A note on compositional refinement

Citation for published version (APA):

Document status and date:
Published: 01/01/1992

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.
If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
A Note on Compositional Refinement

by

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

Computing Science Note 92/01
Eindhoven, January 1992
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

All rights reserved
editors: prof.dr.M.Rem
        prof.dr.K.M.van Hee.
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.

---

1E-mail: zwiers@cs.utwente.nl
2Supported by NWO/SION project 612-316-022: "Fault Tolerance: Paradigms, Models, Logics, Construction." E-mail: wsinjoc@win.tue.nl
3Partially supported by ESPRIT project 3096: "SPEC." E-mail: wpr@informatik.uni-kiel.dbp.de
Such specifications have the form $S \models \chi$, often denoted in this context by $S \mathtt{sat} \chi$, where $S$ is a system as before, but where $\chi$ 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 $\varphi$ 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 $\varphi$ is replaced by process $S$ such that $S \models \varphi$.
- A process $S$ is replaced by another process $T$ such that $T \models S$.
- A specification $\varphi$ is replaced by another specification $\psi$ such that $\psi \models \varphi$, 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 logical 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 \(\rho\) 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 \(p\) on runs, in the form of state sequences \(\sigma_C = (s_0, s_1, s_2, \ldots)\), by pointwise applying \(\rho\). Hence,

\[
\sigma_A = (\rho(s_0), \rho(s_1), \rho(s_2), \ldots).
\]

Similarly, \(\rho\) is then extended to an operation on sets of runs, by applying \(\rho\) 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 \(\rho\), if for any possible \(C\) run there is a corresponding, i.e. \(\rho\) related, \(A\) run. This is easily expressed by the following requirement.

\[
\rho(C) \subseteq A \tag{1}
\]

Note that for the special case that \(\rho\) 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 \(\alpha\) relations \(\alpha\) rather than retrieve functions \(\rho\) 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 \(\alpha\) 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 \(\alpha\) 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 $\alpha$ if

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

Strong simulation requires that for any $C$ run $\sigma_C$ all $\alpha$ related runs $\sigma_A$ are possible runs for the abstract system $A$. Weak simulation only requires that there is at least one such $\alpha$ 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) $\alpha$ is total, but $C \subseteq \alpha^{-1}(A) \Rightarrow \alpha(C) \subseteq A$ if (and only if) $\alpha$ 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:

$$C \subseteq F_\alpha(A) \quad (\text{downward simulation})$$

$$C \subseteq G_\alpha(A) \quad (\text{upward simulation})$$

The operations $F_\alpha$ and $G_\alpha$ 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 $X \circ Y$, weakest preconditions $[X]Y$ and the leads-to operator $X \rightsquigarrow Y$, as follows:

$$F_\alpha(X) = \alpha^{-1} \rightsquigarrow (X \circ \alpha^{-1}),$$

$$G_\alpha(X) = [\alpha](\alpha \circ X).$$

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 \( \alpha^R \) denotes the right adjoint of \( \alpha \). 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 is called *subdistributivity*. An operator \( F \) as above sub-distributes over some \( n \)-ary language operator \( \text{op} \) iff

\[
F(\text{op}(X_1, \ldots, X_n)) \supseteq \text{op}(F(X_1), \ldots, F(X_n)).
\]

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 \( a \), and an implementing program \( C \) operating on corresponding concrete variables \( c \), where \( C \) refines \( A \) with respect to abstraction relation \( \alpha \), 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 finalize, 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 \( \alpha \) 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) if

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

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 \( \rho \) between the overall abstract program and the concrete program may be different from the refinement relations \( \rho_i \) 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 \textit{vertically} composable as well as \textit{horizontally} composable. Vertical composable or \textit{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 composable of retrieve functions. Horizontal composability or \textit{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 composable are readily satisfied, because equality is transitive and sub-
stitutive:

if \( Q = R \) and \( R = S \) then \( Q = S \), and
if \( S_i = T_i \) for \( i = 1, \ldots, n \) then \( S(S_1, \ldots, S_n) = S(T_1, \ldots, T_n) \).

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 \( S \subseteq T \) denotes the set inclusion \( T \subseteq S \) — this means that an operation ‘op’ satisfies the following property.

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

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 \( \alpha_1 \) and \( S \) refines \( R \) with respect to \( \alpha_2 \) than \( S \) refines \( Q \) with respect to \( \alpha_1 \circ \alpha_2 \):

if \( R \subseteq \alpha_1^{-1}(Q) \) and \( S \subseteq \alpha_2^{-1}(R) \) then \( S \subseteq (\alpha_1 \circ \alpha_2)^{-1}(Q) \).

Horizontal composability, however, is not so simple. From \( S_i \subseteq \alpha_i^{-1}(T_i) \) 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 \( \alpha \) (c.f. [CdRZ91]). What we need here is a property related to the notion of subdistributivity as introduced in [HHS87].

Subdistributivity of relation \( \alpha \) for some n-ary program operation \( \text{op} \) means that for any systems \( S_1, \ldots, S_n \) the following inequality holds:

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

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

if \( C_i \subseteq \alpha^{-1}(A_i) \) for \( i = 1, \ldots, n \) then \( S(C_1, \ldots, C_n) \subseteq \alpha^{-1}(S(A_1, \ldots, A_n)) \).

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 $\rho$ between the overall abstract and concrete programs to be different from the refinement relations $\rho_i$ 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 $m_1 m_2 \ldots m_k$ 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 $\pi(m_i) = p_1^i p_2^i \ldots p_k_i$ 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 $p_i^k$ 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 $\rho$, mapping communication histories for $in_C$ and $out_C$ to abstract communication histories. For a history $h$ of the form $\pi(m_1)\pi(m_2)\ldots\pi(m_n)$ 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 $\rho$ forms the basis for an abstraction relation $\alpha$ 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 $\alpha$ 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 \( \alpha \) is a partial function this can also be rephrased as the requirements that \( \alpha \) is *defined* for all \( MED_C \) histories and moreover that

\[ \alpha(MED_C) \subseteq MED_A. \]

We remark that \( \rho \) (and \( \alpha \)) 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:

\[
\text{(NET}_A\text{)}\{Vin_A(i), Vout_A(i), Bin_A\} \text{ where}
\]

\[
\text{NET}_A \triangleq \text{ROUTER}_A \parallel \text{VCHANNELS}_A \parallel \text{MERGER}_A \parallel \text{BUFFER}_A, \text{ and}
\]

\[
\text{VCHANNELS}_A \triangleq \text{VCHAN}_A(1) \parallel \text{VCHAN}_A(2) \parallel \cdots.
\]

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:

\[
\text{chan}(\text{ROUTER}_A) = \{in_A, Vin_A(1), Vin_A(2), \ldots\}
\]

\[
\text{chan}(\text{VCHAN}_A(i)) = \{Vin_A(i), Vout_A(i)\}
\]
\text{chan}(\text{MERGER}_A) = \{\text{Bin}_A, Vout_A(1), Vout_A(2), \ldots\} \\
\text{chan}(\text{BUFFER}_A) = \{\text{Bin}_A, \text{out}_A\}

Except for the \text{in}_A and \text{out}_A channels, all these channels are hidden by the CSP hiding construct \(\text{x}\{\text{Vin}_A(i), \text{Vout}_A(i), \text{Bin}_A\}\). 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 \text{Bin} channel towards the BUFFER which in turn delivers them via \text{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 \text{ROUTER}_A, and to show correctness of the \text{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 \text{ROUTER}_A is replaced by a component \text{ROUTER}_C that behaves much like \text{ROUTER}_A except that it operates on packets rather than messages. At first look we might use essentially the same abstraction relation \(\alpha\) to relate the concrete internal level to the abstract level. (This ‘generic’ \(\alpha\) should relate, by means of the \(\rho\) function, not only \text{in}_C and \text{out}_C to \text{in}_A and \text{out}_A but also the concrete internal channels to their abstract counterparts.) A correct solution could then be specified by the requiring that \text{ROUTER}_C \subseteq \alpha^{-1}(\text{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 \text{MED}_C \subseteq \alpha^{-1}(\text{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 \text{MED}_C via \text{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 \text{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 \text{out}_C.
For this more sophisticated solution we can no longer use $\rho$ (and $\alpha$) as the abstraction function for the internal behaviours. Rather we define a more complex variation $\gamma$ of $\rho$ 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, $\gamma$ and $\rho$ are both defined and yield equal results).

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

$$\gamma(\epsilon) = \begin{cases} 
\epsilon & \text{if there exists an } n: 1 \leq n \leq \#h:
\{p_i^j | 1 \leq j \leq k_i\} \subseteq \{h_i | 1 \leq i \leq n\} \\
\text{and for all } i: 1 \leq i \leq \#h \\
\{p_i^j | 1 \leq j \leq k_i\} \not\subseteq \{h_j | 1 \leq j < n\}\end{cases}$$

As an example, suppose $\pi(m_1) = p_1^2 p_2$ and $\pi(m_2) = p_3^2$ and $p_1^2 p_2 p_3^2$ is transmitted via $in_C$, corresponding to $\rho(p_1^2 p_2 p_3^2) = m_1 m_2$ via $in_A$. On channel $Bin_C$ the sliding window may receive sequences $p_1^2 p_2 p_3^2$ and $p_2^3 p_1 p_1^2$ 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 $p_1^2 p_2 p_3$, which is not allowed on $out_C$, and for which $\rho$ is not defined. The $\gamma$ function is defined for all three sequences, e.g. $\gamma(p_1^2 p_2 p_3^2)$ is computed as follows:

$$\begin{align*}
\gamma(p_1^2 p_2 p_3^2) &= m_2 \gamma(p_1^2 p_2 p_3^2 \backslash \{p_3^2\}) \quad \text{, choose } n = 2 \\
&= m_2 \gamma(p_1^2 p_2) \quad \text{, definition of hiding} \\
&= m_2 m_1 \gamma(p_2 p_1^2 \backslash \{p_1^2, p_1^3\}) \quad \text{, choose } n = 2 \\
&= m_2 m_1 \gamma(\epsilon) \quad \text{, definition of hiding} \\
&= m_2 m_1 \end{align*}$$

Thus, $\gamma$ maps $h_C$ to the abstract history $h_A$ such that message $h_A(i)$ is the $i^{th}$ completely received message. Based on $\gamma$ we formulate a $\beta$ 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

$$\beta(h_C, h_A) \text{ iff } (h_A|\{Vout_A(1), Vout_A(2), \ldots\}) = \gamma(h_C|\{Vout_C(1), Vout_C(2), \ldots\}) \text{ and } (h_A|Bin_A) = \gamma(h_C|Bin_C).$$

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

$$\text{MERGER}_C \subseteq \beta^{-1}(\text{MERGER}_A),$$

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

$$(\text{INT}_C) \subseteq \beta^{-1}(\text{INT}_A).$$

where

$$\text{INT}_A \triangleq (\text{ROUTER}_A \parallel \text{VCHANNELS}_A \parallel \text{MERGER}_A)$$

and where

$$\text{INT}_C \triangleq (\text{ROUTER}_C \parallel \text{VCHANNELS}_C \parallel \text{MERGER}_C).$$

Finally, we define contexts for INT_A and INT_C:

$$\text{Ctx}_A(X) \triangleq (X \parallel \text{BUFFER})\{Vin_A(i), Vout_A(i), Bin_A\}$$

$$\text{Ctx}_C(Y) \triangleq (Y \parallel \text{SW})\{Vin_C(i), Vout_C(i), Bin_C\}$$

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

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

This property can be formulated equivalently as:

$$\text{Ctx}_C(\beta^{-1}(X)) \subseteq \alpha^{-1}(\text{Ctx}_A(X)), \text{ for all } X.$$ 

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

$$\text{Ctx}_C(\text{INT}_C) \subseteq \alpha^{-1}(\text{Ctx}_A(\text{INT}_A)),$$

(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 $\gamma$ and $\beta$ 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) \( \rho \) and \( \alpha \) 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 \( \alpha_j^{-1} \). 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 \( \alpha_0 \) and \( \alpha_1, \ldots, \alpha_n \) iff

\[
\alpha_0^{-1}(S_0(X_1, \ldots, X_n)) \trianglerighteq T_0(\alpha_1^{-1}(X_1), \ldots, \alpha_n^{-1}(X_n))
\]

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 \( S_0 = T_0 \) and where \( F_0 = F_1 = \ldots = F_n \). 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 $\alpha_0$ and $\alpha_1,\ldots,\alpha_n$ if for all $Y_1,\ldots,Y_n$ and all $X_1,\ldots,X_n$:

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

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 $\alpha$ 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 $\alpha$ and $\beta_1, \ldots, \beta_n$. Furthermore, assume that we have equivalent predicates $\chi_A$ and $\chi_C$ 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:

$$\forall h \left( \exists h_1, \ldots, h_n \left( \bigwedge_{i=1}^{n} X_i(\beta_i(h_i)) \land \chi_C(h_1, \ldots, h_n, h) \right) \rightarrow \right.$$  
$$\left. \exists t_1, \ldots, t_n \left( \bigwedge_{i=1}^{n} X_i(t_i) \land \chi_A(t_1, \ldots, t_n, \alpha(h)) \right) \right)$$

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 $\alpha$ 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:

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

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

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

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

\[ \forall h_1, h \left( x_C(h_1, h) \rightarrow x_A(\beta(h_1), \alpha(h)) \right) \]

where \( \alpha \) and \( \beta \) 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 \( \sigma \), a process is said to have an accurate snapshot of \( \sigma' \) if local variables of the process contain a representation of a global state that is a possible successor of \( \sigma' \) and a possible predecessor of \( \sigma \).

Snapshots may be used, for example, to detect whether 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 \( \otimes \) denote the superposition
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, [CLS5]).

(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

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

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

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

(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 pro­
gram 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], $\text{sem}(P)$ denotes the set of all possible execution sequences of $P$ for arbitrary initial states. Let $\text{legsem}(P)$ 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), up to 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 P0 by sending a marker on each outgoing channel. Process P0 may initiate a new round at any time. A round ends when process P0 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 Pi invokes the algorithm KP(c) (algorithm 2). A distributed system P0 || ⋯ || Pn−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 StableSnap(X_1 || ⋯ || X_k) obtained by superposition of the ‘StableSnap’ algorithm is a self-stabilizing extension of the system Snapshot (X_1 || ⋯ || X_k), that is obtained by superposing the Chandy-Lamport algorithm upon X_1 || ⋯ || X_k. Let \( \pi \) 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

\[
\hat{\pi}^{-1}(\text{legsem(Snapshot)}) = \text{legsem(StableSnap)},
\]

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

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

4A 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

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

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

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

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

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

It is easily seen that $Snapshot(X_1 || \cdots || X_k)$ is refined by $StableSnap(X_1 || \cdots || X_k)$
if one chooses $\alpha_0 = (\kappa \circ \pi)$ and for $\alpha_i (i = 1, \ldots, k)$ 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
We discussed the role 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. \textit{State} and \textit{Comp}. The elements \textit{State} are (generalized) states and the elements of \textit{Comp} are computations. For types \(\tau_1, \ldots, \tau_n\) the type \(\textit{Rel}\) is defined by

\[
\textit{Rel}(\tau_1, \ldots, \tau_n) \triangleq \mathcal{P}(\tau_1 \times \cdots \times \tau_n).
\]

Thus the elements of \(\textit{Rel}\) are relations. In case of a unary relation we use \(\textit{Set}(\tau)\) rather than \(\textit{Rel}(\tau)\). The type \(\textit{Trans}\) is only defined for non-basic types \(\tau_1\) and \(\tau_2\):

\[
\textit{Trans}(\tau_1, \tau_2) \triangleq \tau_1 \rightarrow \tau_2
\]

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

\textbf{Definition A.1 (Relations)}

Relations are defined inductively as follows.

- Let \(\chi\) 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 \(\textit{Rel}(\tau_1, \ldots, \tau_n)\).

- Let \(R\) be a relation of type \(\textit{Rel}(\tau_1, \ldots, \tau_n)\), then

  - the complement \(-R \triangleq \overline{R}\) is a relation of type \(\textit{Rel}(\tau_1, \ldots, \tau_n)\).
  - the converse \(R^{-1}\) is a relation of type \(\textit{Rel}(\tau_n, \ldots, \tau_1)\).

- Let \(R_1\) and \(R_2\) be relations of type \(\textit{Rel}(\tau_1, \ldots, \tau_n)\), then the union \(R_1 \lor R_2 \triangleq R_1 \cup R_2\) and the intersection \(R_1 \land R_2 \triangleq R_1 \cap R_2\) are relations of type \(\textit{Rel}(\tau_1, \ldots, \tau_n)\).

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

- A special case of the previous definition is the relational image \(R[\{S\}]\), which abbreviates \(S \circ R\). We employ this abbreviation when \(S\) is a unary relation, i.e. a set. So if \(R\) is a relation of type \(\textit{Rel}(\tau_1, \tau_2, \ldots, \tau_n)\) and \(S\) is of type \(\textit{Set}(\tau_1)\), then the relational image \(R[\{S\}]\) is a relation of type \(\textit{Rel}(\tau_2, \ldots, \tau_n)\).

- Let \(T\) be a transformer of type \(\tau_1 \rightarrow \tau_2\), and let \(R\) be a relation of type \(\tau_1\) then \(T(R)\) is a relation of type \(\tau_2\).
Notice the notation $R_1 \circ R_2$ for composition of relations. Although we do not present any particular programming language here, we denote sequential composition of programs by $S_1 \cdot S_2$. As explained above, the two operations coincide only for the case of sequential programs, where $S_1 \cdot S_2$ (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 $\tau_1$ and $R$ be of type $\tau_2$, possibly with free occurrences of $X$, then $\lambda X.R$ is a transformer of type $\tau_1 \rightarrow \tau_2$.

- Let $T_1$ and $T_2$ be transformers of type $\tau_1 \rightarrow \tau_2$, then
  - $T_1 \land T_2$ and $T_1 \lor T_2$, respectively defined by $(T_1 \land T_2)(X) \triangleq T_1(X) \land T_2(X)$ and $(T_1 \lor T_2)(X) \triangleq T_1(X) \lor T_2(X)$, are transformers of type $\tau_1 \rightarrow \tau_2$.
  - Likewise, $\neg T_1$ is a transformer of type $\tau_1 \rightarrow \tau_2$.

- Let $T_1$ be a transformer of type $\tau_1 \rightarrow \tau_2$ and $T_2$ be a transformer of type $\tau_2 \rightarrow \tau_3$, then $T_2 \circ T_1$ is a transformer of type $\tau_1 \rightarrow \tau_3$.

- Let $T$ be a transformer of type $\tau_1 \rightarrow \tau_2$ then the adjoint transformer $T^!$ is defined by

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

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

$\Box$

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 $\mathcal{S}et(\tau_1)$.

$T^!(X)$ 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.

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

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

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

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 $\mathcal{R}(\tau_1, \ldots, \tau_{n+1})$. Furthermore, let $X$ resp. $Y$ be a variable of type $\mathcal{R}(\tau_1, \ldots, \tau_n, \tau_{n+2}, \ldots, \tau_{n+m})$ resp. $\mathcal{R}(\tau_{n+1}, \ldots, \tau_{n+m})$. We define the so called 'leads to' transformer $\lambda X.R \leadsto X$ of type $\mathcal{R}(\tau_1, \ldots, \tau_n, \tau_{n+2}, \ldots, \tau_{n+m}) \to \mathcal{R}(\tau_{n+1}, \ldots, \tau_{n+m})$ as the adjoint of $R \circ Y$, i.e.

$$\lambda X.R \leadsto X \triangleq (\lambda Y.R \circ Y)^{\dagger}.$$ 

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

- If $R$ is of type $\mathcal{S}(\tau_1)$ and $S$ of type $\mathcal{S}(\tau_2)$, then $R \leadsto S$ is the 'leads to' relation of type $\mathcal{R}(\tau_1, \tau_2)$. If we interpret $R$ and $S$ as precondition and postcondition of a Hoare style formula, then $R \leadsto S$ is the largest program (i.e. relation) that satisfies that Hoare formula.

- If $R$ is of type $\mathcal{R}(\tau_1, \tau_2)$ and $S$ of type $\mathcal{R}(\tau_1, \tau_3)$, then $R \leadsto S$ is the weakest post specification $S/R$ of Hoare and He [HH87] of type $\mathcal{R}(\tau_2, \tau_3)$, (take $n = 1$ and $m = 2$ in the above definition.)

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

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

The following special cases may be more familiar.

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

- And if $R$ is of type $\mathcal{R}(\tau_2, \tau_3)$ and $S$ is of type $\mathcal{R}(\tau_1, \tau_3)$, then $[R]S$ is the weakest prespecification $R \backslash S$ of [HH87] of type $\mathcal{R}(\tau_1, \tau_2)$.

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

$$\langle S \lor T \rangle = \langle S \rangle \lor \langle T \rangle$$

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

$$[S \Diamond T] = [S] \lor [T]$$.
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.

\[
\begin{align*}
[x := a \Diamond x := b](x = a) & \equiv true \\
[x := a \Diamond x := b](x = b) & \equiv true
\end{align*}
\]  

From (4) it follows immediately that $[x := a \Diamond x := b] \subseteq \{(s_0, s_1) | s_1(x) = a\}$, and likewise from (5) it follows that $[x := a \Diamond x := b] \subseteq \{(s_0, s_1) | s_1(x) = b\}$. 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 $\diamond S$.

**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 $\text{Rel}(\tau_1, \ldots, \tau_n)$, 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 $F_1 \land F_2$ are formulae with the obvious interpretation.

- Let $X$ be a variable of type $\text{Rel}(\tau_1, \ldots, \tau_n)$ 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 $\text{Rel}(\tau_1, \ldots, \tau_n)$, and false otherwise.

$\square$

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 $\text{Rel}(\tau_1, \ldots, \tau_n)$, $\text{Rel}(\tau_n, \ldots, \tau_{n+m})$, and $\text{Rel}(\tau_{n+1}, \ldots, \tau_{n+m})$ we define generalized Hoare formulae

\[
(R_1) \quad R_2 (R_3) \triangleq R_1 \triangleright R_2 \subseteq R_3.
\]

In case that $R_1$, $R_2$, and $R_3$ resp. are of type $\text{Set}(\tau_1)$, $\text{Rel}(\tau_1, \tau_2)$, and $\text{Set}(\tau_2)$, then $(R_1) \quad R_2 (R_3)$ coincides with the classical Hoare-style correctness formula $\{R_1\} \quad R_2 \{R_3\}$ (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

\[
R \triangleright C \subseteq A \triangleright R.
\]
Following [HHS87] we define the transformer $F_R$ by

$$F_R(X) \triangleq R \rightsquigarrow (X \triangleright R).$$

From the definition of $\rightsquigarrow$ it follows that

$$R \triangleright F_R(A) \subseteq A \triangleright R.$$

Thus $F_R(A)$ is the largest relation that downward simulates $A$ with respect to $R$. Because, in the prespecification calculus $\triangleright$ coincides with $\triangleright$, 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


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


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
A systolic design for dynamic programming.
P.Struik

89/3 H.M.M. Ten Eikelder
Some category theoretical properties related to
C. Hemerik
a model for a polymorphic lambda-calculus.

89/4 J.Zwiers
Compositionality and modularity in process
W.P. de Roever
specification and design: A trace-state based
approach.

89/5 Wei Chen
Networks of Communicating Processes and their
T.Verhoef P.Struik
(De-)Composition.
J.T.Udding

89/6 T.Verhoef
Characterizations of Delay-Insensitive
P.Struik
Communication Protocols.

89/7 A systematic design of a parallel program for
P.Struik
Dirichlet convolution.

89/8 E.H.L.Aarts
A general theory of genetic algorithms.
A.E.Eiben
K.M. van Hee

89/9 K.M. van Hee
Discrete event systems: Dynamic versus static
P.M.P. Rambags
topology.

89/10 S.Ramesh
A new efficient implementation of CSP with output
S.Ramesh
guards.

89/11 S.Ramesh
Algebraic specification and implementation of infinite
Algebraic specification and implementation of infinite
processes.

89/12 A.T.M.Aerts
A concise formal framework for data modeling.
K.M. van Hee

89/13 A.T.M.Aerts
A program generator for simulated annealing
K.M. van Hee
problems.
M.W.H. Hesen

89/14 H.C.Haesen
ELDA, data manipulatie taal.

89/15 J.S.C.P. van
Optimal segmentations.
der Woude

89/16 A.T.M.Aerts
Towards a framework for comparing data models.
K.M. van Hee

89/17 M.J. van Diepen
A formal semantics for Z and the link between
K.M. van Hee
Z and the relational algebra.

<table>
<thead>
<tr>
<th>No.</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>90/2</td>
<td>K.M. van Hee P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/5</td>
<td>J.A. Brzozowski J.C. Ebergen</td>
<td>On the delay-sensitivity of gate networks, p. 23.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/9</td>
<td>A.T.M. Acrts P.M.E. De Bra K.M. van Hee</td>
<td>Combining the functional and the relational model, p. 15.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P.America F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
<tr>
<td>90/16</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>A fully abstract model for concurrent logic languages, p. 23.</td>
</tr>
<tr>
<td>90/17</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if...,then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of P-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E. Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.</td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem, p. 12.</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built, p. 63.</td>
</tr>
</tbody>
</table>
91/15 A.T.M. Aerts  
K.M. van Hee  

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  

91/21 J. Coenen  
W.-P. de Roever  
J.Zwiers  
Assertional Data Reification Proofs: Survey and Perspective, p. 18.

91/22 G. Wolf  

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. Hoorman  
R. Kuiper  
A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, 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 process 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.
<table>
<thead>
<tr>
<th>Code</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/31</td>
<td>H. ten Eikelder</td>
<td>Some algorithms to decide the equivalence of recursive types, p. 26.</td>
</tr>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
</tbody>
</table>