A note on compositional refinement

Eindhoven University of Technology

Department of Mathematics and Computing Science

A Note on Compositional Refinement

by

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

92/01

Computing Science Note 92/01
Eindhoven, January 1992

COMPUTING SCIENCE NOTES

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

A Note on Compositional Refinement [*]

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

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 zu Kiel
D-2300 Kiel 1, Germany

Abstract

Implementing a (concurrent) program P often requires changing the syntactic structure of P at various levels. We argue and illustrate that in such a situation a natural framework for implementation correctness requires a more general notion of refinement than that of [HHS87], a notion which involves the introduction of separate refinement relations for P's various abstract components. An outline is given of a formal framework for proving implementation correctness that involves these notions.

1 Introduction

Transformational program development, in all its fashions, has become one of the main ways to construct sequential, parallel, and distributed systems. During such a development one constructs a sequence of more and more refined systems Q, R, S, etc. According to some suitable notion of implementation, system Q is implemented by its successor R, which is itself implemented by S etc. In its simplest form, program transformation relies on algebraic equalities of the form $S = T$ or on implementation relations of the form $S \sqsubseteq T$ (*S is refined by T*).

An important property of transformations based on inequalities is that it is relatively easy to incorporate *specification* and *verification* based-on program logics.

[*] To appear in *Proc. of the 5th BCS-FACS Refinement Workshop.* Workshops in Computing, Springer-Verlag 1992.

[†] E-mail: zwiers@cs.utwente.nl

[‡] 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

Such specifications have the form $S \sqsupseteq \chi$, often denoted in this context by S **sat** χ, where S is a system as before, but where χ is a formula of some appropriate program logic [Z89]. Within this framework it is natural to apply process operations '*op*' not only to processes S but also to logic specifications φ or combinations of processes and specifications, resulting in so called *mixed terms* [Z89, O91]. This admits a transformational approach where an initial logical specification is transformed gradually into an implementation as a process. During each step in such a development trajectory one replaces one subterm by another subterm. One of the following three situations applies to each replacement of a subterm by another.

- A *specification* φ is replaced by *process* S such that $S \sqsupseteq \varphi$.
- A *process* S is replaced by another *process* T such that $T \sqsupseteq S$.
- A *specification* φ is replaced by another *specification* ψ such that $\psi \sqsupseteq \varphi$, i.e. $\psi \rightarrow \varphi$ should be a logically valid implication.

It is possible to go one step further in this integration of processes and program logics, by introducing a *single, unified* language of terms that can be composed by means of operations originating from process languages, such as parallel or sequential composition as well as by means of *logical* operations from propositional logic and predicate calculus [ZdR89]. So not only can specifications be combined by means of process operations, but also can processes be combined by means of logic operations such as conjunction. Apart from being more uniform this approach has technical advantages such as the possibility to define more complex process operations by means of simpler logical and process operations. An example is the definition of various forms of parallel composition of processes in terms of logical conjunction and a few simple process operations such as relabeling of actions or projection of communication histories [ZdR89]. Another advantage of the integration of processes and logic is the possibility to deal with higher order constructs such as predicate transformers in a natural way, completely inside a single unified framework. As we rely heavily on such techniques because it also turns out to be the proper framework for expressing and proving reification, we introduce such a unified language in the appendix.

The use of the term reification rather than refinement emphasizes the use of simulation techniques to justify the transformation steps of the development process. Such general techniques are based on *simulation* of an 'abstract' high level specification A by a more concrete lower level implementation C. The idea originates from the well known techniques for data reification, where the relation between abstract and concrete is specified by means of so called abstraction functions, also called retrieve functions, mapping concrete data to abstract data. Data reification can be generalized to simulation where a retrieve function maps complete computations, also called 'runs', from the concrete level to the more abstract level. This is based on the assumption that systems S can *semantically* be interpreted as sets of runs of the system. This view is consistent with many models of computation, both for sequential and concurrent systems. Typical examples are the CSP trace, ready set and failure models, where a run coincides with a finite communication history, possibly decorated or augmented with information concerning termination and deadlock behaviour. Other examples are that of the set of, labeled or unlabeled, state sequences associated with state-transition systems, the runs as defined for Petri nets and event structures, and the traces as introduced by Mazurkiewicz. But maybe the simplest example is the classical model for sequential nondeterministic programs as binary relations on states, where each initial/final state pair (s_0, s_1) represents one possible computation. Retrieve functions ρ which are defined on single states, mapping states s_c of the C system to states $s_a = \rho(s_c)$ of the A system, can be extended straightforwardly to abstraction functions ρ on runs, in the form of state sequences $\sigma_C = (s_0, s_1, s_2, \ldots)$, by pointwise applying ρ. Hence,

$$\sigma_A = (\rho(s_0), \rho(s_1), \rho(s_2), \ldots) \; .$$

Similarly, ρ is then extended to an operation on *sets of runs*, by applying ρ pointwise to each run in turn:

$$\rho(C) \triangleq \{\sigma_A \mid \exists_{\sigma_C \in C}(\sigma_A = \rho(\sigma_C))\} \; .$$

Intuitively, system C implements A or as we will say, *refines* A with respect to retrieve function ρ, if for any possible C run there is a corresponding, i.e. ρ related, A run. This is easily expressed by the following requirement.

$$\rho(C) \subseteq A \tag{1}$$

Note that for the special case that ρ is the identity function, this boils down to $C \subseteq A$, that is, we are back at the simpler form of implementation for mixed terms discussed above. In this case we simply say that C *refines* A.

Although it might not be apparent from (1), it can be shown, see [CdRZ91] that the verification conditions for functional data reification in VDM amount to the same as (1). Data refinement as discussed in [R81] and on pp. 221–222 of [J90] is slightly more general than functional reification in that it allows abstraction *relations* α rather than retrieve *functions* ρ between concrete and abstract data. This means that one particular concrete value can represent several abstract values, a desirable property when dealing with implementation bias. [1] When dealing with abstraction relations α the refinement relation between systems can be formulated as follows.

Definition 1.1 *(Strong simulation)*
System C *strongly simulates* system A with respect to relation α if

$$\alpha(C) \subseteq A$$

□

[1] A specification is said to be implementation biased if it includes more implementation detail than strictly necessary (see e.g. [J90]).

A slightly more general notion of simulation is given in the next definition.

Definition 1.2 *(Weak simulation)*
System C *weakly simulates* system A with respect to α if

$$C \subseteq \alpha^{-1}(A)$$

□

Strong simulation requires that for any C run σ_C *all* α related runs σ_A are possible runs for the abstract system A. Weak simulation only requires that there is *at least one* such α related run that is also a possible run for A. Clearly weak simulation is more liberal than strong simulation, because $\alpha(C) \subseteq A \Rightarrow C \subseteq \alpha^{-1}(A)$ if (and only if) α is *total*, but $C \subseteq \alpha^{-1}(A) \Rightarrow \alpha(C) \subseteq A$ if (and only if) α is *functional*. Whereas retrieve *functions* are not adequate when considering implementation bias, requiring totality is not a real limitation. After all, it is only required that the abstraction relation is total with respect to the admissible states of the concrete data type, which can be achieved by strengthening the data invariant part of the representation invariant that characterizes the abstraction relation.

Both strong and weak simulation are defined in terms of abstraction relations on the level of *computations.* As it turns out, the well known notions of upward and downward simulation are not of this form, i.e. cannot be understood in terms of abstraction relations operating on computations. What *is* possible however is to characterize upward and downward simulation of A by C by means of inequalities of the following form:

$$C \subseteq F_\alpha(A) \qquad \textit{(downward simulation)}$$

$$C \subseteq G_\alpha(A) \qquad \textit{(upward simulation)}$$

The operations F_α and G_α transform processes, i.e. they transform sets of computations, and are defined in [HHS87], relying on weakest prespecifications and strongest postspecifications. Within our unified language they can be expressed in terms of relational composition $X \mathbin{⨾} Y$, weakest preconditions $[X]Y$ and the leads-to operator $X \leadsto Y$, as follows:

$$F_\alpha(X) = \alpha^{-1} \leadsto (X \mathbin{⨾} \alpha^{-1}),$$

$$G_\alpha(X) = [\alpha](\alpha \mathbin{⨾} X).$$

Next we note that both weak and strong simulation can be formulated as inequalities of this form. For weak simulation as defined above this is already the case, and the inequality for strong simulation is expressible in our language as

$$C \subseteq \alpha^R(A),$$

where α^R denotes the *right adjoint* of α. We therefore define in general refinement of A by C with respect to F as the inequality

$$C \subseteq F(A).$$

Departing from this definition of refinement we define in this paper a generalization of it to what can be called *compositional refinement.*
Compositional refinement does not treat abstract and concrete programs as monolithic entities but rather takes their decomposition into smaller programs into account. A limited form of compositionality has been defined in [HHS87], where it it is called *subdistributivity.* An operator F as above sub-distributes over some n-ary language operator *op* iff

$$F(op(X_1, \ldots, X_n)) \supseteq op(F(X_1), \ldots, F(X_n)).$$

Subdistributivity guarantees the following for refinement of complete programs of the form $P(A_1, \ldots, A_m)$ that are built up by means of subdistributive operations from some set of basic programs $A_1, \ldots, A_m$: If each of the A_i is refined by C_i with respect to F, then the whole program $P(A_1, \ldots, A_m)$ is refined by $P(C_1, \ldots, C_m)$ with respect to F. Subdistributivity allows refinement of basic abstract program A_i by means of basic concrete program C_i. But it does not allow for refinement of the (parameterized) abstract program $P(X_1, \ldots, X_m)$ to a 'concrete' program $Q(X_1, \ldots, X_n)$. In this paper we give a precise definition of such 'context refinements' and we provide examples thereof.

Related to context refinement is the idea of a *varying abstraction relation.* The basic idea is that different components of a program might be refined with respect to different abstraction relations, one for each component, rather than using a uniform abstraction relation for the whole program. A very simple example of a varying abstraction relation is provided by variable or channel hiding contexts that are used to declare *local* abstract and concrete variables and (CSP style) channels. The general picture here is that we have an abstract program operating on abstract variables a say, and an implementing program C operating on corresponding concrete variables c, where C refines A with respect to abstraction relation α, i.e. $C \subseteq \alpha^{-1}(A)$. The two programs are placed in contexts $H_A(X)$ and $H_C(Y)$ that declare the a or c as local variables and initialize, and possibly even finialize, those variables. For appropriate contexts we then have that from $C \subseteq \alpha^{-1}(A)$ it follows that $H_C(C) \subseteq H_A(A)$. We regard this as context refinement, where $H_A(X)$ is refined by $H_C(Y)$ and where the abstraction relation α for the components X and Y has been replaced by the *identity* relation on the outer level. In general we do not require the identity at the outer level or 'interface level' as refinement relation; a nontrivial choice for refinement at the interface level enables us to formalize so called *interface refinement.* In the paper we treat an example of context refinement for a communication protocol where there is a shift from rather complicated abstraction functions for components to a relatively simple abstraction function for the interface.

Our definition of compositional refinement takes both context refinement and varying abstraction relations into account. It can be formulated as a simple weak homomorphism property. We say that a program (context) $A(X_1, \ldots, X_n)$ is refined by another program (context) $C(Y_1, \ldots, Y_n)$ with respect to F_0 (for the outer level) and $F_1, \ldots, F_n$ (for the components) iff

$$C(F_1(X_1), \ldots, F_n(X_n)) \subseteq F_0(A(X_1, \ldots, X_n)),$$

for all $X_1, \ldots, X_n$.

The outline of the remainder of this report is as follows. In section 2 we first discuss the rôle of compositionality in refinement and its relation with the notion of subdistributivity in [HHS87]. We give an example that illustrates how the refinement relation ρ between the overall abstract program and the concrete program may be *different* from the refinement relations ρ_i between their (concurrent) abstract and concrete components A_i resp. C_i. Secondly, we introduce a refinement notion which generalizes subdistributivity by allowing *context refinements*. This is illustrated by an example based on the self-stabilizing snapshot algorithm of Katz and Perry [KP90]. In the appendix we present a theory that unifies several refinement methods for both sequential and concurrent programs within one framework. The theory is related to calculus of [HH87] and [HHS87]. Furthermore the theory is applied to the examples of section 2.

2 Compositional refinement

An important question for transformational techniques in general is how they combine with a modular style of system development. Transformations should be *vertically* composable as well as *horizontally* composable. Vertical composability or *transitivity* is the property that if a system A can be transformed into a system B which in turn can be transformed into C then the immediate step from A to C is also a legal transformation. This requirement is readily satisfied for most transformation techniques, including simulation where we rely on composability of retrieve functions. Horizontal composability or *compositionality* requires that if a system $S(S_1, \ldots, S_n)$ can be decomposed into parts $S_1, \ldots, S_n$ and a top level part $S(\ldots)$, then implementing the parts yields also an implementation of the whole. To be more precise, let $S = S(X_1, \ldots, X_n)$ be a program term with free variables $X_1, \ldots, X_n$, for which other programs, say $S_1, \ldots, S_n$ can be substituted which we denote formally as $S[S_1/X_1, \ldots, S_n/X_n]$ and more informally as $S(S_1, \ldots, S_n)$. Then, if S_i is implemented by T_i, for $i = 1, \ldots, n$, compositionality requires that $S(S_1, \ldots, S_n)$ is implemented by $S(T_1, \ldots, T_n)$.

For simple algebraic equalities between processes the requirements of vertical and horizontal composability are readily satisfied, because equality is transitive and substitutive:

$$\begin{array}{ll} \text{if } \ Q = R \text{ and } R = S & \text{then } Q = S, \text{ and} \\ \text{if } \ S_i = T_i \text{ for } i = 1, \ldots, n & \text{then } S(S_1, \ldots, S_n) = S(T_1, \ldots, T_n). \end{array}$$

More complex transformational techniques rely on implementation relations in the form of inequalities between processes rather than equalities. For implementation relations of the form $S \subseteq T$ horizontal composability is guaranteed when programs are built up from smaller parts by means of *monotonic* operations. For systems denoting sets of 'runs' — the implementation relation $S \sqsubseteq T$ denotes the set inclusion $T \subseteq S$ — this means that an operation '*op*' satisfies the following property.

$$\text{if } \ T_i \subseteq S_i \text{ for } i = 1, \ldots, n \ \text{ then } op(T_1, \ldots, T_n) \subseteq op(S_1, \ldots, S_n).$$

Vertical composability follows from the transitivity of the subset relation.

Next we consider the composability requirements that were posed above for refinement notions based on simulation. Again, vertical composability causes no problems: if R refines Q with respect to α_1 and S refines R with respect to α_2 than S refines Q with respect to $\alpha_1 \circ \alpha_2$:

$$\begin{array}{ll} \text{if} & R \subseteq \alpha_1^{-1}(Q) \text{ and } S \subseteq \alpha_2^{-1}(R) \\ \text{then} & S \subseteq (\alpha_1 \circ \alpha_2)^{-1}(Q) \end{array}$$

Horizontal composability, however, is not so simple. From $S_i \subseteq \alpha^{-1}(T_i)$ for $i = 1, \ldots, n$ it does in general *not* follow that $S(S_1, \ldots, S_n) \subseteq \alpha^{-1}(S(T_1, \ldots, T_n))$. A notorious counterexample is sequential composition: $\alpha^{-1}(S_i) \subseteq T_i$, $(i = 1, 2)$, does not necessarily guarantee that $\alpha^{-1}(S_1); \alpha^{-1}(S_2) \subseteq T_1; T_2$, unless some restrictions are imposed upon α (c.f. [CdRZ91]). What we need here is a property related to the notion of *subdistributivity* as introduced in [HHS87].

Subdistributivity of relation α for some n-ary program operation *op* means that for any systems $S_1, \ldots, S_n$ the following inequality holds:

$$op(\alpha^{-1}(S_1), \ldots, \alpha^{-1}(S_n)) \subseteq \alpha^{-1}(op(S_1, \ldots, S_n)) \ .$$

If $S(X_1, \ldots, X_n)$ is built with monotonic subdistributive operations only, then it follows easily that

$$\begin{array}{ll} \text{if} & C_i \subseteq \alpha^{-1}(A_i) \text{ for } i = 1, \ldots, n \\ \text{then} & S(C_1, \ldots, C_n) \subseteq \alpha^{-1}(S(A_1, \ldots, A_n)). \end{array}$$

So, subdistributivity forms the basis of the compositional treatment of *data refinement* in [HHS87], where abstract operations A_i on abstract data within program S, are implemented by concrete operations C_i, operating on concrete data. Concrete operations of C_i, resp. abstract operations of A_i, can be considered as atomic operations in the syntax trees of $S(C_1, \ldots, C_n)$, resp. $S(A_1, \ldots, A_n)$. I.e. apart from their atoms these trees have the same syntactic structure.

However, in some situations a more general notion of subdistributivity is required that allows the refinement relation ρ between the overall abstract and concrete programs to be *different* from the refinement relations ρ_i between the concrete and abstract components C_i resp. A_i. This is illustrated in the following example.

Example 2.1

Consider the an abstract communication medium MED_A. Messages $m_1m_2\ldots m_k$ enter MED_A via some channel in_A and leave via channel out_A. Messages cannot get lost or duplicated, but they can leave MED_A in a different order than they entered it. We want to sketch a few development steps, both *vertically*, by refining the representation of messages and *horizontally*, by indicating how an abstract medium could be built up from a sender process that routes messages via a number of (lower level) channels to a receiver process that merges them into a single stream which leaves the communication module through a buffer. What we want to illustrate is that a relatively simple message representation for the *interface* of the whole module has to be replaced by a more complicated representation inside. Moreover, the *context* that puts together the sender, the channels and the receiver has to be refined into a more complicated context that includes a sliding window process. Thus we see here an example of context refinement with a varying abstraction relation.

First we consider a 'vertical' development step, where MED_A is refined into a more concrete one MED_C. For this refined medium MED_C we take into account that abstract messages m_i, which can be of arbitrary length, are to be split into sequences of fixed-length packets $\pi(m_i) = p_i^1p_i^2\ldots p_i^{k_i}$ for the concrete level. The channels in_A and out_A are for the concrete level replaced by similar channels in_C and out_C, and an abstract message m traveling along in_A or out_A is replaced by the sequence $\pi(m)$ that travels along in_C or out_C. We specify a simple protocol that requires that packets for a given message enter and leave MED_C *as a contiguous, ordered sequence.* In order to reconstruct a message from its packets we assume that each packet p_i^j carries a message identification as well as the total number of packets for that message. The relation between the abstract the concrete level is easily formalized by a retrieve function ρ, mapping communication *histories* for in_C and out_C to abstract communication histories. For a history h of the form $\pi(m_1)\pi(m_2)\cdots\pi(m_n)$ we define $\rho(h) = m_1, m_2, \ldots m_n$. If h' is a history like h as above except that for the last message only a few packets have been communicated we can isolate the longest prefix h'' of h' of 'complete' messages and we define $\rho(h') = \rho(h'')$. This retrieve function ρ forms the basis for an abstraction relation α that relates the complete concrete behaviour, i.e. the combined history h_C for the in_C and out_C channels together, to the complete abstract behaviour h_A, thus:

$$\alpha(h_C, h_A) \text{ iff } (h_A|in_A) = \rho(h_C|in_C) \text{ and } (h_A|out_A) = \rho(h_C|out_C).$$

Note that α is functional, in that there is at most one h_A value for any h_C value, which will be denoted by $\alpha(h_C)$. For histories h_C that do not conform to the protocols introduced above the value of $\alpha(h_C)$ is not defined.

We can now specify the required *interface refinement* as follows:

$$MED_C \subseteq \alpha^{-1}(MED_A).$$

Due to the fact that α is a partial function this can also be rephrased as the requirements that α is *defined* for all MED_C histories and moreover that

$$\alpha(MED_C) \subseteq MED_A.$$

We remark that ρ (and α) map histories to histories and that a mapping from (single) concrete communications to abstract ones does not suffice.

Thus far we have described a rather standard simulation relation for concurrent programs. Note that the refinement relation defines the representation of messages on the module *interface* only; nothing is said yet about message representation inside the module. In fact this representation is determined only after some 'horizontal' development steps are made, where we develop the medium ***on the abstract level***, i.e. without taking any message representation into account. Then, after this horizontal development, *compositional refinement* comes in when we refine abstract internal messages. The idea of the horizontal step is that we use standard *process refinement* techniques to implement the abstract communication medium as a network of processes consisting of a router and a merge process communicating via an asynchronous network consisting of a number of (virtual) channels. Because the messages may be routed through the network via different routes, it is possible that they are received out of order. We can describe this by means of a program in a CSP style process language:

$$(NET_A)\backslash\{Vin_A(i), Vout_A(i), Bin_A\} \text{ where}$$

$$NET_A \triangleq ROUTER_A \parallel VCHANNELS_A \parallel MERGER_A \parallel BUFFER_A, \text{ and}$$

$$VCHANNELS_A \triangleq VCHAN_A(1) \parallel VCHAN_A(2) \parallel \cdots.$$

The (synchronous) transfer of messages from one component to another can be described by means of CSP style communication via CSP channels. We remark here that CSP communication channels should not be confused with communication media such as MED_A or the $VCHAN$ channels. CSP communication is used here exclusively as a *mathematical* device to describe the transfer of messages from one component to another, whereas processes like $VCHAN_A(i)$ are (simplified) models of certain components of communication networks or distributed operating systems. The CSP 'channels' connected to these processes are as follows:

$$chan(ROUTER_A) = \{in_A, Vin_A(1), Vin_A(2), \ldots\}$$

$$chan(VCHAN_A(i)) = \{Vin_A(i), Vout_A(i)\}$$

$chan(MERGER_A) = \{Bin_A, Vout_A(1), Vout_A(2), \ldots\}$

$chan(BUFFER_A) = \{Bin_A, out_A\}$

Except for the in_A and out_A channels, all these channels are *hidden* by the CSP hiding construct "$X \backslash \{Vin_A(i), Vout_A(i), Bin_A\}$". The idea of the design is that incoming messages are forwarded by the ROUTER via one of the VCHANs which act as communication media, of limited capacity and with a low reliability. The ROUTER should take such capacities and potential failures into account and distribute incoming messages in an appropriate way. The MERGER collects the messages from all VCHANs and sends them all via the *Bin* channel towards the BUFFER which in turn delivers them via out_A. Due to nondeterministically determined delays in the reception of messages that are sent via different VCHAN's the order of messages might indeed get lost, as is allowed by the specification. It is not difficult to specify processes like $ROUTER_A$, and to show correctness of the MED_A implementation as above on the basis of these component specifications. One might then continue this 'horizontal' development, by implementing the component processes. At some moment though this has to be followed by a 'vertical' stage, where we take the representation of messages by sequences of packets into account. During this vertical stage a component such as $ROUTER_A$ is replaced by a component $ROUTER_C$ that behaves much like $ROUTER_A$ except that it operates on packets rather than messages. At first look we might use essentially the same abstraction relation α to relate the concrete internal level to the abstract level. (This 'generic' α should relate, by means of the ρ function, not only in_C and out_C to in_A and out_A but also the concrete *internal* channels to their abstract counterparts.) A correct solution could then be specified by the requiring that $ROUTER_C \subseteq \alpha^{-1}(ROUTER_A)$, and similarly for the other components. For in that case the subdistributivity of the CSP parallel composition and hiding constructs would guarantee that $MED_C \subseteq \alpha^{-1}(MED_A)$, as required.

The problem with this proposed solution is that it assumes that the low level $VCHAN$ processes preserve the ordering of messages, which is quite unrealistic, and moreover forces us to send all packets for some particular message via the same $VCHAN$ virtual channel. If we drop the assumption on order preservation and allow the $ROUTER$ to arbitrarily distribute packets the protocol specified above is no longer obeyed, since in general the packets for a message will leave MED_C via out_C out of order and non-contiguous, i.e. intermixed with packets belonging to other messages.

Informally one sees that we can correct the situation by tagging packets with a sequence number and replacing the abstract $BUFFER_A$ process by a process SW implementing a *sliding window protocol*. That is, rather than merely buffering packets, SW will delay incoming packets until it has received all packets for some message and will then deliver all of them, in order and consecutively, via out_C.

For this more sophisticated solution we can no longer use ρ (and α) as the abstraction function for the *internal* behaviours. Rather we define a more complex variation γ of ρ that in some sense incorporates a specification of a sliding window protocol in that it extracts the abstract messages from a 'shuffled' sequence of packets. (For those sequences that *do* conform to our protocol, γ and ρ are *both* defined and yield equal results).

Let $\#h$ denote the length of a communication history h and $h\backslash\{p_1, p_2, \ldots\}$ a variant of the hiding operation, that removes the indicated messages $p_1, p_2, \ldots$ from h. then we define:

$$\gamma(\epsilon) = \epsilon$$

$$\gamma(h) = \begin{cases} m_l\,\gamma(h\backslash\{p_l^j \mid 1 \leq j \leq k_l\}) & , \begin{array}{l}\text{if there exists an } n : 1 \leq n \leq \#h\text{:}\\ \{p_l^j \mid 1 \leq j \leq k_l\} \subseteq \{h_i \mid 1 \leq i \leq n\}\\ \text{and for all } i : 1 \leq i \leq \#h\\ \{p_i^j \mid 1 \leq j \leq k_i\} \not\subseteq \{h_j \mid 1 \leq j < n\}^2\end{array} \\ \\ \epsilon & , \text{otherwise.}\end{cases}$$

As an example, suppose $\pi(m_1) = p_1^1 p_1^2$ and $\pi(m_2) = p_2^1$ and $p_1^1 p_1^2 p_2^1$ is transmitted via in_C, corresponding to $\rho(p_1^1 p_1^2 p_2^1) = m_1 m_2$ via in_A. On channel Bin_C the sliding window may receive sequences $p_1^1 p_1^2 p_2^1$ and $p_2^1 p_1^1 p_1^2$ representing $m_1 m_2$ and $m_2 m_1$ respectively, and which would be legal output for the out_C channel. It may also receive a sequence such as $p_1^2 p_2^1 p_1^1$, which is not allowed on out_C, and for which ρ is not defined. The γ function *is* defined for all three sequences, e.g. $\gamma(p_1^2 p_2^1 p_1^1)$ is computed as follows:

$$\begin{array}{cll} & \gamma(p_1^2 p_2^1 p_1^1) & \\ = & m_2\,\gamma(p_1^2 p_2^1 p_1^1\backslash\{p_2^1\}) & \text{, choose } n = 2\\ = & m_2\,\gamma(p_1^2 p_1^1) & \text{, definition of hiding}\\ = & m_2\,m_1\,\gamma(p_1^2 p_1^1\backslash\{p_1^1, p_1^2\}) & \text{, choose } n = 2\\ = & m_2\,m_1\,\gamma(\epsilon) & \text{, definition of hiding}\\ = & m_2\,m_1 & \end{array}$$

Thus, γ maps h_C to the abstract history h_A such that message $h_A(i)$ is the i^{th} completely received message. Based on γ we formulate a β relation defined on the complete internal behaviour.

[2] One chooses n to be the minimal value for which there exists an index l such that all packets of m_l are received.

For example, for the $MERGER$ component we define

$\beta(h_C, h_A)$ iff

$(h_A | \{Vout_A(1), Vout_A(2), \ldots\}) = \gamma(h_C | \{Vout_C(1), Vout_C(2), \ldots\})$ and

$(h_A | Bin_A) = \gamma(h_C | Bin_C)$.

The criterium for correct refinement of $MERGER_A$ by $MERGER_C$ is then formulated as expected:

$$MERGER_C \subseteq \beta^{-1}(MERGER_A),$$

and similar requirements for the $ROUTER$ and $VCHAN$ components. Subdistributivity is now sufficient to conclude that we have also

$$(INT_C) \subseteq \beta^{-1}(INT_A).$$

where

$INT_A \triangleq (ROUTER_A \parallel VCHANNELS_A \parallel MERGER_A)$ and where

$INT_C \triangleq (ROUTER_C \parallel VCHANNELS_C \parallel MERGER_C)$.

Finally, we define *contexts* for INT_A and INT_C:

$$Ctx_A(X) \triangleq (X \parallel BUFFER) \backslash \{Vin_A(i), Vout_A(i), Bin_A\}$$

$$Ctx_C(Y) \triangleq (Y \parallel SW) \backslash \{Vin_C(i), Vout_C(i), Bin_C\}$$

What we claim here is that context Ctx_C *refines* Ctx_A in the following sense:

$$Y \subseteq \beta^{-1}(X) \Rightarrow Ctx_C(Y) \subseteq \alpha^{-1}(Ctx_A(X)), \text{ for all } X, Y.$$

This property can be formulated equivalently as:

$$Ctx_C(\beta^{-1}(X)) \subseteq \alpha^{-1}(Ctx_A(X)), \text{ for all } X.$$

A proof of this claim will be sketched below, after the formal definition of compositional refinement. From the claim and the refinement relation between INT_A and INT_C it then easily follows that

$$Ctx_C(INT_C) \subseteq \alpha^{-1}(Ctx_A(INT_A)).$$

(Which shows the correctness of the whole design).

What the example shows is that the usual definition of program simulation, which assumes a uniform choice for the retrieve function, is not appropriate within a compositional set-up. For although the γ and β functions *could* have been used at the

interface level too, such is exactly the situation one wants to avoid by the principle of 'separation of concerns': the (simple) ρ and α functions are all that is needed to specify the externally observable behaviour of the communication module, and any complexity related to internal detail is to be avoided for that purpose. Moreover, one might decide later on to re-implement the module much better than our proposal, *while retaining our refined interface.*
(End of example)

We define a more general notion of refinement, avoiding the limitation signalled above by introducing separate refinement relations for every component. Moreover, the definition is in terms of *context refinement.*

Definition 2.1 *(Compositional Refinement)*
We say that $S_0(X_1, \ldots, X_n)$ is refined by $T_0(Y_1, \ldots, Y_n)$ with respect to F_0 and $F_1, \ldots, F_n$ iff

$$T_0(F_1(X_1), \ldots, F_n(X_n)) \subseteq F_0(S_0(X_1, \ldots, X_n)),$$

for all $X_1, \ldots, X_n$.
□

In this paper we mainly concentrate on the case where F_j is of the form α_j^{-1}. We repeat the definition for this special case:

Definition 2.2 *(Compositional refinement based on weak simulation)*
System $S_0(X_1, \ldots, X_n)$ is refined by $T_0(Y_1, \ldots, Y_n)$ with respect to relations α_0 and $\alpha_1, \ldots, \alpha_n$ iff

$$\alpha_0^{-1}(S_0(X_1, \ldots, X_n) \supseteq T_0(\alpha_1^{-1}(X_1), \ldots, \alpha_n^{-1}(X_n)))$$

for all $X_1, \ldots, X_n$.
□

For the simple case that S_0 and T_0 contain no free variables, our definition coincides with the notion of weak simulation introduced in definition 1.2.

Another important special case is that where S_0 equals T_0 and where $F_0 = F_1 = \cdots = F_n$. This is essentially subdistributivity of F_0 for S_0, where we extend this latter notion to complete terms $S_0(X_1, \ldots, X_n)$, rather than just operations $op(X_1, \ldots, X_n)$.
We already noticed that data refinement in the sense of [HHS87] implies a transformation of programs in which only their (atomic) data structure operations are replaced. The reason for this is that although subdistributivity admits implementation of operations or subprograms within a context S, it does not admit transformation of the *context* S itself. For concurrency, this situation is not satisfactory, because there are many cases where one's intuitive notion of implementation implies a change

of context. For instance, take Milner's suggestion (in [M80]) to implement shared variable concurrency using communication based concurrency by modelling shared variables as separate concurrent processes, whose communications correspond to read and write operations. Here, the added shared variable modelling processes are put in parallel with the top syntactic level of the appropriately transformed shared variable program, implying a syntactic change at various levels. Another example of such a context change is contained in [Z90], which gives a correctness proof of a reification where the sequential abstract operations of a program are implemented by concurrent versions, more specifically, where abstract operations are replaced by communications with concurrent processes, implementing the data structures involved in concurrent fashion and running in parallel with the appropriately transformed original program.

Some basic properties of compositional refinement are contained in the following theorems, which we present in the form appropriate for weak simulation.

Theorem 2.3 *(Refinement for monotonic terms)*
Let $T_0(Y_1, \ldots Y_n)$ be monotonic in each of its Y_i variables. Then $S_0(X_1, \ldots, X_n)$ is refined by $T_0(Y_1, \ldots, Y_n)$ with respect to relations α_0 and $\alpha_1, \ldots, \alpha_n$ if for all $Y_1, \ldots, Y_n$ and all $X_1, \ldots, X_n$:

$$(\bigwedge_{i=1..n} Y_i \subseteq \alpha_i^{-1}(X_i)) \;\Rightarrow\; T_0(Y_1, \ldots, Y_n) \subseteq \alpha_0^{-1}(S_0(X_1, \ldots, X_n)).$$

■

Theorem 2.4 *(Horizontal composability)*
Let the following conditions be satisfied.

- $S_0(X_1,\ldots,X_n)$ is refined by $T_0(X_1,\ldots,X_n)$ with respect to α_0 and $\alpha_1,\ldots,\alpha_n$.
- $S_i(Y_1,\ldots,Y_m)$ is refined by $T_i(Y_1,\ldots,Y_m)$ with respect to α_i and $\beta_1,\ldots,\beta_m$, for $i=1,\ldots,n$. (We assume here that the variables of each of the systems S_i is contained in a common list $Y_1,\ldots,Y_m$.)
- T_0 is monotonic in each of the X_i variables.

Then the composed system

$$S_0(S_1(Y_1,\ldots,Y_m),\ldots,S_n(Y_1,\ldots,Y_m))$$

is refined by

$$T_0(T_1(Y_1,\ldots,Y_m),\ldots,T_n(Y_1,\ldots,Y_m))$$

with respect to α_0 and $\beta_1,\ldots,\beta_m$.
■

Our definition of compositional refinement is rather general and moreover formulated in terms of parameterized, i.e. higher order, programs. Thus it might seem complicated to prove refinement on the basis of actual program texts and assertional specifications. For concrete specification- and programming languages it is possible though to have simpler criteria for checking the refinement conditions. We give a sketch of this for the case of CSP style processes and assertional trace specifications, within a "Programs as Predicates" setting [Ho85],[ZdR89]. That is, we use a mixed formalism that includes both processes and specifications as special case.

An assertional trace specification of a process P is a (first order) formula $\chi(h)$ with a special designated variable h denoting the communication history of the specification. The programs as predicates paradigm can be paraphrased as follows. Programs P can *semantically* be regarded as predicates on traces too, which we sometimes indicate by the notation $P(h)$. (*Syntactically* though, the h variable does not occur at all in the program text of P.) A *context* $Ctx(X_1,\ldots,X_n)$ can be regarded in this way as a predicate of the form $\chi(h_1,\ldots,h_n,h)$, where the h_i denote traces of the components X_i, and where the last variable h denotes the trace of Ctx put around those components. Proof systems such as [Z89], [ZdR89] allow one to prove implications of the form $P(h)\rightarrow\chi(h)$. Such implications denote exactly the same as $P\subseteq\chi$ in the notation of this paper, i.e. program P should satisfy specification χ.

Now for predicates as above, there is a simple characterization for inverse images of the form $\alpha^{-1}(\chi)$, by means of substitution:

Lemma 2.5
For functions α and predicates $\chi(h)$ on traces:

$$\alpha^{-1}(\chi(h)) \;=\; \chi(\alpha(h))$$

■

Assume that we want to prove that some context $Ctx_A(X_1,\ldots,X_n)$ is refined by $Ctx_C(Y_1,\ldots,Y_n)$ with respect to α and $\beta_1,\ldots,\beta_n$. Furthermore, assume that we have *equivalent* predicates χ_A and χ_C for Ctx_A and Ctx_C. The refinement condition of the form

$$Ctx_C(\beta_1^{-1}(X_1),\ldots,\beta_n^{-1}(X_n)) \subseteq \alpha^{-1}(Ctx_A(X_1,\ldots,X_n))$$

can then be rewritten as follows:

$$\forall h \left(\exists h_1,\ldots,h_n(\bigwedge_{i=1..n} X_i(\beta_i(h_i)) \;\wedge\; \chi_C(h_1,\ldots,h_n,h)) \;\;\rightarrow \right.$$

$$\left. \exists t_1,\ldots,t_n(\bigwedge_{i=1..n} X_i(t_i) \;\wedge\; \chi_A(t_1,\ldots,t_n,\alpha(h)) \right)$$

This implication should be valid for all $X_1,\ldots,X_n$. A sufficient condition for the implication to hold is obtained by choosing $t_i = \beta_i(h_i)$, followed by simplification of the formula:

$$\forall h\, (\forall h_1,\ldots,h_n(\chi_C(h_1,\ldots,h_n,h) \;\rightarrow\; \chi_A(\beta_1(h_1),\ldots,\beta_n(h_n),\alpha(h)))) \,.$$

What we have shown is the following theorem:

Theorem 2.6 *(Compositional refinement for trace specifications)*
For trace predicates $\chi_A(h_1,\ldots,h_n,h)$ and $\chi_C(h_1,\ldots,h_n,h)$ a sufficient condition for refinement of Ctx_A by Ctx_C with respect to α and $\beta_1,\ldots,\beta_n$ is:

$$\forall h\, (\forall h_1,\ldots,h_n(\chi_C(h_1,\ldots,h_n,h) \;\rightarrow\; \chi_A(\beta_1(h_1),\ldots,\beta_n(h_n),\alpha(h)))) \,.$$

■

Example 2.2
As an example we consider the contexts introduced at the end of example of the communication medium. We recall the definition of these contexts:

$$Ctx_A(X) \triangleq (X \;\|\; BUFFER)\backslash\{Vin_A(i), Vout_A(i), Bin_A\}$$

$$Ctx_C(Y) \triangleq (Y \;\|\; SW)\backslash\{Vin_C(i), Vout_C(i), Bin_C\}$$

Techniques as for instance discussed in [Z89] allow one to prove equivalence of these contexts with the following predicates:

$$\chi_A(h_1,h) \triangleq \exists t'(h = (t'|\{in_A, out_A\}) \wedge chan(t') = \{in_A, out_A, Bin_A\}$$

$$\wedge (t'|\{in_A, Bin_A\}) = h_1 \wedge out_A \leq (t'|Bin_A))$$

and:

$$\chi_C(h_1, h) \triangleq \exists h'(h = (h'|\{in_C, out_C\}) \wedge chan(h') = \{in_C, out_C, Bin_C\}$$

$$\wedge (h'|\{in_A, Bin_A\}) = h_1 \wedge out_C \leq \gamma(t'|Bin_C))$$

What has to be shown to prove the refinement relation that we claimed at the end of the previous example can now be reduced to the following straightforward verification condition:

$$\forall h_1, h(\chi_C(h_1, h) \;\rightarrow\; \chi_A(\beta(h_1), \alpha(h)),$$

where α and β are the abstraction relations introduced in the example.
(End of example)

Finally we provide another example of context refinement, which can be considered a special case of compositional refinement.

Example 2.3 *(Self-stabilizing snapshot algorithm of [KP90])*
Consider a distributed system in which processes communicate by asynchronous message passing via directed channels. The communication network is strongly connected, and the channels are FIFO buffers of sufficient capacity. The *global state* of a distributed system is the product of all the local states plus the contents of the channels. An *accurate snapshot* is defined as follows [KP90].

> *At any global state σ, a process is said to have an* accurate snapshot *of σ' if local variables of the process contain a representation of a global state that is a* possible *successor of σ' and a* possible *predecessor of σ.*[3]

Snapshots may be used, for example, to detect wether a distributed algorithm has terminated or to retrieve information contained in the local states of the processes.

In [CL85] Chandy and Lamport presented the algorithm for obtaining accurate snapshots, which is used as a basis for the 'StableSnap' algorithm of Katz and Perry [KP90]. We will briefly outline the Chandy-Lamport algorithm. For a more complete discussion of the algorithm and its correctness argument we refer to [CL85, D83]. Process P_0 may invoke the snapshot algorithm by recording its local state and sending a *marker* along each outgoing channel. From this moment on the process records for each incoming channel the messages it receives. On receiving a marker along an incoming channel c a process P_i executes the subroutine $CL(c)$ (algorithm 1). If process P_0 has received all reports, it may initiate another snapshot. Let CL_i denote the snapshot algorithm for process P_i and $\frac{S}{T}$ denote the superposition

[3] It is implicitly understood that σ and σ' are states within the same computation.

```
IF $P_i$ has not yet recorded its local state
   THEN $P_i$ records its local state and starts recording
        the incoming messages on channel $c$;
        $P_i$ sends a marker on each outgoing channel
   ELSE $P_i$ stops recording the incoming messages on channel $c$
FI;
IF $P_i$ has stopped recording the messages on all incoming channels
   THEN $P_i$ sends a report to $P_0$
FI
```

Algorithm 1: $CL(c)$ (Snapshot, [CL85]).

(or superimposition), see e.g. [BF88], of program S upon T, then the distributed system $P_0 \parallel \cdots \parallel P_{n-1}$ is transformed into

$$\frac{CL_0}{P_0} \parallel \cdots \parallel \frac{CL_{n-1}}{P_{n-1}} .$$

The structure of this system can be described by

$$C(P_0, \ldots, P_{n-1}, CL_0, \ldots, CL_{n-1}) , \tag{2}$$

where the *context* $C(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1})$ is defined as

$$\frac{Y_0}{X_0} \parallel \cdots \parallel \frac{Y_{n-1}}{X_{n-1}} .$$

We describe an algorithm called 'StableSnap' that can be viewed as a refinement of (2), where the context $C(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1})$ is transformed into $K(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1})$, thereby leaving $P_0, \ldots, P_{n-1}$ and $CL_0, \ldots, CL_{n-1}$ unchanged. So, the resulting system can be described as

$$K(P_0, \ldots, P_{n-1}, CL_0, \ldots, CL_{n-1}) , \tag{3}$$

Compositional refinement allows one to show a refinement relation between $C(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1})$ and $K(X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1})$ without paying attention to the structure of P_i and CL_i $(i = 1, \ldots, n)$, all in order to prove that (2) is refined by (3). Similarly, one could refine P_i into some process P_i' or CL_i into CL_i' $(i = 1, \ldots, n)$. Subdistributivity as in [HH87] would allow only these latter refinements, but *not* context refinement.

The 'StableSnap' algorithm is a *self-stabilizing* snapshot algorithm based upon the Chandy-Lamport algorithm. Paraphrasing Katz and Perry, a self-stabilizing program is a program that eventually resumes normal behaviour even if its execution is initiated from an illegal state. In this sense a self-stabilizing program can tolerate transient faults. Following [KP90], *sem*(P) denotes the set of all possible execution sequences of P for arbitrary initial states. Let *legsem*(P) denote the subset of

$sem(P)$ with all execution sequences starting in *legitimate* initial states (initial states satisfying some characteristic predicate.) Given the definitions for self-stabilization:

> *Program P is* self-stabilizing *if each sequence in $sem(P)$ has a non-empty suffix that is identical to a suffix of some sequence in $legsem(P)$,*

and program extension:

> *Program Q is an* extension *of program P if for each global state in $legsem(Q)$ there is a projection onto all variables and messages of P such that the resulting set of sequences is identical to $legsem(P)$, upto stuttering,* [4]

a program Q is defined to be a *self-stabilizing extension* of program P if Q is self-stabilizing and also an extension of P (see [KP90]).

Below we give a brief description of the core of the 'StableSnap' algorithm. Each *round* of the 'StableSnap' algorithm is initiated by process P_0 by sending a marker on each outgoing channel. Process P_0 may initiate a new round at any time. A round ends when process P_0 has received a *report* for that round from all processes. Due to the lack of synchronization each process may be involved in a different round, Therefore it is assumed that each marker and report contains the round number of the originating process at the moment of sending. Upon receiving a marker via channel c a process P_i invokes the algorithm $KP(c)$ (algorithm 2). A distributed system $P_0 \parallel \cdots \parallel P_{n-1}$ is transformed by superposition of the 'StableSnap' algorithm into

$$\frac{KP_0(CL_0)}{P_0} \parallel \cdots \parallel \frac{KP_{n-1}(CL_{n-1})}{P_{n-1}} ,$$

where $KP_i(CL_i)$ means that KP_i uses the subroutines of CL_i.

It can be shown, c.f. [KP90], that a distributed system $\mathit{StableSnap}(X_1 \parallel \cdots \parallel X_k)$ obtained by superposition of the 'StableSnap' algorithm is a self-stabilizing extension of the system *Snapshot* $(X_1 \parallel \cdots \parallel X_k)$, that is obtained by superposing the Chandy-Lamport algorithm upon $X_1 \parallel \cdots \parallel X_k$. Let π be the function that maps each execution sequence s of 'StableSnap' to the unique execution sequence s' obtained by first projecting the states of s to the variables and messages of 'Snapshot' and then removing duplicate immediate-successors of states. Then from the fact that 'StableSnap' is an extension of 'Snapshot' it follows that

$$\hat{\pi}^{-1}(\mathit{legsem}(\mathit{Snapshot})) = \mathit{legsem}(\mathit{StableSnap}) ,$$

where $\hat{\pi}^{-1}$ is defined by

$$\hat{\pi}^{-1}(\Sigma) \triangleq \{s \mid s' = \pi(s),\ s' \in \Sigma\} .$$

[4] A sequence is stuttering if there exist two identical consecutive states in that sequence.

```
IF $P_i$ is recording this channel and
   the marker has the same round number as $P_i$
   THEN start $CL(c)$;
        IF $P_i$ stopped recording all channels
           THEN send a report to process $P_0$
        FI
   ELSE IF the marker was received before
           THEN skip
           ELSE IF the marker has a higher round number than $P_i$
                   THEN propagate the marker along all channels;
                        adapt the round number and restart $CL(c)$
                   ELSE propagate the marker along all channels;
                        IF $P_i$ stopped recording all channels
                           THEN send a report to process $P_0$
                        FI
                FI
        FI
FI
```

Algorithm 2: $KP(c)$ (StableSnap, [KP90]).

Furthermore, let κ be the relation that relates all execution sequences s with each execution sequence s' that is a non-empty suffix of s then $\hat{\kappa}^{-1}$ is defined by

$$\hat{\kappa}^{-1}(\Sigma) \triangleq \{s \mid (s,s') \in \kappa,\ s' \in \Sigma\} \ .$$

Given the fact that 'StableSnap' is self-stabilizing w.r.t. 'Snapshot', it follows that ($legsem(X) \subseteq sem(X)$ for all X)

$$\hat{\pi}^{-1}(\hat{\kappa}^{-1}(sem(Snapshot))) \supseteq sem(StableSnap) \ ,$$

It is easily seen that $Snapshot(X_1 \parallel \cdots \parallel X_k)$ is refined by $StableSnap(X_1 \parallel \cdots \parallel X_k)$ if one chooses $\alpha_0 = (\kappa \circ \pi)$ and for α_i ($i = 1, \ldots, k$) the identity relation in definition 2.1.

The superposed programs can only affect its own variables, but not variables of the underlying program. Furthermore, the underlying program and the superposed program can identify whether a message belongs to the underlying or the superposed program, so that no interference can occur. These restrictions guarantee that the superposed program can not block or affect the control of the underlying program. Therefore we may conclude that (this kind of) superposition is monotonic. Because superposition is monotonic, it follows from theorem 2.4 that if we replace the macro's of the Chandy-Lamport algorithm by correct implementations, then that will not affect the correctness of the Katz-Perry algorithm, *independently* of the underlying

program.
(End of example)

3 Conclusions

We discussed the rôle of compositionality in refinement and argued that a more general notion of refinement than subdistributivity is needed for compositional refinement. The motivation for such a generalized notion of refinement was illustrated by some non-trivial examples.

Furthermore, we introduced a general framework that unifies several refinement methods for sequential and concurrent programs. The expressive power of the resulting theory has been investigated, which resulted in the conclusion that several well-known theories, such as the prespecification calculus, are embedded within the theory.

A Formal framework

In this section we sketch some of the underlying principles and techniques that are used throughout the paper. One of the problems that we encountered when studying the literature on reification and simulation was the large variety of theories and methods being proposed, both for sequential and for parallel systems. Quite often there is a strong relationship between methods, and it is one of our aims to clarify such relationships. For sequential systems we presented a more uniform framework for several well known reification methods in [CdRZ91]. For instance, it was shown how Reynolds' reification method and VDM-style reification can be related within one predicate transformer framework. Here we extend these results by proposing a language that allows for the formulation of several definitions of simulation, such as upwards or downwards simulation, applicable to both sequential and parallel systems. The language resembles more classical formalisms such as predicate transformer calculi and the relational calculus. It differs from these formalisms in that we make a clear distinction between *composition* of relations and predicate transformers on the one hand and *sequential composition* of programs on the other hand. For instance, in a trace-based formalism we use binary retrieve relations to map concrete communication histories to abstract histories. Composing such relations does not correspond to the sequential composition operator of the programming language. For the latter operation amounts to *concatenation of histories*. However, the predicate transformer framework for sequential programs is embedded in the theory below as a special case in which programs denote binary relations on states. To deal with concurrent programs we use relations not on states but on complete computations, which we sometimes refer to as *generalized states*. In [Z89] a predicate transformer theory and formalism for concurrency based on this notion of generalized states is developed which indicates the wide range of applicability of predicate transformer concepts and which is subsumed in the formalism presented below.

The theory is defined relative to two basic types, viz. State and Comp . The elements State are (generalized) states and the elements of Comp are computations. For types $\tau_1, \ldots, \tau_n$ the type $\mathcal{R}el$ is defined by

$$\mathcal{R}el(\tau_1, \ldots, \tau_n) \triangleq \mathcal{P}(\tau_1 \times \cdots \times \tau_n)\,.$$

Thus the elements of $\mathcal{R}el$ are relations. In case of a unary relation we use $\mathcal{S}et(\tau)$ rather than $\mathcal{R}el(\tau)$. The type $\mathcal{T}rans$ is only defined for non-basic types τ_1 and τ_2:

$$\mathcal{T}rans(\tau_1, \tau_2) \triangleq \tau_1 \rightarrow \tau_2$$

The elements of $\mathcal{T}rans$ are called transformers, because predicate transformers turn out to be special elements of $\mathcal{T}rans$. Besides elements of the types defined above, the theory also includes a class of formulae, which will be discussed later.

Definition A.1 *(Relations)*
Relations are defined inductively as follows.

- Let χ be a n-place predicate on tuples in $\tau_1 \times \cdots \times \tau_n$, then

  $$\{(t_1, \ldots, t_n) \in \tau_1 \times \cdots \times \tau_n \mid \chi(t_1, \ldots, t_n)\}$$

  is a relation of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$.

- Let R be a relation of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$, then
  - the complement $\neg R \triangleq \bar{R}$ is a relation of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$.
  - the converse R^{-1} is a relation of type $\mathcal{R}el(\tau_n, \ldots, \tau_1)$.

- Let R_1 and R_2 be relations of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$, then the union $R_1 \vee R_2 \triangleq R_1 \cup R_2$ and the intersection $R_1 \wedge R_2 \triangleq R_1 \cap R_2$ are relations of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$.

- Let R_1 be a relation of type $\mathcal{R}el(\tau_1, \ldots, \tau_{n-1}, \tau_n)$ and R_2 be a relation of type $\mathcal{R}el(\tau_n, \tau_{n+1}, \ldots, \tau_{n+m})$, then their composition $R_1 \fatsemi R_2 \triangleq R_2 \circ R_1$ is a relation of type $\mathcal{R}el(\tau_1, \ldots \tau_{n-1}, \tau_{n+1}, \ldots, \tau_{n+m})$.

- A special case of the previous definition is the *relational image* $R(\!|S|\!)$, which abbreviates $S \fatsemi R$. We employ this abbreviation when S is a unary relation, i.e. a set. So if R is a relation of type $\mathcal{R}el(\tau_1, \tau_2, \ldots, \tau_n)$ and S is of type $\mathcal{S}et(\tau_1)$, then the relational image $R(\!|S|\!)$ is a relation of type $\mathcal{R}el(\tau_2, \ldots, \tau_n)$.

- Let T be a transformer of type $\tau_1 \rightarrow \tau_2$, and let R be a relation of type τ_1 then $T(R)$ is a relation of type τ_2.

□

Notice the notation $R_1 \fatsemi R_2$ for composition of relations. Although we do not present any particular programming language here, we denote sequential composition of programs by S_1 ; S_2. As explained above, the two operations coincide only for the case of *sequential* programs, where S_1 ; S_2 (also) denotes relational composition.

Definition A.2 *(Transformers)*
For non-basic types $\tau_1, \tau_2 \ldots$. transformers are defined inductively as follows.

- Let X be a variable of type τ_1 and R be of type τ_2, possibly with free occurrences of X, then $\lambda X.R$ is a transformer of type $\tau_1 \longrightarrow \tau_2$.
- Let T_1 and T_2 be transformers of type $\tau_1 \longrightarrow \tau_2$, then
  - $T_1 \wedge T_2$ and $T_1 \vee T_2$, respectively defined by $(T_1 \wedge T_2)(X) \triangleq T_1(X) \wedge T_2(X)$ and $(T_1 \vee T_2)(X) \triangleq T_1(X) \vee T_2(X)$, are transformers of type $\tau_1 \longrightarrow \tau_2$.
  - Likewise, $\neg T_1$ is a transformer of type $\tau_1 \longrightarrow \tau_2$.
- Let T_1 be a transformer of type $\tau_1 \longrightarrow \tau_2$ and T_2 be a transformer of type $\tau_2 \longrightarrow \tau_3$, then $T_2 \circ T_1$ is a transformer of type $\tau_1 \longrightarrow \tau_3$.
- Let T be a transformer of type $\tau_1 \longrightarrow \tau_2$ then the adjoint transformer $T^\dagger$ is defined by

  $$T^\dagger(X) \triangleq \bigcup\{Y \mid T(Y) \subseteq X\}.$$

  It is a transformer of type $\tau_2 \longrightarrow \tau_1$.

□

Because the definitions above are very general, we will discuss some more specific cases of transformers, that capture some of the more familiar (predicate) transformers.

If R is of type $\mathcal{R}el(\tau_1, \tau_2)$ and S of type $\mathcal{S}et(\tau_1)$, then $R(\!|S|\!)$ is the *strongest postcondition* of type $\mathcal{S}et(\tau_1)$.

$T^\dagger(X)$ is easily seen to be the largest Y such that $T(Y) \subseteq X$. For the cases of interest T can be assumed to be completely additive (c.a.), i.e.

$$T(\bigcup\{X \mid \ldots X \ldots\}) = \bigcup\{T(X) \mid \ldots X \ldots\}\ .$$

In that case $T^\dagger$ is also characterized by the following property of right adjoints:

$$T(X) \subseteq Y \text{ if and only if } X \subseteq T^\dagger(Y).$$

The use of the adjoint operation is that it allows for a uniform definition of several distinct predicate transformers, such as weakest preconditions, weakest prespecifications, generalizations thereof for concurrency, the weakest postspecification and the related 'leads to' transformer. We provide some of the details. Let R be a relation of type $\mathcal{R}el(\tau_1, \ldots, \tau_{n+1})$. Furthermore, let X resp. Y be a variable of type $\mathcal{R}el(\tau_1, \ldots, \tau_n, \tau_{n+2}, \ldots, \tau_{n+m})$ resp. $\mathcal{R}el(\tau_{n+1}, \ldots, \tau_{n+m})$. We define the so called 'leads to' transformer $\lambda X.R \rightsquigarrow X$ of type $\mathcal{R}el(\tau_1, \ldots, \tau_n, \tau_{n+2}, \ldots, \tau_{n+m}) \rightarrow \mathcal{R}el(\tau_{n+1}, \ldots, \tau_{n+m})$ as the adjoint of $R \mathbin{⨾} Y$, i.e.

$$\lambda X.R \rightsquigarrow X \triangleq (\lambda Y.R \mathbin{⨾} Y)^\dagger \,.$$

This definition includes the following well known transformers as special cases.

- If R is of type $\mathcal{S}et(\tau_1)$ and S of type $\mathcal{S}et(\tau_2)$, then $R \rightsquigarrow S$ is the 'leads to' relation of type $\mathcal{R}el(\tau_1, \tau_2)$. If we interpret R and S as precondition and postcondition of a Hoare style formula, then $R \rightsquigarrow S$ is the largest program (i.e. relation) that satisfies that Hoare formula.
- If R is of type $\mathcal{R}el(\tau_1, \tau_2)$ and S of type $\mathcal{R}el(\tau_1, \tau_3)$, then $R \rightsquigarrow S$ is the *weakest postspecification* S/R of Hoare and He [HH87] of type $\mathcal{R}el(\tau_2, \tau_3)$, (take $n = 1$ and $m = 2$ in the above definition.)

For relation R of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$ and variables X and Y respectively of type $\mathcal{R}el(\tau_{n+1}, \ldots, \tau_{n+m}, \tau_2, \ldots, \tau_n)$ and $\mathcal{R}el(\tau_{n+1}, \ldots, \tau_{n+m}, \tau_1)$, the transformer $[R]$ of type $\mathcal{R}el(\tau_{n+1}, \ldots, \tau_{n+m}, \tau_2, \ldots, \tau_n) \rightarrow \mathcal{R}el(\tau_{n+1}, \ldots, \tau_{n+m}, \tau_1)$ is defined as the adjoint of $Y \mathbin{⨾} R$, i.e.

$$\lambda X.[R]X \triangleq (\lambda Y.Y \mathbin{⨾} R)^\dagger \,.$$

The following special cases may be more familiar.

- Taking $m = 0$ and $n = 2$ in the above definition, if R is of type $\mathcal{R}el(\tau_1, \tau_2)$ and S is of type $\mathcal{S}et(\tau_2)$, then $[R]S$ denotes the *weakest precondition* transformer of type $\mathcal{S}et(\tau_1)$.
- And if R is of type $\mathcal{R}el(\tau_2, \tau_3)$ and S is of type $\mathcal{R}el(\tau_1, \tau_3)$, then $[R]S$ is the *weakest prespecification* $R \backslash S$ of [HH87] of type $\mathcal{R}el(\tau_1, \tau_2)$.

The transformer $\langle R \rangle$ is as usual defined as the dual of $[R]$, i.e. $\langle R \rangle X \triangleq \neg [R] \neg X$. The purpose of this transformer in theories for reification has been explained in [CdRZ91]. It has the so called 'angelicness' property, which can be formulated as

$$\langle S \vee T \rangle = \langle S \rangle \vee \langle T \rangle$$

Such transformers appear for instance in work by Ralph Back [BvW89]. Back does not make a clear notational distinction between transformers of the form $[S]$ and of the form $\langle S \rangle$. Consequently he must introduce angelic statements, such as the angelic choice operators '$\diamond$' satisfying the (surprising) law

$$[S \diamond T] = [S] \vee [T] \,.$$

A related construct is the angelic assignment of [BvW89] which is used in work on refinement. Such angelic constructs cannot be explained on the level of relations, i.e. if we model statements as binary relations, then the operation $S \Diamond T$ does not exist. For example the statement $x := a \Diamond x := b$ has the following properties.

$$[x := a \Diamond x := b](x = a) \equiv \mathsf{true} \quad (4)$$

$$[x := a \Diamond x := b](x = b) \equiv \mathsf{true} \quad (5)$$

From (4) it follows immediately that $[\![x := a \Diamond x := b]\!] \subseteq \{(s_0, s_1) \mid s_1(x) = a\}$, and likewise from (5) it follows that $[\![x := a \Diamond x := b]\!] \subseteq \{(s_0, s_1) \mid s_1(x) = b\}$. Hence, for all a and b such that $a \neq b$ it follows that $[\![x := a \Diamond x := b]\!] = \emptyset$. For this reason we prefer a theory based on a combined use of the box and diamond operators $[S]$ and $\langle S \rangle$.

Definition A.3 *(Formulae)*
The syntactic class $\mathcal{F}orm$ of (correctness) formulae is defined as follows.

- Let R_1 and R_2 be relations of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$, then $R_1 \subseteq R_2$ is a formula.
- Let $F_1 \in \mathcal{F}orm$ and $F_2 \in \mathcal{F}orm$, then $F_1 \rightarrow F_2$ and $F_1 \wedge F_2$ are formulae with the obvious interpretation.
- Let X be a variable of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$ and F be a formula with free occurrences of X, then $\forall X.F(X)$ is a formula. The formula $\forall X.F(X)$ is true if $F(R)$ is true for all relations R of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$, and false otherwise.

□

We will use abbreviations such as $\exists X.F(X)$ with the usual interpretation.

Some special formulae can be defined as follows. For relations R_1, R_2, and R_3 resp. of type $\mathcal{R}el(\tau_1, \ldots, \tau_n)$, $\mathcal{R}el(\tau_n, \ldots, \tau_{n+m})$, and $\mathcal{R}el(\tau_1, \ldots, \tau_{n-1}, \tau_{n+1}, \ldots \tau_{n+m})$ we define *generalized Hoare formulae*

$$(R_1)\ R_2\ (R_3) \triangleq R_1 ⨾ R_2 \subseteq R_3 \,.$$

In case that R_1, R_2, and R_3 resp. are of type $\mathcal{S}et(\tau_1)$, $\mathcal{R}el(\tau_1, \tau_2)$, and $\mathcal{S}et(\tau_2)$, then $(R_1)\ R_2\ (R_3)$ coincides with the classical Hoare-style correctness formula $\{R_1\}\ R_2\ \{R_3\}$ (c.f. [CdRZ91]). In a similar way [CdRZ91] VDM-style (partial) correctness formulae can also be defined.

As a last example of the expressive power of this framework, we demonstrate how the theory of [HHS87] can be embedded in our theory. A relation C *downward simulates* relation A w.r.t. simulation relation R if

$$R ⨾ C \subseteq A ⨾ R \,.$$

Following [HHS87] we define the transformer F_R by

$$F_R(X) \triangleq R \leadsto (X \fatsemi R) \,.$$

From the definition of $\leadsto$ it follows that

$$R \fatsemi F_R(A) \subseteq A \fatsemi R \,.$$

Thus $F_R(A)$ is the largest relation that downward simulates A with respect to R. Because, in the prespecification calculus '$\fatsemi$' coincides with ';', this is a generalized version of F_R as defined in [HHS87]. In a similar way we can also define the transformer G_S for *upward simulation*, c.f. [HHS87,CdRZ91].

References

[BF88] Bougé L & Francez N. *A Compositional Approach to Superimposition.* Proc. of the 15th Symp. on Principles of Programming Languages, pp. 240–249, 1988.

[BvW89] Back RJR & von Wright J. *A Lattice-theoretical Basis for a Specification Language.* Proc. of the conf. on Mathematics of Program Construction, LNCS 375, van de Snepscheut (Ed.) Springer 1989.

[CL85] Chandy KM & Lamport L. *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems 3(1):63–75, 1985.

[CdRZ91] Coenen J, de Roever WP & Zwiers J. *Assertional Data Reification Proofs: Survey and Perspective.* Proc. of the 4th BCS-FACS Refinement Workshop, Workshops in Computing, pp. 97–114, Springer-Verlag 1991.

[D83] Dijkstra E.W. *The Distributed Snapshot of K.M. Chandy and L. Lamport.* EWD864a.

[Ho85] Hoare C.A.R. *Programs are predicates.* in Mathematical Logic and Programming Languages, Hoare and Shepherdson(eds), Prentice-Hall, 1985.

[HH87] Hoare CAR & He J. *The Weakest Prespecification.* Information Processing Letters 24:127–132, 1987.

[HHS87] Hoare CAR, He J & Sanders JW. *Prespecification in Data Refinement.* Information Processing Letters 25:71–76, 1987.

[J90] Jones CB. *Systematic Software Development using VDM.* Prentice-Hall 1990 (2nd edition).

[KP90] Katz S & Perry JP. *Self-Stabilizing Extensions for Message-passing Systems.* Proc. of the 9th Symp. on Principles of Distributed Computing, pp. 91–101, 1990.

[M80] Milner R. *A Calculus of Communicating Systems.* LNCS 92, Springer-Verlag 1980.

[O91] Olderog ER. *Nets, Terms, and Formulas.* Cambridge Tracts in Computer Science 23, Cambridge University Press 1991.

[R81] Reynolds JC. *The Craft of Programming.* Prentice-Hall 1981.

[Z89] Zwiers J. *Compositionality, Concurrency, and Partial Correctness: Proof Theories for Networks of Processes, and their Relationship.* LNCS 321, Springer-Verlag 1989.

[Z90] Zwiers J. *Refining Data to Processes.* Proc. of VDM '90, LNCS 428, pp. 352–369, Springer-Verlag 1990.

[ZdR89] Zwiers J & de Roever WP. *Predicates are Predicate Transformers: a Unified Compositional Theory for Concurrency.* Proc. of the 8th Symp. on Principles of Distributed Computing, pp. 265–279, 1989.

In this series appeared:

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

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

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

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

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