{x \parallel y \xrightarrow{\alpha} z \parallel y}$	$\dfrac{x \xrightarrow{\alpha} u \quad y \xrightarrow{\beta} v}{x \parallel y \xrightarrow{\alpha\mid\beta} u \parallel v}$
Exception Handler	$\dfrac{x \xrightarrow{\alpha} z}{x \hookrightarrow y \xrightarrow{\alpha} z \hookrightarrow y}$	$\dfrac{x \not\xrightarrow{\alpha} \quad y \xrightarrow{\alpha} z}{x \hookrightarrow y \xrightarrow{\alpha} z}$
Encapsulation	$\dfrac{x \xrightarrow{\alpha} y \ (\alpha \notin H)}{\partial_H(x) \xrightarrow{\alpha} \partial_H(y)}$	
Recovery	$\dfrac{x \xrightarrow{a} y \quad y \xrightarrow{b} z \quad (a \in H,\ b \in a^{\downarrow})}{\mathcal{R}_H(x) \xrightarrow{a\mid b} \mathcal{R}_H(z)}$	
	$\dfrac{x \xrightarrow{a} y \quad y \not\xrightarrow{b} \quad (a \in H, \forall_b(b \in a^{\downarrow}))}{\mathcal{R}_H(x) \xrightarrow{\bot} \delta}$	$\dfrac{x \xrightarrow{\alpha} y \quad (\alpha \notin H)}{\mathcal{R}_H(x) \xrightarrow{\alpha} \mathcal{R}_H(y)}$

Table 2: SOS rules for aprACP$_{\mathsf{E}}$

which are used to specify relative to what fault scenarios the systems is fault resilient, can be described as a parallel process. (see Section 3 for an example.) The scope within which an error or exception must be caught is determined by the recovery operator. E.g., in the process $\mathcal{R}_{\{a\}}(a \cdot \delta \hookrightarrow b \cdot y) \parallel b \cdot z$, the error a can only be caught by the handler $b \cdot y$. The autonomous character of an error can be best illustrated by the following example: consider the process $x = a \cdot \delta + (a \cdot \delta \hookrightarrow b \cdot y)$. Then

$$\mathcal{R}_{\{a\}}(x) \xrightarrow{\bot} \delta$$

is a possible transition because the error generated by the left summand cannot be recovered. So, once an error occurs other alternatives are disregarded.

An exception handler that is activated only if an error occurs can now be modelled as

$$\partial_{\{b\}} \circ \mathcal{R}_{\{a\}}(p \hookrightarrow b \cdot q)\,,$$

with $b \in a^{\downarrow}$ $(a \mid b \neq b)$. Thus we model exception handling analogously to the ACP treatment of concurrency: first, freely generate all potentially possible executions and then restrict this set to the actual ones.

2.2 The Axiom System

The axiomatization is an extension of the usual axiomatization of aprACP which consists of all the axioms of Table 4 and Table 3 but for the axioms concerning $\bot$: $\bot \cdot x = \bot$

and $\alpha|\bot = \bot$.

$x+y$	$=$	$y+x$
$(x+y)+z$	$=$	$x+(y+z)$
$x+x$	$=$	x
$x+\delta$	$=$	x
$\delta\cdot x$	$=$	δ
$\bot\cdot x$	$=$	$\bot$

Table 3: $\mathsf{aprBPA}_{\delta,\bot}$ axiomatization

$x \parallel y$	$=$	$x \between y + y \between x + x\|y$	
$(\alpha\cdot x) \between y$	$=$	$\alpha\cdot(x \parallel y)$	
$(x+y) \between z$	$=$	$x \between z + y \between z$	
$\alpha\cdot x\|\beta\cdot y$	$=$	$(\alpha\|\beta)\cdot(x \parallel y)$	
$(x+y)\|z$	$=$	$x\|z + y\|z$	
$x\|(y+z)$	$=$	$x\|y + x\|z$	
$\alpha\|\beta$	$=$	$\beta\|\alpha$	
$\alpha\|(\beta\|\gamma)$	$=$	$(\alpha\|\beta)\|\gamma$	
$\alpha\|\delta$	$=$	δ	
$\alpha\|\bot$	$=$	$\bot$	$\alpha \neq \delta$
$\partial_H(a\cdot x)$	$=$	δ	$a \in H$
$\partial_H(\alpha\cdot x)$	$=$	$\alpha\cdot\partial_H(x)$	$\alpha \notin H$
$\partial_H(x+y)$	$=$	$\partial_H(x)+\partial_H(y)$	

Table 4: Merge and encapsulation

Tables 5 and 6 extend these axioms. The combined set of equations is also denoted as $\mathsf{aprACP}_{\mathsf{E}}$. The way $\cdot \hookrightarrow \cdot$ is axiomatized is analogous to that of the merge. We introduce auxiliary operators that force the left (right) process to move first, thus allowing choices to be resolved. Two auxiliary operators are needed here because $\cdot \hookrightarrow \cdot$

is not commutative. As an example, consider the following derivation ($a \neq b$):

$$
\begin{aligned}
& (a{\cdot}\delta \hookrightarrow b{\cdot}y) \hookrightarrow b{\cdot}z \\
= \; & (a{\cdot}\delta \mathbin{\leftarrowtail} b{\cdot}y + a{\cdot}\delta \hookrightarrow\!\!\!\!\rightarrow b{\cdot}y) \hookrightarrow b{\cdot}z \\
= \; & (a(\delta \hookrightarrow b{\cdot}y) + (a{\cdot}\delta + \delta) \hookrightarrow\!\!\!\!\rightarrow b{\cdot}y) \hookrightarrow b{\cdot}z \\
= \; & (a(\delta \mathbin{\leftarrowtail} b{\cdot}y + \delta \hookrightarrow\!\!\!\!\rightarrow b{\cdot}y) + \delta \hookrightarrow\!\!\!\!\rightarrow b{\cdot}y) \hookrightarrow b{\cdot}z \\
= \; & (a(\delta + b{\cdot}y) + b{\cdot}y) \hookrightarrow b{\cdot}z \\
= \; & (a{\cdot}b{\cdot}y + b{\cdot}y) \hookrightarrow b{\cdot}z \\
= \; & (a{\cdot}b{\cdot}y + b{\cdot}y) \mathbin{\leftarrowtail} b{\cdot}z + (a{\cdot}b{\cdot}y + b{\cdot}y) \hookrightarrow\!\!\!\!\rightarrow b{\cdot}z \\
= \; & (a{\cdot}b{\cdot}y \mathbin{\leftarrowtail} b{\cdot}z + b{\cdot}y \mathbin{\leftarrowtail} b{\cdot}z) + (a{\cdot}b{\cdot}y + b{\cdot}y) \hookrightarrow\!\!\!\!\rightarrow b{\cdot}z \\
= \; & (a(b{\cdot}y \hookrightarrow b{\cdot}z) + b{\cdot}(y \hookrightarrow b{\cdot}z)) + \delta \\
= \; & a(b{\cdot}y \mathbin{\leftarrowtail} b{\cdot}z + b{\cdot}y \hookrightarrow\!\!\!\!\rightarrow b{\cdot}z) + b{\cdot}(y \hookrightarrow b{\cdot}z) \\
= \; & a{\cdot}(b(y \hookrightarrow b{\cdot}z) + \delta) + b(y \hookrightarrow b{\cdot}z) \\
= \; & a{\cdot}b(y \hookrightarrow b{\cdot}z) + b(y \hookrightarrow b{\cdot}z)
\end{aligned}
$$

The left summand of the conclusion of this derivation describes the situation that after a the handler $b{\cdot}y$ is activated, whereas the right summand describes the immediate activation of the handler $b{\cdot}y$. Note that the handler $b{\cdot}z$ can only be activated after b is executed.

$x \hookrightarrow y$	$=$	$x \mathbin{\leftarrowtail} y + x \hookrightarrow\!\!\!\!\rightarrow y$	
$(x + y) \mathbin{\leftarrowtail} z$	$=$	$x \mathbin{\leftarrowtail} z + y \mathbin{\leftarrowtail} z$	
$\alpha{\cdot}x \mathbin{\leftarrowtail} z$	$=$	$\alpha{\cdot}(x \hookrightarrow z)$	
$\delta \mathbin{\leftarrowtail} x$	$=$	δ	
$\bot \mathbin{\leftarrowtail} x$	$=$	$\bot$	
$x \hookrightarrow\!\!\!\!\rightarrow (y + z)$	$=$	$(x \hookrightarrow\!\!\!\!\rightarrow y) + (x \hookrightarrow\!\!\!\!\rightarrow z)$	
$(\alpha{\cdot}x + y) \hookrightarrow\!\!\!\!\rightarrow \beta{\cdot}z$	$=$	δ	$\alpha = \beta$
$(\alpha{\cdot}x + y) \hookrightarrow\!\!\!\!\rightarrow \beta{\cdot}z$	$=$	$y \hookrightarrow\!\!\!\!\rightarrow \beta{\cdot}z$	$\alpha \neq \beta$
$\delta \hookrightarrow\!\!\!\!\rightarrow x$	$=$	x	

Table 5: Exception handler

The recovery operator resembles the state operator [BW90]. After having seen an $a \in H$ action, the operator changes its behavior: the operator $\mathcal{R}_H^a$ searches for a handler for a, which then is transformed into $a|b$, in case such a handler cannot be found $\bot$ is delivered.

2.3 Soundness

To prove soundness of the axiom system we define a model for the language which associates with each process a labelled transition system. A labelled transition system is a triple $(S, A, \longrightarrow)$ consisting of a set of states S, a set of labels A, and a transition relation $\longrightarrow \subseteq S \times A \times S$. We will use a representation of transition systems in non-well-founded set theory ([Acz88].) The techniques underlying the model construction

$\mathcal{R}_H(\alpha{\cdot}x)$	$=$	$\alpha{\cdot}\mathcal{R}_H(x)$	$\alpha \notin H$
$\mathcal{R}_H(a{\cdot}x)$	$=$	$\mathcal{R}^a_H(x)$	$a \in H$
$\mathcal{R}_H(x+y)$	$=$	$\mathcal{R}_H(x)+\mathcal{R}_H(y)$	
$\mathcal{R}^a_H(\delta)$	$=$	$\bot$	
$\mathcal{R}^a_H(b{\cdot}x)$	$=$	$(a\|b){\cdot}\mathcal{R}_H(x)$	$b \in a^{\downarrow}$
$\mathcal{R}^a_H(\alpha{\cdot}x)$	$=$	$\bot$	$\alpha \notin a^{\downarrow}$
$\mathcal{R}^a_H(\bot+x)$	$=$	$\bot+\mathcal{R}^a_H(x)$	
$\mathcal{R}^a_H(b{\cdot}x+y)$	$=$	$\mathcal{R}^a_H(y)$	$b \notin a^{\downarrow}$
$\mathcal{R}^a_H(b{\cdot}x+c{\cdot}y+z)$	$=$	$(a\|b){\cdot}\mathcal{R}^a_H(x)+\mathcal{R}^a_H(c{\cdot}y+z)$	$b,c \in a^{\downarrow}$

Table 6: Recovery

Left Handler	$\dfrac{x \xrightarrow{\alpha} y}{x \leftarrowtail z \xrightarrow{\alpha} y \hookrightarrow z}$	
Right Handler	$\dfrac{y \xrightarrow{\alpha} z \quad x \not\xrightarrow{\alpha}}{x \hookrightarrow y \xrightarrow{\alpha} z}$	
Handler-Search	$\dfrac{x \xrightarrow{b} y \quad (b \in a^{\downarrow})}{\mathcal{R}^a_H(x) \xrightarrow{a\|b} \mathcal{R}_H(y)}$	$\dfrac{x \not\xrightarrow{b} \quad (\forall b \in a^{\downarrow})}{\mathcal{R}^a_H(x) \xrightarrow{\bot} \delta}$

Table 7: SOS rules for the auxiliary operators

are taken from [Rut92]. One of the advantages of this new approach is that the model is defined directly in terms of the SOS. Given that the SOS is well-founded and in GSOS format so that bisimulation is a congruence, we then can prove soundness of the axioms *without* having to define explicitly the semantic operators corresponding to the operators of the language.

First observe that we can associate SOS rules to the auxiliary operators that were used; see Table 7. These rules, too, are in GSOS format and are well-founded. In other words, bisimulation is a congruence for all operators, including the auxiliary ones.

2–2 DEFINITION. *Let P be the largest class satisfying*
$P = \mathcal{P}(A_\bot \times P)$.
(Here $A_\bot = A \cup \{\bot\}$.)

Formally, P is obtained as the largest fixed-point of the class operator Φ that assigns to every class X the class $\mathcal{P}(A_\bot \times X)$, i.e., the class of all *subsets* of $A_\bot \times X$ (see [Rut92].)

2–3 DEFINITION. *Let $M \in \mathsf{aprACP}_{\mathsf{E}} \to P$ be defined as follows:*
$M(x) = \{(\alpha, M(y)) \mid x \xrightarrow{\alpha} y\}$

This recursive definition can be justified by an application of the *solution lemma* according to which systems of equations of a certain class have a unique solution in non-well-founded set theory [Acz88]. We have the following theorem:

2–4 THEOREM. *For any processes x and y we have*
$x \simeq y \Leftrightarrow M(x) = M(y)$.

For a proof of this theorem we refer to [Rut92]. Since we know that $\simeq$ is a congruence, it suffices for proving soundness to show that for any axiom $x = y$ we have $M(x) = M(y)$.

2–5 THEOREM (Soundness). *For any two processes x and y*
$\mathsf{aprACP_E} \vdash x = y$ implies $x \simeq y$

As explained above we need now only to inspect the individual equations. We treat the following case: $x \hookrightarrow y = x \leftarrowtail y + x \hookrightarrow y$.

$$\begin{aligned}
& M(x \hookrightarrow y) \\
= \; & \{\langle \alpha, M(z)\rangle \mid x \hookrightarrow y \xrightarrow{\alpha} z\} \\
= \; & \{\langle \alpha, M(x' \hookrightarrow y)\rangle \mid x \xrightarrow{\alpha} x'\} \cup \{\langle \alpha, M(y')\rangle \mid x \not\xrightarrow{\alpha},\ y \xrightarrow{\alpha} y'\} \\
= \; & M(x \leftarrowtail y) \cup M(x \hookrightarrow y) \\
= \; & M(x \leftarrowtail y + x \hookrightarrow y)
\end{aligned}$$

Finally we note that every guarded recursive (process) equation has a unique solution in P (see [Rut92].) For example, the equation $x = a{\cdot}x$ is interpreted in non-well-founded set theory as $x = \{\langle a, x\rangle\}$. Let $\pi(x)$ be the unique solution of x. Since P is the largest class satisfying the equation used for its definition we have that $\pi(x) \in P$.

2.4 Completeness

We first prove an elimination lemma. Let, $\mathsf{aprBPA}_{\delta,\perp}$ be the axiom system of Table 3 for basic processes, i.e., processes formulated in the signature $\{\delta\cdot, a\cdot, \perp\cdot, \cdot + \cdot \mid a \in A\}$.

2–6 LEMMA (Elimination). *For any $x \in \mathsf{aprACP_E}$ there is a basic process y such that* $\mathsf{aprACP_E} \vdash x = y$.

Using the axioms we can eliminate all the operators but prefixing and choice starting from the innermost one and "working our way up".

The completeness then follows from the completeness of $\mathsf{aprBPA}_{\delta,\perp}$ (which is a trivial extension of the completeness theorem for aprBPA_δ) and the above soundness theorem: let $x \simeq y$, according to the above lemma there exists basic processes x' and y' such that $\mathsf{aprACP_E} \vdash x = x'$ and $\mathsf{aprACP_E} \vdash y = y'$. By the soundness theorem we have that $x \simeq x'$ and $y \simeq y'$, so $x' \simeq y'$. From the completeness of $\mathsf{aprBPA}_{\delta,\perp}$ it then follows that $\mathsf{aprBPA}_{\delta,\perp} \vdash x' = y'$, and thus $\mathsf{aprACP_E} \vdash x = y$.

3 A distributed fault-tolerant system

In order to achieve a higher degree of reliability, a fault-tolerant system must exploit some form of redundancy. In the example below, which is very much inspired by Peleska's fault-tolerant system [Pel91], a system P is duplicated and embedded in

a protocol that ensures correct behaviour despite the presence of faults. We have modified Peleska's original system in order to preserve correctness under a larger class of fault scenario's. Nevertheless we will refer to this system as Peleska's system.

In this section each exception can be recovered by one action only, i.e. $|a^{\downarrow}| = 1$ for all $a \in A$. Therefore, in this section we simply write $a^{\downarrow}$ instead of the unique action $b \in a^{\downarrow}$ that recovers $a^{\uparrow}$, for all $a^{\uparrow} \in a$.

Peleska's system is built around two duplicates P_1 and P_2 of the basic system P in table 8. The basic system inputs a value x on channel a and then computes the value $\varphi(x)$, which is output on channel b. It is assumed that all values are within a *finite* data domain D, and that $\varphi : D \to D$ is a function on D. Any system that

$$P = \sum_{x \in D} a(x) \cdot b(\varphi(x)) \cdot P$$

Table 8: Basic system

satisfies the equation of the basic system, can be systematically transformed into a more resilient system T that is *weakly* bisimilar with P; i.e., a system that is bisimilar if we abstract from internal actions. This implies that for many applications one may simply replace P by system T.

The transformed system T consists of six components RP_1, RP_2, Q_1, Q_2, RR_1, and RR_2 (see Figure 1.) Components RP_1 and RP_2 are *restartable* (see e.g. [Pra84,

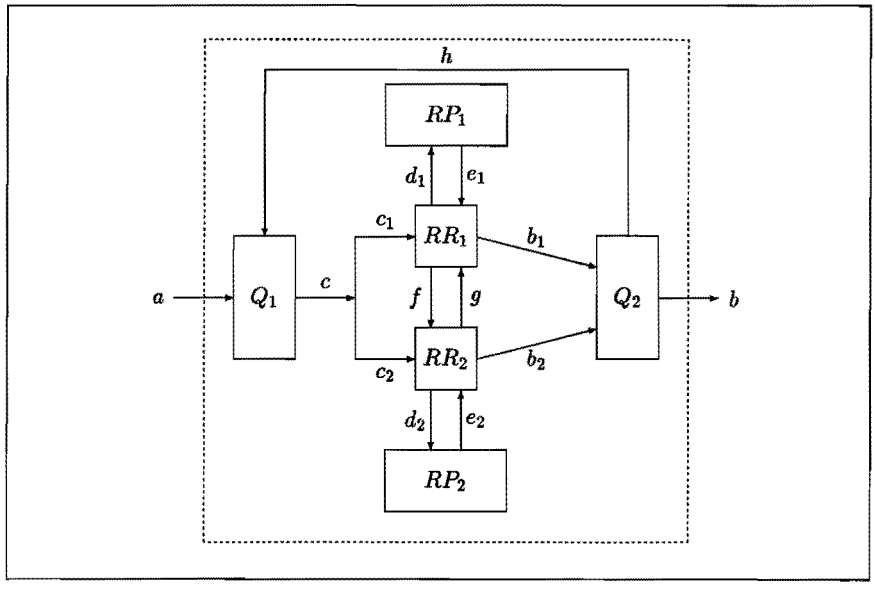

Figure 1: Fault-tolerant system T

Pra87, HH87].) A restartable system can be defined with the exception handler of $\mathsf{aprACP}_{\mathsf{E}}$. For example, in Table 9 a restartable version RP of the basic system P

$$RP \quad = \quad P \hookrightarrow p^{\downarrow} \cdot RP$$

Table 9: A restartable system

is defined. The restartable system RP behaves like the basic system P, until the exception $p^{\uparrow}$ is raised after which it is restarted. Components RP_1 and RP_2 are defined slightly different because they operate in a master-slave configuration. Initially RP_1 is the active master until a fault is detected in P_1 (signalled by exception $z_1{}^{\uparrow}$.) Upon detection of the fault RP_1 is de-activated and RP_2 takes over, thereby switching the rôle of master and slave. If a fault is detected in P_2 (signalled by exception $z_2{}^{\uparrow}$) process RP_1 takes over again. As a matter of fact this is an example of a *dynamic redundant system* in which RP_1 and RP_2 alternately function as *hot-standby components*. The philosophy of such a system is that if faults don't occur too frequently — i.e. the *Mean Time Between Failures* (MTBF) is sufficiently larger than the *Mean Time To Repair* (MTTR) — the de-activated faulty component can be replaced while the other duplicate is operational.

To ensure that no data will be lost if the currently active component P_i ($i \in \{1,2\}$) crashes, the stand-by component should receive a copy of the input data whenever the active component receives an input. Because the stand-by component is not active this might result in a deadlock. For this reason each RP_i is connected to a component RR_i. RR_i is a restartable component with a core process R_i. Processes R_i simultaneously accept the input data and then offer it to their corresponding component RP_i. Because RP_i may not be active, process RR_i might deadlock. For this reason the other process R_j — which is guaranteed to succeed because only one of the components RP_i can be de-activated at the time — sends a reset signal (f or g) after it has forwarded the result obtained from RP_j. Component RR_i restarts R_i when the exception handler $r_i{}^{\downarrow}$ is activated. The exception handler $r_i{}^{\downarrow}$ is synchronized with the exception handler $p_i{}^{\downarrow}$, and therefore triggered by exception $z_i{}^{\uparrow}$. If P_i crashes after accepting an input of RR_i, but before resetting the other component RR_j, it should be willing to accept a reset signal before restarting in order to avoid deadlock.

There is still one problem to be resolved. In case a component P_i crashes just after RR_i has forwarded the result, but before RR_i has sent the reset signal to the other component RR_j, RP_j becomes active and RR_j will forward its output also. To avoid such duplicate outputs an additional layer consisting of components Q_1 and Q_2 is included. Components Q_1 and Q_2 execute an alternating-bit protocol. Process Q_1 receives its inputs from the input channel a of the system T. Upon receipt of a message, it adds an extra bit to the message and forwards it to both components RR_1 and RR_2. A component RR_i removes the additional bit before passing the message to RP_i, but re-appends it again before forwarding the output messages from RP_i. Component Q_2 waits for a message of one of the components R_i. Upon receipt of a message the extra bit is inspected and removed. If the extra bit has the expected value then Q_2 outputs the message, to channel b of T and sends a signal h to Q_1. If the extra bit does not have the expected value then the received message is simply discarded. The signal h, which is not present in Peleska's original example, informs component Q_1 that it may accept a new input. Peleska's original system has a buffer capacity due to internal communications. This results in a communication latency which allows the transformed system T to input more than one message before giving an output message. For this reason Peleska's original transformed system is not weakly bisimilar

with the basic system P.

The specification of the transformed system T and its components is listed in Table 10. The synchronization function is defined in Table 11, and the encapsulation set H is defined in Table 13.

T	$=$	$\partial_H((P_1 \leftarrow p_1^{\downarrow} \cdot RP_1) \parallel RP_2 \parallel Q_1(0) \parallel Q_2(0) \parallel RR_1 \parallel RR_2)$
P_1	$=$	$\sum_{x \in D} d_1(x) \cdot e_1(\varphi(x)) \cdot P_1$
P_2	$=$	$\sum_{x \in D} d_2(x) \cdot e_2(\varphi(x)) \cdot P_2$
RP_1	$=$	$(p_2^{\downarrow} \cdot P_1) \leftarrow p_1^{\downarrow} \cdot RP_1$
RP_2	$=$	$(p_1^{\downarrow} \cdot P_2) \leftarrow p_2^{\downarrow} \cdot RP_2$
$Q_1(n)$	$=$	$\sum_{x \in D} a(x) \cdot c(x,n) \cdot h \cdot Q_1(1-n)$
$Q_2(n)$	$=$	$\sum_{\substack{x \in D \\ i \in \{1,2\}}} (b_i(x,n) \cdot b(x) \cdot h \cdot Q_2(1-n) + b_i(x,1-n) \cdot Q_2(n))$
R_1	$=$	$\sum_{\substack{x \in D \\ n \in \{0,1\}}} c_1(x,n) \cdot (d_1(x) \cdot (\sum_{y \in D} e_1(y) \cdot b_1(y,n) \cdot f \cdot R_1) + g \cdot R_1)$
R_2	$=$	$\sum_{\substack{x \in D \\ n \in \{0,1\}}} c_2(x,n) \cdot (d_2(x) \cdot (\sum_{y \in D} e_2(y) \cdot b_2(y,n) \cdot g \cdot R_2) + f \cdot R_2)$
RR_1	$=$	$R_1 \leftarrow r_1^{\downarrow} \cdot (g \cdot RR_1 + RR_1)$
RR_2	$=$	$R_2 \leftarrow r_2^{\downarrow} \cdot (f \cdot RR_2 + RR_2)$

Table 10: Specification of T and its components

$b_i(x,n) \mid b_i(x,n)$	$=$	$cb_i(x,n)$	$f \mid f$	$=$	cf
$c(x,n) \mid c(x,n)$	$=$	$cc(x,n)$	$g \mid g$	$=$	cg
$c_1(x,n) \mid c_2(x,n)$	$=$	$c(x,n)$	$h \mid h$	$=$	ch
$d_i(x) \mid d_i(x)$	$=$	$cd_i(x)$	$p_i^{\downarrow} \mid p_i^{\downarrow}$	$=$	$r_i^{\downarrow}$
$e_i(x) \mid e_i(x)$	$=$	$ce_i(x)$	$r_i^{\downarrow} \mid r_i^{\downarrow}$	$=$	$z_i^{\downarrow}$
			$z_i^{\uparrow} \mid z_i^{\downarrow}$	$=$	$z_i^{\Downarrow}$
$i \in \{1,2\}, n \in \{0,1\}, x \in D$					

Table 11: Synchronization function

Of course, no system can be guaranteed to function correctly in arbitrary conditions. Therefore we have to make some assumptions about occurrences of faults in a *fault hypothesis*. It then suffices to prove correctness of a system with respect to the fault hypothesis. In $\mathsf{aprACP}_{\mathsf{E}}$ a fault hypothesis can be modelled as a process. As such one may think of the fault hypothesis as Cristian's *adverse environment* [Cri85]. To prove correctness of the system T with respect to a fault hypothesis modelled by process FH, we have to verify the property in Table 12 ($\simeq_\tau$ denotes weak bisimulation

$P \simeq_\tau \tau_I \circ \mathcal{R}_J(T \parallel FH_1)$

Table 12: Proof obligation

and τ_I renames the action in I as τ.) The set I of internal moves and the set J of actions that must be recovered are defined in Table 13. In order to prove the property in Table 12 we need additional axioms for weak bisimulation and hiding. These axioms are included in Table 14 and their justification can be found in e.g. [BW90]. Note that $\mathsf{aprACP_E}$ allows P and $\mathcal{R}_J(T \parallel FH)$ to be reduced to $\mathsf{aprBPA}_{\delta,\perp}$-terms and the axioms in Table 14 suffice for weak bisimulation on $\mathsf{aprBPA}_{\delta,\perp}$.

H	$=$	$\{b_i(x,n), c(x,n), c_i(x,n), d_i(x), e_i(x), f, g, h, p_i^{\downarrow}, r_i^{\downarrow}\}$
I	$=$	$\{cb_i(x,n), cc(x,n), cd_i(x), ce_i(x), cf, cg, ch, z_i^{\Downarrow}\}$
J	$=$	$\{z_i^{\uparrow}\}$
$i \in \{1,2\}, n \in \{0,1\}, x \in D$		

Table 13: Encapsulation, hide, and recovery set

$a{\cdot}\tau{\cdot}x$	$=$	$a{\cdot}x$	
$\tau{\cdot}x + x$	$=$	$\tau{\cdot}x$	
$a{\cdot}(\tau{\cdot}x + y)$	$=$	$a{\cdot}(\tau{\cdot}x + y) + a{\cdot}x$	
$\tau_I(\delta)$	$=$	δ	
$\tau_I(\perp)$	$=$	$\perp$	
$\tau_I(a{\cdot}x)$	$=$	$\tau{\cdot}\tau_I(x)$	$a \in I$
$\tau_I(a{\cdot}x)$	$=$	$a{\cdot}\tau_I(x)$	$a \notin I$
$\tau_I(x + y)$	$=$	$\tau_I(x) + \tau_I(y)$	

Table 14: Axioms for weak bisimulation and hiding

Peleska's system is weakly bisimilar with the basic system P for the trivial fault hypothesis δ, which means that the *normal behaviour* of T satisfies the property in Table 12. Peleska's original system ([Pel91]) can tolerate a single failure of one of its basic components P_1 or P_2, which is expressed by the fault hypothesis $FH = z_1^{\uparrow}{\cdot}\delta + z_2^{\uparrow}{\cdot}\delta$. The system we present can tolerate any number of faults of P_1 and P_2 provided the interval between consecutive faults is large enough and faults occur in active components only. This is modelled by synchronizing the fault hypothesis with the feedback signal h which results in the new fault hypothesis FH_1 (see Table 15.) The corresponding proof obligation is also given in that Table (It is assumed that $ch|ch = cch$.) A formal verification of the above protocol is provided in [Ham93] together with a discussion of the encountered problems.

FH_i	$=$	$z_i^{\uparrow} \cdot ch \cdot FH_{3-i} + ch \cdot FH_i$	Fault hypothesis
P	$\simeq_\tau$	$\tau_{I \cup \{cch\}} \circ \mathcal{R}_J \circ \partial_{\{ch\}}(T \parallel FH_1)$	Proof obligation

Table 15: Proof obligation for extended fault hypothesis

4 Conclusions

We have defined an exception handling primitive and recovery operator that have properties that are more in line with what is found in the fault tolerance literature [Cri85]; specifically, handlers are invoked innermost out and handlers can only become active through the occurrence of an error. We have developed an algebraic theory for these operators based on ACP [BK84]. We choose ACP because it is well developed, uniform theory. However, nothing stands in the way of developing a similar theory based on CCS [Mil89] or TCSP [Hoa85]. We have used this theory to specify a generalization of a fault resilient system of Peleska's [Pel91]. Finally, we want to note our use of non-well founded sets [Acz88] to construct models for our axiomatization. The standard method in ACP is to use the *process graph model*. In this model, elements are bisimulation equivalence classes of graphs and this fact makes the process graph model more difficult to use than the concrete model we introduce in this paper in which bisimilar process terms map onto the same element in the model, which element is straightforwardly determined by the SOS.

Future work includes further working out the example towards a formal proof of weak bisimilarity and extending the theory. We need to investigate more closely the connection of our theory with others such as the one in [Pra87]. Another question is whether a process algebra with prioritized actions [BW90] can be used to model exception handling. We already have some preliminary results. Finally, we want to extend our axiomatization to congruences coarser than bisimulation; specifically to maximal trace congruence.

Acknowledgement

We would like to thank Jos Baeten for his helpful comments.

References

[Acz88] P. Aczel. *Non-well-founded sets.* Number 14 in CSLI Lecture Notes. 1988.

[Ada83] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* LNCS 155, Springer-Verlag 1983.

[BK84] J.A. Bergstra & J.W. Klop. *Process Algebra for Synchronous Communication.* Information and Control 60:109–137, 1984.

[BW90] J.C.M. Baeten & W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science, Vol **18**, 1990.

[Cri85] F. Cristian. *A Rigorous Approach to Fault-Tolerant Programming.* IEEE Transactions on Software Engineering 11:23–31, 1985.

[Dix83] T.I. Dix. *Exceptions and Interrupts in CSP.* Science of Computer Programming 3:189–204, 1983.

[Gro90] J.F. Groote. *Transition System Specifications with Negative Premises.* Proc. CONCUR '90, LNCS 443 pp. 332-341, 1990.

[Gro91] J.F. Groote *Process Algebra and Structured Operational Semantics.* PhD Thesis University of Amsterdam, 1991.

[Ham93] A.M.R. Hamers. *The Proof of an Algebraic Specification of a Distributed Fault-Tolerant System.* Master's Thesis, Dept. of Real-Time Systems, Faculty of Math. and Computer Science, University of Nijmegen, 1993.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall 1985.

[HH87] He Jifeng & C.A.R. Hoare. *Algebraic Specification and Proof of a Distributed Recovery Algorithm.* Distributed Computing 2:1–12, 1987.

[LS90] K. Lodaya & R.K. Shyamasundar. *Proof Theory for Exception Handling in a Tasking Environment.* Acta Informatica 28:7–42, 1990.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall 1989.

[Pel91] J. Peleska. *Design and Verification of Fault Tolerant Systems with CSP.* Distributed Computing 5:95–106, 1991.

[Pra84] K.V.S. Prasad. *Specification and Proof of a Simple Fault Tolerant System in CCS.* Internal Report CSR-1178-84, Department of Computer Science, University of Edinburgh, 1984.

[Pra87] K.V.S. Prasad. *Combinators and Bisimulation Proofs for Restartable Systems.* PhD Thesis University of Edinburgh, 1987.

[Rut92] J.J.M.M. Rutten. *Processes as terms: non-well-founded models for bisimulation.* Mathematical Structures in Computer Science 2:257–275, 1992.

Chapter 5

Top-down Development of Layered Fault Tolerant Systems and Its Problems — a Deontic Perspective

This chapter is a revised version of:

J. COENEN.
Top-down Development of Layered Fault Tolerant Systems and Its Problems — a Deontic Perspective.
Annals of Mathematics and Artificial Intelligence **9**, pp. 133–150,
Special Issue on The First Intern. Workshop on Deontic Logic in Computer Science
(R. Wieringa & J.-J.Ch. Meyer, eds.), 1993.

Top-down Development of Layered Fault Tolerant Systems and Its Problems — a Deontic Perspective

J. COENEN [1]
Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

Abstract. Although the top-down development paradigm has successfully been applied to master the complexity of large systems, it has not yet been accepted as a useful paradigm for fault tolerant system design. This is mainly due to a problem that is sometimes referred to as the 'lazy programmers' paradox. The 'lazy programmer' paradox was already present and solved in top-down development methods for non-critical systems. However, the problem has re-appeared in an even more serious variant for critical systems. A few 'toy' examples concerning exception handling in an Ada-like language are used to explain and illustrate the paradox.

One possible solution to the problem is to use a specification language in which one can express that certain behaviours of a system are preferred over others. This paper proposes deontic logic as such a specification language. Therefore, a short and rather informal introduction to deontic logic is included. A non-trivial example is included to illustrate how deontic logic can be used to solve the 'lazy programmer' paradox.

Keywords: Deontic logic, Exception handling, Fault tolerance, Layered systems, Lazy programmer paradox, System specification, Top-down development.

[1]Supported by NWO/SION Project 612-316-022: "Fault Tolerance: Paradigms, Models, Logics, Construction."

1 Introduction

As computing systems are used more often for critical applications the importance of formal design methods for fault tolerant systems becomes more apparent (cf. [deRoever91]). Such design methods should provide not only formal specification and verification methods, but also a *design methodology* which supports the structuring of the system under development *and* the development process itself. Formal methods that meet these requirements adopt the top-down development paradigm. Top-down development methods incorporate some refinement method which is used to gradually transform a high level abstract specification into a low level concrete implementation. Each transformation step creates a new layer beneath the previous generated layers of the system, hence the name layered systems. One of the earliest descriptions of a layered system can be found in [Dijkstra68].

To overcome the complexity of its design, a fault tolerant system may, like most complex systems, be structured in layers. On the one hand, a layer may use the services delivered by its lower level layer to provide a service to its upper level layer. On the other hand, a layer may receive an exception from its lower level layer or raise an exception to signal its upper layer that it cannot provide a requested service. At each level, the system tries to handle the exceptions raised by the layer below. If the current layer is unable to cope with the current situation it may decide to raise an exception itself. In this way a malfunctioning of the underlying execution mechanism may gradually propagate to a layer which can deal with it in a satisfactory manner. A layer can therefore be regarded as an *ideal fault tolerant component* in the sense of Anderson and Lee [Anderson90], see figure 1. The arc directed from 'exceptional

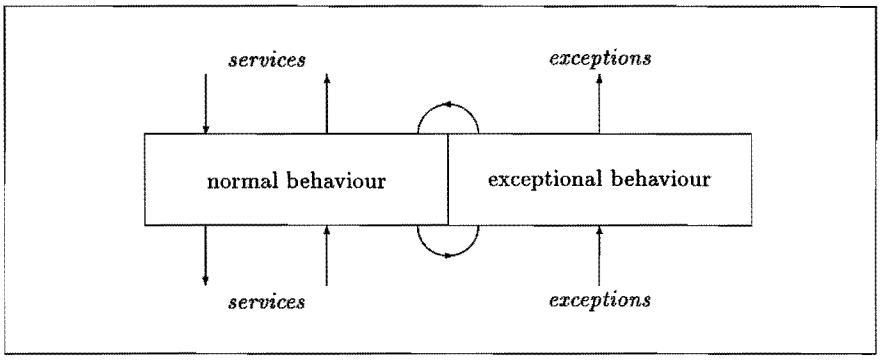

Figure 1: *Layer viewed as an ideal fault tolerant component*

behaviour' to 'normal behaviour' represents the case that the current layer handles an exception raised by a lower level (or the current level). The arc directed from 'normal behaviour' to 'exceptional behaviour' represents the case that an exception is raised by a lower level (or the current level). Notice, that in order to achieve a layered structure as described above, it must be possible to program a deliberately raised exception.

Any formal method that supports top-down development of layered fault tolerant systems has to solve the following two problems. Firstly, the method must provide a formal language to reason over faults and their effects. For example, Hoare's proof

system as it was presented in [Hoare69] can not deal with fault tolerance, because in this proof system a program is considered correct, if it behaves according to its specification under the assumption of a faultless execution mechanism.

In [Cristian85] Cristian extended Hoare's logic to deal with exceptions. However, in Cristian's formalism it is not possible to distinguish between deliberately raised exceptions and exceptions due to a physical fault in the executing hardware. Now, consider a specification of a program that computes the factorial $N!$ for input N. If an intermediate result of the computation causes an integer overflow, signalled by the exception *ovf*, it is specified that the result is zero. A lazy programmer might be tempted to write a program that outputs zero immediately and raises the exception *ovf* deliberately. This is of course not an acceptable implementation — the exception should only be raised due to an overflow in the underlying hardware — which can be avoided by explicitly stating that the programmer is not allowed to raise the exception *ovf*. This works well for this particular example, but it was already mentioned that it should be allowed to raise certain exceptions deliberately, e.g. to prevent undefined results. Because it is in general not possible to predict when such exceptions may occur, the lazy programmer cannot be prohibited from abusing his privilege to raise exceptions deliberately. This is a particular case of the second problem that has to be solved in any top-down development method for fault tolerant programs. The more general case of this problem is referred to as the 'lazy programmer' paradox, and will be discussed in more detail in section 4.

This paper is a first step towards a deontic specification language for fault tolerant systems. It does not include a semantic model nor does it include a complete proof theory. It merely discusses and illustrates the problems encountered when specifying the operations of fault tolerant system when adapting a top-down development strategy. This is unlike the work of [Meyer88] where (monadic) deontic logic is reduced to dynamic logic thereby obtaining a logic suitable for specifying the behaviour of programs without considering faults.

The merits of a dyadic deontic specification language is that it is possible to distinguish the behaviour in a perfect world (i.e. a computation without faults) from the (preferred) one in a less than perfect world. For example, if a program should satisfy a property φ, but due to some fault it does not we can specify a property ψ it should satisfy instead. Using dyadic deontic logic this can be specified as follows.

$$O\varphi \wedge (\neg\varphi)O\psi \ .$$

The conjunct $O\varphi$ is used instead of simply φ, because φ is not always satisfied but it *ought* to be if possible. The second conjunct specifies that if φ is not satisfied then ψ ought to be satisfied instead. If one would replace the second conjunct by an implication $(\neg\varphi) \rightarrow \psi$ the program that satisfies $\neg\varphi \wedge \psi$ would be a correct implementation, which was not intended. Replacing $(\neg\varphi)O\psi$ by $O(\neg\varphi \rightarrow \psi)$ or $\neg\varphi \rightarrow O\psi$ causes similar problems (see [Follesdal71]). For example, a specification φ for a system embedded in a perfect environment leaves no room for reasoning over the same system in a malicious environment that prohibits the system from satisfying φ, because $\varphi \rightarrow (\neg\varphi \rightarrow O\mathsf{false})$ forces the system to do the impossible in case it does not satisfy φ.

The remainder of this paper is organized as follows. In section 2, a programming language is defined and an intuitive explanation of the language constructs is given. In this section three small programs are explained. These programs are also used in section 3 to motivate the introduction, and explain the meaning of, dyadic modalities in the deontic logic specification language. Section 3 introduces deontic logic. The 'lazy programmer' paradox is discussed in somewhat more detail in section 4. Section 5 includes an informal description of a non-trivial fault tolerant system. The application of deontic logic as a specification language to solve the 'lazy programmer' paradox is illustrated in section 6 by specifying part of the example outlined in section 5. Finally, section 7 contains a comparison with related work and some suggestions for future work.

2 Program Notation

In this section a small subset of an Ada-like 'programming' language [Ada83], called $\mathcal{P}rog$ is defined. This programming language is also used in section 5 to describe some of operations used in the example. The main feature of the programming language $\mathcal{P}rog$ is that it provides a notation for exception handling.

Given the following basic sets:

- $\mathcal{V}ar$, the set of program variables, with typical element x;
- $\mathcal{E}xc$, the set of exceptions, with typical element exc;
- $\mathcal{E}xpr$, the set of expressions with occurrences of program variables, with typical element exp;
- $\mathcal{B}exp$, the set of boolean expressions with occurrences of program variables, with typical element b;

the syntactic class $\mathcal{P}rog$ of programs, with typical element S, is defined by

$$\begin{aligned} S ::= \ & \textbf{null} \mid x := exp \mid \textbf{raise } exc \mid \textbf{begin } S \textbf{ end} \mid S_1;\ S_2 \\ & \mid \textbf{if } b \textbf{ then } S \textbf{ fi} \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \\ & \mid \textbf{begin } S_0 \textbf{ exception when } exc_1 \Rightarrow S_1 \ldots \textbf{when } exc_k \Rightarrow S_k \textbf{ end} \end{aligned}$$

The meaning of the programming language constructs in $\mathcal{P}rog$ is as follows.

- The empty statement **null** has no effect other than skipping to the next statement.
- The assignment statement $x := exp$ assigns the value of the expression exp to the program variable x.
- The raise statement **raise** exc raises the exception exc. As a side effect it causes the execution of the program to continue at the innermost enclosing exception handler, that handles exc exceptions. If such enclosing exception handler does not exist, program execution is aborted.

- The simple block statement **begin** S **end** groups the statements in S in a single block. It may be regarded as a pair of parentheses.
- S_1; S_2 is the sequential composition of the programs S_1 and S_2. First S_1 is executed, and if S_1 terminates successfully, then S_2 is executed.
- In case of the alternative statement **if** b **then** S_1 **else** S_2 **fi**, the subprogram S_1 is executed if the boolean guard b is true, and S_2 is executed otherwise. The construct **if** b **then** S **fi** is an abbreviation of **if** b **then** S **else null fi**.
- The iterative statement **while** b **do** S is skipped if b is initially false. If b initially is true, then execution of S is repeated until b becomes false.
- **begin** S_0 **exception when** $exc_1 \Rightarrow S_1 \ldots$ **when** $exc_k \Rightarrow S_k$ **end** is executed as follows. The program starts with the execution of S_0. If during the execution of S_0 an exception exc_i $(i = 1, \ldots, k)$ is raised, then the execution of S_0 is aborted and the program resumes with the execution of S_i. If an exception other than exc_i $(i = 1, \ldots, k)$ is raised, then execution of S_0 is aborted, and the exception is passed to the next enclosing block. If there isn't an enclosing block the program is aborted. If S_0 terminates without raising an exception, then the program terminates normally.

For example, the programs listed in figure 2 are executed as follows. Program a assigns the factorial of N to variable x unless an *ovf* exception occurs — meaning that an overflow has been detected —in which case x is set to zero. Program b sets x to zero and then raises *ovf* deliberately. Program c assigns $N!$ to x if initially N is less or equal than K, and sets x to zero in case N is larger than K.

(a) **begin** $x := N!$ **exception when** *ovf* $\Rightarrow$ $x := 0$ **end**

(b) **begin** $x := 0$; **raise** *ovf* **end**

(c) **begin if** $N \leq K$ **then** $x := N!$ **else** $x := 0$ **fi end**

Figure 2: *Running examples*

3 Deontic Logic

The specification language combines deontic logic with first-order predicate logic, and is inspired by the logic used in [vEck82]. A systematic introduction to deontic logic in general is given in [Aqvist83]. The basic modality of the deontic logic used in this paper is the *dyadic obligation* $\varphi O \psi$. A more philosophical motivation of dyadic deontic logic can be found in [vWright71] and [vWright81]. The first-order predicates in the specification language are used to quantify over logical variables only.

Assume that the following sets are defined:

- $\mathcal{E}xpr'$, the extended set of expressions over program variables, which may be decorated with a prime. Thus $\mathcal{E}xpr \subset \mathcal{E}xpr'$.
- $\mathcal{L}var$, the set of logical variables, such that $\mathcal{L}var \cap \mathcal{E}xpr = \emptyset$. Logical variables never have primes attached to them.

A primed program variable x' refers to the value of the variable x before executing a program, whereas an unprimed variable x refers to the value of x after the execution of the program. The use of primed and unprimed variables in expressions captures the concept of initial and final states syntactically.

Given the sets above, the syntax of assertions $\varphi, \psi \in \mathcal{A}ssn$ is defined by ($exp_0, exp_1 \in \mathcal{E}xpr'$, $exc \in \mathcal{E}xc$, and $g \in \mathcal{L}var$)

$$\varphi ::= \mathsf{true} \mid exp_0 = exp_1 \mid exp_0 \leq exp_1 \mid \delta(exc) \mid \neg\varphi \mid \varphi \rightarrow \psi \mid \exists_g(\varphi) \mid \varphi O \psi$$

Notice that quantification is only allowed over logical variables. Besides the usual abbreviations for predicate logic (such as $\forall_g(\varphi)$ for $\neg\exists_g(\neg\varphi)$), the following derived operators are defined

$$\begin{array}{lcllcl} O\varphi & \triangleq & \mathsf{true} O\varphi & & & \\ \varphi F\psi & \triangleq & \varphi O\neg\psi & , & F\varphi & \triangleq & \mathsf{true} F\varphi \\ \varphi P\psi & \triangleq & \neg(\varphi O\neg\psi) & , & P\varphi & \triangleq & \mathsf{true} P\varphi \end{array}$$

The meaning of $\delta(exc)$ is that exception exc was raised. The notation δ is used to stress the difference with variables that refer to states instead of events. The meaning of $\varphi O \psi$ is that in all φ-perfect worlds (worlds that are perfect ϵ case) ψ is true. Hence, $O\varphi$ expresses that φ is the case in all perfect worlds. Similarly, $\varphi P\psi$ and $\varphi F\psi$ express that in all φ-perfect worlds ψ is respectively permitted and forbidden.

A formula with primed and unprimed variables specifies a relation between the initial and final state of a program. Hence, it can not distinguish between the individual actions of a program. The primed variables provide the specification language with the dynamic aspect needed to reason about programs. For instance $x = x' + 1$ specifies an action that increases the value of program variable x by one.

Below two standard derivation rules of deontic logic are given (see e.g. [Aqvist83]).

$$\frac{\varphi,\ \varphi \rightarrow \psi}{\psi} \quad \text{(Modus Ponens)} \qquad \frac{\vdash \psi}{\vdash \varphi O \psi} \quad \text{(Necessitation)}$$

The axioms below are more typical for the application discussed in the introduction. The first two are still quite common axioms, that should cause no problems. The third axiom is more typical for the logic. It expresses that all relative perfect worlds are perfect alternatives to themselves. Or more loosely, there is only one perfect alternative

for each world. It is motivated by the fact that the set of possible executions of a program does not change unless new faults are introduced.

$$\vdash \varphi O(\psi \rightarrow \chi) \rightarrow (\varphi O\psi \rightarrow \varphi O\chi)$$
$$\vdash \varphi O(\psi \wedge \chi) \leftrightarrow (\varphi O\psi \wedge \varphi O\chi)$$
$$\vdash \varphi O(O\psi) \rightarrow \varphi O\psi$$

This is of course not intended to be a complete axiomatization. The axiomatization of the logic itself is part of ongoing research in which there are still a lot of questions to be settled.

The most characteristic difference between the deontic logic defined above and the ones that can be found in the literature about system specification is that the above logic includes *dyadic* modalities. For example, Khosla [Khosla88] uses the monadic modalities $O\alpha$ respectively $P\alpha$ to express that the action α *must* respectively *may* be performed. Thus the deontic aspect of the specification language in [Khosla88] is used only to reason over the freedom of choice. In particular, a predicate $O\alpha$ is defined such that α is the *only* action that is obliged. Hence the formula $O\alpha \wedge O\beta$ is equivalent to false per definition if $\alpha \neq \beta$. When specifying fault tolerant systems with monadic modalities this causes a problem, because in a less than perfect world one can get several, sometimes conflicting, duties. In the more general context of deontic logic this is known as the Chisholm paradox (see [Aqvist67]).

Consider the following specification for a program that tries to anticipate a possible division by zero, when computing $1/x$ for input x.

$$O(x' \neq 0) \wedge O(x' \neq 0 \rightarrow y = 1/x') \wedge (x' = 0 \rightarrow O(y = 0)) \qquad (1)$$

This specification expresses that the input x is expected not to be zero, and it should be the case that if input x is not zero then y is $1/x$, and if x is zero then y ought to be zero. This seemingly correct specification is inconsistent in case the input x is zero. Using the above axioms and proof rules only it is possible to derive $O(y = 1/x')$ from $O(x' \neq 0)$ and $O(x' \neq 0 \rightarrow y = 1/x')$, and $O(y = 0)$ from $x' = 0$ and $(x' = 0 \rightarrow O(y = 0))$. The problem is that the monadic modalities refer to perfect worlds only, which may lead to a conflict of duties once one finds oneself in a less than perfect world. The behaviour of a fault tolerant system in less than perfect conditions should be specified, as the predicate 'fault tolerant' suggests.

Of course one might argue that if in the above specification $O(x' \neq 0 \rightarrow y = 1/x')$ is replaced by $x' \neq 0 \rightarrow O(y = 1/x')$ or $(x' = 0 \rightarrow O(y = 0))$ by $O(x' = 0 \rightarrow y = 0)$ then there is no problem if x is initially zero. The philosophical objections to do so (see e.g. [Follesdal71]) might be irrelevant to system specifications. The specification of fault tolerant systems is a difficult task even if one does not have to bother with such subtle paradoxes. Therefore, it is preferable to use a specification language in which such paradoxes can easily be avoided.

Dyadic deontic logic allows one to make assertions about less than perfect worlds. For example, the last conjunct of (1) may be replaced by $(x' = 0)O(y = 0)$, which does not

result in an inconsistency if $x' = 0$ because there is no detachment rule which allows one to derive $O(y = 0)$ from $x' = 0$ and $(x' = 0)O(y = 0)$. Notice that in the intuitive meaning of $\varphi O\psi$ it is implicit that ψ ought to be established, but φ is 'provided' for and not to be established. This observation is the key to the solution of the 'lazy programmer' paradox in section 4.

In this paper the use of dyadic modalities is restricted to the special case in which only exceptions occur on the left side of the modality, i.e. dyadic modalities occur in specifications only according to the format $\delta(exc)O\psi$. However, it is permitted to have predicates on the left side also. This is illustrated in the next section.

A standard technique to obtain a higher degree of reliability is the duplication of system components. For example, one may use two different algorithms to compute a certain value and compare the outcomes, say x and y, of these computations. In case $x \neq y$ at least one of the computations resulted in an error, and in case $x = y$ either both computations were correct or both computations yielded the same erroneous result. If one assumes that the probability of the latter case occurring is zero, the system sketched above is fault tolerant. A schematic drawing of a component that compares the outputs x and y is pictured in figure 3.

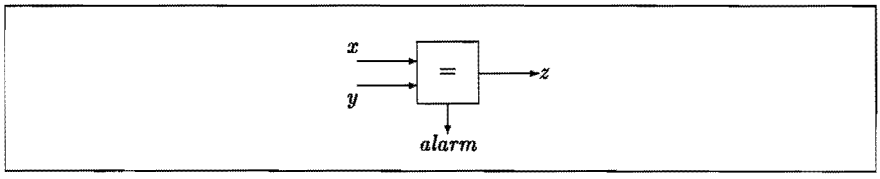

Figure 3: *Comparator*

The comparator may be specified by

$$O(z = x \wedge z = y \wedge \neg alarm) \wedge (x \neq y)O(alarm) ,$$

According to its specification, the comparator ought to set z equal to x and y, and set *alarm* to false . In case $x \neq y$ — and hence, it is not possible to set z equal to both x and y — *alarm* must be set to true .

4 The 'Lazy Programmer' Paradox

Lazy programmers were already a problem in Hoare's logic, because it is a *partial* correctness formalism which means that it is not possible to specify that a program must terminate. Hence, each *divergent* program is a correct implementation of every specification. This particular version of the 'lazy programmer' paradox is solved in *total* correctness formalisms in which one can specify that a program must terminate.

The particular formulation of the 'lazy programmer' paradox for fault tolerance has a

striking similarity with the 'Good Samaritan' paradox [2] (see [Aqvist67]). A program that is designed to tolerate only faults intentionally caused by that program itself, hardly deserves the predicate 'fault tolerant', just as little as a thief who salvages his own victims deserves to be called a Good Samaritan.

The programs in figure 2 serve to illustrate the previous discussion. Consider the following naive specification for a program that is to compute the factorial of N.

$$O((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)) . \tag{2}$$

The specifier has anticipated that, due to hardware limitations, it is possible that during the computation of $N!$ an overflow, signalled by exception *ovf*, occurs. If the overflow indeed occurs then x should be set to zero, otherwise x ought to be equal to $N!$. However, nothing prevents the lazy programmer from simulating an overflow as in program b of figure 2. Because program b ought to raise the exception *ovf* and set x to zero it satisfies

$$O(x = 0 \wedge \delta(ovf)) . \tag{3}$$

Unfortunately (3) specifies a correct implementation of (2), which can formally be proved as follows.

1. $\vdash (x = 0 \wedge \delta(ovf)) \rightarrow ((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0))$
2. $\vdash O((x = 0 \wedge \delta(ovf)) \rightarrow ((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)))$
3. $\vdash O((x = 0 \wedge \delta(ovf)) \rightarrow ((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)))$
   $\rightarrow (O(x = 0 \wedge \delta(ovf)) \rightarrow O((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)))$
4. $\vdash O(x = 0 \wedge \delta(ovf)) \rightarrow O((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0))$

The individual steps of the above derivation are justified as follows:
1. is a valid predicate logic formula;
2. is obtained by the application of the Necessitation rule to 1;
3. is an instance of the first axiom listed on page 73;
4. is obtained by applying Modus Ponens to 2 and 3.

Using dyadic modalities one can specify program a as follows.

$$O(x = N!) \wedge \delta(ovf)O(x = 0) . \tag{4}$$

This specification expresses that is preferred to set x equal to $N!$, and if this is not possible due to an overflow x ought to be zero. Provided that the axiomatization of the deontic logic does not allow one to derive (4) from (3), it is not possible to prove that program b is a correct implementation of (4). As a matter of fact, program b can be excluded more explicitly by adding the conjunct $F\delta(ovf)$ to (4). Hence, the lazy programmer has to think of other means to avoid working.

[2] The Good Samaritan ought to help a man who has been robbed. Thus there ought to be man who has been robbed.

Basically, the 'lazy programmer' paradox is solved by making the specification language more expressive. This imposes some requirements on the semantics and axiomatization of the programming language to avoid the situation in which an intuitively correct program does not satisfy a given specification. For example, suppose that the maximum number, say *MaxInt*, that can be computed without causing an overflow is known. If K is chosen such that $K! \leq MaxInt < (K+1)!$, then program c in figure 2 is intuitively a correct implementation of (4). The specification of program c is, however, as follows.

$$O((N \leq K \wedge x = N!) \vee (\neg N \leq K \wedge x = 0)) \tag{5}$$

The only way to prove that (5) specifies a correct implementation, i.e. to prove that (5) implies (4), is by making the knowledge about the hardware limitation explicit. For instance by including the following axioms.

$$\vdash O(N \leq K) \tag{6}$$
$$\vdash O(\neg N \leq K \rightarrow \varphi) \rightarrow \delta(ovf)O\varphi \tag{7}$$

Axiom (6) expresses that it ought to be the case that $N \leq K$. Axiom (7) expresses that if one is obliged to establish φ if $\neg N \leq K$ in a faultless world, this implies that φ ought to be established even if an overflow occurred. The second axiom is motivated by the knowledge that the overflow would have occurred anyway if $\neg N \leq K$. Because (5) is equivalent with

$$O(N \leq K \rightarrow x = N!) \wedge O(\neg N \leq K \rightarrow x = 0) \tag{8}$$

This can be proved as follows. Let ψ, φ_1, and ψ_2 be defined by

$$\begin{aligned} \psi &\triangleq (N \leq K \wedge x = N!) \vee (\neg N \leq K \wedge x = 0) \\ \varphi_1 &\triangleq N \leq K \rightarrow x = N! \\ \varphi_2 &\triangleq \neg N \leq K \rightarrow x = 0 \end{aligned}$$

We give the major steps of the derivation of (8) from (5):

$$\begin{aligned} &\vdash \psi \leftrightarrow (\varphi_1 \wedge \varphi_2) && \text{, Predicate logic.} \\ &\vdash O(\psi \leftrightarrow (\varphi_1 \wedge \varphi_2)) && \text{, Necessitation.} \\ &\vdash O\psi \leftrightarrow O(\varphi_1 \wedge \varphi_2) && \text{, Axioms and Modus Ponens.} \\ &\vdash O\psi \leftrightarrow (O\varphi_1 \wedge O\varphi_2) && \text{, Axioms and Modus Ponens.} \end{aligned}$$

Because it is easily seen that 8 implies

$$(O(N \leq K) \rightarrow O(x = N!)) \wedge O(\neg N \leq K \rightarrow x = 0)$$

we may conclude from (6) and (7) that program c is a correct implementation of (4), *provided* that above assumptions hold. Thus only if the hardware limitations are such that the axioms are justified the above reasoning holds.

It is possible to think of clever variations on the programs in figure 2, e.g. the ones in figure 4, for which the correct arguments to accept or reject them as correct implementations of (4) are not so easily found. For example, program d should be rejected, but just including $F\delta(ovf)$ in the specification would also exclude program e which might be acceptable. However, these problems should be solved in the semantics and the axiomatization of the programming language. The purpose of the previous discussion is to demonstrate that dyadic deontic logic, if provided with adequate semantics, can be expressive enough to distinguish deliberate errors from unintentional ones.

(d) **begin** $x := N!$; **raise** *ovf* **exception when** $ovf \Rightarrow x := 0$ **end**

(e)
```
begin
    if N ≤ K
        then x := N!
        else raise ovf
    fi
exception
    when ovf ⇒ x := 0
end
```

Figure 4: *The lazy programmer strikes back*[3]

5 A Stable Storage

An important concept in fault tolerant computing is the atomicity of actions. An action is atomic if it is either executed successfully or not executed at all. Atomic actions can be implemented by creating a checkpoint before the action is executed, and if an error is detected by recovering the original state from this checkpoint. The checkpoint should be recorded on a reliable medium, called a stable storage. This section contains a summary of some aspects of a particular stable storage and focuses on the implementation of the read operation. A more complete description of the stable storage, described below is given in [Schepers91].

The stable storage consists of three layers. At the lowest level, the stable storage is implemented by a number of physical disks. These physical disks, with the appropriate operations on them, are grouped in the so called 'physical disk' layer. Each physical disk has a corresponding logical disk, that abstracts from the physical location of sectors on the physical disk, by maintaining a flexible mapping between logical addresses and physical sector numbers. The logical disks are grouped together in the 'logical disk' layer. The layer at the top level is called the 'reliable disk' layer. The reliable disk layer provides a single stable storage, which is implemented by several logical disks.

[3]Or is it a too diligent programmer?

It is assumed that the only relevant errors are caused by damaged sectors of the physical disks. In the remainder of this section the layers are examined in somewhat more detail.

5.1 Reliable Disk Layer

The reliable disk layer provides a *read_sector* operation, with the intention that the contents of the sector with logical address *address* is retrieved in the variable *sector*. For this purpose, the reliable disk layer records which logical disks are still operational, i.e. which logical disks have not yet caused a *logical_disk_crash* exception. The numbers of the operational logical disks are administered in the set *operational_disks*. On invocation of the *read_sector* operation, an operational logical disk is selected on which a *read_logical_disk* operation is performed.

The reliable disk layer must anticipate two exceptions that may be raised by the logical disk layer. The exception *logical_sector_lost* indicates that this logical disk is unable to return the contents of the sector with logical address *address*. The exception *logical_disk_crash* is raised when the logical disk layer can no longer guarantee consistency of the information stored in the logical disk. In case of a *logical_sector_lost* exception, the reliable disk layer attempts to retrieve the sector from another logical disk. The retrieve operation will be left unspecified, but notice that retrieving the lost sector might include a recursive call of *read_sector*.

The *logical_disk_crash* exception is handled simply by deleting the corresponding disk number from the set *operational_disks*. If the reliable disk layer runs out of operational logical disks it raises a *reliable_disk_crash* exception. See also figure 5.

```
begin
  success := false ;
  while ¬success do
  begin
    disknr := a member of operational_disks ;
    read_logical_disk(disknr, address) ;
    success := true
  exception
    when logical_sector_lost ⇒
      retrieve the lost sector
    when logical_disk_crash ⇒
      operational_disks := operational_disks − {disknr} ;
      if operational_disks = ∅
        then raise reliable_disk_crash
      fi
  end
end
```

Figure 5: *read_sector*

Notice that the nondeterminism in the selection of an operational disk needs to be

resolved. This freedom of choice may be exploited to obtain a more efficient *read_sector* operation.

5.2 Logical Disk Layer

Whereas the reliable disk layer achieves a higher degree of reliability through the redundancy of the logical disks, the logical disk layer, in turn, achieves a higher degree of reliability through the redundancy of so called spare sectors on each logical disk. The spare sectors are recorded in the set *spare_sectors*. Furthermore, the logical disk layer abstracts from the physical location of sectors by maintaining a mapping *log_to_phys* between logical addresses and sector numbers.

```
begin
  read_physical_sector(log_to_phys(address))
exception
  when invalid_crc ⇒
    if spare_sectors = ∅
      then raise logical_disk_crash
      else new_sector := a member of spare_sectors ;
        spare_sectors := spare_sectors - {new_sector} ;
        update log_to_phys ;
        raise logical_sector_lost
    fi
end
```

Figure 6: *read_logical_disk*

The read operation at the logical disk level is listed in figure 6. The logical disk layer simply calls the *read_physical_disk* operation with the converted address. If the physical disk layer raises the *invalid_crc* exception and there are no spare sectors left, then the logical disk layer raises a *logical_disk_crash* exception. If the *invalid_crc* exception is raised and there are spare sectors, then one of the spare sectors is selected and the mapping *log_to_phys* is updated, and the *logical_sector_lost* exception is raised.

5.3 Physical Disk Layer

The physical disk layer achieves reliability by using information redundancy. The contents of each logical sector is augmented with a cyclic redundancy code. It is assumed that all relevant faults can be detected with this code. Or more precisely, the probability of not detecting a relevant error is sufficiently small. This means that faults like damaged disk drives etc. are not considered relevant in this example. The *read_physical_sector* operation is listed in figure 7.

The cyclic redundancy code is checked by the function *cyc_red_check*, which may be implemented by special purpose hardware.

```
begin
  sector := physical_disk[sector_nr] ;
  if ¬cyc_red_check(sector)
    then raise invalid_crc
  fi
end
```

Figure 7: *read_physical_sector*

6 Deontic Logic Specifications of the Read Operations

A specification of an operation of a fault tolerant system typically has the following format

$$\varphi_1 O\psi_1 \wedge \ldots \wedge \varphi_n O\psi_n \ .$$

Each ψ_i specifies how the operation of this layer should behave, provided the lower level created the condition φ_i. Because the upper level layer cannot interfere with the actions of the lower level layer, the conditions φ_i are established facts for the upper level layer to which it is supposed to react according to ψ_i. For example, at the top level of the stable storage the read operation may have been specified as follows.

$$O(sector = reliable\_disk'(address')) \wedge \delta(reliable\_disk\_crash)O\psi \ ,$$

where ψ is left open for the moment. Thus it is specified that the read operation ought to assign the initial contents of the stable storage at address *address* to *sector*. In case a *reliable_disk_crash* exception was raised, ψ ought to be established. Of course, one might also have specified that e.g. the address or contents of the storage ought to be left unchanged.

Because the physical disk layer is the lowest level of the stable storage and it is assumed that *cyc_red_check* detects all errors, there are no faults (from lower levels) that must be anticipated by this layer. Therefore, the specification of the *read_physical_sector* operation (figure 7) contains only monadic modalities. The *read_physical_sector* operation (for physical disk i) is specified by

$$O(sector = physical\_disk_i'[sectornr']) \\ \wedge O(\delta(invalid\_crc) \rightarrow \neg cyc\_red\_check(sector')) \ .$$

The first conjunct expresses that if the underlying execution mechanism functions correctly then *sector* is set equal to the contents of physical disk i at location *sectornr*. The second conjunct of this specification can be rewritten as

$$F(\delta(invalid\_crc) \wedge cyc\_red\_check(sector')) \ ,$$

which forbids to raise the *invalid_crc* exception when the sector passes the cyclic redundancy check. Now suppose an *invalid_crc* exception ought to be raised, i.e.

$$O\delta(invalid\_crc)\ .$$

From the specification of the *read_physical_sector* operation it follows that

$$O(\delta(invalid\_crc) \rightarrow \neg cyc\_red\_check(sector'))\ .$$

This together with the following axiom instance

$$O(\delta(invalid\_crc) \rightarrow \neg cyc\_red\_check(sector')) \\ \rightarrow (O\delta(invalid\_crc) \rightarrow O\neg cyc\_red\_check(sector'))$$

is sufficient to derive

$$O\delta(invalid\_crc) \rightarrow O\neg cyc\_red\_check(sector')$$

with modus ponens. One more application of modus ponens results in

$$O\neg cyc\_red\_check(sector')\ .$$

Hence, under the assumption that the physical disk functions correctly it is established that the *invalid_crc* exception ought to be raised only if the sector didn't pass the cyclic redundancy check.

Notice that if the second conjunct in the specification of *read_physical_sector* is replaced by

$$\delta(invalid\_crc) \rightarrow O\neg cyc\_red\_check(sector')$$

then an *invalid_crc* exception ensures that *sector* didn't pass the cyclic redundancy check regardless whether the exception was raised by *read_physical_sector* operation itself or by another operation.

The logical disk layer must anticipate an *invalid_crc* exception, but is allowed to raise a *logical_disk_crash* exception or a *logical_sector_lost* exception depending on whether there are any spare sectors available (figure 6). The *read_logical_sector* operation (for logical disk i) is specified by

$$O(sector = logical\_disk_i'(address')) \wedge \\ \delta(invalid\_crc)O((\delta(logical\_disk\_crash) \wedge spare\_sectors_i' = \emptyset) \vee \\ (\delta(logical\_sector\_lost) \wedge spare\_sectors_i' \neq \emptyset))\ .$$

A single logical disk cannot handle an *invalid_crc* exception by itself, but achieves graceful degradation through the discrimination between the fatal situation in which

there aren't any spare sectors left, and the less harmful situation when there are enough redundant sectors. Assuming that this layer functions correctly, it follows that a *logical_disk_crash* exception is raised if an *invalid_crc* exception was detected and initially the number of spare sectors was zero. To ensure that a *logical_disk_crash* or *logical_sector_lost* is raised only in the situation described above, the specification may be strengthened by adding the conjunct $F(\delta(logical\_disk\_crash) \wedge \delta(logical\_sector\_lost))$, which forbids raising these exceptions deliberately. Notice that this specification is not complete because it does not specify that the mapping *log_to_phys* should be updated before raising the *logical_sector_lost* exception.

Although the reliable disk layer must handle both exceptions that may possibly be raised by the logical disk layer, the specification below only anticipates the occurrence of a *logical_disk_crash* exception. Therefore also this specification is not complete. The *read_sector* operation (figure 5) of the reliable disk layer is specified by

$$\begin{aligned} & O\exists_i(i \in operational\_disks' \wedge sector = logical\_disk_i'(address')) \\ \wedge\ & \delta(logical\_disk\_crash)O(\delta(reliable\_disk\_crash) \rightarrow operational\_disks = \emptyset)\ . \end{aligned}$$

Suppose that it is forbidden to raise the *reliable_disk_crash* exception deliberately, which may be accomplished by adding the conjunct $F\delta(reliable\_disk\_crash)$ to the specification above. Then it follows that a *reliable_disk_crash* exception is only raised if there are no other operational disks left and a *logical_disk_crash* was raised. Thus the only initially operational disk doesn't have the appropriate information.

7 Conclusions

The previous section illustrates how deontic logic provides the possibility to specify fault tolerant systems in a natural way. It turns out that to derive certain properties of a specified system, one needs to make the assumptions about faults and their effect on the behaviour of the system explicit. The possibility to express the preference of some behaviors over others, allows one to distinguish between conditions created by a possible malfunctioning of a lower level, and the conditions created by the layer under discussion itself. Although deontic logics have been suggested for system specification before, e.g. in [Khosla88], the application to fault tolerant systems seems to be new, which partly explains the differences between the specification language used in this paper and those appearing in the literature about system specification.

The deontic logic described in this paper differs from the deontic logics for system specification in the existing literature mainly in two ways. Firstly, the logic in this paper is a *dyadic* deontic logic, whereas the logics in e.g. [Meyer88] and [Khosla88] are *monadic* deontic logics. Secondly, primed and unprimed variables are used to capture the dynamic aspect of programs in the specification language, whereas Meyer [Meyer88] and Khosla [Khosla88] use a dynamic logic in combination with the deontic logic.

The first difference, which seems to be the most essential one, can be explained by the particular application to fault tolerant systems. An important concept in fault tolerance is *graceful degradation*, which allows a system to temporarily sacrifice a service

in favor of a more important one if a fault occurs. This corresponds in a natural way with deontic logic specification of the format $\varphi_1 O\psi_1 \wedge \ldots \varphi_n O\psi_n$ that specify the behaviour ψ_i of a system under less than perfect conditions φ_i $(i = 1, \ldots, n)$. Moreover, dyadic deontic logic offers a solution to the 'lazy programmer' paradox described in section 4. And, although the examples used to illustrate this paradox may be regarded as 'toy' examples, it should be evident from the example in section 5 that this problem becomes more important as the complexity of a system increases.

The second difference concerns primed variables. A nice property of the logic is that it captures *state* predicates as well as *action* predicates. State predicates are predicates with either only primed variables or only unprimed variables. Action predicates are predicates with both primed and unprimed variables. A serious disadvantage of the primed and unprimed variables is that it is not clear how this method can be extended to deal with (distributed) real-time systems, which is an important application area of fault tolerance. Such systems may be specified in a logic that mixes deontic logic with a temporal logic, or in a logic with combined deontic-temporal modalities like the one in [vEck82].

The next step which must be taken is the definition of an adequate formal semantics for the deontic logic discussed in this paper. A first study shows that a Kripke semantics can be obtained by introducing residuals of reachability relations.

Acknowledgment. The author is grateful to Henk Schepers for providing the stable storage example, and Tijn Borghuis and Wim Koole for many helpful discussions.

References

[Ada83] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983, LNCS 155. Springer-Verlag, 1983.

[Anderson90] T. Anderson & P.A. Lee. *Fault Tolerance: Principles and Practice*, 2nd. revised edition. Springer-Verlag, 1990.

[Aqvist67] L. Åqvist. *Good Samaritans, Contrary-to-Duty Imperatives, and Epistemic Obligations.* Noûs 2, pp. 361–379, 1967.

[Aqvist83] L. Åqvist. *Deontic Logic.* In "Handbook of Philosophical Logic" Vol. II, pp. 605–714. D. Gabbay & F. Guenthner (Eds.), Reidel, 1983.

[Cristian85] F. Cristian. *A Rigorous Approach to Fault-Tolerant Programming.* IEEE Trans. on Softw. Eng. Vol. SE-11 pp. 23–31, 1985.

[Dijkstra68] E.W. Dijkstra. *The Structure of the "THE"-Multiprogramming System.* Comm. of the ACM 11:341–346, 1968.

[vEck82] J.A. van Eck. *A System of Temporally Relative Modal and Deontic Predicate Logic and Its Philosophical Applications.* Logique et Analyse 100, pp. 249–381, 1982.

[Follesdal71] D. Føllesdal & R. Hilpinen. *Deontic Logic: an Introduction.* In "Deontic Logic: Introductory and Systematic Readings", pp. 1–35. R. Hilpinen (Ed.), Reidel 1971.

[Hoare69] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, Vol. 12 pp. 576-580, 1969.

[Khosla88] S. Khosla. *System Specification: a Deontic Approach.* Ph.D. Thesis University of London (Imperial College of Science and Technology), 1988.

[Meyer88] J.-J.Ch. Meyer. *Using Programming Concepts in Deontic Reasoning.* Report IR-161 Free University Amsterdam, 1988.

[deRoever91] W.-P. de Roever. *Foundations of Computer Science: Leaving the Ivory Tower.* Bulletin of the EATCS 44:455–492, 1991.

[Schepers91] H. Schepers. *Terminology and Paradigms for Fault Tolerance.* In 'Formal Techniques in Real-Time and Fault Tolerant Systems', J. Vytopil (Ed.), pp. 3–31, Kluwer 1993.

[vWright71] G.H. von Wright. *A New System of Deontic Logic.* In "Deontic Logic: Introductory and Systematic Readings", pp. 105–120. R. Hilpinen (Ed.), Reidel 1971.

[vWright81] G.H. von Wright. *Problems and Prospects of Deontic Logic: a Survey.* In "Modern Logic — a Survey: Historical, Philosophical, and Mathematical Aspects of Modern Logic and Its Applications", pp. 399–423. E. Agazzi (Ed.), Reidel 1981.

Samenvatting

Foutbestendige computersystemen hebben als speciale eigenschap dat ze correct blijven functioneren indien er fouten optreden. Het is natuurlijk niet mogelijk om een systeem zo te ontwerpen dat het tegen iedere denkbare fout opgewassen is. Daarom eist men slechts dat het systeem bestand is tegen een bepaalde klasse van fouten. De eigenschap dat een systeem foutbestendig is maakt het geschikt voor toepassingen in kritische omgevingen waar het falen van een systeem of een component ongewenste gevolgen kan hebben.

Bij het ontwerpen van een foutbestendig systeem moeten geen nieuwe fouten geïntroduceerd worden. Het belang van een formele ontwikkelingsmethode is daarom juist voor foutbestendige systemen van belang. Een formele methode biedt de mogelijk om de gewenste eigenschappen van het systeem op hoog niveau te specificeren waarbij afgezien kan worden van implementatiedetails. Tevens bevat de methode een taal waarmee concrete implementaties beschreven kunnen worden. De methode moet daarnaast de mogelijkheid bieden om formeel af te leiden of de uiteindelijke implementatie inderdaad de gespecificeerde eigenschappen heeft. Idealiter ondersteunt de methode een ontwerper bij het structureren van zowel het beoogde systeem als ontwerpproces zelf.

Het onderhavige proefschrift beschrijft het onderzoek dat zich richtte op vinden van een formele ontwikkelingsmethode voor foutbestendige systemen. Het eerste hoofdstuk bevat een korte inleiding tot en een overzicht van het verrichte onderzoek. Het tweede hoofdstuk beschrijft hoe twee bekende formalismen voor programmaontwikkeling aan elkaar gerelateerd zijn. Het belang hiervan is dat het aantoont hoe op het eerste oog verschillende methoden uiteindelijk dezelfde klasse van systemen — in dit geval sequentïle programma's — bschrijven. Het derde hoofdstuk definieert een interpretatie van een taal waarin foutbestendige tijdkritische systemen beschreven kunnen worden. Tijdkritische systemen zijn systemen waaraan tijdseisen gesteld worden. Het vierde hoofdstuk bevat een algebraïsche methode voor de verificatie van foutbestendige systemen. Hoofdstuk vijf tenslotte, wordt een algemeen probleem voor formele methode voor foutbestendigheid beschreven. Een eerste aanzet tot de oplossing van dit probleem met behulp van een bijzondere logica wordt gegeven.

Promotiereglement
Artikel 15.3b

The EUT "promotiereglement" requires that if a thesis contains co-authored papers it should be indicated which parts are based on active contributions of the author of the thesis.

Both chapter 3 and chapter 4 are typical joint articles that are difficult to entangle. The basic computational model of chapter 3 is due to Jozef Hooman; the original idea and all adaptations to fault tolerance and simplifications are due to the author of this thesis. In the case of chapter 4, the first set-up for an algebraic theory for exception handling and its application to the specification and verification of fault tolerant systems is due to the author of this thesis. The formalization in ACP was proposed by Frank de Boer.

It is not feasible to provide any further quantitative or qualitative division of the efforts reported on in these chapters. All co-authors have been actively involved in parts of the research.

STELLINGEN

behorende bij het proefschrift

FORMALISMS FOR PROGRAM REIFICATION AND FAULT TOLERANCE

van

JOS COENEN

I. Er bestaat geen volledige regel voor programmareïficatie in Floyd-Hoare logica, zoals gedefinieerd in [Hoa69], zonder logische variabelen.

II. Iedere Hoare formule $\{\varphi\}\ S\ \{\psi\}$ kan als volgt in een equivalente normaalvorm gebracht worden

$$\{x = y_0\}\ S\ \{\varphi[y_0/x] \rightarrow \psi\}$$

Hierin stelt x de lijst van programmavariabelen in φ en ψ voor, en y_0 een even lange lijst van verse logische variabelen.

III. De volgende bewijsregel voor neerwaartse simulatie (downward simulation) is volledig voor gesloten relationele termen R_1, R_2 en R_3, d.w.z. relationele termen zonder vrije voorkomens van specificatievariabelen..

$$\frac{\{\exists_a(\chi_\alpha \wedge x = y_0)\}\ R_1\ \{\exists_a(\chi_\alpha \wedge (\varphi[y_0/x] \rightarrow \psi))\},\ \{\varphi\}\ R_2\ \{\psi\} \rightarrow R_2 \subseteq R_3}{\alpha^{-1};\ R_1 \subseteq R_3;\ \alpha^{-1}}$$

IV. De volgende bewijsregel voor opwaartse simulatie (upward simulation) is volledig voor gesloten relationele termen R_1, R_2 en R_3, met gesloten als gedefinieerd als in stelling III.

$$\frac{\{x = y_0\}\ R_1\ \{\forall_a(\chi_\alpha \rightarrow (\exists_a(\chi_\alpha[y_0/x]) \wedge (\varphi[y_0/x] \rightarrow \psi)))\},\ \{\varphi\}\ R_2\ \{\psi\} \rightarrow R_2 \subseteq R_3}{R_1;\ \alpha \subseteq \alpha;\ R_3}$$

V. De volgende niet compositionele bewijsregel voor simulatie is op zich volledig voor gesloten relationele termen R_1, R_2 en R_3, met gesloten als gedefinieerd in stelling III.

$$\frac{\{x = y_0\}\ R_1\ \{\exists_a(\chi_\alpha \wedge (\exists_a(\chi_\alpha[y_0/x]) \wedge (\varphi[y_0/x] \rightarrow \psi)))\},\ \{\varphi\}\ R_2\ \{\psi\} \rightarrow R_2 \subseteq R_3}{R_1 \subseteq \alpha;\ R_3;\ \alpha^{-1}}$$

VI. Door toevoeging van de volgende verificatieconditie voor opwaartse simulatie (upward

simulation) kan VDM (zie [Jon90]) volledig gemaakt worden m.b.t. programmareïficatie.

$$\forall \overline{c}, c \in C, a \in A \cdot rel(a, c) \wedge pre\text{-}C(\overline{c}) \wedge post\text{-}C(\overline{c}, c) \Rightarrow \exists \overline{a} \in A \cdot rel(\overline{a}, \overline{c}) \wedge pre\text{-}A(\overline{a}) \wedge post\text{-}A(\overline{a}, a)$$

VII. Back's algemene verificatieconditie voor programmareïficatie (zie [Bac88]), waarin fortuinlijk nondeterminisme (angelic nondeterminism) is toegelaten, is volledig voor neerwaartse simulatie (downward simulation) indien geeist wordt dat de abstractie relatie totaal is.

VIII. Superpositie van programma's is een vorm van contextverfijning waarvoor de eis van subdistributiviteit te sterk is [ZCdR92].

IX. Dat de keuze van een symmetrisch symbool voor gelijkheid in de rekenkunde niet voor iedereen intuïtief hoeft te zijn kan als volgt experimenteel aangetoond worden. Neem twee repen chocola en breek van ieder reep een derde deel af en verdeel het resultaat gelijkelijk onder drie jonge kinderen. Twee van de drie kinderen zullen verongelijkt reageren omdat het andere kind twee stukken heeft.

X. Vanuit het oogpunt van betrouwbaarheid verdient de toepassing van het TMR-principe (Triple Modular Redundancy) in grootschalige uniforme netwerken de voorkeur boven het principe van zelfcontrole door duplicatie (self-checking logic) [Coe94].

XI. De werking van een remmende synaps in een neuraal netwerk correspondeert niet met die van één enkele invertor in een logisch circuit [Coe93].

XII. Het onderscheid tussen 'natte' en 'droge' horeca is zeer bedenkenswaardig in een debat over gokverslaving waarbij drankmisbruik niet aan de orde komt.

Referenties

[Bac88] Back, R.J.R.: *Data Refinement in the Refinement Calculus.* Reports on Computer Science & Mathematics 68, Åbo Akademi, 1988.

[Coe93] Coenen, J.: Modelling Reliable Neural Networks. *Proc. 3rd ICYCS*, 3.01–3.06, Tsinghua University Press 1993.

[Coe94] Coenen, J.: Simulating Large Neural Assemblies of Unreliable Components. *Cybernetics & Systems*, **25**, 335-342 (1994).

[Hoa69] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. *C. ACM*, **12**, 576–580 (1969).

[Jon90] Jones, C.B.: *Systematic Software Development Using VDM* (second edition). Prentice-Hall, 1990.

[ZCdR92] Zwiers, J.; Coenen, J. & De Roever, W.-P.: A Note on Compositional Refinement. *Proc. 5th BCS-FACS Refinement Workshop*, 342–366, Workshops in Computing, Springer 1992.