Methodological Aspects of Action Refinement*

Arend Rensink

Institut für Informatik, University of Hildesheim
Postfach 101363, D–31113 Hildesheim
e-mail: rensink@informatik.uni-hildesheim.de

The principle of action refinement refers to the implementation of abstract actions through more complex, concrete processes. In this paper we study the fundamental question how to use this principle in process algebraic system design. We formulate a methodological framework to express design under action refinement, and present two applications of this framework. The first application corresponds to the well-known interpretation of action refinement as an operator in the algebra, requiring a non-interleaving semantics; the second application is new, and results in a much more flexible notion of design, which is moreover compatible with the standard interleaving semantics.

Keyword Codes: D.3.1; F.3.1; F.3.3
Keywords: Programming Languages, Formal Definitions and Theory; Specifying and Verifying and Reasoning about Programs; Studies of Program Constructs


1. INTRODUCTION

It is a common statement in papers on action refinement that its basic principle, replacing abstract actions with more complex concrete behaviour, intuitively corresponds to the ideas of top-down design. However, the question how exactly action refinement is to be integrated in a design methodology has hardly been addressed so far, and hence this intuitive correspondence has not been realised in practice.

In current methodologies, formal design is based on a binary implementation relation (in fact, a preorder) between specifications and their implementations, such that an implementation is “correct,” in whatever sense one is interested in, if and only if it is related to the specification. In other words, the correctness criterion is encoded in the implementation relation. Naturally one would expect a similar arrangement when the implementation is obtained by action refinement. Immediately the problem arises that action refinement typically brings a change of alphabet, since abstract actions are replaced by more concrete actions; hence none of the usual implementation relations are appropriate, since they all imply trace inclusion (see Van Glabbeek [12,13]). In fact, it is obvious that the actual mapping from abstract actions to concrete behaviour will have an overriding influence on

*Based on Rensink [24, Chapters 4 and 6]. This work was initiated while the author was employed at the University of Twente, and was partially supported by the Esprit BRA Project 3096 SPEC (Formal Methods and Tools for the Development of Distributed and Real Time Systems) and the Esprit Working Group 6067 CALIBAN (Causal Calculi Based on Nets)
the correctness criterion—implementation can only be considered correct under a given mapping—and hence information about the mapping must somehow be an integral part of the implementation relation.

In Section 2 we introduce ternary relations, called vertical implementation relations, to deal with design under action refinement: the third component is the relevant refinement mapping. This concept is quite general and makes no assumptions whatsoever about the actual correctness criterion, that is, the nature of design by action refinement. We then proceed to define two actual vertical implementation relations, on the basis of a language introduced in Section 3. The first application (Section 4) is based on the by now standard substitution-like operator for action refinement (which we call expansion to avoid a too intensive overloading of the term refinement). The idea is that a system can be implemented by expanding it according to a given refinement function. It turns out that this is well-defined if and only if the systems are interpreted up to a congruence with respect to expansion. Since this is exactly the property that has been studied intensively in the literature (some good references are [1,10,14,19,25]), we feel justified in stating that the standard work on action refinement fits into our framework.

The second application (Section 5) is inspired by the principles of data refinement. The idea is that the user or environment of a system is expanded rather than the system itself, intuitively because the operations invoked by the user are implemented by procedures. The system itself, being the virtual machine executing the invoked actions, should then be designed in such a way that the interaction between the expanded user and the implementation is equivalent to the interaction between the original, unexpanded user and the specification.

These two approaches are illustrated using a common example. It is shown that the first approach suffers from two drawbacks: it is not compatible to the interleaving semantics (due to the congruence problem mentioned above) and it is not very flexible (due to the fact that it is completely constructive). The second approach allows some useful implementations that are impossible to derive using expansion. Conclusions and ideas for further work are collected in Section 6.

Because the approach presented in this paper substantially differs from the usual treatment of action refinement, we have preferred explanatory text over theory. The results are given without formal proof. The theory (which in point of fact is not very deep) has been worked out in detail in Rensink [21].

2. VERTICAL IMPLEMENTATION RELATIONS

We will consider what it means to implement a system under action refinement. In particular we will develop a methodological framework in which to express such implementation steps in general, without tying ourselves down to a particular notion of refinement.

Throughout this paper, we assume that systems are described by terms \( B, I, S \) (for Behaviour, Implementation and Specification, respectively) in some process algebra \( L \). According to standard methodology, the notion of implementation is formalised by a preorder \( \preceq \subseteq L \times L \) (in many circumstances actually an equivalence) such that a system described by \( I \in L \) implements a system described by \( S \in L \) if and only if \( I \preceq S \). (There is no consensus in the literature about the “direction” of implementation relations: we
choose to write the implementation on the left hand side of the relation symbol.) The
transitivity of the preorder ensures that design steps can be composed: if a sequence of
$n$ design steps results in a series of implementations $I_n \preceq I_{n-1} \preceq \cdots \preceq I_1 \preceq S$, then also
$I_n \preceq S$, hence we still have an implementation of $S$.

The preorders investigated in the literature are however inadequate for dealing with
implementations where the alphabet of the term changes. Since this is typically what
happens during action refinement, it is clear that we have to adapt this standard frame-
work. Instead of expressing implementation under action refinement as a simple binary
relation, somehow information about the refinement itself, in particular which actions are
refined and into what, should be taken into account. A natural choice is to include this
information as an index to the relation. For this purpose we assume a universe $r$ of
refinements, ranged over by $r$; implementation will then be represented by a ternary rather
than a binary relation $\preceq \subseteq r \times L \times L$, usually expressed as an indexed family $(\preceq^r)_r$ of
binary relations. We call such an object a \textit{vertical implementation relation}. In contrast,
an implementation relation of the ordinary kind is sometimes called \textit{flat}.

We have not yet stated what the objects in $r$ are, except that they somehow represent
refinements. Typically (albeit by no means necessarily) we may think of them as \textit{functions}
from a domain of actions, such that for all $a$ in the domain, $r(a)$ represents the concrete
behaviour that implements $a$. In this paper, the elements of $r$ will be functions $A \to L$,
henceforth called \textit{refinement functions}.

The composition of design steps changes accordingly. Instead of requiring transitivity
of the relations $\preceq^r$, as in standard implementation relations, we require that a design step
based on a refinement $r_1 \in r$, followed by a design step based on another refinement $r_2 \in r$,
corresponds to a single design step based on $r_2 \cdot r_1$, where the latter denotes \textit{composition}
of refinements (where the order of application is from right to left). One final assumption
about $r$ is that it contains an object $1$ representing the \textit{identity refinement}, which does
not really chance the actions at all; hence $\preceq^1$ is expected to be a standard implementation
relation and $1$ is expected to be a unit with respect to composition, i.e., $r \cdot 1 = 1 \cdot r = r$
for all $r \in r$. This brings us to the following formal definition.

\begin{definition}
A \textit{vertical implementation relation} is family $(\preceq^r)_r$ of binary relations over
$r$ such that $\preceq^1$ is reflexive and for all $r_1, r_2 \in r$, $\preceq^{r_2} \circ \preceq^{r_1} \subseteq \preceq^{r_2 \cdot r_1}$ (where $\circ$ denotes
relation composition).
\end{definition}

Note that since composition of relations is associative, for the well-definedness of this
\textit{generalised transitivity} property it is necessary that composition of refinement functions
is also associative; in other words, $(r, \cdot, 1)$ is a monoid.

\begin{definition}
A \textit{refinement universe} is a monoid $(r, \cdot, 1)$.
\end{definition}

Note that since $1$ is a unit of $r$, the corresponding relation $\preceq^1$ is indeed transitive, i.e., a
preorder: $\preceq^1 \circ \preceq^1 \subseteq \preceq^{1 \cdot 1} = \preceq^1$. We will call $\preceq^1$ the (flat) \textit{basis} of $(\preceq^r)_r$ and often drop the
index. The terms \textit{vertical} and \textit{flat} are based on the intuition that action refinement \textit{lowers the level of abstraction}; this suggests that the level of abstraction is a new parameter in
the design methodology that is at “right angles” to the ordinary direction of design. This
idea is expressed graphically in Figure 1.
Vertical implementation relations extend flat implementation relations in the sense that the latter are retained as a special case (the basis). In fact the extension is conservative in the sense that the sets of r-implementations respectively r-specifications of a given system B do not provide additional distinguishing power along the basis:

\[ B_1 \preceq B_2 \iff (\forall r. \{ I \mid I \preceq B_1 \} \supseteq \{ I \mid I \preceq B_2 \}) \]

Note that for nontrivial refinement functions, \( \preceq \) is in general asymmetric and irreflexive, and unlike for preorders, the symmetric closure does not yield a useful notion. We reserve the term vertical equivalence for vertical implementation relation where just the basis is symmetric.

In [24] we have developed the theory of vertical implementation relations somewhat further. For this paper, the only remaining notions of interest are two classes of vertical implementation relations that can be represented by (partial) functions between abstraction levels, either transforming a given specification into an r-implementation of it or vice versa. The first case corresponds to a top-down design strategy, the second to bottom-up. If such a function can be formulated constructively, it offers a clear advantage. In the following, \( L \rightarrow_s L \) denotes the space of partial functions from \( L \) to \( L \).

**3 Definition.** Let \( (\preceq_r) \) be a vertical implementation relation.

- \( (\preceq_r) \) is called top-down if there exists a mapping \( D: r \rightarrow (L \rightarrow_s L) \) such that for all \( r \in r \), \( I \preceq_r S \) if and only if \( I \preceq D(r)(S') \preceq_r S' \preceq S \) for some \( S' \).
- \( (\preceq_r) \) is called bottom-up if there exists a mapping \( U: r \rightarrow (L \rightarrow_s L) \) such that for all \( r \in r \), \( I \preceq_r S \) if and only if \( I \preceq I' \preceq U(r)(I') \preceq S \) for some \( I' \).

Hence with a top-down vertical implementation relation, given a specification \( S \) and a refinement \( r \) there is a characteristic (\( \preceq \)-maximal) implementation, viz. \( D(r)(S) \): apart from choosing \( r \), no further design choices are involved. On the other hand, in the bottom-up case, the correctness of the design can be verified a posteriori, that is, a given implementation \( I \) can be proved correct by mapping it onto its characteristic (\( \preceq \)-smallest) specification, \( U(r)(I) \), which should then \( \preceq \)-implement \( S \); but in general there is more
than one \( I \) for which this holds, and even if \( \mathcal{U}(I_i) \preceq S \) for \( i = 1, 2 \) then in general \( I_1 \) and \( I_2 \) are \( \preceq \)-incomparable; hence apart from choosing \( r \), a further design choice has to be made by selecting an \( I_i \). The latter allows for greater design freedom and hence appears an advantage, even if bottom-up design in itself does not conform to current wisdom.

3. THE LANGUAGE

The study in this paper is undertaken on the basis of a process algebraic language \( \mathbb{L} \), based on a collection of operators from CCS [22], CSP [8] and LOTOS [5], and generated by the following grammar:

\[
B ::= \text{stop} | a | \tau | B \parallel B | B; B | B[A] | B[\varphi] | X .
\]

Here \text{stop} is a constant denoting deadlock; \( a \in \mathbb{A} \) denotes an action, \( \tau \notin \mathbb{A} \) an invisible activity of the system; we use \( \mu \) to range over \( \mathbb{A} \cup \{\tau\} \). The choice between two terms \( B_1 \) and \( B_2 \) is denoted \( B_1 \parallel B_2 \), whereas \( B_1 ; B_2 \) denotes their sequential composition. \( B_1 [A] B_2 \) denotes parallel composition, where the actions in \( A \subseteq \mathbb{A} \) take place in \( B_1 \) and \( B_2 \) at the same time, in other words, constitute synchronisation actions. \( B_1 || B_2 \) will be used to abbreviate \( B_1 [\emptyset] B_2 \), i.e., parallel composition without synchronisation. \( B[\varphi] \) denotes the renaming of \( B \) according to a function \( \varphi : \mathbb{A} \rightarrow (\mathbb{A} \cup \{\tau\}) \); we use hide \( A \) in \( B \) to abbreviate \( B[\lambda a \mid a \in A \land \tau \notin A] \). Finally, \( X \in \mathbb{X} \) denotes a process name, used to specify infinite behaviour and interpreted according to an implicit process environment \( \theta : \mathbb{X} \rightarrow \mathbb{L} \). We also write \( X ::= \theta \ B \) for \( \theta(X) = B \).

To deal with sequential composition we need an auxiliary constant \text{skip} denoting successful termination. On the basis of this constant we define a (postfix) termination predicate \( \checkmark \in \mathbb{L} \) as the smallest such that

- \( \text{skip} \checkmark \);
- if \( B_1 \checkmark \) and \( B_2 \checkmark \) then \( (B_1 [A] B_2) \checkmark \) for all \( A \subseteq \mathbb{A} \);
- if \( B \checkmark \) then \( B[\varphi] \checkmark \) for all \( \varphi \).

The standard semantics of \( \mathbb{L} \) is given in the form of transition rules in Table 1. We also use the usual “double arrow” relation defined for all \( n \geq 0 \) by

\[
B \xrightarrow{a_1 \ldots a_n} B' \iff B \xrightarrow{\tau} \xrightarrow{a_1} \xrightarrow{\tau} \cdots \xrightarrow{\tau} \xrightarrow{a_n} \xrightarrow{\tau} B' .
\]

Since we will not always use the standard semantics in this paper, in addition we define a notion of strong congruence, being the smallest congruence relation over \( \mathbb{L} \) satisfying the axioms in Table 2. This relation will serve as a “bottom line” in the sense that we expect every equivalence relation to satisfy these axioms; hence we are always safe in interpreting \( \mathbb{L} \) up to \( \equiv \). It is for instance easy to check that transition system isomorphism (extended to deal with the termination predicate \( \checkmark \)) is (strictly) coarser than \( \equiv \).

4. REFINEMENT BY EXPANSION OF THE SYSTEM

Now we come to concrete applications of the framework introduced in Section 2; that is, we will define several actual vertical implementation relations and investigate them in terms of the properties discussed in general above.
Table 1: Standard operational semantics of $L$

<table>
<thead>
<tr>
<th>$\mu \in A \cup {\tau}$</th>
<th>$\mu \xrightarrow{\varphi} \text{skip}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1 \xrightarrow{\mu} B'$</td>
<td>$B_1 \parallel B_2 \xrightarrow{\mu} B'$</td>
</tr>
<tr>
<td>$B_2 \xrightarrow{\mu} B'$</td>
<td>$B_1 \parallel B_2 \xrightarrow{\mu} B'$</td>
</tr>
<tr>
<td>$B_1 \xrightarrow{\mu} B'_1 \neg B'_1 \checkmark$</td>
<td>$B_1 ; B_2 \xrightarrow{\mu} B'_1 ; B_2$</td>
</tr>
<tr>
<td>$B_1 \xrightarrow{\mu} B'_1$</td>
<td>$B_1 ; B_2 \xrightarrow{\mu} B_2$</td>
</tr>
<tr>
<td>$B_1 \xrightarrow{\mu} B'_1$</td>
<td>$B_2 \xrightarrow{\mu} B'_2 \mu \notin A$</td>
</tr>
<tr>
<td>$B_2 \xrightarrow{\mu} B'_2 \mu \notin A$</td>
<td>$B_1 \parallel [A] B_2 \xrightarrow{\varphi} B_1 \parallel [A] B_2$</td>
</tr>
<tr>
<td>$B_1 \xrightarrow{\mu} B'_1$</td>
<td>$B_2 \xrightarrow{\mu} B'_2 \mu \in A$</td>
</tr>
<tr>
<td>$B \xrightarrow{\mu} B'$</td>
<td>$B[\varphi] \xrightarrow{\varphi(\mu)} B'[\varphi]$</td>
</tr>
<tr>
<td>$\theta(X) \xrightarrow{\mu} B$</td>
<td>$X \xrightarrow{\mu} B$</td>
</tr>
</tbody>
</table>

The first of these is based on the expansion operator introduced first by Van Glabbeek and Goltz; see e.g., [15]. (This is non-standard terminology: traditionally both this operator and the associated notion of design were called refinement. Since we separate the two, and would like to keep the term refinement for design-related notions, we choose to refer to the operator as expansion.) In its conception, expansion was intended primarily as a criterion to generate nonstandard semantics by investigating coarsest congruences, and indeed, as mentioned in the introduction, this “congruence question” has been studied in depth, and it also turns up in this application of our framework.

4.1. Refinement functions and expansion

In this and the next section, refinements $r \in r$ correspond to functions $A \rightarrow L$, where the $a \in A$ are thought of as abstract actions and $r(a)$ as the corresponding concrete behaviour. The empty refinement $1 \in r$ corresponds to the identity function over $A$. Since actions are also terms, refinement functions generalise the renaming functions of $L$.

A term of the form $B[r]$ where $B \in L$ and $r \in r$ denotes the expansion of $B$ according to $r$. Expansion corresponds to some sort of substitution driven by the mapping $r$, where however simple syntactic substitution is not always satisfactory (see [16] for an extensive discussion). Table 3 gives a number of properties for expansion which are assumed to hold under any reasonable equivalence.

It is a well-known fact that the standard (interleaving) semantics is not adequate

Table 2: Laws of strong equivalence

<table>
<thead>
<tr>
<th>$B \parallel \text{stop}$</th>
<th>$\equiv B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1 \parallel B_2$</td>
<td>$\equiv B_2 \parallel B_1$</td>
</tr>
<tr>
<td>$B_1 \parallel (B_2 \parallel B_3)$</td>
<td>$\equiv (B_1 \parallel B_2) \parallel B_3$</td>
</tr>
<tr>
<td>$B_1 ; (B_2 ; B_3)$</td>
<td>$\equiv (B_1 ; B_2) ; B_3$</td>
</tr>
<tr>
<td>$B_1 \parallel [A] B_2$</td>
<td>$\equiv B_2 \parallel [A] B_1$</td>
</tr>
<tr>
<td>$X$</td>
<td>$\equiv \theta(X)$</td>
</tr>
</tbody>
</table>
Table 3: Strong equivalence for expansion

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>stop[r] ≡ stop</td>
<td>μ[r] ≡ r(μ) (μ ∈ A U {r})</td>
</tr>
<tr>
<td>B[r₁][r₂] ≡ B[r₂ ∗ r₁]</td>
<td></td>
</tr>
</tbody>
</table>

for modelling expansion compositionally: for instance, the terms a || b and a; b || b; a are isomorphic in the standard semantics, whereas their expansions under a → a₁; a₂ correspond (modulo ≡) to a₁; a₂ || b and a₁; a₂; b || b; a₁; a₂, respectively, which are not even trace equivalent. In other words, interleaving equivalences are not congruent with respect to refinement. Whether or not this is a problem depends on how one intends to use expansion, as we will see in the course of this paper.

In order to apply the theory of vertical implementation relations we need a refinement universe in the sense of Definition 2, meaning that we have to define an associative composition of refinement functions with respect to which 1 is a unit. Ordinary function composition is not appropriate since the domain and codomain of refinement functions are dissimilar. Instead we define

\[ r₂ ∗ r₁ = λa. (r₁(a))[r₂] \]  

To get a refinement universe we now interpret refinement functions up to ≡ (where \( r₁ ≡ r₂ \) if \( r₁(a) ≡ r₂(a) \) for all \( a ∈ A \)).

4 Proposition. \( (r, ∗, 1) \) is a monoid modulo ≡.

4.2. Vertical design by expansion

A given preorder \( ≤ S ⊆ L × L \) satisfying the ≡-laws in Tables 2 and 3 can be extended in the vertical dimension by defining for all \( r ∈ r \)

\[ I ≤ S :⇔ I ≤ S[r] \]  

According to this definition, the way to implement a given behaviour vertically under \( r \) is expanding it. This notion of vertical design is therefore constructive in the sense that given \( S \) and \( r \), the vertical design problem can be reduced to a flat design problem; in fact we will show below that \( (≤ S^r) \) is top-down in the sense of Definition 3. But first we have to establish whether it is a vertical implementation relation at all, in the sense of Definition 1.

5 Theorem. \( (≤ S^r) \) as defined in (2) is a vertical implementation relation if and only if for all \( B₁, B₂ ∈ L \) and \( r ∈ r \), \( B₁ ≤ B₂ \) implies \( B₁[r] ≤ B₂[r] \).

In other words, in this interpretation of vertical design, congruence with respect to expansion is derived as a necessary side-condition.

It is not difficult to see that any vertical implementation relation defined according to (2) will be top-down in the sense of Definition 3. In fact the mapping \( D: r → (L → L) \)
required in the definition precisely corresponds to expansion itself: \( D(r) = \lambda S, S[r] \). It follows that the top-down nature of \((\preceq^r)\), is not even dependent on the actual definition of the expansion operator: the mere fact that it is based on an operator of some kind immediately implies it will be top-down.

4.3. Example: implementation of a buffer

To demonstrate this kind of vertical design, we take an example from Langerak [20] concerning the implementation of a buffer. For the flat implementation relation \( \preceq \) that is extended vertically we choose \( \equiv \). The set \( A \) contains actions \( \text{wr}_x \) and \( \text{rd}_x \) for all \( x \in D \), where \( D \) is the set of data values that can be buffered. \( \text{wr}_x \) denotes the action of writing (i.e. inserting) \( x \) into the buffer, and \( \text{rd}_x \) denotes reading (removing) \( x \) from the buffer. Moreover there are actions \( \text{wg}_1, \text{wg}_2, \text{rg}_1, \text{rg}_2 \in A \) serving as guards; see below. Now in [20] approximately the following specification is developed (\( \Sigma \) denotes a generalised version of the choice operator):

\[
\begin{align*}
\text{Cell} & := \emptyset \text{wg}_1; \Sigma_{x \in D} \text{wr}_x; (\text{wg}_2 || \text{rg}_1; \text{rd}_x; \text{rg}_2) \\
\text{Chain} & := \emptyset \\text{hide} \ \text{wg}_2, \text{rg}_2 \ \text{in} \ \text{Cell} \ [(\text{wg}_2, \text{rg}_2)] \ \text{Chain}[\text{wg}_1 \mapsto \text{wg}_2, \text{rg}_1 \mapsto \text{rg}_2] \\
\text{Buf} & := \emptyset \\text{hide} \ \text{wg}_1, \text{rg}_2 \ \text{in} \ \text{Chain}
\end{align*}
\]

\( \text{Cell} \) describes a buffer cell which allows writing a value and then reading it. The actions \( \text{wg}_1 \) and \( \text{rg}_1 \) ensure that the cell awaits its turn to write resp. read; \( \text{wg}_2 \) and \( \text{rg}_2 \) signal that the writing resp. reading has been done and therefore the next cell may start. In \( \text{Chain} \) an unbounded number of such cells are put in parallel, synchronising in such a way that the “end guards” \( \text{wg}_2 \) and \( \text{rg}_2 \) of one cell synchronise with the “begin guards” \( \text{wg}_1 \) and \( \text{rg}_1 \) of the next. Finally, \( \text{Buf} \) “turns off” the very first guards by hiding them; hence the first cell in the chain can start immediately. In Figure 2, the left half depicts the partial order behaviour of \( \text{Cell} \) and \( \text{Buf} \); the nodes represent actions (the unlabelled, open nodes being \( \tau \)-actions) and the arrows causal relationships. The right half shows the standard semantics of \( \text{Buf} \), where again we have left out the \( \tau \)-labels.

Now we want to implement this buffer in the following manner: instead of putting each data value in a single buffer cell, we use two cells (for instance because the values \( x \in D \) are too large). For this purpose we assume that for all \( x \in D \) we have unique values \( y \in D_1 \) and \( z \in D_2 \) such that \( x = yz \), and we have new actions \( \text{wr}_y, \text{wr}_z, \text{rd}_y, \text{rd}_z \) for all \( y \in D_1, z \in D_2 \) (where \( D_1 \) and \( D_2 \) are assumed to be disjoint). The proposed design step can then be characterised by the following refinement function \( r \):

\[
\begin{align*}
\text{wr}_y & \mapsto \text{wr}_y; \text{wr}_z \\
\text{rd}_y & \mapsto \text{rd}_y; \text{rd}_z
\end{align*}
\]

This gives rise to an implementation \( \text{Dbl}_1 \preceq^r \text{Buf} \) such that \( \text{Dbl}_1 \equiv \text{Buf}[r] \). By applying the \( \equiv \)-distribution laws for expansion in Table 3 it can be proved that \( \text{Dbl}_1 \) equals \( \text{Buf} \) above after replacing \( \text{Cell} \) with

\[
\begin{align*}
\text{Cell}_1 & := \emptyset \text{wg}_1; \Sigma_{y \in D_1, z \in D_2} \text{wr}_y; \text{wr}_z; (\text{wg}_2 || \text{rg}_1; \text{rd}_y; \text{rd}_z; \text{rg}_2)
\end{align*}
\]

This behaviour is depicted in Figure 3. Note that in \( \text{Dbl}_1 \), data values are written and read two at a time. In particular it is not possible to read the first part \( y \) of a data value \( yz \) before the second part \( z \) has been written, whereas intuitively there would be nothing
Figure 2: Partial order and interleaving behaviour of a parallel buffer specification

Cell:

\[ \begin{align*}
&rg_1 \quad \text{rd}_x \quad rg_2 \\
&wg_1 \quad wr_x \quad wg_2 
\end{align*} \]

Buf:

\[ \begin{align*}
&\text{wr} \quad \text{rd} \\
&Dbl_{2} := \emptyset \quad wg_1; \sum_{y \in D_1, z \in D_2} \text{wr}_y; (\text{wr}_z; wg_2 \parallel \text{rg}_1; \text{rd}_y); \text{rd}_z; \text{rg}_2 ,
\end{align*} \]

Intuitively also implements Buf; nevertheless it cannot be generated using expansion. (Recently a more flexible notion of expansion has been developed in Wehrheim [26], where behaviour similar to Figure 4 can be obtained if we explicitly state that the \text{rd}_y- and \text{wr}_z- actions are independent.)

It is interesting to note that \textit{Dbl}_1 and \textit{Dbl}_2 are not even trace equivalent. Hence no top-down vertical implementation relation will be able to derive both \textit{Dbl}_1 and \textit{Dbl}_2 as implementations of Buf.

As a final remark, note that if we had started out with a different buffer specification, not \(\equiv\)-equivalent to Buf, then the expansion would generally have been quite different.

Figure 3: Partial order and interleaving behaviour of a parallel buffer implementation

\[ \begin{align*}
&Dbl_{1} \\
&\text{wr} \quad \text{rd} \\
&Dbl_{2} := \emptyset \quad wg_1; \sum_{y \in D_1, z \in D_2} \text{wr}_y; (\text{wr}_z; wg_2 \parallel \text{rg}_1; \text{rd}_y); \text{rd}_z; \text{rg}_2 ,
\end{align*} \]
also. For instance, consider a set of process variables $Buf_{\sigma} \in X$ for all strings $\sigma \in D^*$ (where $\varepsilon$ denotes the empty string), defined as follows:

$$Buf_{\varepsilon} := \emptyset \quad Buf_{x} := \emptyset \quad \Sigma_{x \in D} wr_x; Buf_{x}$$

$$Buf_{x's} := \emptyset \quad rd_x; Buf_{x} \quad \Sigma_{x \in D} wr_x; Buf_{x's}$$

Now $Buf \approx Buf_{\varepsilon}$ (where $\approx$ denotes observation congruence, see e.g. Milner [22]), but $Buf \not\approx Buf_{v}$, and the corresponding implementation $Dbl_{\varepsilon} \not\approx Buf_{\varepsilon}$, given by $Buf_{\varepsilon}[v]$ and depicted in Figure 5, is not observation congruent or even trace equivalent to either $Dbl_1$ or $Dbl_2$:

$$Dbl_{\varepsilon} := \emptyset \quad \Sigma_{y \in D_1, z \in D_2} wr_y; wr_z; Dbl_{yz}$$

$$Dbl_{v's} := \emptyset \quad rd_y; rd_z; Dbl_{v} \quad \Sigma_{y \in D_1, z \in D_2} wr_y; wr_z; Dbl_{v's}$$

5. REFINEMENT BY EXPANSION OF THE ENVIRONMENT

We see several disadvantages to the refinement by expansion of the system. Firstly, as discussed in the previous section, the resulting design notion is for some purposes
too rigorous. Secondly, much of the existing process algebraic design methodology is based on the standard interleaving semantics of systems, whereas Theorem 5 vertical implementation relations generated by expansion of the system require a more concrete semantics. In this section we propose a different setup.

5.1. Data refinement

Basically, instead of expanding a system to obtain an implementation, we expand the users or environment of that system and require the system implementation to interact with the expanded users in the same way as the system itself interacts with the original users. This approach can be justified by comparing it to data refinement as known from sequential programming (see e.g. Hoare [18]). Consider the diagram in Figure 6. Behaviour is modelled by transition systems where the states are functions mapping program variables to their current values and the transitions are simple statements of the language, which in the concrete systems are turned into procedures. abstraction is a binary relation between concrete states and abstract states. An implementation is considered correct if the diagram commutes:

\[ abstraction \circ statement = procedure \circ abstraction. \]

We recall a well-known example in order to make this more explicit and to make the connection to our vertical design clearer. Consider a virtual stack machine \( S \) over a set \( X \) of data values, with states \( X^* \) ranged over by \( \sigma \), and for all \( x \in X \) transitions

\[ \sigma \xrightarrow{push(x)} a \quad x\sigma \xrightarrow{pop(x)} a \quad \varepsilon \xrightarrow{pop:err} a \varepsilon. \]

This can be implemented on a concrete machine \( I \) consisting of a counter ranging over \( \mathbb{N} \) and an array ranging over the functions \( \mathbb{N} \to X \); hence with states \( \mathbb{N} \times [\mathbb{N} \to X] \) and transitions

\[
\begin{align*}
\langle n, f \rangle &\xrightarrow{rd} \langle n, f \rangle \\
\langle n, f \rangle &\xrightarrow{wr[m]} \langle m, f \rangle \\
\langle n, f \rangle &\xrightarrow{get(i):f(i)} \langle n, f \rangle \\
\langle n, f \rangle &\xrightarrow{put[i,x]} \langle n, \lambda j. \text{if } i = j \text{ then } x \text{ else } f(j) \rangle
\end{align*}
\]

The abstract operations \( push \) and \( pop \) can be implemented on this machine as procedures with the following definitions (in Pascal-like pseudo-code):

```pascal
procedure push(x: data); var n: integer;
begin
  n := rd; pdl(n + 1, x); wr(n + 1)
end

function pop: data; var n: integer;
begin
  n := rd; if n > 0 then wr(n - 1); pop := get(n) else pop := err
fi end
```
The corresponding abstraction mapping is then given by

\[ \text{abstraction: } (n, f) \mapsto f(0)f(1) \cdots f(n-1). \]

It is not difficult to see that (3) is satisfied. The question is of course what we have achieved by this. The point is the following: for the input/output-behaviour of any sequential program \( B \), it is completely irrelevant whether \( B \) runs on the abstract virtual machine where \textit{push} and \textit{pop} are elementary operations, or on the concrete machine where they are procedures. In other words, for all sequential \( B \)

\[ B[\text{\texttt{pop, push}}] S \cong B[\text{\texttt{r, wr, get, put}}] I \tag{4} \]

where we have equated procedure calls with expansion and the invoking of machine operations with synchronisation over those operations. \( \cong \) denotes input/output-equivalence.

### 5.2. Vertical bisimulation

We will use data refinement as inspiration for vertical design. For this purpose some things have to be adapted. For one thing, input/output-equivalence is not sufficient for reactive systems; indeed a large spectrum of relations has been developed to replace \( \cong \) in process equations such as (4) (cf. Van Glabbeek in [12,13]). More importantly however, even for \( \cong \) the equation (4) immediately and massively fails once we start allowing \textit{distributed} environments \( B \). For instance, for

\[ B = \text{push}(1); (n := \text{pop} \parallel m := \text{pop}) \]

we may obtain \( n = m = 1 \) on the concrete machine \( B[\text{\texttt{r, wr, get, put}}] I \), which is an impossible outcome in \( B[\text{\texttt{pop, push}}] S \). This (and similar) effects are due to the interference between the two \textit{pop}-procedures. Such interference is usually prevented by requiring that the refinements of \( \text{pop} \) and \textit{push} may not overlap at all—which would correspond to the implementation \( Dbl \) in the buffer example (Figure 5). We find this too strict and take a different approach, based solely on observational criteria.

In order to obtain something similar to (4) we have to strengthen the original criterion (3) by explicitly taking parallel programs into account. For this purpose we introduce two auxiliary relations over terms of \( L \).

\[ B \bot B' :\Leftrightarrow \exists \sigma \in A^*, t' \in L. B \not\cong t \sqsupseteq B' \land t' \cong \sigma \]

\[ B \text{ fails } t :\Leftrightarrow \exists \sigma \in A^*, B', t' \in L. B \not\cong t \sqsupseteq B' \land t' \not\cong \sigma \]

When \( B \bot B' \), meaning that \( B \) may do an (arbitrary) completed trace of \( t \), we say that the system \( B \) \textit{executes} or \textit{runs} the test \( t \), resulting in \( B' \). This relation is very similar to one defined by Boudol in [6]. On the other hand, if \( B \text{ fails } t \), meaning that \( B \) may do an initial part of \( t \) but fail to execute the remainder, we say that \( B \) \textit{deadlocks} on \( t \). This is the negation of \textit{must}-testing in De Nicola and Hennessy [11], at least for non-divergent systems. Note that \( \bot \) only requires the existence of \textit{some} completed trace of \( t \), and does not quantify over all runs. Hence \( B \) may both execute and deadlock on the same test, although on the other hand the inability to run a test implies deadlock. For example, \( a; b \overset{a}{\preceq t} \text{skip} \), \( a; b \overset{b}{\preceq t} \text{skip} \) and \( a \parallel b \overset{a}{\preceq t} \text{skip} \parallel \text{skip} ; \) however \( a; b \not\overset{a; b}{\preceq t} \) and hence \( a; b \text{ fails } a; b \), and finally both \( a \parallel a; b \overset{a}{\preceq t} \text{skip} \) and \( a \parallel a; b \text{ fails } a; b \).

Using these relations, and for the sake of simplicity disregarding the effect of termination in the specification and implementation, we obtain the following definition.
6 Definition. For all \( r \in \mathcal{R} \), the relation \( \preceq^r \subseteq \mathcal{L} \times \mathcal{L} \), called vertical bisimilarity under \( r \), is the largest relation such that for all \( I \preceq^r S \) and all \( t \in \mathcal{L} \):

- If \( I \xrightarrow{[t]} L \) then there exists an \( S' \) such that \( S \Downarrow S' \) and \( L \xrightarrow{[t]} S' \);
- If \( S \Downarrow S' \) then there exists an \( L' \) such that \( I \xrightarrow{[t]} L' \) and \( L' \xrightarrow{[t]} S' \);
- \( I \) fails \( t[\mathcal{R}] \) if and only if \( S \) fails \( t \).

It follows that the implementation can be “observed” only using terms of the form \( t[\mathcal{R}] \), i.e., expanded terms. Such terms are evaluated on the one hand according to their completed runs (transitions \( \xrightarrow{[t]} \)) and on the other according to their deadlock properties (predicates \( \text{fails } t[\mathcal{R}] \)). The fact that only expanded terms are allowed decreases the distinguishing power of the observations on \( I \) markedly. In particular, it is in general not possible to observe single action transitions \( =_{a} \). Instead the best one can do is to set \( t = a \), in which case the observations correspond to \( \xrightarrow{a} \), the completed traces of which are longer than just single actions. For instance, if \( r(a) = a_1, a_2 \) then \( I \xrightarrow{[t]} L' \) if and only if \( I \xrightarrow{a_1,a_2} L' \); we cannot observe \( I \xrightarrow{a_1} L'' \) and \( L'' \xrightarrow{a_1} L' \) separately.

In other words, the “grain of observation” on the implementation side is not atomic. This corresponds with the idea that the user will always be of the form \( B[\mathcal{R}] \) (see (4)), i.e., can access the concrete system only after expansion. On the specification side, however, we do have atomic observations at our disposal. Since in fact the relations \( \Downarrow \) and \( \text{fails } t \) can be derived completely from \( \preceq^\mathcal{R} \), this means that we can simplify Definition 6 in this respect. We return to this subject in Section 5.4 below.

7 Theorem. \((\preceq^r), \) is a bottom-up vertical equivalence.

The basis \( \preceq^1 \) corresponds to observation congruence. To see this, note that since \( a[1] \equiv a \) for all \( a \in \mathbb{A} \), now the grain of observation on the implementation side is atomic, too; hence we can restrict ourselves to observations \( \xrightarrow{a} \) in Definition 6, which collapses the definition to that of observation congruence.

5.3. Examples

We discuss some examples in order to develop some intuition for vertical bisimulation. In the following, \( r \) is defined by \( a \mapsto a_1; a_2 \) and \( b \mapsto b \) for all \( b \neq a \).

\[
\begin{align*}
I_1 &= a_1; a_2 \parallel b \\
S_1 &= a \parallel b
\end{align*}
\]

We have connected the important \( \preceq^r \)-related states with dotted lines. Note the unrelated states on the implementation side, represented by open nodes. These states are at most “passed through” on the way to a connected state. We will call the connected states of the implementation complete, and the others intermediate. The above implementation \( I_1 \) corresponds to \( S_1[r] \); hence we have not yet progressed beyond what we could already do in the previous section. Now consider the following:
\[ I_2 = a_1; a_2; b \parallel b; a_1; a_2 \preceq^r S_2 = a; b \parallel b; a \]

This implementation \( I_2 \) has one interleaving less than \( I_1 \) above and corresponds to \( S_2[r] \) where \( S_2 \approx S_1 \) is the “interleaving version” of \( S_1 \). We may conclude that also \( I_1 \preceq^r S_2 \) and \( I_2 \preceq^r S_1 \). It follows that the first disadvantage of expansion-based vertical design mentioned above has disappeared; \( \preceq^r \) is insensitive to the \( \approx \)-representative chosen on the abstract level. The fact that the additional path of \( I_1 \) does not make a difference w.r.t. \( \preceq^r \) is due to the limited observations \( t[r] \) allowed on the implementation. The path \( a_1ba_2 \) of \( I_1 \), in which \( b \) “interferes” with the refinement of \( a \), can be taken only by refining a test of the form \( t = a \parallel b \); however, this test cannot distinguish \( I_2 \) from \( I_1 \), since \( I_2 \preceq^r[a \parallel b][r] \), also holds due to \( I_1 \preceq^r[a \parallel a][r] \).

On the other hand, it is not the case that such a “mixed path” in which refinements of different actions are interleaved cannot make a difference at all. Consider the following:

\[ I_3 = a_1; (a_2; b \parallel c; a_2) \parallel b; a_1; a_2 \preceq^r S_1 = a \parallel b \]

Here the path \( a_1ca_2 \) of \( I_3 \) can be taken by the parallel test \( (a \parallel c)[r] \), but there is no serialisation of \( a \parallel c \) which yields a comparable transition of \( S_1 \). In particular, \( I_3 \not\preceq^r[a \parallel c][r] \) and \( I_3 \not\preceq^r[a \parallel c][c] \). As a consequence, not only is \( I_3 \preceq^r S_1 \) but there does not even exist an \( S \) such that \( I_3 \preceq^r S \).

So far we have not shown the influence of the predicate \texttt{fails} \( t \). Now consider the following example.

\[ I_4 = I_2 \parallel a_1; a_2; b \preceq^r S_1 = a \parallel b \]

\( I_4 \) is trace equivalent to \( I_1 \), but now a choice is made initially between two \( a_1 \)-transitions. It follows that \( I_4 \) \texttt{fails} \( a[r] \) since \( I_4 \not\preceq^r[a \parallel a][r] \) if the wrong initial choice is made. In other words, a program \( B = a \) using the concrete machine \( I \) after expansion (left hand side of (4)) behaves differently (viz. it may deadlock) than when \( B \) is applied directly to the abstract machine \( S \). On the other hand, still also \( I_4 \not\preceq^r[a \parallel a][c] \), and if we had not included the \texttt{fails}-condition in Definition 6 then \( I_4 \) would have been considered correct.

The final example, also to do with the deadlock predicate, shows that in some cases, the \textit{expansion} of a specification is not vertically bisimilar to that specification, i.e. is not
an implementation in the sense of this section. Let \( r \) be given by \( a \mapsto c'; a' \) and \( b \mapsto c'; b' \).

\[
I = c'; a' \parallel c'; b' \quad S = a \parallel b \quad S^* = I' = c'; (a' \parallel b')
\]

It can be seen that \( I \approx S[r] \) fails \( a[r] \) if the wrong choice is made for \( c' \). Since not \( S \) fails \( a \) it follows that \( I \not\approx S \). On the other hand, \( I' \approx S \). The point is that the difference between \( a \) and \( b \) cannot be seen in the initial parts of their refinements: both \( r(a) \) and \( r(b) \) start with the same concrete action \( c' \) and differ only afterwards. Below we discuss a restriction on refinement functions which rules out this kind of situation.

### 5.4. A summary of results

We have obtained a number of results with regard to vertical bisimulation in Rensink [24], the detailed presentation of which is unfortunately out of the scope of this paper. We present a summary here.

One may restrict the refinement functions \( r \in \mathbf{r} \) to be studied. A perhaps surprising result is that we can interpret refinement functions up to observation congruence (where \( r_1 \approx r_2 \) if and only if \( r_1(a) \approx r_2(a) \) for all \( a \in \mathbf{A} \)). This can be understood by observing that in the relation \( \triangleq \), the term \( t \) is evaluated only up to completed trace equivalence and in \( \text{fails} \) \( t \) up to failure equivalence. A somewhat more complicated proof shows that \( r_1 \approx r_2 \) implies \( t[r_1] \approx t[r_2] \) for all \( t \). Since observation congruence is stronger than failure or completed trace equivalence, this leads to the following.

**8 Theorem.** If \( r_1 \approx r_2 \) then \( \approx_{r_1} = \approx_{r_2} \).

Since every term can be rewritten modulo \( \approx \) to a sequential term, an immediate consequence is that we only have to investigate sequential refinement functions. Now let us call \( r \) distinct if for all \( a \in \mathbf{A} \) there is at most one \( b \in \mathbf{A} \) such that \( a \) occurs in \( r(b) \), and furthermore \( a \) occurs in \( r(b) \) exactly once. Of the refinement functions discussed so far, only the one of the last example was not distinct. Distinct refinement functions enjoy many pleasant properties, one of which is the following.

**9 Theorem.** If \( r \) is distinct then \( B[r] \preceq B \) for all \( B \in \mathbf{L} \).

Moreover, every refinement function \( r \) can be decomposed into a distinct refinement function \( r' \) followed by a renaming function \( \varphi \) such that \( r = \varphi \cdot r' \) and \( \approx_{r} \preceq \approx_{\varphi \cdot r'} \) (note that the \( \subseteq \)-part of the latter equality follows from Theorem 7 and Definition 1). Hence to study vertical bisimilarity we can restrict ourselves to distinct sequential refinement functions and renaming functions. For the latter, a linear time algorithm for finite state systems is given in [24].

The most objectionable feature of vertical bisimilarity is certainly the fact that the observations \( t \) to be tested for range over the whole of \( \mathbf{L} \). In fact Definition 6 can be improved in several ways.
1. There exists a property called atomicity up to \( r \), such that \( I \preceq^r S \) if and only if \( I \) is atomic up to \( r \) and \( I \preceq_n S \), where the latter relation is defined by removing the fails-condition from Definition 6. Intuitively, \( I \) is atomic up to \( r \) if \( I \) cannot get stuck halfway a refinement. This property is independent of \( S \), and decidable for finite state \( I \). If \( I \) is atomic up to \( r \) then we may interpret \( r \) up to completed trace equivalence; this further reduces the work of deciding \( \preceq^r \).

2. There exists a property called serialisability up to \( r \), which holds of a given \( I \) if and only if there exists a specification \( S \) such that \( I \preceq^r S \). Moreover, \( S \) is then determined up to observation congruence, and straightforward to derive from \( I \). Intuitively, serialisability of \( I \) means that its behaviour under a given parallel observation can always be “explained” by comparable behaviour under some sequential observation. This corresponds with the intuition from the database world that a system is serialisable if its behaviour can be “thought of” as sequential. Just as atomicity, serialisability is independent of \( S \). Deciding serialisability is in general still quite complicated, but we have formulated sufficient conditions which appear to be applicable to a reasonable number of examples.

3. There exists a test generator algorithm which produces sufficiently many tests to establish \( \preceq^r \). The tests generated by this algorithm are such that they contain no synchronisation over actions that are changed by \( r \): this means we can rewrite every \( t[r] \) modulo \( \equiv \) to a flat term (without refinement), and hence we do not need partial order semantics at all.

5.5. Buffer implementation

Back to the buffer example of the previous section. We argue that \( Dbl_x \preceq^r Buf \) for all \( x \in \{1, 2, \varepsilon\} \). It should be clear that \( Buf \approx Buf \). Since \( Dbl_1 \equiv Buf[r] \) and \( r \) is distinct it follows (Theorem 9) that \( Dbl_1 \preceq^r Buf \), also \( Dbl \equiv Buf[r] \preceq^r Buf \). Finally, it can be proved that \( Dbl_2 \preceq^r Buf \). The important related states are depicted in Figure 7.

It follows that vertical bisimulation is indeed more flexible than vertical design by expansion, since there \( Dbl_2 \) could not be obtained as an implementation of \( Buf \) or \( Buf \).
However, $\preceq_r$ also rules out some implementations that are indeed intuitively incorrect. Consider for instance the parallel composition of two independent buffers sketched in Figure 8. The corresponding transition system is too elaborate to represent; however also from the partial order behaviour it can be seen that it is possible to mix up values of different abstract write-actions into a single read-action. For instance, the test

$$t = wr_{y_1, z_1} ; wr_{y_2, z_2} \parallel (rd_{y_1, z_2} \parallel rd_{y_2, z_1})$$

deadlocks on $Buf_\varepsilon$, but its refinement does not deadlock on $Dbl_3$.

6. CONCLUSIONS

The two main contributions of this paper are: the development of a framework to express design by action refinement (vertical design), and the definition within this framework of a ternary relation called vertical bisimilarity. The standard approach, refining a system by expanding it, fits into our framework provided that the implementation relation is a congruence with respect to expansion. This congruence property is studied intensively in the literature; the fact that it is a natural consequence of our framework is a point in favour. (One could also reason the other way around: studying this congruence property makes sense because it is natural in our framework.)

Vertical bisimulation on the other hand is compatible with the interleaving semantics since it does not require congruence with respect to expansion. This is an important advantage because it means we can use the large body of theory that exists for interleaving semantics. Finally, vertical bisimulation is more flexible than design by expansion, in the sense that it allows more implementations, including some that are intuitively appealing. On the other hand, two drawbacks of vertical bisimulation are: firstly, that it is bottom-up rather than top-down, in other words that an implementation cannot be constructed from specification and refinement function only; and secondly, that it is in its current conception not very tractable: proving vertical bisimulation involves proving that two systems match each other for arbitrarily long transitions.

One can think of other applications of the general framework. In particular the following two ideas seem worthwhile.

- Refining only hidden actions. If we know beforehand that the actions we want to
refine are hidden immediately afterward, we have much more freedom in the design. The refinement universe \( r \) then consists of functions \( A \to 2^A \) rather than \( A \to L \), and

\[
I \preceq^r S \iff (\text{hide } r(A) \text{ in } I) \preceq (\text{hide } A \text{ in } S)
\]

For instance, Langerak has proposed to replace internal synchronous communication (single actions) by asynchronous communication (a small protocol). Because the communication is internal, one does not have to take external interfering influences into account, and hence all sorts of precautions can be ignored in the implementation.

- Interface refinement. Brinksma and others have proposed a notion of interface refinement in [7], which can be interpreted in our framework if we take the refinement universe \( r \) to consist of terms \( F \in L \) called interfaces, with two associated disjoint sets \( A_F, C_F \) of abstract and concrete actions, respectively. The corresponding vertical implementation relation is

\[
I \preceq^F S \iff (\text{hide } C_F \text{ in } I \| [C_F] F) \preceq S
\]

which corresponds to the idea that the interface \( F \) is “taken out of” \( S \). As a consequence, under some additional conditions on \( \preceq \) we have that for all “users” \( B \in L \) in which none of the actions in \( A_C \) occur

\[
I \preceq^F S \implies (\text{hide } A_C \text{ in } (B \| [A_F] F) \| [A_C] I) \preceq B \| [A_F] S
\]

In other words, just as in (4), the user \( B \) is changed (here by inserting the interface \( F \) which “absorbs” the abstract actions \( A_F \) and “turns them into” concrete actions \( A_C \)) and the implementation \( I \) interacts with the changed user in the same way as the specification \( S \) with the original user.

There are many questions raised by this work, the answers to some of which are already known but do not appear in this paper for lack of space.

- Which implementation relations can usefully be extended vertically? We have based this paper on bisimulation, and in the thesis [24] a similar extension is given to testing. We expect that the technique on which vertical bisimulation is based—expanding the environment, rather than the system—can be applied more generally; in fact all interleaving relations that abstract from invisible actions seem amenable to this approach. In particular, we expect that extending trace inclusion in this way should yield something very close to existing “linear time” refinement methodologies; see for instance Back [2,4], He [17].

- What congruence properties do we want vertical design to satisfy? The data refinement property (4) generalises to

\[
I \preceq^r S \iff \forall B \in L. (\text{hide } A \text{ in } B \| [A] I) \preceq (\text{hide } r(A) \text{ in } B[r] \| [r(A)] S)
\]  

(5)
where $A$ should include at least the “active domain” of $r$, i.e., those actions that actually change during refinement, and $r(A)$ contains the concrete actions occurring in any of the $r(a)$. Indeed, Definition 6 would seem to guarantee that (5) holds for vertical bisimulation. Surprisingly, this turns out not to be the case: for instance if $I = a_1; (a_2 \parallel a_3; b), r; a \mapsto a_1; a_2, S = a \parallel a; b$ then $I \not\lesssim S$ but if $B = a; b \parallel c$ then $\langle \text{hide } a_1, a_2 \text{ in } B \parallel \lfloor a_1, a_2 \rfloor \rangle \not\approx \langle \text{hide } A \text{ in } B \parallel \lfloor a \rfloor \rangle$. The problem appears to be that bisimulation too strictly preserves the “moment of choice”, even if it occurs within a series of internal actions. (A similar observation was made by Lynch and Vaandrager in [21] in connection with timed bisimulation.) The problem disappears when we move to failure inclusion (see [24]); it may even be sufficient to move to a weaker bisimulation-like relation such as coupled simulation [23]. (Thanks to Rob van Glabbeek for suggesting this.)

- Can vertical bisimulation be made more tractable? We have already mentioned one alternative characterisation through serialisability, for which some tractable sufficient conditions exist, but which in general is as difficult to prove as the original definition.

- Do there exist constructive algorithms to generate $\lesssim$-implementations, apart from traditional expansion? In other words, can we give correctness-preserving transformations for vertical design? An interesting approach is taken in Wehrheim [26], who adds information about dependencies to the refinement functions. This information can be used as a “control parameter” in expansion, such that causalities are added only between dependent actions.

We conclude that our framework for vertical design gives a useful insight in the methodological aspects of design by action refinement. It also raises many nontrivial questions.

Acknowledgement. Many thanks are due to my ex-colleagues at the University of Twente, especially Ed Brinksma, Rom Langerak and Bart Botma, discussing this work with whom has been a great pleasure and inspiration. Also thanks to Ulla Goltz and Roberto Gorrieri for lending their ears and minds.

REFERENCES

2. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In de Bakker et al. [3], pages 67-93.


