Analysis and Applications of the XDI model

Willem C. Mallon, Jan Tijmen Udding
Department of Computing Science
University of Groningen
E-mail: {willem,jtu}@cs.rug.nl

Tom Verhoeff
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
wstomv@win.tue.nl

Keywords: Delay Insensitivity, Verification, Derivation, Factorization, Communicating Processes

Abstract

It is not always straightforward to implement a network that is robust enough to be functionally independent of communication delay. In order to specify and verify so called Delay Insensitive networks, numerous models and formalisms have been developed. In this paper we analyze one of the most expressive models. We show how based on rewrite rules we can compute, rather than invent parts of a network. We implemented these computations in a tool. We also show how healthiness, finite execution models and a distributive parallel composition cannot coexist.

0. Introduction

In [Ver94] and [Ver98a] Verhoeff introduces the eXtended DI model, a proposal to model delay-insensitive processes and their environments. The model distinguishes itself from other models in that it allows one to place progress requirements on both processes and their environments. Previous models did not allow progress requirements [Ebe91], or allowed progress requirements to be placed on the process only [Luc94, Jos92, Dil89]. Specifications in the XDI model can be graphically represented in a lucid way, and help in our understanding of typical asynchronous concepts like dynamic nondeterminism [Ver98a].

In [Ver98a] it is shown how processes represented in this model can be analyzed. In this paper we concentrate on the model itself. We present an analysis of the structural properties of the model. We relate the model to the failure set model that is used as semantics for DI algebra expressions [Luc94, JU93].

The $\mathbf{X^2DI}$ model is an extension of the $\mathbf{XDI}$ model. We define and investigate parallel composition as an operator in the $\mathbf{X^2DI}$ model.

We show how the $\mathbf{X^2DI}$ model is generally expressive enough to compute an unknown P for given Q, R such that the parallel composition of P and Q implements R. If R is a specification, and Q is a component that we intend to use, we can compute what the least deterministic specification is that remains to be implemented. We show how healthiness, a distributive composition operator and a finitary execution model cannot coexist.

We have implemented parallel composition and other operators on $\mathbf{X^2DI}$ processes in a tool called `ludwig`. The `ludwig` tool enables us to perform various verifications of compositions as well as derivations: i.e. starting from a specification we have been able to realize implementations in small steps, where most of the steps were automated.

0.1 Notation

We denote concatenation by juxtaposition. Hence the concatenation of traces x and y is denoted by xy. Concatenation binds strongest of all operations. We use $.$ for function application. Hence $P.x$ means the function P applied to argument x. We use $\sqcap$ and $\sqcup$ to denote greatest lower bound (infimum) and least upper bound (supremum) respectively. We use the family notation from [DS90]. Hence $(\sqcap i : d.i : P.i)$ is the greatest lower bound of the set X, where $P.i \in X \equiv d.i$, for any i.

1 The $\mathbf{XDI}$ model

We repeat the definition of the XDI model here, as it was given in [Ver98a]. However, we take a different approach w.r.t. some of the properties for reasons that will become clear throughout this paper.

Let Λ be the set of labels $\{\bot, \Delta, \square, \nabla, \top\}$ with associated total ordering $\bot \sqsubseteq \Delta \sqsubseteq \square \sqsubseteq \nabla \sqsubseteq \top$. The labels model properties of the current state of the process. Their interpretation is as follows:

Label	Name	Interpretation
$\top$	top	Unreachable
∇	transient	Process must send an output
$\square$	quiescent	No obligations
Δ	demanding	Process must receive an input
$\perp$	bottom	An error has occurred

Let A be an alphabet partitioned in input alphabet I and output alphabet O. With A^* we denote the set of all finite traces over A. The relation $x \preceq y$ is defined as the strongest transitive reflexive relation such that:

$$(0) \qquad a \in I \vee b \in O \quad \Rightarrow \quad sabt \preceq sbat$$

Operationally, trace $x \preceq y$ holds iff x can be created from y by moving inputs in y to the front, and outputs in y to the back. For example, if $a \in I$ and $b \in O$ we have $bca \preceq acb$.

The relation $\preceq$ captures the fact that in communication with a DI process, the trace that is observed by the environment may differ from the trace as it is observed by the process. This is typical for asynchronous communicating processes. If there are no signals in transit, and the environment observed trace x, and the process observed trace y, then x and y are related as $x \preceq y$.

By P/x (pronounced P after x, and defined as $(P/x).y = P.xy$) we denote the process that behaves like process P after trace x has been communicated with it.

The partial ordering $\sqsubseteq$ on processes is defined as:

$$(1) \qquad P \sqsubseteq Q \quad \equiv \quad (\forall x : x \in A^* : P.x \sqsubseteq Q.x)$$

If $P \sqsubseteq Q$ we consider Q to be an adequate substitute for P in any network, and we say that 'Q refines P', or 'Q is more deterministic than P'.

We now define the $\mathbf{X^2DI}$ space as all functions P of type $(I \cup O)^* \to \Lambda$ such that:

$$(2) \qquad P.s = \perp \quad \Rightarrow \quad P.st = \perp$$
$$(3) \qquad P.s = \top \quad \Rightarrow \quad P.st = \top$$
$$(4) \qquad P.xb = \perp \wedge b \in O \quad \Rightarrow \quad P.x = \perp$$
$$(5) \qquad P.xa = \top \wedge a \in I \quad \Rightarrow \quad P.x = \top$$
$$(6) \qquad x \preceq y \quad \Rightarrow \quad P/x \sqsubseteq P/y$$
$$(7) \qquad P.xcc = \top \quad \vee \quad P.xcc = \perp$$

Further motivation for these requirements can be found in [Ver98a], [Ver94]. With (2) and (3) we require that error, and unreachable states are persistent. With (4) we require that the process cannot reach an error state by producing outputs. With (5) we require receptiveness. The process cannot refuse inputs. With (7) we require that sending two

signals in succession through the same channel is not normal behaviour. With (6) we require that whenever process and environment differ in their opinion on the current trace ($x \preceq y$), the process must behave at least as deterministic as the environment expects it to behave.

In [Ver98a] Verhoeff places two more requirements on processes, which we call the healthiness constraints. They are formally stated as:

$$(8) \qquad P.x = \nabla \quad \Rightarrow \quad (\exists b : b \in O : P.xb \neq \top)$$
$$(9) \qquad P.x = \Delta \quad \Rightarrow \quad (\exists a : a \in I : P.xa \neq \perp)$$

We refer to Property (8) and (9) as $\nabla - healthy$ and $\Delta - healthy$ respectively. The reason for introducing these constraints, is that processes that do not satisfy these properties do not fit easily into our operational interpretation. If a process is in a state that is labeled ∇, it must send an output. If all outputs lead to unreachable states, it cannot send an output. This seems to be conflicting, and is also very hard to implement. However, as we show in this paper, unhealthy processes can function as intermediaries when designing an implementation, analogue to how partial commands can help when designing a program. [Hes92]

Furthermore, placing these two constraints on processes complicates both the model and the definitions enormously. We therefore separately investigate the space $\mathbf{X^2DI}$ where both requirements need not hold, and the space $\mathbf{XDI}$ where both requirements hold. Of course we have $\mathbf{XDI} \subseteq \mathbf{X^2DI}$

1.1 Finite Automata

Some of the processes in $\mathbf{XDI}$ and $\mathbf{X^2DI}$ can easily be viewed as finite automata. For example, consider the wire process W given in figure 0.

The initial state is 0 and can be recognized by the small triangle pointing towards it. The shape of the state that we reach after following a path labeled x, is the value of $W.x$. For example, when communicating trace aba with the Wire, we find that we end in state 1 that is labeled ∇. Hence $W.aba = \nabla$.

When drawing $\mathbf{X^2DI}$ and $\mathbf{XDI}$ processes we often leave out the $\perp$ and $\top$ states and transitions to and from those states. Because of the constraints on processes these can be inferred from the other transitions and labels. If a state in a graph drawing has no outgoing a input transition, the a transition that is not shown must be a transition to $\perp$, otherwise (5) would be violated. A similar argument holds for output transitions. The only situation where this reconstruction is not possible is when the initial state is labeled $\top$ or $\perp$, in which case we are explicit in our drawings. The wire process can hence be drawn as in Figure 0.
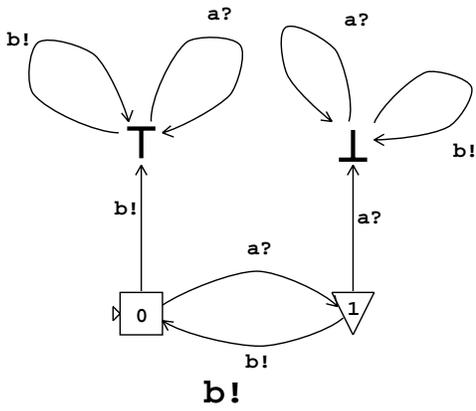
Figure 0. The wire process W (top), and the same process with implicit $\top$ and $\bot$ (bottom)

Figure 1. Process P **(top left),** Q **(top right), their pointwise infimum (bottom left) and their healthy infimum**

1.2 Complete Lattices

A poset $(X, \sqsubseteq)$, where X is a set and $\sqsubseteq$ is a partial order, is a complete lattice if it is closed under arbitrary infimum $\sqcap$ and supremum $\sqcup$. Complete lattices have various properties that can be exploited. For example: Every monotonic function $X \to X$ on a complete lattice has a least and a greatest fixed point. This means that, if G is a monotonic function, we can give meaning to recursive definitions of the shape $P = G.P$, and, for example, say that the expression defines P as the least (or greatest) fixed point of G.

It is easy to show [MU98b] that the space $\mathbf{X^2DI}$ is a complete lattice, and that infimum and supremum are pointwise lifted infimum and supremum on labels. That is, for S a set of $\mathbf{X^2DI}$ processes we have:

$$(10) \quad (\sqcap P : P \in S : P).x \;=\; (\sqcap P : P \in S : P.x)$$
$$(11) \quad (\sqcup P : P \in S : P).x \;=\; (\sqcup P : P \in S : P.x)$$

Note that the $\sqcap$ occurring on the lefthand side denotes infimum on processes. The $\sqcap$ on the righthand side denotes infimum on labels.

By adding the healthiness constraints infimum and supremum are no longer pointwise lifted. Consider for example processes P and Q, both with empty output alphabet and singleton set $\{a\}$ as input alphabet, given in Figure 1. The pointwise infimum is also given in Figure 1, and clearly violates (9). However the healthy infimum in $\mathbf{XDI}$ exists, and is given in Figure 1.
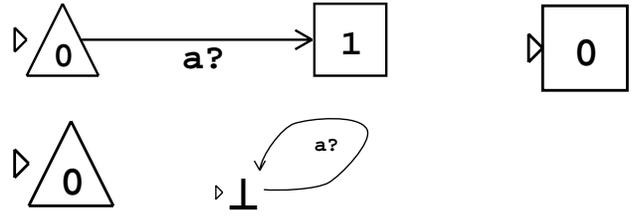
Proving that $\mathbf{XDI}$ is a complete lattice therefore takes more effort. In [MU98b] a complete proof has been given.

The fact that healthy infimum and supremum are not pointwise lifted from labels is a disappointment. It makes it so much harder to prove some of their properties, and to work with them. It is one of the reasons for looking at $\mathbf{X^2DI}$.

1.3 Relation to Failure sets

Failure sets [Jos92] are less expressive than the $\mathbf{X^2DI}/\mathbf{XDI}$ models. Failure sets cannot express progress conditions on the environment, but they can express progress conditions on the process. Failure sets are used as a model for the DI-algebra. We can define a structure preserving function Map [MU98b] from failure set space to $\mathbf{XDI}$ such that for failure sets F and G we have:

$$(12) \quad F \sqsubseteq G \;\equiv\; Map.F \sqsubseteq Map.G$$

It follows that the failure set space is isomorphic with the codomain of Map. This means that we can switch with impunity between the two formalisms. The transformation from DI algebra expression to $\mathbf{XDI}$ process has been implemented in the digg tool [MU98b].

Furthermore, if we want to design an operator on $\mathbf{X^2DI}$, that already exists in the DI algebra, we can use (12) to anchor our new definition. Typically, we want for a new operator $\otimes$ that should behave like an existing operator $\oplus$, that we have for failure sets F, G:

$$(13) \quad Map.(F \oplus G) \;=\; Map.F \otimes Map.G$$

2 Testing and Composition in General

There are basically two approaches to modeling:

- Define the entities in the model (such as 'certain functions of type $A^* \to \Lambda$') and study their inherited properties (such as '$\langle \mathbf{XDI}, \sqsubseteq \rangle$ is a complete lattice');

- Impose reasonable or useful properties on the model, and find entities that realize these properties.

We shall take the second approach in this section (details, such as proofs, can be found in [Ver98b]).

So far, we have dealt with models whose entities are processes and where the only operators are the binary refinement relation $\sqsubseteq$, and the after operator. To reason about the composition of processes, one needs a parallel composition operator. We will denote the binary composition operator on processes by $\|$.

The composition operator should abstract from internal communication events and $P \| Q$ should be just another process in our model. In particular, we are interested in the relationship between composition and refinement.

Reasonable properties for composition are that it is commutative and associative, and that it has a unit element E:

$$(14) \qquad P \| Q \;=\; Q \| P$$
$$(15) \qquad (P \| Q) \| R \;=\; P \| (Q \| R)$$
$$(16) \qquad P \| E \;=\; P$$

Of course, in some process domains, $\|$ is only associative when certain restrictions on alphabets are taken into account. This means that care has to be taken if the theory in this section is instantiated for models where this is an issue.

Another reasonable, though more imposing, property is that composition distributes over infima.

$$(17) \qquad P \| (\sqcap V) \;=\; (\sqcap Q : Q \in V : P \| Q)$$

for any set V of processes. In other words, $\|$ distributes over arbitrary $\sqcap$.

Finally, we assume that refinement is fully abstract with respect to a form of testing. Let D be a special process. We say that process P passes the test in the context of process R whenever

$$(18) \qquad P \| R \;\sqsupseteq\; D$$

For delay-insensitive processes, absence of interference and deadlock can be captured by such a test (see [Ver94] for details).

We abbreviate the set of processes R for which $P \| R \sqsupseteq D$ by:

$$(19) \qquad R \in pass_D.P \;\equiv\; P \| R \sqsupseteq D$$

and leave out D when it is obvious. That refinement is fully abstract with respect to this form of testing is captured by

$$(20) \qquad P \sqsubseteq Q \;\equiv\; pass.P \subseteq pass.Q$$

that is, Q is better than P whenever Q passes at least all the tests that P passes.

One reason to be interested in the relationship between composition and refinement is the following practical situation. Given processes Q (a guess for a partial implementation) and R (a specification to be implemented), the question is which processes P satisfy

$$(21) \qquad P \| Q \sqsupseteq R$$

that is, which processes P can be composed with the guessed part Q to realize R? Equation (21) is called the design equation. Given the assumptions that we have made about the process model, the design equation can be solved explicitly.

First, one can introduce a reflection operator $\backsim$ on processes by

$$(22) \qquad \backsim Q \;=\; \sqcap pass.Q$$

Reflection has such properties as:

$$(23) \qquad \backsim Q \;\in\; pass.Q$$
$$(24) \qquad P \in pass.Q \;\equiv\; P \sqsupseteq \backsim Q$$
$$(25) \qquad P \sqsubseteq Q \;\equiv\; \backsim P \sqsupseteq \backsim Q$$
$$(26) \qquad P \sqsupseteq Q \;\equiv\; \backsim Q \in pass.P$$
$$(27) \qquad \backsim\backsim P \;=\; P$$

Using these properties, one readily derives the so called factorization equation:

$$(28) \qquad P \| Q \sqsupseteq R \;\equiv\; P \sqsupseteq \backsim(Q \| \backsim R)$$

$\qquad P \| Q \sqsupseteq R$
$\equiv \quad \{ (20) \}$
$\qquad (\forall X : X \in pass.R : X \in pass(P \| Q))$
$\equiv \quad \{ (19) \}$
$\qquad (\forall X : X \in pass.R : X \| (P \| Q) \sqsupseteq D)$
$\equiv \quad \{ \text{property } \sqcap \}$
$\qquad (\sqcap X : X \in pass.R : X \| (P \| Q)) \sqsupseteq D$
$\equiv \quad \{ \| \text{ is associative and distributes over } \sqcap \}$
$\qquad (\sqcap X : X \in pass.R : X \| Q) \| P \sqsupseteq D$
$\equiv \quad \{ (19, 27) \}$
$\qquad \backsim\backsim(\sqcap X : X \in pass.R : X \| Q) \in pass.P$
$\equiv \quad \{ (26), \| \text{ distributes over } \sqcap \}$
$\qquad P \sqsupseteq \backsim((\sqcap X : X \in pass.R : X) \| Q)$
$\equiv \quad \{ (22) \}$
$\qquad P \sqsupseteq \backsim(\backsim R \| Q)$

Hence, the $\sqsubseteq$-least solution of the design equation (21) is

$$(29) \qquad \backsim(Q \| \backsim R)$$
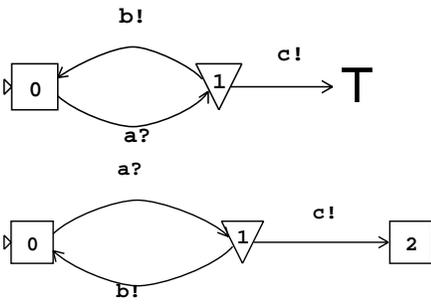
Figure 2. definitions of P **(top) and** Q

Figure 4. Composition $S = I \parallel R_i$

which we shall also denote by $R - Q$. Note that in this notation we have $\smile Q = D - Q$.

Finally, it is worth observing that if, in a process model, every design equation has a least solution, then composition must distribute over arbitrary infima. i.e. $(28) \Rightarrow (17)$.

3 Parallel Composition in $\mathbf{X^2DI}$ and $\mathbf{XDI}$

Because of the symmetry of the $\mathbf{X^2DI}$ model, it is likely that $\smile$ is a simple mirror operation. That is $\smile P$ can be obtained from P by simply transforming inputs to outputs, outputs to inputs, ∇ states to $\triangle$ states, $\top$ states to $\bot$ states, etc. If $\parallel$ is not too difficult either, we can use the factorization equation and, for example, implement it.

Hence we would like our parallel composition to distribute over arbitrary infima. This, however, is not possible in $\mathbf{XDI}$ space! In the rest of this section we will not be explicit in our definition of $\parallel$, but use properties that it ought to have, if it is to be a faithful model of connecting wires on on circuit level.

We show this by counter example. Consider the processes P and Q with input alphabet $\{a\}$ and output alphabet $\{b, c\}$. Both P and Q are initially quiescent. Upon reception of input a, P will produce output b and return to its initial state. This is also allowed behaviour for process Q, but process Q can upon reception of input a choose to produce output c and then stop. See Figure 2. Hence, P implements Q, $Q \sqsubseteq P$.

Now consider the parallel composition of P and Q with I-wire I. An I-wire starts by producing an output, and then behaves like the usual wire. If we want the parallel composition to by distributive it has to be at least monotonic. Hence $Q \parallel I \sqsubseteq P \parallel I$. Now consider the label of the initial state of $P \parallel I$ as observed by an environment. Clearly $P \parallel I$ cannot be initially $\top$, since that would mean by (3) that $P \parallel I$ maps every trace to $\top$ and hence that by definition of $\sqsubseteq$ (1), it would be a sufficient replacement for all processes. This is operationally very unsatisfactory. $P \parallel I$ cannot be initially ∇, because it will never produce an output, and would hence be violating (8). We conclude that the initial state of
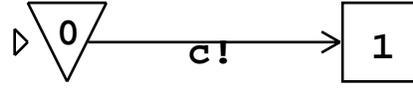
$P \parallel I$ is at most labeled $\square$. Hence the initial state of $Q \parallel I$ is at most $\square$ too.

Now consider the family of processes R_i where R_i is informally defined as:

R_0 = upon reception of input a produce output c, then stop

R_{i+1} = upon reception of input a produce output b, then behave like R_i

We give R_3 in Figure 3.

Consider for any i the parallel composition of an I-wire with R_i. This composition will engage in i internal handshakes, but will eventually produce output c. When abstracting from internal communications we find that for any i the process $I \parallel R_i$ is the process that will initially produce output c and then stop. We give $S = I \parallel R_i$ in Figure 4. Hence we find $(\sqcap i :: R_i \parallel I) = S$, and initially ∇. However, it is not hard to prove that $(\sqcap i :: R_i) = Q$. This is consequence of the fact that we only observe finite traces in the model.

We therefore conclude:

(30) $\qquad (\sqcap i :: R_i) \parallel I \quad \neq \quad (\sqcap i :: R_i \parallel I)$

which is unfortunate. Note that this is a very general counter example. In Failure set space we have healthiness constraints and a finitary model. As a consequence parallel composition cannot distribute over arbitrary infima in failure set space. In Negulescu's process spaces [Neg98] composition operators are defined that distribute over arbitrary infima. This means that instantiation with finitary execution sets must lead to unhealthy processes.

If we want the factorization equivalence to be valid in our model we have two options based on the counter example above:

- Drop the healthiness requirements

- Include infinite traces in the model

In this paper we concentrate on the first option. It is possible to define a parallel composition operator on $\mathbf{X^2DI}$ space [MU98b] that has the following properties:

(31) (9) holds for $P \parallel Q$

(32) $\qquad P \parallel Q \sqsupseteq R \quad \equiv \quad P \sqsupseteq \smile(Q \parallel \smile R)$

Furthermore, if there is no infinite chatter (livelock) between failure sets F and G, and $\parallel_{DI}$ is the DI algebra com-
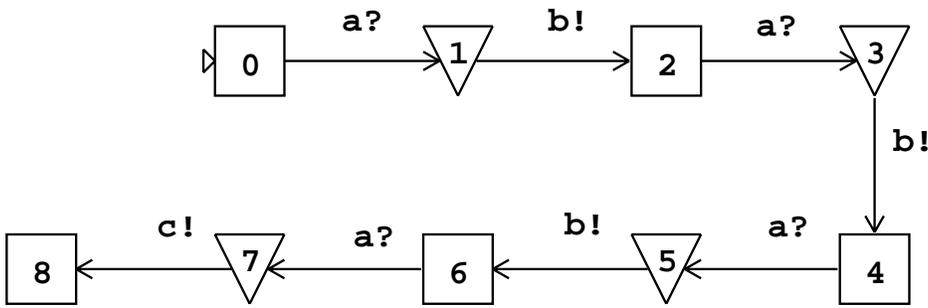
Figure 3. R_3

position, we can prove [MU98b]:

$$(33) \qquad Map.F \| Map.G \quad = \quad Map(F \|_{DI} G)$$

which in a sense is a validation for our choice of parallel composition. The DI algebra composition is basically a weave, so our composition is basically a weave on the codomain of Map. Of course if we want (33) to hold without premises, we would have to map infinite chatter to $\bot$ as is done in the failure set composition definition [Luc94]. This however will interfere with the factorization equation. We can use the same counterexample we used earlier to show that. It is crucial that property (31) holds for parallel composition as we show later. Hence our $\|$ models composition on circuit level faithfully in those cases where there is no possibility of livelock. Thus we can use our composition to establish partial correctness, but have to be careful in those situations where livelock may occur.

4 Consequences

The basic DI components that we work with, like C-elements, or sequencer arbiters, are healthy. The specifications and larger components that we work with are often generated from DI algebra expressions and are healthy as well. If we accept the DI algebra parallel composition, we can use our definition of composition, as long as we take care not to introduce infinite chatter.

This leads to the following approach to solve the design equation: Suppose we want to implement specification R, and use component Q to do so. Then we need to find a healthy process P such that there is no infinite chatter between P and Q and such that $P \sqsupseteq \backsim(Q \| \backsim R)$.

We have written tool support to help us find such P. A small example of the tools functionality is given in a later section. There are more ways in which this particular factorization equation can help us when designing delay-insensitive implementations, as we show in the next subsection.

4.1 Analysis

The healthy processes that are impossible to implement are the processes that map all traces to $\top$. We denote these processes with $\top$, if it is clear from the context that we mean the process, and not the label.
All other healthy processes are considered implementable.

This means that the predicate 'R can be implemented using Q' can be transformed to 'There is a healthy process P that is not equal to $\top$ such that P has no infinite chatter with Q, and $P \sqsupseteq \backsim(Q \| \backsim R)$'.

We concentrate on the latter part. Suppose $\backsim(Q \| \backsim R)$ is unhealthy. Then $Q \| \backsim R$ must be violating (8) because of (31). Hence $\backsim(Q \| \backsim R)$ must be violating $\Delta - healthy$. By simply mapping all Δ states to $\square$ states we find a greater process[1] that satisfies all healthiness constraints. Therefore, the predicate 'There is a healthy P not equal to $\top$ such that $P \sqsupseteq \backsim(Q \| \backsim R)$' transforms to the predicate $\backsim(Q \| \backsim R) \neq \top$. Recalling the abbreviation $\backsim(Q \| \backsim R) = R - Q$, we find that

Q can be used to implement R

$\equiv$

$(\exists P : \top \sqsupseteq P \sqsupseteq R - Q :$
there is no infinite chatter between Q and $P)$

We have optimized our tool to compute $R - Q \neq \top$ quickly. The worst case efficiency will always be in the order of the product of the number of states of Q and R though. On the bright side: worst case occurs only if the predicate holds.

One direct application of (4.1) is the comparison of specifications. $P_0 \sqsubseteq P_1$ means that P_1 is an adequate substitute of P_0. A necessary condition for $P_0 \sqsubseteq P_1$ is that the alphabets of P_0 and P_1 are equal. As a consequence, the alphabet of $P_1 - P_0$ is empty. Since no process with empty alphabet can engage in infinite chatter it suffices to check whether the alphabets of P_0 and P_1 are equal, and compute $P_1 - P_0 \neq \top$.

Coming back to (4.1), it is quite often possible to guarantee that there will be no infinite chatter between a com-

---

[1]This is where property (31) is crucial

ponent, and the remainder. Consider for example a specification R, with an input a and an output b. If we try to implement R with a fork, such that the fork has input a and one of the fork outputs is mapped to b, then whatever remainder we get cannot engage in infinite chatter with Q. It suffices to calculate $R - Q \neq \top$ to verify that Q can be used.

Experience shows it to be very useful, given a specification R, to compute $R - Q_i \neq \top$ for a large set of basic components Q_i where for any Q_i it is easy to guarantee that $R - Q_i$ and Q_i will not engage in infinite chatter.

In some cases, for example Counterflow Pipeline control [SSM94], we could even find implementations by just applying the following algorithm:

> { R is the specification }
> while $R \neq$ neutral element for composition
>   do
>   Choose a Q_i such that $R - Q_i \neq \top$,
>   and there is no livelock
> $R := R - Q_i$
> $I := I \cup Q_i$
>   od
> { I is the set of components implementing R }

Of course, this algorithm does not necessarily terminate, and it depends heavily on the set of Q_i processes. However running the algorithm partially and observing the result can give useful insight in the structure of the process R we are trying to implement.

4.2 Weakening and Strengthening

Typically when designing an implementation we make 'design choices'. A design choice amounts to implementing something more deterministic than required by the specification. Sometimes a specification is quite nondeterministic, and it can be difficult to find a more appropriate and deterministic specification. Consider:

$$
\begin{aligned}
&true \\
\Rightarrow\ &\{ \text{reflexivity} \} \\
&SPEC - X \sqsupseteq SPEC - X \\
\equiv\ &\{ (32) \} \\
&(SPEC - X)\|X \sqsupseteq SPEC
\end{aligned}
$$

The other way around works as well. We find:

(34) $\quad SPEC \ \sqsubseteq\ (SPEC - X)\|X$

(35) $\quad SPEC \ \sqsupseteq\ (SPEC\|X) - X$

Here is an abstract example. Consider a large specification that might contain a lot of nondeterminism. The specification has an input signal a and an output signal b. Our
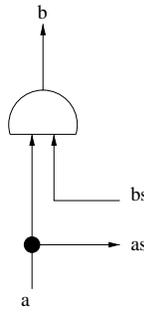

Figure 5. X

design decision is that output b should only occur if an input a has occurred earlier. Input a should not occur again before it has been confirmed by b. Then we can try to create an X from a C element and a fork as in Figure 5, and compute $(SPEC - X)\|X$ to see if that specification is easier to implement. We demonstrate how this strategy can be applied in practice in the next section.

5 Example

As a small demonstration of how the theory developed thus far has been applied in a number of synthesis and analysis tools we discuss the example of the Nacking Arbiter. More information on the tools used in this section can be found at [GLM+].

The following informal description has been taken from EDIS [Ver]. "A Nacking Arbiter has two input terminals (r_0 and r_1) and four output terminals (a_0, n_0, a_1, and n_1). For each i in 0, 1, the signals cycle through either $r_i\ a_i\ r_i\ a_i$ or $r_i\ n_i$. The first r_i signal of each cycle can be interpreted as a request (to access a critical section). The remaining three signals in a cycle $r_i\ a_i\ r_i\ a_i$ can be interpreted as grant, done, and acknowledge (of done), respectively. The grant outputs are mutually exclusive. After request r_i the cycle is closed with n_i, interpreted as nack (negative acknowledge), when the other party has already been granted. When a Nacking Arbiter receives two requests, it will grant exactly one of them (and nack the other). The specification leaves the choice open."

A specification in DI-algebra would be

$$
\begin{aligned}
NA &= [r0? \to a0!; G0\ \Box\ r1? \to a1!; G1] \\
G0 &= [r0? \to a0!; NA\ \Box\ r1? \to n1!; G0] \\
G1 &= [r0? \to n0!; G1\ \Box\ r1? \to a1!; NA]
\end{aligned}
$$

Feeding this specification to `digg` [MU98a], the result is a state graph, expressed in AND/IF-notation [EMN+], that can then be analyzed and synthesized with `ludwig`.

As shown in [Ver98a] this specification is more deterministic than necessary since the DI algebra considers pre-

cisely one environment, and the intention of the nacking arbiter is to be used in two independent environments.

We can apply alternations [MU98c] to make the specification less deterministic. An alternation consists of two disjoint sets of symbols $[S, T]$. Applying alternation $[S, T]$ to process P puts the extra restriction on the environment that in communication with the process P symbols from S and T must alternate, and start with a symbol from S. If the environment violates the alternation by sending illegal inputs, or by sending inputs that may allow the process to send illegal outputs, the process enters the error state $\perp$. Alternations can be seen as a generalization of handshaking [Ber92].

In this case the alternation is that no $r0$ should be sent unless the previous one has been acknowledged by either $a0$ or $n0$. This is expressed by the alternation $[r0?, \{a0!, n0!\}]$ (which also states that the alternation is to start with $r0$). A similar alternation is imposed for the other party.

The beginning of the input file for `ludwig` looks like:

```
# Ludwig input file for the synthesis
# of a Nacking arbiter.
# Let the file spec.ndf contain
# the AND/IF specification generated
# by digg1.0 from the DI-algebra
# specification:

S = FILE "spec.ndf"

# We now apply the alternations to S to
# arrive at the final
# (after minimization) specification S.

S = APPLY [ r0? , {a0!, n0!} ] TO S
S = APPLY [ r1? , {a1!, n1!} ] TO S

S = MINIMIZE S

PRINT "andif" S
```

This text is self-explanatory. The last statement prints the specification in AND/IF format. The specification turns out to be equivalent to the XDI-specification given in [Ver98a].

At such a point, and we shall see that more often throughout the synthesis process, the designer needs to have some idea for part of an implementation. In this case the obvious thing to try is to use a sequencer arbiter to arbitrate between the request r_0 and r_1, and allow a next arbitration to take place after one of the signals a_0, a_1, n_0, or n_1 has been generated. This suggests a 4-input Merge that feeds each output back to the done-signal of the arbiter, cf. Figure 6.

What we are then interested in is the remainder. So the input file for ludwig continues in the following way:

```
# As a first step in the implementation
# we try to use a sequencer arbiter, a
# 4-way Merge, implemented as three
# 2-way Merges, and four Forks.
```

```
# The remainder we call COMP.

F = FILE "fork.ndf"
A = FILE "sequencer.ndf"
M = FILE "merge.ndf"

COMP =  S -
(   RENAME [a? b! c!]
      TO [ia0? a0! p0!] IN F)
|| (RENAME [a? b! c!]
      TO [ia1? a1! p1!] IN F)
|| (RENAME [a? b! c!]
      TO [in0? n0! p2!] IN F)
|| (RENAME [a? b! c!]
      TO [in1? n1! p3!] IN F)
|| (RENAME [a? b? c!]
      TO [p0? p1? s0!] IN M)
|| (RENAME [a? b? c!]
      TO [p2? p3? s1!] IN M)
|| (RENAME [a? b? c!]
      TO [s0? s1? d!] IN M)
|| A)

PRINT "andif" COMP
```

The state graph of the remainder turns out to be very simple , and is given in Figure 7. This is what is left to be implemented. It shows three quiescent states, 8, 3, and 7, where 8 is the initial state and 3 and 7 are complementary states. Looking at state 3 we see that input $g0$ should produce output $in0$ which leads to state 3 again. Moreover it is the only way that ever output $in0$ is produced. Therefore, a next step in the synthesis seems to be to use some sort of state holding device that in one state will, upon reception of $g0$, produce output $in0$, and that in the other state will produce some internal output that is to be processed by the remainder. Obviously, a set-ack pair to this component is necessary, in order to switch states.

Therefore, we introduce a switch element that can be switch to the other state using input signal s that is acknowledged by a. Upon a read request r the current value is output, which we call *on* and *off*. The specification in DI-algebra is:

$$
\begin{aligned}
SCoff &= [s? \to a!; SCon \square r? \to off!; SCoff] \\
SCon &= [s? \to a!; SCoff \square r? \to on!; SCon]
\end{aligned}
$$

After applying alternation $[\{r?, s?\}, \{on!, off!, a!\}]$ to ensure sequentiality of the handshakes, we arrive at the state graph given in Figure 8. The `ludwig` input file could now continue like:

```
# We continue the synthesis by
# trying to factor a switch
# cell.

SC = FILE "switch.ndf"

COMP1 = COMP -
   (RENAME [r? on!] TO [g0? in0!] IN SC)
```
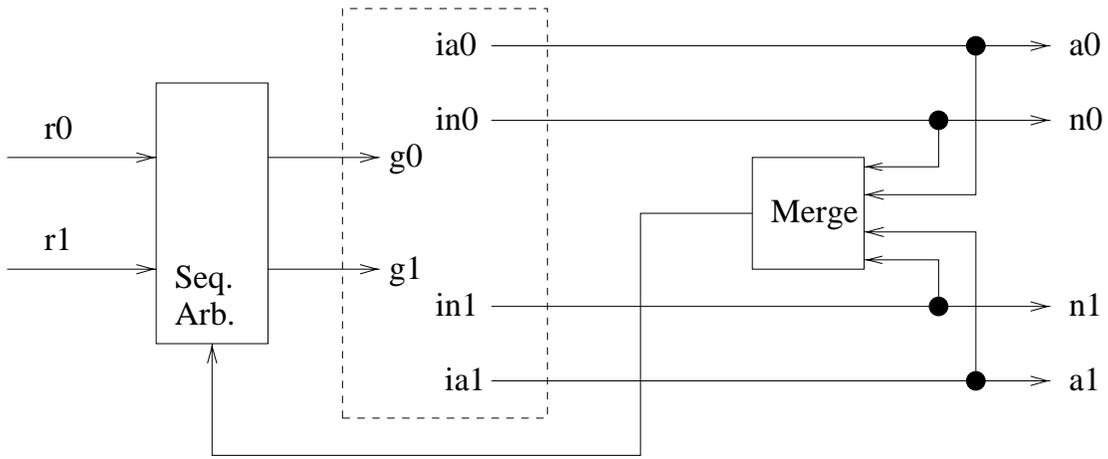
8

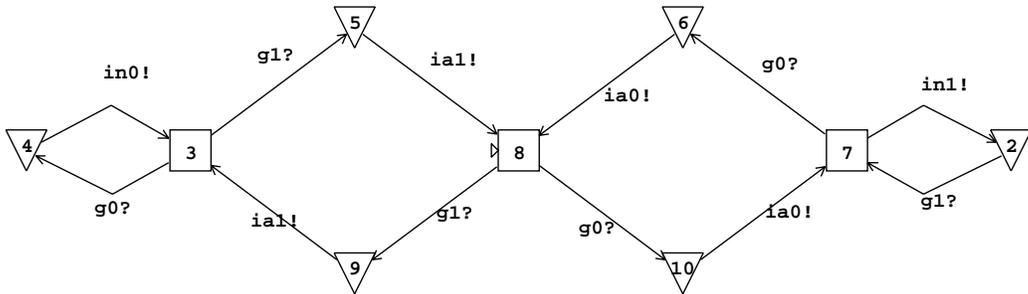**Figure 6. Decomposition using Sequencer Arbiter and Merge**

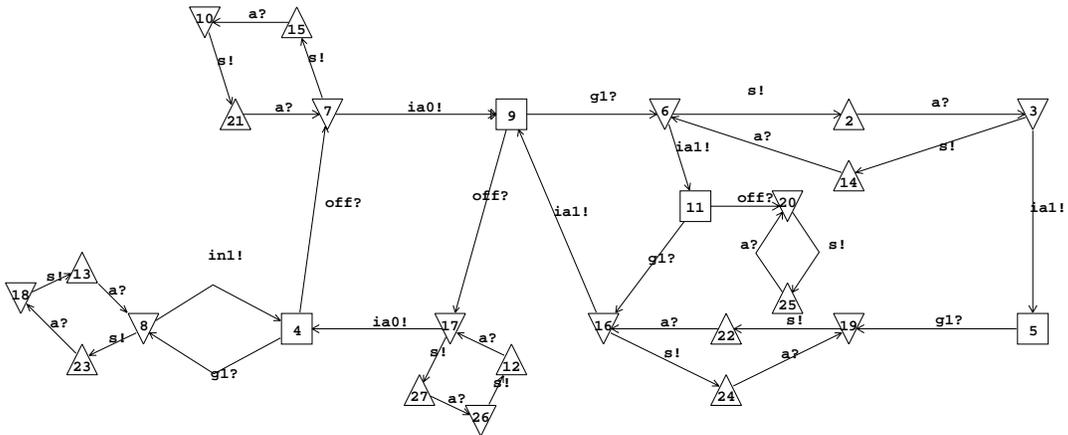**Figure 7. Result after taking out Arbiter and Merge**

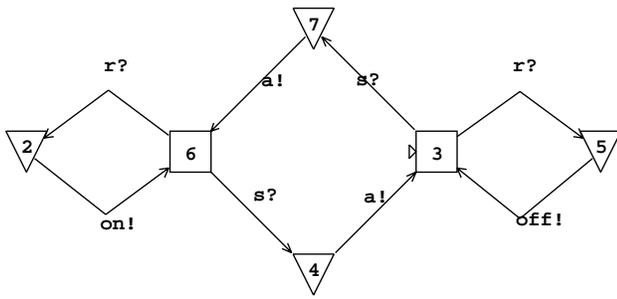**Figure 9. Remainder after taking out the Switch**

Figure 8. State graph of the On-Off switch

This results in the state graph of Figure 9.

Notice that each communication of $s!$ leads to a demanding state, since it is known that the switch *will* produce output $a!$ after reception of an $s?$. This graph entails another interesting phenomenon. At various places we see s, a cycles. Obviously, if we were to implement these cycles, we would introduce infinite chatter.

The question is how we can systematically generate a more deterministic specification, such that an implementation of this specification would not engage in cycles of s, a communications. Here we can use that a specification S is refined by $(S - X) \parallel X$ for any X, as stated in (34). It is easy to see from Figure 9 that $ia1!$ can safely be postponed until reception of (exactly one) $a?$. In this case, we use the X element that is given in Figure 5, where we rename b to $ia1$. The input to ludwig continues in the following way.

```
# COMP1 turns out to be too
# nondeterministic, leading to
# internal chatter on s and a. We
# refine the specification by requiring
# a and ia1 to alternate

X = FILE "X.ndf"
X = RENAME [b!] TO [ia1!] in X

COMP1 = MINIMIZE ((COMP1 - X) || X)

PRINT "andif" COMP1
```

The resulting state graph is depicted in Figure 10. Clearly, all infinite chatter has been removed.

One more piece of the implementation now becomes obvious. Output $ia1$ is produced upon reception of a. This can easily be implemented by a wire, and the input to ludwig continues:

```
# Clearly, ia1! is produced upon reception
# of an a, so we factor away a wire.

W = FILE "wire.ndf"

COMP1 = COMP1 -
   (RENAME [a? b!] TO [a? ia1!] IN W)
```

```
COMP1 = MINIMIZE COMP1
PRINT "andif" COMP1
```
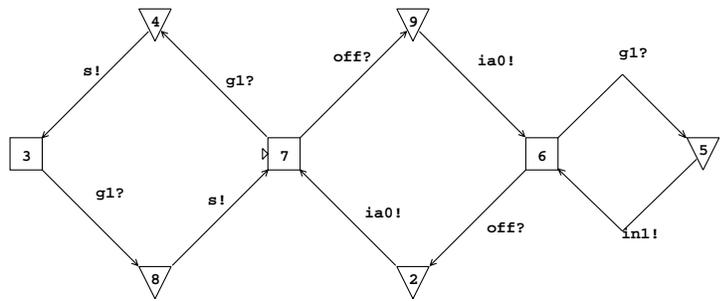
The final remainder is given in Figure 11.

Figure 11. The final remainder

Looking at the specification of the switch, it is almost the same, the difference being that in the original specification, communicating $r0; off$ would lead back to the initial state, whereas in this remainder, $g1; s$ must be communicated twice, before the initial state is reached again.

It follows that this remainder can be implemented using a second switch. An environment could communicate $r0; off$ twice, and be back in the initial state, but also that this is a genuine refinement. We could do with a switch element that is specified by the state graph in Figure 8. Actually, Bill Coates [Coa98] cleverly uses this asymmetry to use an and-element, instead of a waiton (latch) and a xor, in the implementation of this particular switch.

The original remainder from Figure 6 can hence be implemented using two switches, as shown in Figure 12. These switches must behave at least as deterministic as specified in Figure 11.
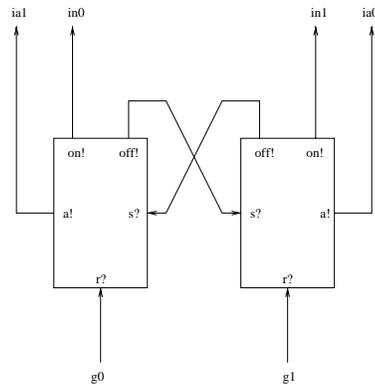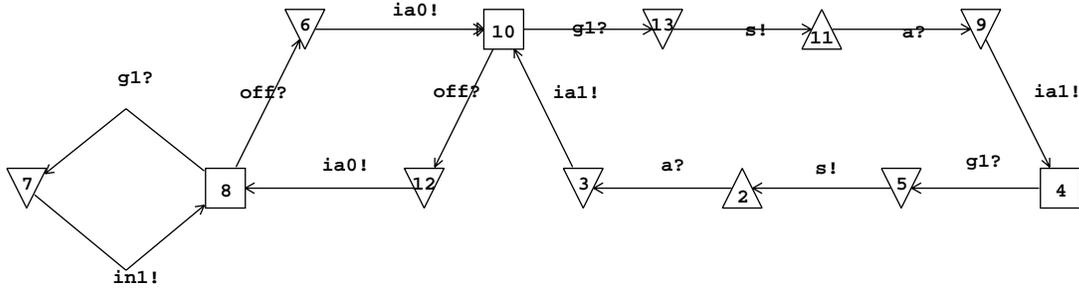

Figure 12. Two switches

6 ia0! 10 g1? 13 s! 11 a? 9

g1? off? off? ia1! ia1!

7 8 ia0! 12 3 a? 2 s! 5 g1? 4

in1!

Figure 10. A more deterministic remainder

6 Conclusion and Further Work

We have shown that the two models $\mathbf{X^2DI}$ and $\mathbf{XDI}$ are complete lattices. A consequence is that monotonic functions have a least fixed point, which we can use for recursive schemes. We have shown that in models with limited progress expressiveness and finitary executions, parallel composition in general does not distribute over arbitrary infima without introducing unhealthiness. The counter example is general enough to hold for a family of models.

We have shown that the unhealthy processes can be used as intermediaries. We have implemented the operations in a tool. This tool does not use optimized graph representation but is still fast enough to help with non trivial designs.

Based on the factorization equation DI synthesis seems to be feasible. The idea is to find components that are small enough to implement speed independently, and together form a DI decomposition of the original specification. This strategy results in an asynchronous circuit where all timing constraints are local and easy to verify. The connections between the components are DI in the sense that no timing assumptions need to be made on these wires.

On the other hand a complete DI decomposition down to basic elements like C-elements, forks, merges, decision-waits and arbiters is useful to have anyway, since it can be compared to a speed independent or otherwise asynchronous implementation to help find or verify all timing assumptions.

The guarded choice as a Galois adjoint of the after operator is another issue that needs to be investigated. The after operator distributes over $\sqcap$ and $\sqcup$ in both $\mathbf{X^2DI}$ and $\mathbf{XDI}$ model. Hence there exist functions f, g such that

(36) $\qquad P/a \sqsubseteq Q \quad \equiv \quad P \sqsubseteq f.Q$

(37) $\qquad P \sqsubseteq Q/a \quad \equiv \quad g.P \sqsubseteq Q$

It is interesting to find out how these functions are related to the DI algebra guarded choice. If we can add a guarded choice operator to the $\mathbf{X^2DI}$ model, we are able to specify processes in the same easy manner as they are specified in the DI algebra, but have the added benefit of the factorization rewrite rule.

There are a lot of open questions still in the $\mathbf{X^2DI}$ model. Many question have been raised in [Ver94]. In this paper we have given some answers, but some questions remain. For example:

- Can all specifications that do not contain dynamic non-determinism be implemented by a composition that does not contain arbiters ?

- Is it decidable whether nondeterminism is dynamic ? And if so, is a witness constructible ?

The parallel composition that we have defined and analyzed in this paper may help in answering these questions.

The current interface to ludwig is as it is given in this paper. We are working on a graphical user interface.

7 Acknowledgments

References

[Ber92] Kees van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[Coa98] Bill Coates. Design of naking arbiter steering logic. Technical Report SML# 98:0263, Sun Microsystems Laboratories, 1998.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

11

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[Ebe91] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[EMN$^+$] Jo Ebergen, Charles Molnar, Radu Negulescu, Huub Schols, Bob Sproull (editor), Jan Tijmen Udding, and Tom Verhoeff. And/if a file format for exchanging finite automata descriptions. `http://edis.win.tue.nl/and-if`.

[GLM$^+$] Rix Groenboom, Paul Lucassen, Willem Mallon, Indra Polak, and Jan Tijmen Udding. Asynchronous research group groningen. `http://www.cs.rug.nl/~willem/ARGG`.

[Hes92] Wim H. Hesselink. *Programs, Recursion and Unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[Jos92] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[JU93] M. B. Josephs and J. T. Udding. An overview of DI algebra. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[Luc94] Paul G. Lucassen. *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*. PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, May 1994.

[MU98a] Willem C. Mallon and Jan Tijmen Udding. Building finite automatons from DI specifications. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193, 1998.

[MU98b] Willem C. Mallon and Jan Tijmen Udding. Delay insensitive composition and decomposition. Technical Report CSN 98-10-19, Dept. of Comp. Science, Univ. of Groningen, 1998.

[MU98c] Willem C. Mallon and Jan Tijmen Udding. Divergence extensions and alternations. Technical Report in preparation, Dept. of Comp. Science, Univ. of Groningen, 1998.

[Neg98] Radu Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, Dept. of Computer Science, Univ. of Waterloo, Canada, August 1998.

[SSM94] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.

[Ver] Tom Verhoeff. Encyclopedia of Delay-Insensitive Systems (EDIS). `http://edis.win.tue.nl/`.

[Ver94] Tom Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[Ver98a] Tom Verhoeff. Analyzing specifications for delay-insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 172–183, 1998.

[Ver98b] Tom Verhoeff. Factorization in process domains. Technical Report 98/10, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1998.