Assertional data reification proofs : surveys and perspective

Eindhoven University of Technology

Department of Mathematics and Computing Science

Assertional Data Reification Proofs: Survey and Perspective

by

J. Coenen W.-P. de Roever J. Zwiers

Computing Science Note 91/21
Eindhoven, September 1991

COMPUTING SCIENCE NOTES

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

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

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

Assertional Data Reification Proofs: Survey and Perspective *

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

W.-P. de Roever‡
Institut für Informatik und
Praktische Mathematik
Christian-Albrechts-Universität Kiel
D-2300 Kiel, Fed. Rep. Germany

J. Zwiers§
University of Twente
P.O. Box 217
7500 AE Enschede, The Netherlands

Abstract

In this survey we discuss three methods for program development, which incorporate data reification: VDM, Reynolds' method, and Back's method and develop a modest predicate transformer based framework to relate them. At first we consider partial correctness only, and discuss Reynolds' method and a partial correctness version of VDM. Later we also consider total correctness in order to cover (part of) Back's refinement calculus and the full notion of specification and associated refinement methods in VDM.

Keywords: Data reification, Program transformation.

1 Introduction

Already in the early seventies it became clear that data reification is a powerful principle in program development [Milner71][Hoare72] [Gerhart75]. It was already present in the methods presented in [Back78][Jones80] and [Reynolds81], which laid the foundations for the program development methods like e.g. VDM [VDM86][VDM89].

In this survey we discuss three methods for program development, which incorporate data reification: VDM, Reynolds' method, and Back's method and develop a

*Appeared in *Proc. 4th BCS-FACS Refinement Workshop*, pp. 97–114. Workshops in Computing, Springer-Verlag 1991.

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

‡Partially supported by ESPRIT project 3096: "SPEC." E-mail: wpr@informatik.uni-kiel.dbp.de

§E-mail: zwiers@cs.utwente.nl

modest predicate transformer based framework to relate them. The Vienna Development Method is included, because it has become one of the major design methods. Back's refinement calculus is included as a representative of the class of refinement calculi such as [Morgan90] and [Morris89]. Finally, we consider the method advocated by Reynolds in chapter 5 of [Reynolds81], because in our opinion it is the one presented most elegantly along with the best worked out examples.

As mentioned already, besides surveying the above mentioned methods, we also set up a modest theory based on predicate transformers and relations which we use to relate them. At first we consider partial correctness only, and discuss Reynolds' method and a partial correctness version of VDM. Later we extend our theory to total correctness in order to cover Back's refinement calculus and the full notion of refinement in VDM.

Maybe this is the proper place to explain our usage of the term reification instead of refinement. In our opinion *refinement* is best defined by Gardiner and Morgan in [Gardiner90]: "One module is said to be refined by a second if no program using the second module can detect that it is not using the first." As such it can be conceived of as a compositional notion, related to the notion of subdistributivity by Hoare, He and Sanders [Hoare87]. However in actual program development sometimes noncompositional "refinement" techniques occur such as e.g. optimization in compilers. Another example of noncompositional refinement, in the style of Reynolds, is the implementation of the program fragment

$$\textbf{begin new}\, y;\ y := \textit{ a member of } U;\ U := U - \{y\}\, \textbf{end}$$

by $b := b - 1$, where the set U is represented by an array segment and b denotes the upper bound of that segment. The statement implements neither of the two abstract operations and is only correct for this particular abstract program fragment. As a last example illustrating a noncompositional aspect of refinement we mention a notion of Abadi and Lamport which is related to the introduction of prophecy variables that add stuttering (cf. section 5.3 of [Abadi88]). To cover both these cases and action refinement we prefer to use the term reification, for want of a better term.

Actually there is multitude of refinement notions in program development:

- **compositional** or **process refinement** exploiting the compositional nature of a formalism, see e.g. [Gries81][Hehner84] [Morgan90][Zwiers89],

- **action refinement** typically applying to the implementation of atomic actions in a concurrent programming environment, see e.g. [Lamport83] [Abadi88], or the many references in [deBdeRRoz90],

- **data refinement**, see e.g. [Hoare72][Gerhart75] [Jones80][Reynolds81][Back88].

We will be concerned with data refinement only in this paper.

The outline of the remainder of this paper is as follows. In section two we discuss refinement within Hoare's logic. Section three is devoted to partial correctness preserving refinement. Total correctness preserving refinement is the subject of section four, and we conclude with a discussion and some ideas for future work.

2 Reification in Hoare's logic

We start with the presentation of a modest theory of relations and predicate transformers. Next, we define the meaning of correctness formulae and specifications. Furthermore, we discuss four simulation notions, and show how these simulations can be used to prove refinement in Hoare's logic. Most of the results are already well known, although they are sometimes presented in a different framework.

Although, in this section, we are mainly concerned with reification in a Hoare-style proof system we set up the theory in such a way that also a concise meaning can be given to our version of partial correctness formulae in VDM.

2.1 Relations and predicate transformers

Relations appear in four ways in our theory. Firstly, programs are interpreted as relations on states, i.e. we use a relational semantics. Therefore, secondly, we also interpret specification as relations. Thirdly, the postconditions in VDM specifications and correctness formulae denote relations on states. For this reason they are often referred to as post-relations. And fourthly, representation invariants, which are used to prove reification steps correct, define relations between states of two possibly different state spaces.

We use the following notations for relations. Throughout this paper p and q denote sets of states, and (indexed) r denotes a relation on states.

$$\begin{array}{lcl} r^{-1} & \hat{=} & \{(\tau,\sigma) \mid (\sigma,\tau) \in r\} \ , \\ p \rightsquigarrow q & \hat{=} & \{(\sigma,\tau) \mid \sigma \in p \rightarrow \tau \in q\} \ , \\ p \rightsquigarrow r & \hat{=} & \{(\sigma,\tau) \mid \sigma \in p \rightarrow (\sigma,\tau) \in r\} \ , \\ p;\, r & \hat{=} & \{(\sigma,\tau) \mid \sigma \in p \wedge (\sigma,\tau) \in r\} \ , \\ r_0;\, r_1 & \hat{=} & \{(\sigma,\tau) \mid \exists_{\sigma'}((\sigma,\sigma') \in r_0 \wedge (\sigma',\tau) \in r_1)\} \ . \end{array}$$

The symbol $\rightsquigarrow$ is pronounced as "leads to", r^{-1} denotes the converse of r, $p;\, r$ and $r_0;\, r_1$ respectively denote the sequential composition of a filter and a relation and the sequential composition of two relations.

Furthermore, we use the following notations for sets:

$$\begin{array}{lcl} \langle r \rangle p & \hat{=} & \{\sigma \mid \exists_\tau((\sigma,\tau) \in r \wedge \tau \in p)\} \ , \\ [r]p & \hat{=} & \{\sigma \mid \forall_\tau((\sigma,\tau) \in r \rightarrow \tau \in p)\} \ , \\ r(p) & \hat{=} & \{\tau \mid \exists_\sigma((\sigma,\tau) \in r \wedge \sigma \in p)\} \ . \end{array}$$

These sets correspond with the notions of strongest postcondition ($r(p)$ or $\langle r^{-1} \rangle p$) and weakest precondition ($[r]p$). Lemma 2.1.1 lists some useful properties of the above predicate transformers.

Lemma 2.1.1

$$\begin{array}{lcl} \langle r^{-1}\rangle p & = & r(p) \\ \langle r_0; r_1\rangle p & = & \langle r_0\rangle\langle r_1\rangle p \\ [r_0; r_1]p & = & [r_0][r_1]p \end{array}$$

□

A relation r is *total* if for every state σ in the domain of r there exists a state τ such that $(\sigma, \tau) \in r$.

Lemma 2.1.2
If, and only if r is total then

$$\begin{array}{lcl} (p \rightsquigarrow q); r^{-1} & = & p \rightsquigarrow (\langle r\rangle q) \\ r; (p \rightsquigarrow q) & = & ([r]p) \rightsquigarrow q \end{array}$$

□

2.2 Specifications and correctness formulae

We assume a syntactic class $\mathcal{E}xpr$ of expressions with occurrences of program variables $x \in \mathcal{V}ar$, a disjoint set of 'hooked' program variables $\overline{x}$, and another set, disjoint with the previous ones, of logical variables $g \in \mathcal{L}var$. 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 $\mathcal{E}[\![\,.\,]\!] : \mathcal{E}xpr \to (\Gamma \to ((\Sigma \times \Sigma) \to \mathcal{V}al))$ is defined such that

$$\begin{array}{lcl} \mathcal{E}[\![\, x \,]\!]\gamma(\sigma,\tau) & \hat{=} & \tau(x) \\ \mathcal{E}[\![\, \overline{x} \,]\!]\gamma(\sigma,\tau) & \hat{=} & \sigma(x) \\ \mathcal{E}[\![\, g \,]\!]\gamma(\sigma,\tau) & \hat{=} & \gamma(g) \end{array}$$

The syntactic class $\mathcal{A}ssn$ of assertions, with typical elements χ, is defined by ($e_1, e_2 \in \mathcal{E}xpr$)

$$\chi \ ::= \ \text{true} \mid e_1 = e_2 \mid \neg\chi \mid \chi_1 \to \chi_2 \mid \exists_g(\chi)$$

We will use the usual abbreviations such as e.g. $\chi_1 \vee \chi_2$. Assertions are interpreted by a truth-valued function $\mathcal{T}[\![\,.\,]\!] : \mathcal{A}ssn \to (\Gamma \to ((\Sigma \times \Sigma) \to \{tt, ff\}))$, which defined as follows.

$$\begin{array}{lcl} \mathcal{T}[\![\,\text{true}\,]\!]\gamma(\sigma,\tau) & \hat{=} & tt \\ \mathcal{T}[\![\, e_1 = e_2 \,]\!]\gamma(\sigma,\tau) & \hat{=} & \mathcal{E}[\![\, e_1 \,]\!]\gamma(\sigma,\tau) = \mathcal{E}[\![\, e_2 \,]\!]\gamma(\sigma,\tau) \\ \mathcal{T}[\![\, \neg\chi \,]\!]\gamma(\sigma,\tau) & \hat{=} & \text{not } \mathcal{T}[\![\, \chi \,]\!]\gamma(\sigma,\tau) \\ \mathcal{T}[\![\, \chi_1 \to \chi_2 \,]\!]\gamma(\sigma,\tau) & \hat{=} & \mathcal{T}[\![\, \chi_1 \,]\!]\gamma(\sigma,\tau) \Rightarrow \mathcal{T}[\![\, \chi_2 \,]\!]\gamma(\sigma,\tau) \\ \mathcal{T}[\![\, \exists_g(\chi) \,]\!]\gamma(\sigma,\tau) & \hat{=} & \begin{cases} tt & , \text{ there exists a } v \in \mathcal{V}al \text{ such that} \\ & \quad \mathcal{T}[\![\, \chi \,]\!]\gamma[v/g](\sigma,\tau) \\ ff & , \text{ otherwise .} \end{cases} \end{array}$$

Often assertions are interpreted as sets of pairs of states.

$$[\![.]\!] : \mathcal{A}ssn \to (\Gamma \to \mathcal{P}(\Sigma \times \Sigma))$$
$$[\![\chi]\!]\gamma \hat{=} \{(\sigma,\tau) \mid \mathcal{T}[\![\chi]\!]\gamma(\sigma,\tau)\}$$

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

$\varphi, \psi \in$	$\mathcal{A}ssn_H$,	assertions in which 'hooked' variables do not occur, to be used in Hoare-style correctness formulae.
$\rho \in$	$\mathcal{A}ssn_R$,	assertion without free occurrences of logical variables, but with 'hooked' ones, to be used in post relations of VDM-style correctness formulae.
$\pi \in$	$\mathcal{A}ssn_J$,	assertion in $\mathcal{A}ssn_H \cap \mathcal{A}ssn_R$, to be used as preconditions in VDM-style correctness formulae.

For these assertions the following interpretation functions are defined.

$$[\![.]\!]_H : \mathcal{A}ssn_H \to (\Gamma \to \mathcal{P}(\Sigma))$$
$$[\![.]\!]_R : \mathcal{A}ssn_R \to \mathcal{P}(\Sigma \times \Sigma)$$
$$[\![.]\!]_J : \mathcal{A}ssn_J \to \mathcal{P}(\Sigma)$$

$$\begin{aligned}
[\![\varphi]\!]_H\gamma &\hat{=} \bigcap_{\sigma}\{\tau \mid \mathcal{T}[\![\varphi]\!]\gamma(\sigma,\tau)\} \\
[\![\rho]\!]_R &\hat{=} \bigcap_{\gamma}\{(\sigma,\tau) \mid \mathcal{T}[\![\rho]\!]\gamma(\sigma,\tau)\} \\
[\![\pi]\!]_J &\hat{=} \bigcap_{\gamma,\sigma}\{\tau \mid \mathcal{T}[\![\pi]\!]\gamma(\sigma,\tau)\}
\end{aligned}$$

Let R denote a relation — R possibly stands for a program $S \in \mathcal{L}an$ of some programming language with a relational semantics $\mathcal{R}[\![.]\!] : \mathcal{L}an \to \mathcal{P}(\Sigma \times \Sigma)$, but not necessarily — then the syntactic class $\mathcal{F}orm$ of correctness formulae is defined by

$$f ::= (\varphi)\ R\ (\psi)_H \mid (\pi)\ R\ (\rho)_J \mid \forall_g(f) \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid f_1 \to f_2$$

We use $\mathcal{F}orm_H$ for the subclass of correctness formulae that are obtained by deleting $(\pi)\ R\ (\rho)_J$ from the above definition, and $\mathcal{F}orm_J$ for the subclass which is obtained by deleting $(\varphi)\ R\ (\psi)_H$ and $\forall_g(f)$ from the above definition. The basic formulae in $\mathcal{F}orm_H$ and in $\mathcal{F}orm_J$ are respectively Hoare correctness formulae and VDM correctness formulae.

The semantics of correctness formulae is defined by the truth-valued function $\mathcal{F}[\![.]\!] : \mathcal{F}orm \to (\Gamma \to \{tt, ff\})$:

$$\begin{aligned}
\mathcal{F}[\![(\varphi)\ R\ (\psi)_H]\!]\gamma &\hat{=} [\![R]\!]([\![\varphi]\!]_H\gamma) \subseteq [\![\psi]\!]_H\gamma \\
\mathcal{F}[\![(\pi)\ R\ (\rho)_J]\!]\gamma &\hat{=} [\![\pi]\!]_J ; [\![R]\!] \subseteq [\![\rho]\!]_R \\
\mathcal{F}[\![\forall_g(f)]\!]\gamma &\hat{=} \begin{cases} tt & , \text{ for all } v \in \mathcal{V}al\ \ \mathcal{F}[\![f]\!]\gamma[v/g] \\ ff & , \text{ otherwise .} \end{cases} \\
\mathcal{F}[\![f_1 \vee f_2]\!]\gamma &\hat{=} \mathcal{F}[\![f_1]\!]\gamma \text{ or } \mathcal{F}[\![f_2]\!]\gamma \\
\mathcal{F}[\![f_1 \wedge f_2]\!]\gamma &\hat{=} \mathcal{F}[\![f_1]\!]\gamma \text{ and } \mathcal{F}[\![f_2]\!]\gamma \\
\mathcal{F}[\![f_1 \to f_2]\!]\gamma &\hat{=} \mathcal{F}[\![f_1]\!]\gamma \Rightarrow \mathcal{F}[\![f_2]\!]\gamma
\end{aligned}$$

Note that this definition deviates from the standard convention, as e.g. in [Apt81], in that logical variables are not universally quantified in $(\varphi)\ R\ (\psi)_H$, and therefore the truth of this formula is relative to a logical state. As result reasoning over correctness formulae becomes more natural (see [Zwiers89], page 125).

Observe also that VDM correctness formulae are interpreted as partial correctness formulae, which is unlike the total correctness interpretation in e.g. [VDM89]. It is not until we deal with total correctness that we capture the complete total correctness meaning of VDM correctness formulae.

Now, consider a Hoare-style specification $\{\varphi\}\ name\ \{\psi\}_H$. There are two important differences with a correctness formula $\forall_g((\varphi)\ S\ (\psi)_H)$. The first one is that *name* is just a name for the program that is specified, whereas S is a program. As a consequence, the meaning of the correctness formula depends on S, but the meaning of the specification does not depend on *name*. The second difference is that a correctness formula is either true or false, but a specification defines the set of programs that satisfy that specification.

There is of course also a strong relationship between correctness formulae and specifications that becomes clear if we define what it means if a program satisfies a given specification. Intuitively a program S satisfies the specification $\{\varphi\}\ name\ \{\psi\}_H$, if the corresponding correctness formula $\forall_g((\varphi)\ S\ (\psi)_H)$ is true. Thus one would like to have a definition of satisfaction such that

$$R \textbf{ sat } \{\varphi\}\ name\ \{\psi\}_H \Leftrightarrow \forall_g((\varphi)\ R\ (\psi)_H)\ .$$

Following a similar argument as for Hoare-style specifications, we find the corresponding requirement for the definition for VDM-style (partial correctness) specifications.

$$R \textbf{ sat } \{\pi\}\ name\ \{\rho\}_J \Leftrightarrow (\pi)\ R\ (\rho)_J\ .$$

The following lemma about correctness gives a sound basis for the definition of specifications and satisfaction.

Lemma 2.2.1

$$\forall_g((\varphi)\ R\ (\psi)_H) \quad \text{if, and only if,} \quad [\![R]\!] \subseteq \bigcap_\gamma([\![\varphi]\!]_H\gamma \rightsquigarrow [\![\psi]\!]_H\gamma)$$

$$(\pi)\ R\ (\rho)_J \quad \text{if, and only if,} \quad [\![R]\!] \subseteq [\![\pi]\!]_J \rightsquigarrow [\![\rho]\!]_R$$

□

Lemma 2.2.1 defines the maximal relation, that satisfies a correctness formula of either type. Because the maximal relation which satisfies a specification characterizes the set of all relations which satisfy that specification, we define the semantics of a specification accordingly.

$$[\![\{\varphi\}\ name\ \{\psi\}_H]\!] \ \hat{=}\ \bigcap_\gamma([\![\varphi]\!]_H\gamma \rightsquigarrow [\![\psi]\!]_H\gamma)$$

$$[\![\{\pi\}\ name\ \{\rho\}_J]\!] \ \hat{=}\ [\![\pi]\!]_J \rightsquigarrow [\![\rho]\!]_R$$

If specifications denote relations, satisfaction becomes simply inclusion between relations, and can be defined for arbitrary relations.

Definition 2.2.1

R_0 **sat** R_1 if, and only if, $R_0 \subseteq R_1$

□

2.3 Simulation

It is good programming practice to define a representation invariant when replacing an abstract data type by a more concrete one, c.f. [Hoare72]. A representation invariant is a predicate that defines how the abstract data type is related to the concrete one. It typically is conjunction of a characteristic predicate of an abstraction relation and a data invariant of the concrete data type. The abstraction relation relates the abstract variables with the concrete variables and the data invariant defines the allowed states of the concrete data type.

As we already pointed out in the introduction, we say that a concrete program refines an abstract one if no program using the concrete one can detect it is not using the abstract program. This notion of refinement is also used in e.g. [Hoare87]. Let $P(A)$ denote a program P using an abstract data type A and its operations. And, likewise, let $P(C)$ denote the same program P, with the abstract data type and its operations replaced by the concrete data type C and corresponding operations. Let AI and AF be the initialization and the finalization statements of the abstract data type, and CI and CF be the initialization and finalization of the concrete data type. Then, C refines A if, and only if,

$$[\![CI;\ P(C);\ CF]\!] \subseteq [\![AI;\ P(A);\ AF]\!]$$

for all programs P, cf. [Hoare87].

Proving a reification step correct can be done by showing that there exists a simulation relation. For data reification, the simulation relation is the abstraction relation, which is defined by the representation invariant.

There are different kinds of simulation that are used for reification.

Definition 2.3.1
Let $S^A \subseteq \Sigma^A \times \Sigma^A$ be an abstract relation (program) and $S^C \subseteq \Sigma^C \times \Sigma^C$ be a concrete relation, and $\alpha \subseteq \Sigma^C \times \Sigma^A$ an abstraction relation. Then

S^C L-simulates S^A w.r.t. α	if, and only if,	$\alpha^{-1};\ S^C \subseteq S^A;\ \alpha^{-1}$
S^C L^{-1}-simulates S^A w.r.t. α	if, and only if,	$S^C;\ \alpha \subseteq \alpha;\ S^A$
S^C U-simulates S^A w.r.t. α	if, and only if,	$\alpha^{-1};\ S^C;\ \alpha \subseteq S^A$
S^C U^{-1}-simulates S^A w.r.t. α	if, and only if,	$S^C \subseteq \alpha;\ S^A;\ \alpha^{-1}$

□

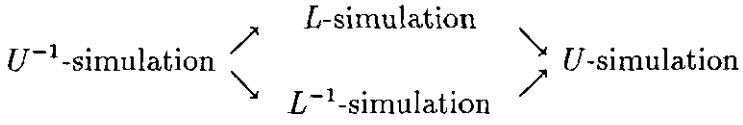
L-simulation and L^{-1}-simulation are respectively called downward simulation and

upward simulation in [Hoare87]. U-simulation is used in [Back88], and U^{-1}-simulation is used in [Lamport83] for concurrent systems.

From the definition above it follows that any two relations simulate each other, because one can choose the empty relation for α. Therefore we demand that the abstraction relation is total on the concrete state space defined by the data invariant of the concrete data type. The requirement that an abstraction relation must be total causes no problems, because one can always choose a sufficiently strong data invariant at the implementation level.

Although the different versions of simulation are in general incomparable — e.g. some refinements can be proven by L^{-1}-simulation but not by L-simulation cf. [Gardiner90], and vice versa — they can be compared if one requires the abstraction relation to be total or functional.

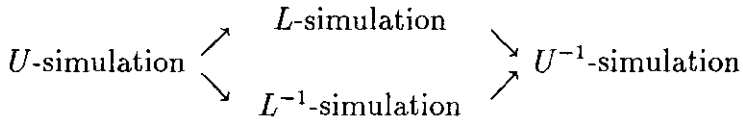
Lemma 2.3.1
If, and only if, α is functional then



□

This diagram should be read as follows. E.g. if S^C U^{-1}-simulates S^A then S^C also L-simulates S^A.

Lemma 2.3.2
If, and only if, α is total then



□

In [Hoare87] it was proven that L- and L^{-1}-simulation together are sufficient to prove refinement (see also [He89]). This implies that U^{-1}-simulation is sufficient to prove refinement if only total abstraction relations are used, and U-simulation is sufficient if only abstraction functions are used. However, requiring that the abstraction relation must be functional causes a problem if one allows implementation bias, a notion which can be very useful in practical situations.

A specification is implementation biased if it contains more information than strictly necessary to specify the desired operation. Although, implementation bias can always be avoided it may be used to give a specification that is easier to understand. There is also a practical reason for permitting implementation bias. Suppose you have designed a large and complex system. However, you 're quite unhappy with the inefficiency of some of the operations. Because, you don't want to redesign

the complex algorithms used for these operations, you might decide to apply some optimization techniques to the operations instead. This means you consider the implementations of the operations as specifications, thereby introducing implementation bias.

It is tempting to conclude from the above that U^{-1}-simulation is the simulation one should use to prove refinement. This is true if one considers refinement of a single abstract operation, but when refining a large program this is no longer true. Suppose that S_1^C simulates S_1^A and S_2^C simulates S_2^A, and P is a program with statements S_1^A and S_2^A. Then P with S_1^A and S_2^A replaced by S_1^C and S_2^C should also simulate P. This property is called subdistributivity in [Hoare87], where L- and L^{-1}-simulation where shown to be subdistributive. Unfortunately, U- and U^{-1}-simulation are in general not subdistributive, and can therefore not be used to prove refinement of a complete program by proving the refinement of the operations in isolation. The problem is sequential composition. In lemma 2.3.3, sufficient and necessary conditions are given under which these simulations are subdistributive.

Lemma 2.3.3
If S_1^C U-simulates S_1^A and S_2^C U-simulates S_2^A, then

$S_1^C;\ S_2^C$ U-simulates $S_1^A;\ S_2^A \Leftrightarrow \alpha$ is total

If S_1^C U^{-1}-simulates S_1^A and S_2^C U^{-1}-simulates S_2^A, then

$S_1^C;\ S_2^C$ U^{-1}-simulates $S_1^A;\ S_2^A \Leftrightarrow \alpha$ is functional

□

Thus the abstraction relation must be total and functional, if we insist that U- or U^{-1}-simulation is subdistributive and sufficient to prove refinement in the sense of [Hoare87]. A similar problem arises if one uses L-simulation and L^{-1}-simulation for proving reification of different parts of the same program. If S_1^C L-simulates S_1^A and S_2^C L^{-1}-simulates S_2^A, then it is not guaranteed that $S_1^C;\ S_2^C$ L- or L^{-1}-simulates $S_1^A;\ S_2^A$.

2.4 Proving reification

We want to answer the following question. Suppose we are provided with a specification $\{\pi\}\ op^A\ \{\psi\}$ of an abstract operation op^A. How should we specify the concrete operation op^C such that every program that satisfies the specification of op^C is a correct implementation op^A w.r.t. a given abstraction relation α? This question is answered in theorem 2.4.1, for each of the simulation notions discussed in section 2.3.

We include $\langle\alpha\rangle\psi$ and $[\alpha]\psi$ in the assertion language $\mathcal{A}ssn_H$ as abbreviations of respectively $\exists_a(\chi_\alpha \wedge \psi)$ and $\forall_a(\chi_\alpha \rightarrow \psi)$, where χ_α ($\in \mathcal{A}ssn_J$) denotes the characteristic predicate of α and a is the list of all free abstract variables in this predicate.

Theorem 2.4.1

- *(U-simulation).*

$$\begin{array}{ll} & \alpha^{-1};\ S;\ \alpha\ \mathbf{sat}\ \{\varphi\}\ op^A\ \{\psi\}_H \\ \Leftrightarrow & S\ \mathbf{sat}\ \{\langle\alpha\rangle\varphi\}\ op^C\ \{[\alpha]\psi\}_H \end{array}$$

- *(U^{-1}-simulation).* If, and only if, α is total then

$$\begin{array}{ll} & S\ \mathbf{sat}\ \alpha;\ (\{\varphi\}\ op^A\ \{\psi\}_H);\ \alpha^{-1} \\ \Leftrightarrow & S\ \mathbf{sat}\ \{[\alpha]\varphi\}\ op^C\ \{\langle\alpha\rangle\psi\}_H \end{array}$$

- *(L-simulation).* If, and only if, α is total then

$$\begin{array}{ll} & \alpha^{-1};\ S\ \mathbf{sat}\ (\{\varphi\}\ op^A\ \{\psi\}_H);\ \alpha^{-1} \\ \Leftrightarrow & S\ \mathbf{sat}\ \{\langle\alpha\rangle\varphi\}\ op^C\ \{\langle\alpha\rangle\psi\}_H \end{array}$$

- *(L^{-1}-simulation).* If, and only if, α is total then

$$\begin{array}{ll} & S;\ \alpha\ \mathbf{sat}\ \alpha;\ (\{\varphi\}\ op^A\ \{\psi\}_H) \\ \Leftrightarrow & S\ \mathbf{sat}\ \{[\alpha]\varphi\}\ op^C\ \{[\alpha]\psi\}_H \end{array}$$

□

Suppose we want to prove that $S_0;\ S_1$ is an implementation of $\{\varphi_0\}\ op^A\ \{\varphi_2\}_H$ under abstraction relation α. If we use L-simulation it suffices to prove [1] $(\langle\alpha\rangle\varphi_0)\ S_0\ (\langle\alpha\rangle\varphi_1)_H$ and $(\langle\alpha\rangle\varphi_1)\ S_1\ (\langle\alpha\rangle\varphi_2)_H$ for some assertion φ_1. If we use U-simulation we have to find an assertion φ_1 such that $(\langle\alpha\rangle\varphi_0)\ S_0\ ([\alpha]\varphi_1)_H$ and $(\langle\alpha\rangle\varphi_1)\ S_1\ ([\alpha]\varphi_2)_H$ *and* $[\alpha]\varphi_1 \rightarrow \langle\alpha\rangle\varphi_1$. This extra proof obligation is needed because U-simulation is not subdistributive. In case that α is total, and hence U-simulation is subdistributive, $[\alpha]\varphi_1 \rightarrow \langle\alpha\rangle\varphi_1$ is always true. Thus subdistributivity guarantees nice proof rules, but it is not always necessary to require subdistributivity. In a correctness proof one is interested in a particular assertion φ for which $[\alpha]\varphi \rightarrow \langle\alpha\rangle\varphi$ (or $\langle\alpha\rangle\varphi \rightarrow [\alpha]\varphi$) may very well be true.

3 Partial correctness preserving transformations

We will show how the verification conditions for reification in Reynolds' method and in VDM can be derived within the theory of section two.

[1] This is an example where one explicitly reasons over correctness formulae within a given logical state and not about implicitly quantified formulae.

3.1 Reynolds' reification method

Each reification step in Reynolds' method consists of four smaller steps. First, the concrete variables and the representation invariant are introduced. Second, each assignment that affects the representation invariant is augmented with one or more assignments to the concrete variables such that the representation invariant is reestablished (or achieved in case of initialization). Third, expressions that contain abstract variables, but occur outside of an assignment to abstract variables, are replaced by expressions that do not contain abstract variables but are guaranteed by the representation invariant to have a value that could have been the result of the abstract expression. Fourth, the declarations of and the assignments to abstract variables that have become auxiliary by the previous step are eliminated. The second and the third step are the most interesting ones, because they implicitly describe what correctness of a reification step in Reynolds' method means.

According to step two, a simultaneous assignment $a := e^A(a, x)$ is augmented with assignments to concrete variables to reestablish the representation invariant. The assignments to the concrete variables can be contracted into a single simultaneous assignment $c := e^C(c, x)$. Thus, one has to prove that

$$(\chi_\alpha)\ a := e^A(a, x);\ c := e^C(c, x)\ (\chi_\alpha)_H$$

holds, where χ_α denotes the representation invariant. This boils down to proving $\chi_\alpha \rightarrow \chi_\alpha[e^C(c, x)/c][e^A(a, x)/a]$, or because the abstract and concrete variables are disjoint

$$\chi_\alpha \rightarrow \chi_\alpha[e^C(c, x)/c, e^A(a, x)/a]\ .$$

Observe that in this verification condition nothing is said about the pre- and post-condition of the abstract assignment itself. In this respect there is a difference between Reynolds' method and Back's method on one hand and ours and VDM on the other. In VDM operations are characterized by their specifications, whereas Reynolds characterizes them through their semantics. Thus, Reynolds interprets the meaning of an abstract program and this is in principle independent of the proof outline of that program (of course validity of a proof outline does depend on the semantics of the abstract operations).

If we specify the abstract assignment by $\{a = g\}\ op^A\ \{a = e^A(g, x)\}_H$ — with the restriction that assignment to the variables in x is not allowed — we can use the rule for L-simulation to obtain $\{\exists_a(\chi_\alpha \wedge a = g)\}\ op^C\ \{\exists_a(\chi_\alpha \wedge a = e^A(g, x))\}_H$. For a concrete assignment $c := e^C(c, x)$ this means we have to prove

$$\exists_a(\chi_\alpha \wedge a = g) \rightarrow \exists_a(\chi_\alpha[e^C(c, x)/c] \wedge a = e^A(g, x))\ .$$

In Reynolds' method the abstract operation $a := e^A(a, x)$ always terminates, and therefore we have to prove that $\exists_a(\chi_\alpha \wedge a = g) \rightarrow \chi_\alpha[e^C(c, x)/c, e^A(g, x)/a]$, which is equivalent to the verification condition above.

Next, we turn to step three of Reynolds' method. In case that an abstract variable occurs in an expression outside an assignment to an abstract variable, we

concentrate on nondeterministic assignments of the form $x := x'.\pi^A(a, x')$. The intended meaning of this statement is that the variable x is assigned a value such that $\pi^A(a, x)$ is true if possible, otherwise it does not terminate. According to step three, proving that $x := x'.\pi^C(c, x')$ is a correct implementation, can be done by verifying that

$$\chi_\alpha \rightarrow (\pi^C(c, x) \rightarrow \exists_a(\chi_\alpha \wedge \pi^A(a, x))) .$$

This verification condition can be deduced within our framework. The abstract operation satisfies $\{\text{true}\}\ op^A\ \{\pi^A(a, x)\}_H$, with the restriction that only assignment to x is allowed. The rule for L-simulation requires a proof of

$$(\exists_a(\chi_\alpha))\ x := x'.\pi^C(c, x')\ (\exists_a(\chi_\alpha \wedge \pi^A(a, x)))_H .$$

Using the axiom for nondeterministic assignment this generates the following verification condition

$$\exists_a(\chi_\alpha) \rightarrow \forall_g(\pi^C(c, g) \rightarrow \exists_a(\chi_\alpha \wedge \pi^A(a, g))) .$$

And this is equivalent with Reynolds' verification condition. The replacement of abstract variables by concrete variables in boolean expressions is completely analogous.

From the considerations in the previous paragraphs one may conclude that Reynolds uses L-simulation to prove reification, and therefore the method of Reynolds is incomplete.

3.2 Reification in VDM: partial correctness

To investigate the relationship between reification in VDM and reification in the other methods we use a translation from VDM specifications to Hoare specifications and a translation the other way around. Of course, the same translations can be applied to correctness formulae.

A VDM specification $\{\pi\}\ op^A\ \{\rho\}_J$ denotes the same relation as the Hoare specification $\{\pi \wedge x = g\}\ op^A\ \{\rho[g/\overline{x}]\}_H$, where $x = g$ is a freeze predicate that "freezes" the values of all hooked program variables in ρ.

Thus, if logical variables are used then primed variables become superfluous. The opposite is also true. If primed variables are used one does not need logical variables, because a Hoare specification $\{\varphi\}\ op^A\ \{\psi\}_H$ denotes the same relation as $\{\text{true}\}\ op^A\ \{\forall_g(\varphi[\overline{x}\ /x] \rightarrow \psi)\}_J$, where g is the list of free logical variables in φ and ψ. In [VDM89], Jones gives two proof rules for reification. One that is used if the abstraction relation is functional, i.e. it is a retrieve function. And, secondly, a proof rule that is used in case that the abstraction relation is not functional, i.e. it is a retrieve relation. In case of a retrieve function, Jones uses U-simulation as a basis for his proof rule, and therefore has a complete proof rule. However the retrieve functions used in [VDM89] are also total and adequate — i.e. for each instance of the abstract datatype there is an instance of the concrete datatype that represents

it and vice versa — so that the simulations in section two are equal. For this reason we move directly to the more interesting case of nonfunctional abstraction relations. It is already known, see e.g. [Gardiner90] that the proof rule for retrieve relations expresses L-simulation, but we give a proof of this fact within the theory of section two.

To prove in VDM that $\{\pi^C\}\ op^C\ \{\rho^C\}_J$ specifies a correct implementation of $\{\pi^A\}\ op^A\ \{\rho^A\}_J$ under α, one has to verify that ([VDM89], page 222)

$$\begin{array}{ll} & \chi_\alpha \wedge \pi^A \rightarrow \pi^C \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{, } \textit{domain rule;} \\ \text{and} & \chi_\alpha[\overleftarrow{a}\ /a, \overleftarrow{c}\ /c] \wedge \pi^A[\overleftarrow{a}\ /a] \wedge \rho^C \rightarrow \exists_a(\rho^A \wedge \chi_\alpha) \quad \text{, } \textit{result rule.} \end{array}$$

The above conditions are satisfied if the concrete operation satisfies

$$\{\chi_\alpha \wedge \pi^A\}\ op^C\ \{\exists_a(\rho^A \wedge \chi_\alpha)\}_J\ .$$

This is also the weakest specification that guarantees that the domain rule and result rule are satisfied. This can be explained as follows. From the domain rule it follows that $\chi_\alpha \wedge \pi^A$ is the strongest precondition that is allowed in the specification of op^C. Because,

$$[\![\, \chi_\alpha[\overleftarrow{a}\ /a, \overleftarrow{c}\ /c] \wedge \pi^A[\overleftarrow{a}\ /a] \rightarrow \exists_a(\rho^A \wedge \chi_\alpha)\,]\!]_R = [\![\, \{\chi_\alpha \wedge \pi^A\}\ op\ \{\exists_a(\rho^A \wedge \chi_\alpha)\}_J\,]\!]$$

it follows that $\exists_a(\rho^A \wedge \chi_\alpha)$ is also the weakest postcondition that is admissible in the specification of the concrete operation.

The specification above can be translated in the Hoare specification

$$\{\chi_\alpha \wedge \pi^A \wedge x = g\}\ op^C\ \{\exists_a(\rho^A[g/\ \overleftarrow{a}] \wedge \chi_\alpha)\}_H\ .$$

Note that the abstract variables a do not occur free in the postcondition, thus we can apply the adaptation rule (see e.g. [Olderog83]) to obtain

$$\{\exists_a(\chi_\alpha \wedge \pi^A \wedge x = g)\}\ op^C\ \{\exists_a(\rho^A[g/\ \overleftarrow{a}] \wedge \chi_\alpha)\}_H\ .$$

To obtain the previous specification again, just apply the consequence rule. And this is exactly what we would have obtained by first translating the abstract specification and then applying the rule for L-simulation. So, also the VDM verification conditions are deducible within our formalism.

4 Total correctness preserving transformations

Although partial correctness preserving refinement is interesting and can be used to illustrate many ideas about refinement, it has the disadvantage that every abstract operation may be implemented by a nonterminating program at the concrete level. Therefore, we will investigate in this section how our theory might be extended to total correctness preserving reification.

A straightforward way to adapt the theory in section two to total correctness, is to introduce a special state $\perp$ to denote divergence, use a strict interpretation —

evaluation of a predicate in $\perp$ yields false — and require that relations are total. To distinguish the correctness formula with the new interpretation from the old ones we add the superscript $\perp$. Thus, the *total* correctness formula $(\varphi)\ S\ (\psi)^{\perp}_{H}$ is only true if for each initial state that satisfies φ the program S will terminate in a state satisfying ψ. Although this is the approach taken in VDM and by Back, we also want to mention an alternative approach.

The approach sketched in the previous paragraph is in a sense an all-or-nothing approach because one cannot specify that a program may not terminate for certain initial states. However, for concurrent systems it is sometimes desirable to specify that a program does not terminate for some initial values [Zwiers90a]. Therefore one may, again, introduce $\perp$ and a special predicate **fin** which is only false in $\perp$. The total correctness formula above and the partial correctness formulae of section two are respectively obtained as the special cases $(\varphi)\ S\ (\mathbf{fin} \wedge \psi)_H$ and $(\varphi)\ S\ (\mathbf{fin} \rightarrow \psi)_H$. It is also possible to consider correctness formulae with a meaning different from both partial and total correctness. For example, the formula $(x = g)\ S\ (g \neq 0 \leftrightarrow \mathbf{fin})_H$ is true if S terminates only if x is initially not zero.

In the remainder of this section we investigate how total correctness preserving refinement is achieved in VDM and Back's refinement calculus, and how this could be applied to the theory of section two.

4.1 Reification in VDM: total correctness

We already discussed partial correctness preserving reification in VDM, so we can be rather brief about total correctness. In VDM the preservation of total correctness in a reification step is guaranteed by the restriction that the implementation of an abstract data type is *adequate*, which means that for each state that satisfies the data invariant at the abstract level there is a state at the concrete level that represents it. Together with the domain rule this is sufficient to ensure that if the precondition of the abstract operation is true for a particular state, then there exists a corresponding state at the concrete level such that the precondition of the concrete operation is true. Because specifications in VDM specify programs that must terminate for states that satisfy the precondition, the concrete program must terminate also. The result rule ensures that the program terminates in a correct state.

Thus to adapt the theory of section two to total correctness, it is sufficient to interpret the correctness formulae as total correctness formulae and add the requirement that abstraction relations are adequate.

4.2 Reification in Back's refinement calculus

Before we proceed with data reification in Back's calculus, we must admit that it is not possible to cover Back's calculus completely with the theory of section two. The reason for this is that Back allows angelic conditional statements of the form $S_1 \Diamond S_2$ with a weakest precondition semantics such that $[S_1 \Diamond S_2]p \mathrel{\hat{=}} [S_1]p \vee [S_2]p$. This statement doesn't have a relational semantics, and is therefore not covered by

the theory of section two. However, in data refinement angelic nondeterminism is used in a restricted way, which can be handled in a relational framework, indeed.

In the refinement calculus S implements S' if, and only if, $[S']\pi \rightarrow [S]\pi$ for all π ($\in \mathcal{A}ssn_J$). Likewise, S data refines S' w.r.t. abstraction relation α if, and only if [2], $[S']\pi \rightarrow [\alpha^{-1}; S; \bar{\alpha}]\pi$. The statement $\bar{\alpha}$ is a angelic nondeterministic statement whose predicate transformer semantics is defined by $[\bar{\alpha}]\pi \hat{=} \exists_a(\chi_\alpha \wedge \pi)$, cf. [Back90].

If the abstract statement is specified by $\{\varphi\}\ op^A\ \{\psi\}^{\perp}_H$ an application of the characterization theorem [Back90] results in the following proof obligation for the implementation S^C

$$(\exists_a(\chi_\alpha) \wedge \forall_a(\chi_\alpha \rightarrow \varphi))\ S^C\ (\exists_a(\chi_\alpha \wedge \psi))^{\perp}_H\ .$$

To extend the theory of section two to total correctness in the same way as Back, this means that we only have to replace $[\alpha]\varphi$ by $\langle\alpha\rangle\text{true} \wedge [\alpha]\varphi$ in the proof rules of theorem 2.4.1.

5 Conclusions

Three well-known methodologies for proving data refinement for sequential programs, due to Reynolds [Reynolds81], Jones [VDM89], and Back [Back88] have been presented up to now separately in the literature without mentioning the underlying principles that relate them. We have investigated how the afore mentioned methodologies are related by

1. Developing a modest predicate transformer framework.

2. Relating four known varieties for proving refinement and express them as verification conditions within this framework.

3. Analyzing Reynolds' method and VDM-style refinement proofs, and stating their associated verification conditions for partial correctness, and then, through an extension of our framework to include nontermination, to total correctness.

4. Mentioning how a restricted form [Back88] of Back's general theory can also be characterized within our formalism.

Although we considered data reification in sequential systems, most issues are also relevant for concurrent systems. Back has applied the same techniques used in data reification to refinement of action systems [Back90a]. Also in [Zwiers90] data refinement was used for the implementation of a queue by a dynamic network of processes. Therefore it is worthwhile to have an overview of the main methods and issues in data refinement as a basis for the refinement of distributed systems.

[2] To avoid confusion, readers familiar with Back's notation should be aware that his abstraction relations are defined on $\Sigma^A \times \Sigma^C$, whereas we have defined them on $\Sigma^C \times \Sigma^A$.

Future work, includes the generalization of the results for concurrency possibly using a framework similar to the one in [Zwiers90a]. This would enable us to integrate e.g. Hehner's work on concurrency [Hehner84b].

We want to emphasize that many results, especially concerning predicate transformers, are but variations upon work reported in existing literature, and mention that we especially learned a lot from Gardiner and Morgan [Gardiner90], although our framework was developed independently from theirs. Our sincere thanks go to John Reynolds, Cliff Jones, and Ralph Back for their fine work in this area, whose study continues to be a source of enjoyment and inspiration for us.

References

[Abadi88] M. Abadi & L. Lamport. *The Existence of Refinement Mappings.* Proceedings of the 3rd IEEE Conference on Logic in Computer Science (LICS), pp. 165–175, 1988.

[Apt81] K.R. Apt. *Ten Years of Hoare's Logic: A Survey—Part I.* ACM Transactions on Programming Languages and Systems 4:431–483, 1981.

[Back78] R.J.R. Back. *On the Correctness of Refinement Steps in Program Development.* Report A-1978-4, Dept. of Computer Science, University of Helsinki, 1978.

[Back88] R.J.R. Back. *Data Refinement in the Refinement Calculus.* Reports on computer science & mathematics 68, Åbo Akademi, 1988.

[Back90] R.J.R. Back & J. von Wright. *Refinement Calculus, Part I: Sequential Nondeterministic Programs.* In Stepwise Refinement of Distributed Systems, LNCS 430, pp. 42–66. Springer-Verlag, 1990.

[Back90a] R.J.R. Back. *Refinement Calculus, Part II: Parallel and Reactive Programs.* In Stepwise Refinement of Distributed Systems, LNCS 430, pp. 67–93. Springer-Verlag, 1990.

[deBdeRRoz90] J.W. de Bakker, G. Rozenberg & W.-P. de Roever. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness.* Proceedings of the NFI/REX Workshop, LNCS 430. Springer-Verlag 1990.

[vanDiepen86] N.W.P. van Diepen & W.-P. de Roever. *Program Derivation through Transformations: The Evolution of List-Copying Algorithms.* Science of Computer Programming 6:213–272, 1986.

[Gardiner90] P. Gardiner & C. Morgan. A Single Complete Rule for Data Refinement, submitted to ACM Transactions on Programming Languages and Systems.

[Gerhart75] S.L. Gerhart. *Correctness Preserving Program Transformations.* Proceedings 2nd Symposium on Principles of Programming Languages, pp. 54–66, 1975.

[Gries81] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[He89] He Jifeng. *Process Simulation and Refinement.* Formal Aspects of Computing 1:229–241, 1989.

[Hehner84] E.C.R. Hehner. *The Logic of Programming.* Prentice-Hall, 1984.

[Hehner84a] E.C.R. Hehner. *Predicative Programming Part I.* Communications of the ACM 27:134–143, 1984.

[Hehner84b] E.C.R. Hehner. *Predicative Programming Part II.* Communications of the ACM 27:144–151, 1984.

[Hoare72] C.A.R. Hoare. *Proofs of Correctness of Data Representation.* Acta Informatica 1:271–281, 1972.

[Hoare87] C.A.R. Hoare, He Jifeng & J.W. Sanders. *Prespecification in Data Refinement.* Information Processing Letters 25:71–76, 1987.

[Jones80] C.B. Jones. *Software Development: a Rigorous Approach.* Prentice-Hall, 1980.

[Lamport83] L. Lamport. *Specifying Concurrent Program Modules.* ACM Transactions on Programming Languages and Systems 2:190–220, 1983.

[Lee79] S. Lee, W.-P. de Roever & S.L. Gerhart. *The Evolution of List-Copying Algorithms and The Need for Structured Program Verification.* Conf. Rec. 6th Ann. ACM Symp. on Principles of Progr. Languages, pp. 53–67, 1979.

[Milner71] R. Milner. *An Algebraic Definition of Simulation between Programs.* Proceedings of 2nd Int. Joint Conf. on Artificial Intelligence, pp. 481–489, 1971.

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

[Morris89] J.M. Morris. *Laws of data refinement.* Acta Informatica 26:287–308, 1989.

[Olderog83] E.-R. Olderog. *On the Notion of Expressiveness and the Rule of Adaptation.* Theoretical Computer Science 24:337–347, 1983.

[Reynolds81] J.C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.

[VDM86] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1986.

[VDM89] C.B. Jones. *Systematic Software Development using VDM*, 2nd edition. Prentice-Hall, 1989.

[Zwiers89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes, and Their Relationship.* LNCS 321, Springer-Verlag, 1989.

[Zwiers90] J. Zwiers. *Predicates, Predicate Transformers and Refinement.* In Stepwise Refinement of Distributed Systems, LNCS 430, pp. 759–776. Springer-Verlag, 1990.

[Zwiers90a] J. Zwiers & W.-P. de Roever. *Predicates are Predicate Transformers: A Unified Compositional Theory for Concurrency.* To appear in the proc. of Principles of Distributed Computing '89.

In this series appeared:

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

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

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

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