

**SOFTWARE VERIFICATION RESEARCH CENTRE**  
**DEPARTMENT OF COMPUTER SCIENCE**  
**THE UNIVERSITY OF QUEENSLAND**

Queensland 4072  
Australia

**TECHNICAL REPORT**

No. 94-32

**Using the Refinement Calculus  
for Dataflow Processes**

**Brendan P. Mahony**

**October 1994**

**Phone: +61 7 3365 1003**

**Fax: +61 7 3365 1533**

**Note:** Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

# Using the Refinement Calculus for Dataflow Processes

Brendan P. Mahony  
Department of Computer Science  
University of Queensland

## Abstract

The refinement calculus, based on predicate transformer semantics, is proving useful in the construction of sequential programs. It is argued that the refinement calculus provides a suitable development formalism for (possibly real-time) dataflow-like processes. Conventional precondition and postcondition specifications of sequential programs are generalised to assumption and effect specifications of processes. Such specifications may be given predicate transformer semantics and then formally refined to implementations in much the same way as precondition and postcondition specifications of sequential programs. A minimal collection of dataflow-process operators and associated refinement laws is considered. Small examples of dataflow-process development are presented to demonstrate the utility of the approach. The compositionality of the approach is considered in detail.

## 1 Introduction

The aims of formal process development methods are firstly to ensure our expectations of the desired system are unambiguously specified, and secondly to provide formal support for the construction of said processes, that we might have greater confidence that they implement the desired system.

This paper considers the algebraic properties of a formal method which supports a refinement calculus approach to the construction of dataflow-like processes. The refinement calculus is based on a broad-spectrum language which allows the expression of all products of the development process, specifications, designs, and executables. Highly abstract requirements specifications are transformed by the application of *refinement laws* through a series of increasingly detailed design specifications to a final executable specification expressed in a distinguished subset of the specification language known as the *target language*. The executables which are refinements of a given specification are called its *implementations*.

The refinement calculus encourages a separation of specification, design and implementation concerns, so that the appropriateness of specification, design and implementation decisions may be demonstrated at a convenient level of abstraction. In this way the complexity of each design and verification step is considerably reduced. One novel contribution of this paper is the recognition that the refinement calculus can be made applicable to a wide variety of application domains simply by varying the subset of the specification language identified as the target language and adopting a specialised selection of process operators and related refinement laws.

The main focus of this paper is on demonstrating the utility of the refinement approach to the development of dataflow processes through some small examples. The emphasis is on the use of the refinement laws associated with the operators of the specification language and in particular the parallel composition operator. Consideration is given to the issue of compositionality, in particular with respect to the dangers associated with introducing trivial feedback loops into networks. The semantics of the composition operator and the proofs of the associated refinement laws are the subject of a companion paper [18]. A large case study in the use of the dataflow refinement calculus may be found elsewhere [20].

The balance of this section is devoted to the introduction and definition of various concepts important to the rest of the paper. In Section 2 a dataflow specification statement is introduced and some small examples demonstrate its use and also the generality of the approach. Section 3 introduces the process refinement relation together with a collection of dataflow process operators that are compositional with respect to refinement. Some small case studies in process refinement and design are undertaken. The methods described in these sections are general in their application and may be used in conjunction with a wide variety of process implementation languages, even hardware components may be treated. In Section 4 a simple dataflow target language is considered and the it is argued that the refinement calculus provides a suitably compositional development method for that language.

## 1.1 Processes act on systems

The philosophical starting point for applying the refinement calculus to the development of dataflow processes is to (perhaps artificially) sharply distinguish between system and process, the notion of system being existential, that of process being constructive. The relationship between process and system may succinctly be stated:

*A process acts upon an existing system for the purpose of constructing a new system exhibiting a desired structure.*

This is suggestive of the relationship between function and domain. In support of this distinction a formalism is proposed which views systems as basic objects

and processes as mappings between systems. The advantage of this approach lies in its generality. Whatever the methods used to model a class of systems, the class of mappings between those systems is an appropriate formalism for describing the construction of said systems.

The predicate calculus provides a convenient formalism for the description of many systems. The components of a system are represented as variables and the predicates allow statements about the structure of the system. It follows then that an appropriate formalism for the description of the processes which construct such systems lies in the functions from predicates to predicates, the *predicate transformers*. The view is adopted that a process may be described by a mapping from required systems to existing systems. This is a *weakest environment assumption* view, it tells us the kind of environment we must apply the process to in order to create a system with a desired structure.

In the case of dataflow processes the environment observables will always be a proper subset of the observables in the required system. The observables of the environment are the *inputs* and the extra observables in the required system are the *outputs* or *constructions* of the dataflow process.

## 1.2 The refinement calculus

The predicate transformer model has been successfully used to aid in the development of the subclass of processes called sequential programs. The weakest precondition program semantics of Dijkstra [6] have been extended by several authors [4, 25, 26] into a wide-spectrum *refinement calculus* that supports the top-down development paradigm for sequential programs. Both specifications and programs are expressed within the same calculus and related via the *refinement* relation.

The thesis of this paper is that the refinement calculus may also be used to model dataflow processes.

### Notation for predicates and transformers

This paper makes use of predicates for the description of system models. In decreasing order of precedence, the predicate operators are  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$ , and  $\Leftrightarrow$ . The operator  $\Rightarrow$  associates to the right; that is  $Q_1 \Rightarrow Q_2 \Rightarrow Q_3$  means  $Q_1 \Rightarrow (Q_2 \Rightarrow Q_3)$ . Quantified variables are bound to the end of the current scope. For example  $\forall \vec{v} \bullet Q_1 \wedge Q_2$  means  $\forall \vec{v} \bullet (Q_1 \wedge Q_2)$  and not  $(\forall \vec{v} \bullet Q_1) \wedge Q_2$ . The order on predicates is *entailment*, written  $Q_1 \Rightarrow Q_2$  and true if and only if the predicate  $Q_1 \Rightarrow Q_2$  is valid. Entailment equivalence of predicates is written  $Q_1 \equiv Q_2$ .

The *Z* specification language [7, 29] provides a convenient formal notation for the predicate calculus. *Z* schemas consist of a list of variable declarations (the *signature*) and a predicate that describes properties satisfied by the variables.

In this paper the notion of predicate is identified with that of the Z schema. Predicates presented outside of schema boxes have implicit signatures.

A signature is a set of variable names assigned to types. The assignment of a variable name to a type is written,  $v : T$ . The signature of a predicate (schema) is often called the *state vector* of the system being discussed. By convention vectorised variables (for example  $\vec{v}$ ) are used to represent signatures.

It is useful to place a strong typing system on predicates, based on their variable signatures, writing  $\text{PRED}_{\vec{v}}$  for the collection of predicates/schemas over the set of observables  $\vec{v}$ . A Z schema with signature  $\vec{u}$  belongs to  $\text{PRED}_{\vec{v}}$  for all  $\vec{v} \supseteq \vec{u}$ .  $\text{PRED}$  is the collection of all predicates/Z-schemas and when ordered by entailment  $\text{PRED}$  forms a lattice with top and bottom elements **true** and **false** respectively.

Following Back and von Wright [5] the notation  $\text{MTRAN}_{\vec{u} \rightarrow \vec{v}}$  is used to represent the class of monotonic predicate transformers from  $\text{PRED}_{\vec{v}}$  to  $\text{PRED}_{\vec{u}}$ . The reverse order of the signatures  $\vec{v}$  and  $\vec{u}$  is indicative of the fact that the transformers are used to support a weakest assumption process model. The process modelled by a transformer in  $\text{MTRAN}_{\vec{u} \rightarrow \vec{v}}$  acts on a system with observables  $\vec{u}$  to construct a system with observables  $\vec{v}$ , but is modelled by a mapping which for each desired effect on the observables  $\vec{v}$  returns the weakest required assumption about the observables  $\vec{u}$ . In addition,  $\text{MTRAN}_{\vec{u} \xrightarrow{+} \vec{z}}$  represents those predicate transformers which map  $\text{PRED}_{\vec{u} \cup \vec{z}}$  to  $\text{PRED}_{\vec{u}}$ , with  $\vec{z}$  disjoint from  $\vec{u}$ , in such a way as to preserve the original properties of  $\vec{u}$ . These are used to model processes which construct outputs  $\vec{z}$  from inputs  $\vec{u}$ . This typing scheme for predicate transformers is discussed in full by Mahony [17].

## 2 Writing specifications

The basic specification tool of the dataflow refinement calculus is the specification statement which corresponds closely to the program specification statement of Morgan [25], but rather than describing a transition between initial and final states, it describes a transition from inputs to outputs. The main differences are: the frame indicates the *necessary* outputs of the process, rather than the variables that *may* be updated; in place of preconditions there are assumptions about the process environment (the assumption may refer only to the input variables); and in place of postconditions there are required effects on the outputs.

**Definition 2.1** (Dataflow specification statement)

A (*dataflow*) *specification statement* consists of an output frame,  $\vec{x}$ , detailing the observables constructed by the process, an assumption,  $A : \text{PRED}_{\vec{u}}$  ( $\vec{x}$  disjoint from  $\vec{u}$ ), and an effect,  $E : \text{PRED}_{\vec{u} \cup \vec{x}}$ , and is written

$$+\vec{x} [A, E].$$

This expression denotes the predicate transformer

$$\lambda \phi : \text{PRED}_{\vec{u} \cup \vec{x}} \bullet (A \wedge \forall \vec{x} \bullet E \Rightarrow \phi),$$

which is the least refined element of  $\text{MTRAN}_{\vec{u} \rightarrow \vec{x}}$  which constructs outputs  $\vec{x}$  satisfying the property  $E$ , provided its inputs,  $\vec{u}$ , satisfy the property  $A$ .  $\heartsuit$

The semantics of the specification statement are presented here because of its importance in calculating specifications for system subcomponents, as demonstrated in Example 3.5 and Example 3.6. The semantics of the other dataflow operators fall beyond the scope of this paper.

The specification statement is a general tool and may be used to specify the requirements of processes in a wide variety of application domains.

**Example 2.2 (A square-root function)** A function which calculates the square-root of a non-negative integer,  $i$ ,

<i>NonNeg</i>
$i : \mathbb{Z}$
$i \geq 0$

and places it in an output variable,  $o$ ,

<i>SQRT</i>
$i, o : \mathbb{Z}$
$o = \lfloor \sqrt{i} \rfloor$

can be specified by

$$+o [\text{NonNeg}, \text{SQRT}].$$

□

**Example 2.3 (A summing machine)** A machine which processes a stream of integers,  $a$ , and produces a stream  $s$  consisting of the successive partial sums of the input stream,

<i>Sum</i>
$a, s : \mathbb{N} \rightarrow \mathbb{Z}$
$\forall n : \mathbb{N} \bullet s(n) = \sum_{j : 0..n} a(j)$

can be specified by

$$+s [\text{true}, \text{Sum}].$$

□

The summing machine specification describes the behaviour of the process only in the case that the input stream is non-terminating. This is somewhat unusual in that it is common practice to specify the behaviour of processes on finite inputs and to then deduce infinite behaviours through some form of continuity assumption. Instead, we specify the infinite behaviours and rely on the causality of all implementations to ensure the correct behaviour for finite input streams. No implementation can ever determine that its input stream has terminated or that it terminate. All implementations must therefore treat every input stream as a prefix of an infinite stream.

The time model used here is a simple time-ordered enumeration of the important system events on each stream,  $s(n)$  is simply the  $n^{\text{th}}$  communication on the  $s$  stream. This is no implied notion of a system clock or a regular progression of events.

**Example 2.4 (A Bit)** A bit stores a single binary digit,

$$BIN ::= hi \mid lo$$

constantly providing the value of the digit on a digital wire

$$Q : \mathbb{R} \longrightarrow BIN.$$

The real numbers ( $\mathbb{R}$ ) are used to model time.

The value of the bit is set to *lo* by sending a *hi* signal on a reset line,  $R$ , while maintaining a *lo* signal on a set line,  $S$ . Once reset, the value of the bit should remain *lo* for as long as the set signal is *lo*. The value of the bit may be set *hi* via a similar process, except with the *hi* signal on  $S$  and the *lo* on  $R$ . The bit is only required to respond properly when the input signals  $R$  and  $S$  are complements and are stable for at least  $\delta : \mathbb{R}$ . In other circumstances, the behaviour is not specified, allowing for example the possibility of meta-stable states resulting from some input signals. Any delay in setting the outputs must be no more than  $\delta$ .

<i>Bit</i>
$R, S, Q : \mathbb{R} \longrightarrow BIN$
$\forall x, y, z : \mathbb{R} \bullet x + \delta < y < z \Rightarrow$
$(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow R(t) = hi) \wedge$ $(\forall t : \mathbb{R} \bullet x < t < z \Rightarrow S(t) = lo) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow Q(t) = lo)$
$(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow S(t) = hi) \wedge$ $(\forall t : \mathbb{R} \bullet x < t < z \Rightarrow R(t) = lo) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow Q(t) = hi)$

The bit constructor process can be specified by

$$+Q [\text{true}, \text{Bit}].$$

□

### 3 Using the refinement calculus

The refinement calculus supports top-down development through the use of the refinement relation [5, 25], which determines when an implementation may safely replace a specification, and through the use of process operators which allow complex specifications to be expressed as combinations of simple specifications.

**Definition 3.1** (Refinement)

A process,  $T : \text{MTRAN}_{\bar{u} \rightarrow \bar{z}}$ , refines a specification,  $S : \text{MTRAN}_{\bar{u} \rightarrow \bar{z}}$ , written

$$S \sqsubseteq T,$$

if and only if for any required system behavior,  $R : \text{PRED}_{\bar{u} \cup \bar{z}}$ ,

$$S(R) \Rightarrow T(R).$$

That is, wherever  $S$  would achieve a required system behavior,  $T$  also achieves that behavior. ♡

The refinement relation allows abstract dataflow process specifications to gradually be replaced by networks of more concrete process designs, until finally one arrives at a description of a process that is constructed entirely as an expression in the chosen target language (see Section 4). In the general course of using the refinement calculus refinement is not verified directly, but rather inferred from a collection of high level rules known as *refinement laws* which characterise the interaction between the refinement relation and the process network constructors.

It should be noted that this refinement relation and the refinement laws developed from it are essentially safety criteria. It ensures that any implementation identified by the refinement process satisfies the original specification, but it does not ensure that each refinement step makes progress toward an implementation. It is possible to make legal refinement steps which can never lead to an implementation, just as it is possible in the first instance to describe a top-level specification of an unimplementable process. It remains part of the design craft to ensure that at each refinement step progress is made toward an implementation.

This section develops a minimal language for the specification and construction of process networks and introduces the associated refinement laws. The utility of the language and associated refinement laws are demonstrated in some

small refinement examples. It is convenient to present these laws as inference rules. A law is written in the form

$$\frac{P}{P'} \quad [ C ]$$

and is read as follows: wherever the expression  $P$  occurs it may be replaced by the expression  $P'$ , provided the side condition  $C$  is true. The ability to replace  $P$  wherever it occurs in a larger expression derives from the principle of compositionality.

### 3.1 The principle of compositionality

The main tool which makes this top down design method possible is the *principle of compositionality*. It ensures that any process specification expression may be legally implemented by implementing each of its component parts in isolation.

The principle of compositionality requires in the first instance that each process operator be monotonic with respect to the refinement relation. Though the existing program refinement calculus literature provides a range of compositional process operators [23, 32] which are equally applicable in the dataflow case, the primary tools for constructing process networks are the parallel composition operator and the hiding operator, both of which obey the principle of compositionality with respect to refinement.

Another aspect of the principle of compositionality identifies itself at the end of the development process. The process operators must be such that any composition of process implementations is itself a process implementation, regardless of any interactions between the component processes. It is in this context that the parallel operator provides a considerable challenge to the construction of compositional refinement methods. It is usual to model the parallel operator as logical conjunction in a predicate-based process semantics [1, 3], but conjunction is not a compositional operator. Consider the predicate specification  $x > 3 \wedge x < 5$ . The predicate  $x = 4$  is one acceptable implementation of this specification, so it is implementable. Equally, the predicate  $x = 6$  is an implementation of  $x > 3$  and the predicate  $x = 2$  an implementation of  $x < 5$ . In a compositional system the conjunction of these two implementations would form an implementation of the original specification. However the conjunction  $x = 6 \wedge x = 2$ , whilst certainly a (magical) refinement of the original specification, is clearly not an implementation, even though its component processes are both implementations of their respective component specifications.

Obviously this example relies strongly on the fact that both processes control the same observable, but even with a strict separation of state similar effects are possible, as demonstrated in detail in Section 4.

In order to overcome this difficulty, existing methods have generally opted to place severe restrictions on the specification language and/or upon the use of

decomposition [22, 1, 28]. We prefer to leave the specification language relatively free of constraints and instead to adopt a strict target language. Again the details are discussed in Section 4

The parallel operator described in this section is compositional with respect to refinement, but requires no a priori restrictions on the assumption and effect predicates that may be used in the specification language. The operator supports a powerful network decomposition rule which allows the introduction of behavioral properties of component processes as environment assumptions in their sibling processes, but which does place simple restrictions on the way in which effects may be decomposed.

### 3.2 Specification statements

For specification statements the refinement relation corresponds to weakening the assumptions

**Law R1** (Weaken assumption)

$$\frac{+\vec{x} [A_1, E]}{+\vec{x} [A_2, E]} \quad [A_1 \Rightarrow A_2]$$

or to strengthening the effects.

**Law R2** (Strengthen effect)

$$\frac{+\vec{x} [A, E_1]}{+\vec{x} [A, E_2]} \quad [A \Rightarrow (\forall \vec{x} \bullet E_2 \Rightarrow E_1)]$$

In sequential program refinement [25] a third law allows the removal of variables from the frame, but this is not allowed in dataflow refinement since any refinement must construct the same outputs as its specification.

**Example 3.2 (The stream square-root process)** Using R2 the effect *SQRT* of Example 2.2 may be transformed to ensure a result even for negative inputs

$$\begin{aligned} &+o [NonNeg, SQRT] \\ \sqsubseteq &+o [NonNeg, o = \lfloor \sqrt{(|i|)} \rfloor] \end{aligned}$$

and then the assumption predicate may be relaxed using R1.

$$\sqsubseteq +o [\text{true}, o = \lfloor \sqrt{(|i|)} \rfloor]$$

□

### 3.3 Process composition

The basic tool for constructing process networks is parallel composition.

**Definition 3.3** For predicate transformer specifications,

$$S_1 : \text{MTRAN}_{\vec{u} \cup \vec{y} \xrightarrow{+} \vec{x}} \text{ and } S_2 : \text{MTRAN}_{\vec{u} \cup \vec{x} \xrightarrow{+} \vec{y}}$$

( $\vec{u}$ ,  $\vec{x}$ , and  $\vec{y}$  mutually disjoint), their parallel composition

$$S_1 \parallel S_2 \in \text{MTRAN}_{\vec{u} \xrightarrow{+} \vec{x} \cup \vec{y}}$$

forms a process network with inputs  $\vec{u}$  and outputs  $\vec{x} \cup \vec{y}$ , in which the outputs of  $S_1$  form part of the environment of  $S_2$  and vice versa.  $\heartsuit$

Within the refinement calculus method, parallel composition is used to decompose the assumptions and effects of a complex process specification statement so as to create a network of simpler subprocesses which may then be refined in isolation.

The simplest possible decomposition is to partition the parent process completely, making no use of the properties of its siblings in the development of each component.

**Law R3**

$$\frac{+\vec{x} \cup \vec{y} [A, E_{\vec{x}} \wedge E_{\vec{y}}]}{(+\vec{x} [A, E_{\vec{x}}]) \parallel (+\vec{y} [A, E_{\vec{y}}])}$$

However, in the development of individual subprocesses, it is efficient to be able to make use of outputs from sibling processes within the network, even to develop feedback loops. This is supported through a refinement law which allows the introduction the effects of sibling processes as assumptions in each subprocess.

**Law R4** For  $A \in \text{PRED}_{\vec{u}}$ ,  $E_{\vec{x}} \in \text{PRED}_{\vec{u} \cup \vec{x}}$ ,  $E_{\vec{y}} \in \text{PRED}_{\vec{u} \cup \vec{y}}$ ,

$$\frac{+\vec{x} \cup \vec{y} [A, E_{\vec{x}} \wedge E_{\vec{y}}]}{(+\vec{x} [A \wedge E_{\vec{y}}, E_{\vec{x}}]) \parallel (+\vec{y} [A \wedge E_{\vec{x}}, E_{\vec{y}}])} \quad \left[ \begin{array}{l} A \Rightarrow (\exists \vec{x} \bullet E_{\vec{x}}) \\ A \Rightarrow (\exists \vec{y} \bullet E_{\vec{y}}) \end{array} \right]$$

It is instructive to pause to consider the restrictions that this rule places on the ways in which processes may be decomposed.

The side condition ensures that the assumption predicates introduced into component subprocesses do not constitute a strengthening of the assumptions made in the original specification, in the sense that they assume only things whose existence can be deduced from the original assumptions. If the original specification is “sensible” in the sense that its assumptions are logically sufficient

to enable the satisfaction of its effects, this side condition will naturally be satisfied.

A more subtle, but greater restriction on the application of the decomposition rule lies in the typing constraints placed on the effects predicates which may be introduced as assumptions. Neither predicate may mention the outputs of its sibling process. Thus it is only possible to make use of commitments made to the global environment and hence it is not possible to make assumptions about the reactions of sibling components in the development of a component. All such interactions must be determined and made use of prior to decomposition. Whilst this represents a strong discipline on the use of network decomposition, it does not represent a significant intellectual overhead to the decomposition rule since this requirement can be easily checked statically and is readily amenable to automatic checking through well understood type checking technology.

The parallel composition operator is associative and hence R3 and R4 may be generalised to arbitrary finite compositions of processes.

## Local variables

When local constructions are required to support the action of a process they may be hidden from the external world.

**Definition 3.4** (Local constructions)

For  $S : \text{MTRAN}_{\vec{u} \rightarrow \vec{w} \cup \vec{z}}$  ( $\vec{w}$  disjoint from  $\vec{u}$  and  $\vec{z}$ ), the process

$$[[S \setminus \vec{w}]] \in \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}$$

uses the outputs  $\vec{w}$  as local constructions. ♡

Local constructions are outputs of an internal process which are hidden from the outside world. In general they will be used for protected communications between sibling processes.

The notion of hiding local variables is common to many formalisms. The corresponding construct for sequential programs is the local variable, which is both a local input and and a local output [4, 23]. In [4] local variables are implemented through an operation to add a new variable and an operation to remove an existing one, the latter of which is similar to the hiding operator described above.

Local constructions may be freely introduced if they are not already mentioned in a specification statement.

**Law R5** (Introduce local construction)

$$\frac{+\vec{z} [A, E]}{[[+\vec{w} \cup \vec{z} [A, E] \setminus \vec{w}]]} \quad [ \vec{w} \text{ new variables } ]$$

**Example 3.5 (Designing the summing machine)** Using the process constructors and related refinement laws introduced above a first-level design decomposition of the summing machine introduced in Example 2.3 is demonstrated.

The summing machine may be implemented with the use of an adding unit

$$\frac{\textit{Add}}{a, b, s : \mathbb{N} \rightarrow \mathbb{Z}} \quad \forall n : \mathbb{N} \bullet s(n) = a(n) + b(n)$$

with the input stream  $a$  attached to one of its inputs.

The required properties of the second input  $b$  can be deduced by calculating the weakest assumption required for the adder to achieve the *Sum* effect.

$$\begin{aligned} & +s [\textit{true}, \textit{Add}](\textit{Sum}) \\ & \equiv \forall s : \mathbb{N} \rightarrow \mathbb{Z} \bullet \\ & \quad (\forall n : \mathbb{N} \bullet s(n) = a(n) + b(n)) \Rightarrow \\ & \quad (\forall n : \mathbb{N} \bullet s(n) = \sum j : 1..n \bullet a(j)) \\ & \equiv \forall s : \mathbb{N} \rightarrow \mathbb{Z} \bullet \\ & \quad (\forall n : \mathbb{N} \bullet s(n) = a(n) + b(n)) \Rightarrow \\ & \quad (\forall n : \mathbb{N} \bullet s(n) = a(n) + \sum j : 1..n-1 \bullet a(j)) \\ & \Leftarrow \forall n : \mathbb{N} \bullet b(n) = \sum j : 1..n-1 \bullet a(j) \end{aligned}$$

Thus if  $b$  is at each step the sum of the values on  $a$  up to the previous step, the adder will achieve the *Sum* effect.

$$\frac{\textit{Subsum}}{a, b : \mathbb{N} \rightarrow \mathbb{Z}} \quad b(n) = \sum j : 0..n-1 \bullet a(j)$$

This can clearly be achieved by placing the previous value of  $s$  on  $b$ .

$$\frac{\textit{Feedback}}{b, s : \mathbb{N} \rightarrow \mathbb{Z}} \quad \begin{array}{l} b(0) = 0 \\ \forall n : \mathbb{N}_1 \bullet b(n) = s(n-1) \end{array}$$

This design is pictured in Figure 1.

The formal refinement sequence is begun by introducing the local construction  $b$ .

$$\begin{aligned} & +s [\textit{true}, \textit{Sum}] \\ & \sqsubseteq [+b, s [\textit{true}, \textit{Sum}] \setminus b] \end{aligned} \quad [\text{R5}]$$

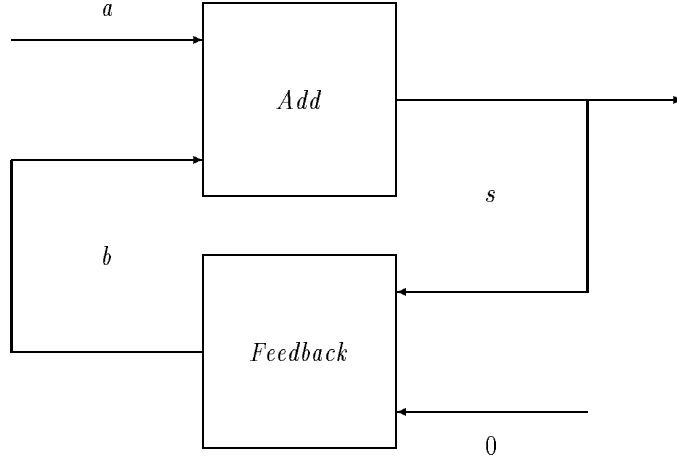


Figure 1: First level design of summing machine.

Next, strengthen the effect with the required behavior calculated for  $b$ .

$$\sqsubseteq \llbracket +b, s \text{ [true, Sum} \wedge \text{Subsum]} \setminus b \rrbracket \quad [\text{R2}]$$

Now the specification may be split into a parallel combination, one subprocess to construct  $s$  and the other to construct  $b$ .

$$\sqsubseteq \llbracket (+s \text{ [Subsum, Sum]} \parallel (+b \text{ [Sum, Subsum]} \setminus b) \rrbracket \quad [\text{R4}]$$

As already calculated,  $\text{Subsum} \wedge \text{Add} \Rightarrow \text{Sum}$  and  $\text{Sum} \wedge \text{FeedBack} \Rightarrow \text{Subsum}$ , so the subprocesses of the above parallel composition may be refined with the adder process and the feedback process respectively,

$$\sqsubseteq \llbracket (+s \text{ [true, Add]} \parallel (+b \text{ [true, Feedback]} \setminus b) \rrbracket \quad [\text{R2, R1} \times 2]$$

giving us a design that at each stage calculates the next partial sum by adding the value of the last partial sum to the current input.  $\square$

**Example 3.6 (Designing the bit)** The first-level design of the bit process specified in Example 2.4 uses two nor gates, the output of each gate connected to one input of the other, as illustrated in Figure 2.

A nor gate calculates the logical nor of its input wires, with a reaction delay of no more than  $\epsilon : \mathbb{R}$ , provided that its inputs remain stable for this period.

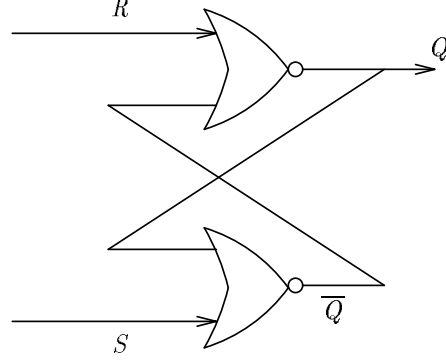


Figure 2: Circuit diagram for a flipflop.

<i>Nor</i>
$i_1, i_2, o : \mathbb{R} \rightarrow BIN$
$\forall x, y : \mathbb{R} \bullet x + c < y \Rightarrow$ $(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow i_1(t) = hi \vee i_2(t) = hi) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + c < t < y \Rightarrow o(t) = lo)$ $(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow i_1(t) = lo \wedge i_2(t) = lo) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + c < t < y \Rightarrow o(t) = hi)$

From the symmetry of the design it can be seen that, if it operates correctly,  $\overline{Q}$  will essentially be the inverse of the  $Q$ , that is  $\overline{Q}$  is set *lo* when  $Q$  is set *hi* and vice versa, modulo some small delays.

$\overline{Bit}$
$R, S, \overline{Q} : \mathbb{R} \rightarrow BIN$
$\forall x, y, z : \mathbb{R} \bullet x + \delta < y < z \Rightarrow$ $(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow R(t) = hi) \wedge$ $(\forall t : \mathbb{R} \bullet x < t < z \Rightarrow S(t) = lo) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow \overline{Q}(t) = hi)$ $(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow S(t) = hi) \wedge$ $(\forall t : \mathbb{R} \bullet x < t < z \Rightarrow R(t) = lo) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow \overline{Q}(t) = lo)$

The correct behavior of the circuit may be determined by considering the weakest assumption semantics of the nor gate process. Correct behavior requires

that the condition

$$\begin{aligned} &+ Q [\text{true}, \text{Nor}[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}]](\text{Bit}) \\ &\equiv \forall Q : \mathbb{R} \longrightarrow \text{BIN} \bullet \text{Nor}[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}] \Rightarrow \text{Bit}. \end{aligned}$$

be ensured by the environment.

Assume  $\overline{\text{Bit}}$  and choose arbitrary  $x, y, z : \mathbb{R}$  so that  $x + \delta < y < z$ , then when

$$\begin{aligned} &(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow R(t) = hi) \wedge \\ &(\forall t : \mathbb{R} \bullet x < t < z \Rightarrow S(t) = lo) \end{aligned}$$

it follows that

$$(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow \overline{Q}(t) = hi),$$

and  $\text{Nor}[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}]$  will ensure that

$$(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow Q(t) = lo)$$

provided that  $\epsilon < \delta$ .

When

$$\begin{aligned} &(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow S(t) = hi) \wedge \\ &(\forall t : \mathbb{R} \bullet x < t < z \Rightarrow R(t) = lo) \end{aligned}$$

$\overline{\text{Bit}}$  ensures that

$$(\forall t : \mathbb{R} \bullet x + \delta < t < z \Rightarrow \overline{Q}(t) = lo),$$

but here  $\text{Nor}[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}]$  can only assure us that

$$(\forall t : \mathbb{R} \bullet x + \delta + \epsilon < t < z \Rightarrow Q(t) = hi)$$

since  $Q$  will not change to  $hi$  until both  $R$  and  $\overline{Q}$  are  $lo$ . Thus the requirements of  $\overline{Q}$  must be strengthened to remove the extra delay of time  $\epsilon$ .

$\begin{aligned} &\overline{S \text{ resets } \overline{Q}} \\ &S, \overline{Q} : \mathbb{R} \longrightarrow \text{BIN} \\ &\forall x, y : \mathbb{R} \bullet x + \epsilon < y \Rightarrow \\ &\quad (\forall t : \mathbb{R} \bullet x < t < y \Rightarrow S(t) = hi) \Rightarrow \\ &\quad (\forall t : \mathbb{R} \bullet x + \delta - \epsilon < t < y \Rightarrow \overline{Q}(t) = lo) \end{aligned}$
--

By symmetry, it is also necessary to strengthen the required effect on  $Q$ .

$\frac{RresetsQ}{R, Q : \mathbb{R} \rightarrow BIN}$ $\forall x, y : \mathbb{R} \bullet x + \epsilon < y \Rightarrow$ $(\forall t : \mathbb{R} \bullet x < t < y \Rightarrow R(t) = hi) \Rightarrow$ $(\forall t : \mathbb{R} \bullet x + \delta - \epsilon < t < y \Rightarrow Q(t) = lo)$
---

Again the weakest assumption semantics of the nor gate process can determine whether this extra effect is achieved by this design. It may be seen that

$$\forall Q : \mathbb{R} \rightarrow BIN \bullet Nor[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}] \Rightarrow RresetsQ.$$

provided that  $\epsilon < \delta - \epsilon$ , which is to say that  $2 * \epsilon < \delta$ .

Having ensured the correctness of the design through the appropriate calculations the formal refinement sequence may be presented.

The first step is to introduce the intermediate wire  $\overline{Q} : \mathbb{R} \rightarrow BIN$ .

$$\begin{aligned} &+Q [\text{true}, Bit] \\ \sqsubseteq &[[+\overline{Q}, Q [\text{true}, Bit] \setminus \overline{Q}]] \end{aligned} \tag{R5}$$

Next, strengthen the effect as calculated above.

$$\sqsubseteq [[+\overline{Q}, Q [\text{true}, Bit \wedge RresetsQ \wedge \overline{Bit} \wedge Sresets\overline{Q}] \setminus \overline{Q}]] \tag{R2}$$

The process may now be decomposed according to the circuit diagram in Figure 2.

$$\begin{aligned} \sqsubseteq &[[+\overline{Q} [Bit \wedge RresetsQ, \overline{Bit} \wedge Sresets\overline{Q}]] \\ &+Q [\overline{Bit} \wedge Sresets\overline{Q}, Bit \wedge RresetsQ] \\ &\setminus \overline{Q}]] \end{aligned} \tag{R4}$$

As calculated above,

$$\overline{Bit} \wedge Sresets\overline{Q} \wedge Nor[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}] \Rightarrow Bit \wedge RresetsQ$$

and by symmetry

$$Bit \wedge RresetsQ \wedge Nor[\frac{Q}{i_1}, \frac{S}{i_2}, \frac{\overline{Q}}{o}] \Rightarrow \overline{Bit} \wedge Sresets\overline{Q}$$

so each subprocess may be refined by an instance of the nor gate process.

$$\begin{aligned} \sqsubseteq &[[+\overline{Q} [\text{true}, Nor[\frac{Q}{i_1}, \frac{S}{i_2}, \frac{\overline{Q}}{o}]]] \\ &+Q [\text{true}, Nor[\frac{R}{i_1}, \frac{\overline{Q}}{i_2}, \frac{Q}{o}]] \\ &\setminus \overline{Q}]] \end{aligned} \tag{R2, R1 \times 2}$$

As in the original specification, this design remains a description of the behavior of the bit when correct operating protocols are observed. Behaviours when operated incorrectly (such as the possibility of meta-stable states) are not considered and are left to later design stages.  $\square$

## 4 Compositionality of implementations

As observed in Section 3, the parallel operator poses a unique problem with respect to the provision of a compositional design method. Care must be taken to ensure that each network of specifications is implemented by every network of implementations of the respective specifications. In order to investigate the compositionality of the parallel operator with respect to implementations, it is necessary to first consider the role of implementations within the refinement calculus formalism.

The program refinement calculus provides a broad-spectrum specification, design, and implementation language in which a particular subclass of specifications is identified with the target programming language. This subclass then forms the goal of every design activity, the aim being to find a refinement of a given specification which falls within this implementation class. Similarly, in the dataflow refinement calculus it is necessary to identify a collection of process specifications which correspond to some class of real-world processes, referred to in the following as the *implementation domain*. The dataflow case is complicated somewhat by the wide range of possible target implementation domains, ranging from communicating sequential programs on some multi-processing digital computing device as might be used to implement the summing machine to hardware devices such as solid state transistors as might be used to implement the flipflop.

This section takes up the summing machine example and develops an appropriate implementation domain for this and similar processes. The parallel operator is compositional with respect to this implementation domain in a weak sense.

### 4.1 Recursively enumerable networks

The summing machine example of Example 2.3 admits some surprising refinements, which highlight some interesting aspects of and possible traps associated with the parallel composition operator.

**Example 4.1 (A “bad” summing machine)** The intermediate design of the summing machine developed in Example 3.5, ie

$$[[ (+s [Subsum, Sum]) \parallel (+b [Sum, Subsum]) \setminus b ],$$

apparently admits some unexpected “implementations”.

Consider the left hand subprocess above. Since the  $b$  channel includes all the required sums, the  $s$  channel can be given the required properties by ignoring the first value on  $b$  and simply repeating subsequent values verbatim. Likewise, the  $b$  channel may be implemented by initially emitting a 0 and then relaying

each of the values received on the  $s$  channel. That is,

$$\begin{aligned} & \| (+s [\text{true}, \forall n : \mathbb{N} \bullet a(n) = b(n+1)]) \| \\ & (+b [\text{true}, b(0) = 0 \wedge \forall n : \mathbb{N} \bullet b(n+1) = a(n)] \setminus b \|, \end{aligned}$$

is a valid refinement of  $+s [\text{true}, \text{Sum}]$ .  $\square$

The “implementation” of Example 4.1 is unnatural in that neither process actually bothers to calculate the desired sums, but instead opts simply to pass on the values “calculated” by the other.

Perhaps even more disturbing is the following refinement path for the summing machine.

**Example 4.2 (All-purpose machine)**

$$\begin{aligned} & +s [\text{true}, \text{Sum}] \\ & \sqsubseteq \| [+s [\text{true}, \text{Sum}] \setminus b \| \\ & \sqsubseteq \| [+s [\text{true}, \text{Sum} \wedge \text{Sum}[\frac{b}{s}]] \setminus b \| \\ & \sqsubseteq \| [+s [\text{Sum}[\frac{b}{s}], \text{Sum}]] \| (+b [\text{Sum}, \text{Sum}[\frac{b}{s}]] \setminus b \| \\ & \sqsubseteq \| [+s [\text{true}, s = b]] \| (+b [\text{true}, b = s]) \setminus b \| \end{aligned}$$

Since no properties of  $\text{Sum}$  are used in this development, the final “implementation” can obviously be used to implement *any* initial specification at all and yet, the individual subprocesses do appear reasonable candidates for an implementation domain!  $\square$

In order to identify an implementation domain that is closed under network composition it is sufficient to identify a weak property that is in some sense preserved by network composition. One candidate for such a property is *causality*. A process is causal if the value of its observables at each point in time depends only on the *previous* values of its observables. Clearly the above “implementations” fail this property, the first because one of the subprocesses looks at future values of  $b$  to calculate each value of  $s$  and the second because both processes make use of the current value of their inputs to calculate each output value. There is no time-ordering of the events in each process which allow them to be causal.

One class of causal processes capable of implementing processes such as the summing machine is the class of recursively enumerable dataflow machines.

**Definition 4.3** A dataflow process  $S : \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}$ , with each of the observables  $\vec{u}, \vec{z}$  integer channels, is a recursively enumerable dataflow (RED) machine if and only if for each  $z \in \vec{z}$  there exists a constant  $C_z$  and an integer expression  $F_z$  in the process variable names such that

$$S = +\vec{z} [\text{true}, \bigwedge z : \vec{z} \bullet \mathcal{RED}(z, C_z, F_z)],$$

where

$$\begin{aligned} \mathcal{RED}(z, C_z, F_z) == z(0) = C_z \wedge \\ \forall n : \mathbb{N} \bullet z(n+1) = F_z[\frac{\vec{u}(n)}{\vec{u}}, \frac{\vec{z}(n)}{\vec{z}}]. \end{aligned}$$

♡

The RED machines rely on a model of time which requires each stream to have its  $n^{th}$  event before any have their  $(n+1)^{th}$  event. Thus all previous stream values are available to the next round of events. This is essentially a synchronous view of computation, each component proceeding in lock step with all others. Using this model of computation is not however a theoretic constraint on our methods, since asynchronous systems can be implemented on such machines through the use of communications buffers and null-valued communications.

Clearly the problem “implementations” discussed above are not recursively enumerable, so a large class of potential anomalies are avoided by adopting the RED machines as an implementation domain. Unfortunately, this class is not closed under network composition as shall be seen in the next section. Nevertheless, by adopting a slightly weakened form of the refinement relation in characterising specifications which correspond to RED machines, the situation can be recovered satisfactorily.

## 4.2 Finite observation equivalence

In this section, the summing machine example is further investigated in order to demonstrate the need for a weaker form of process equivalence when developing RED machines. This weaker form corresponds to an “acceptable up to time  $t$ ” view of process equivalence.

The summing machine is recast in an asynchronous framework and as in the last section is shown to admit two RED “implementations”, one which seems “natural” and another which seems “unnatural” in the context of their infinite behaviors, but for both of the implementations all finite behaviors are acceptable.

Before proceeding it is necessary to introduce some concepts for discussing asynchronous dataflow processes. In asynchronous process networks, each process operates under an autonomous system clock and need not communicate at every tick of the global clock. Consequently, the channels in an asynchronous network are represented as partial functions  $\mathbb{N} \mapsto \mathbb{Z}$ , with undefined values corresponding to no communication at that time. The times at which no communication occurs are called *stutters*. The communications on such a channel may be infinite as in the synchronous case above or the channel may eventually fall into permanent disuse after only a finite number of communications.

$$\text{seq}_\infty X == (\mathbb{N} \longrightarrow X) \cup \text{seq } X$$

In order to distill the useful information from these partially defined channels two extraction functions are used.

$$\begin{array}{l}
\boxed{[X]} \\
\hline
\chi : (\mathbb{N} \rightarrow X) \rightarrow (\text{seq}_\infty X) \\
\tau : (\mathbb{N} \rightarrow X) \rightarrow (\text{seq}_\infty \mathbb{N}) \\
\hline
\forall s : \mathbb{N} \rightarrow X \bullet \\
\tau_s = \{(n, m) : \mathbb{N} \times (\text{dom } s) \mid n = \#((0..m) \triangleleft s) - 1\} \\
\chi_s = \tau_s \hat{\circ} s
\end{array}$$

For any asynchronous channel  $s$ ,  $\tau_s$  is the (possibly infinite) sequence of times at which  $s$  communicates, and  $\chi_s$  is the (possibly infinite) sequence of communications on  $s$ .

**Example 4.4 (The asynchronous summing machine)** An asynchronous version of the summing machine may be designed by requiring that each communication on the  $s$  channel should be the sum of the previous communications on the  $a$  channel.

$$\begin{array}{l}
\boxed{ASum} \\
\hline
a, s : \mathbb{N} \rightarrow \mathbb{Z} \\
\hline
\text{dom } \chi_s = \text{dom } \chi_a \wedge \\
\forall n : \text{dom } \chi_s \bullet \chi_s(n) = \sum j : 0..n \bullet \chi_a(j)
\end{array}$$

Note that, since timing information is not considered, no hard deadlines are required of the summing machine so that this is essentially a safety requirement, the only progress property required is that the system *eventually* calculate each partial sum.

Once again it is possible to proceed to a design which introduces a local variable  $b$ , which holds the previous partial sum and decomposes the process into an adding unit and a feedback unit.

$$\begin{array}{l}
\boxed{ASubsum} \\
\hline
a, b : \mathbb{N} \rightarrow \mathbb{Z} \\
\hline
0 \in \text{dom } \chi_b \wedge \\
\forall n : \mathbb{N} \setminus \{0\} \bullet n \in \text{dom } \chi_b \Leftrightarrow n - 1 \in \text{dom } \chi_a \wedge \\
\forall n : \text{dom } \chi_a \bullet \chi_b(n) = \sum j : 0..n - 1 \bullet \chi_a(j)
\end{array}$$

$$+s : [\text{true}, ASum] \sqsubseteq \\
[[ (+s [ASubsum, ASum]) \parallel (+b [ASum, ASubsum]) \setminus b ] ]$$

### Good implementation

Making use of the feedback channel,  $b$ , it is possible to calculate the next partial sum at each iteration by adding the next input value to the previous partial sum. These values become available at (possibly) different times, so that it may be necessary to wait until after both values are available.

$\begin{array}{l} \textit{AAdd} \\ \hline a, b, s : \mathbb{N} \rightarrow \mathbb{Z} \\ \hline \text{dom } \chi_s = (\text{dom } \chi_a \cap \text{dom } \chi_b) \wedge \\ \forall k : \text{dom } \chi_s \bullet \\ \quad \chi_s(k) = \chi_a(k) + \chi_b(k) \\ \quad \tau_s(k) = \max\{\tau_a(k), \tau_b(k)\} + 1 \end{array}$
--

The feedback loop is implemented through the simple expediency of passing each succeeding partial sum along the channel  $b$  as they arrive on  $s$ .

$\begin{array}{l} \textit{AFeedback} \\ \hline b, s : \mathbb{N} \rightarrow \mathbb{Z} \\ \hline 0 \in \text{dom } \chi_b \wedge \\ \forall n : \mathbb{N} \setminus \{0\} \bullet n \in \text{dom } \chi_b \Leftrightarrow n - 1 \in \text{dom } \chi_s \wedge \\ b(0) = 0 \\ \forall k : \text{dom } \chi_s \bullet \\ \quad \chi_b(k + 1) = \chi_s(k) \\ \quad \tau_b(k + 1) = \tau_s(k) + 1 \end{array}$
---

The network consisting of these two processes can be shown to implement the summing machine via the above design.

$$+s : [\text{true}, \textit{ASum}] \sqsubseteq [ (+s [\text{true}, \textit{AAdd}]) \parallel (+b [\text{true}, \textit{AFeedback}]) \setminus b ]$$

In order to implement the above process network as a RED machine it is necessary to add buffers to each of the channels, but this is a relatively straight forward exercise. For any finite execution time this network is equivalent to the RED machine determined by the predicate  $\textit{AAdd} \wedge \textit{AFeedback}$ .

### Bad implementation

Another way to make use of the known properties of the feedback channel is simply to pass them on to the environment, thus leaving the hard calculation to the feedback process. The first value on  $b$  is dropped and the rest passed on as they arrive.

$b\text{-Feed-s}$ $b, s : \mathbb{N} \leftrightarrow \mathbb{Z}$ <hr style="border: 0.5px solid black;"/> $\forall n : \mathbb{N} \bullet n \in \text{dom } \chi_s \Leftrightarrow n + 1 \in \text{dom } \chi_b \wedge$ $\forall k : \text{dom } \chi_s \bullet$ $\chi_s(k) = \chi_b(k + 1)$ $\tau_s(k) = \tau_b(k + 1) + 1$
--

Using the original design decomposition it can be shown that the network consisting of the above process and a feedback loop is an “implementation” of the asynchronous summing machine.

$$+s : [\text{true}, A\text{Sum}] \sqsubseteq [ (+s [\text{true}, b\text{-Feed-s}] \parallel (+b [\text{true}, A\text{Feedback}]) \setminus b ) ]$$

Once again this design can be implemented as a RED machine through the introduction of channel buffers. For any finite execution time, the resulting process is equivalent to the machine determined by the predicate  $b\text{-Feed-s} \wedge A\text{Feedback}$ .  $\square$

The good implementation in Example 4.4 seems a “natural” implementation in that it has the property that the partial sums are calculated in near synchronicity with the input stream, at least up to a fixed and bounded time delay. The results are reported in “real-time” so to speak.

The “bad” implementation at least has the advantage over those of Example 4.1 and Example 4.2 that it has a well-defined function, it stutters forever. On the other hand, like Example 4.1 and Example 4.2, it seems “unnatural” in the sense that both component processes leave the hard work to the other, with the result that nothing is done by either in any finite period of time. However this is a “reasonable” implementation because the original specification was lax enough to allow such behavior. The asynchronous summing machine specification made no requirements as to the timeliness of calculation. There is no finite time at which you can definitely state that this implementation is not performing the correct calculation, it satisfies the specification by postponing the communication of its results until the “end of time”!

In [8], Hehner argues that the impossibility of observing non-termination is a vital flaw in total correctness approaches to program development. He argues that it is superior to abandon the total correctness approach of encoding termination into the correctness relation and to instead include explicit timing requirements in the specification. All of the work necessary to establish an explicit timing bound must be done to establish total correctness, but is usually lost in the process. In any case failure to strictly specify the desired timing behavior is bad practice in a practical sense since there is no feasible method of testing for compliance if there is no specific time by which compliance is required. Progress properties which require some behavior to occur, but which do require strictly bounded timing behavior are termed *liveness* properties in

the literature and have generally been considered the hard part of producing a compositional method for developing process networks [1].

In any case, the problem posed by the bad implementation of Example 4.4 may be solved satisfactorily by adopting a finite observations view of process equivalence when determining the predicate transformer expressions which correspond to RED machines.

**Definition 4.5** (Finite observation equivalence)

A predicate  $\phi$ , over process variables  $\vec{v}$  representing integer channels, is finitely refutable if and only if for all states  $\sigma$  such that  $\phi$  is false at  $\sigma$ , there exists  $N \in \mathbb{N}$  such that  $\phi$  fails for all states whose behaviors are the same as  $\sigma$  for the first  $N$  communications. That is a finitely refutable predicate can be falsified by some finite time.

A process  $S$ , over process variables  $\vec{v}$  representing integer channels, is finite observations refined by another process  $T$ , written  $S \sqsubseteq T$ , if and only if, for all finitely refutable predicates  $\phi$ ,  $S(\phi) \Rightarrow T(\phi)$ .

$S$  is finite observations equivalent to  $T$ , written  $S \stackrel{f}{=} T$ , if and only if  $S \sqsubseteq T$  and  $T \sqsubseteq S$ . ♡

If two processes are finite observations equivalent, they cannot be distinguished by any finite observation and in every practical sense may be used interchangeably in the construction of a RED network.

**Definition 4.6** A predicate transformer  $S : \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}$  is a RED implementation if and only if there exists a RED machine  $R : \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}$  such that  $S \stackrel{f}{=} R$ . ♡

The class of RED implementations is closed under network composition [18] and thus provides a suitable target implementation domain for a fully compositional RED machine development method. Each RED implementation is uniquely associated with a RED machine which embodies all of its finitely observable behaviors. Making use of finite observation equivalence only in the identification of implementations allows the development method to retain its ability to incorporate the use of liveness properties in the specification and design process. The onus is placed on the system developer to replace liveness properties with quantified progress requirements before arriving at the implementation level. Implementations are only guaranteed never to contradict a liveness requirement rather than to actually satisfy it. It is worth noting that it remains perfectly safe to make use of liveness properties at high levels of abstraction and also to include them in process assumptions when decomposing networks.

## 5 Discussion and conclusions

The traditional view of sequential programs sees them beginning computation on an existing state vector and proceeding uninterrupted to produce a new state vector. Accordingly the program's requirements of its environment can be stated in terms of this initial state and referred to as *preconditions*. Similarly the requirements of the program can be stated in terms of the final state and referred to as *postconditions*. The behavior of sequential programs may be fully specified by preconditions and postconditions.

For parallel or non-terminating processes this paradigm becomes restrictive.

When the process cannot be assumed to proceed uninterrupted (for example parallel processes with shared variables) it is generally not possible to state the obligations of the environment simply as preconditions. The *rely* and *guarantee* conditions of VDM [14] represent an attempt to address this problem.

When the process does not terminate there is no final state on which to express a postcondition. Process algebra formalisms such as CSP [11] and CCS [21] and various temporal logics [31, 13, 16, 27] address the problem of specifying the behavior of non-terminating processes but they unfortunately ignore the necessity to also specify the obligations the process may place on its environment, vital to a compositional design method.

Compositional methods demand some method of including in the specification of each subprocess an abstract description of the behavior of interacting processes. Only in this way can the *in situ* behavior of a process be analysed in isolation from the implementation details of interacting processes. The inclusion of environment assumptions provides a valuable mechanism for the consideration and documentation of conditions under which a process may fail, at a high level of abstraction. Moreover it is possible to make use of and discharge these assumptions at appropriate levels in the design process, removing the necessity to take these high-level properties into consideration at lower levels of abstraction.

Recognising this, attempts to develop a compositional development method for networks of processes have for the most part been inspired by the Hoare Logic style of program specification [9]. A simple approach is to augment the Hoare triple specification of a sequential program with some additional information about its progress through intermediate states. This style of method is typified by the work of Hooman [12].

Another key inspiration has been the trace based semantics of the process algebra CSP. By describing processes in terms of traces of communications [10] it is feasible to assign a relational semantics to Hoare triple specifications in the non-terminating case. This approach was pioneered by Misra and Chandy [22], wherein a simple compositional method for verifying safety properties of networks is presented. Subsequent work has aimed at using similar relational trace semantics to address progress properties with varying degrees of success [28, 1, 2, 3].

A notable compositional method of this style is that of Abadi and Lamport [1], in which processes are viewed as a collection of agents which act sequentially to effect state changes in a global state vector. A system trace is a record of all the state transitions, annotated with the effecting agents. High level specifications are expressed in terms of *assumptions* about the traces allowed by their environment and *guarantees* about the traces allowed by the process. Abadi and Lamport [1] address the circular reasoning problem by restricting the class of allowed specifications in such a way that each specification essentially describes a causal relationship between actions of the environment and actions of the process. This causal relationship effectively encodes a well-founded induction principle into the network decomposition rule, leading to a simple network decomposition rule upon a restricted class of process specifications. This allows their method to be compositional, but at the cost of specification language expressiveness. For example, one restriction is that all processes admit arbitrary stuttering (here a stutter is an action performed by an environment agent). Thus, though the technique allows the verification of liveness properties as well as safety properties, it does not appear possible to verify quantified timing obligations for process reactions. In their critique of existing compositional methods, Abadi and Lamport note that [1, p76],

“Specifications that involve only safety properties are not very satisfying, since any safety property is satisfied by a system that does nothing. Liveness properties must be added to rule out trivial implementations.”

However, as argued in Section 4.2 and in [8] simple liveness properties can be subjected to similar criticism from a practical viewpoint. Quantified timing obligations are to be preferred over liveness properties in the general case and would appear essential to the real-time case.

The dominant alternative to relational semantics for program verification is the predicate transformer semantics based refinement calculus developed independently by several authors [4, 24, 26]. The refinement calculus approach makes use of a wide-spectrum language, capable of expressing both specifications and implementations. Program development proceeds through a series of correctness preserving refinements, until the specification is finally transformed into an expression in the programming language subset of the refinement calculus.

This paper demonstrates the feasibility of applying predicate transformer semantics to the compositional construction of networks of processes. As in the axiomatic verification methods described above, the key is the adoption of traces as the basic model for describing process behavior. A departure from existing methods is the use of individual traces for each process observable as opposed to a single global state trace. An integral part of the specification of a process is the identification of the observables it constructs as distinct from the observables constructed in its environment. The specification style adopted is a

natural transformation of the standard sequential style to this view of process as constructor. The notion of precondition is generalised to allow the expression of arbitrary environmental obligations, *assumptions* about the environment observables that the process may make during computation, and the notion of postcondition generalised to allow the expression of arbitrary requirements on the behavior of the constructed system, *effects* the process must achieve during computation. A specification statement consists of a list of constructions, an assumption predicate in which only the environment observables appear free, and an effect predicate in both the constructed and the environment observables. The semantics of the specification statement is essentially the same as for the sequential case [25], modulo the syntactic restriction on the assumption predicate.

Following [23] the interaction of the refinement relation with the various language elements is described in terms of a series of transformation templates known as refinement laws. For the most part the language operators of the sequential program calculus and their associated refinement laws may be applied directly to the dataflow case. This paper has concentrated on the features specific to the dataflow case, the slightly variant specification statement, the hiding mechanism for local observables, and most importantly network decomposition.

The refinement compositionality of network composition is achieved through the strict partitioning of the system state and through the encoding of an induction principle into the parallel composition operator. Implementation compositionality is achieved by adopting a target language that is closed under composition, the basic requirement being that there be a causal relationship between input streams and output streams. By way of contrast, Abadi and Lamport [1] achieve compositionality through restricting the class of verification properties and encoding a causality principle into the specification statement, effectively allowing consideration only of causal specifications.

A prime advantage of the refinement calculus method over that of Abadi and Lamport is that makes no restrictions on the classes of properties that may be considered. In addition to the ability to treat both liveness and safety properties, it is possible to verify explicit timing properties as demonstrated in Example 3.6. Arguably, a drawback is the interaction of liveness properties with network composition which makes it desirable that they be replaced with explicit timing requirements at some stage of the development process. Conversely, Abadi and Lamport do not allow liveness properties in environment assumptions.

The small case studies in process development presented in this paper have served to demonstrate some of the power of the refinement calculus method.

The refinement calculus method is able to treat both software and hardware components, be they analogue or digital. Abadi and Lamport also make this claim of their method, but the partitioning of the system state in the refinement calculus method means that it is far more straightforward to utilize differing implementation technologies for individual subprocesses. The full power of this capability has been demonstrated by a significant case study in the design of a

boiler control system [20]. Here the ability to support multiple implementation technologies in the high level specifications was critical to satisfactory treatment of a complex network of interacting hardware and software components. Conversely, the Abadi and Lamport method would appear more convenient when the target language is a CSP-like language with global trace semantics.

The verification argument for the nor gate implementation of a bit presented in Example 3.6 is greatly simplified by the assumption predicates introduced to the parallel components. A simple logical proof [19] required an explicit real induction argument. The ability, supported by refinement rule R4, to introduce environmental responsibilities that are to be discharged by other parallel components is a powerful feature of any composition method. Both Example 3.5 and Example 3.6 demonstrate how the refinement calculus enhances this feature with the ability to calculate what these responsibilities must be in order to utilise a proposed implementation component.

As observed in Section 4, R4 does not prevent the introduction of circular dependencies at some stage in the design process. The existence of such refinement paths does not lead to a collapse of the formal system as it would in a predicate calculus process model (where it is equivalent to claiming  $\text{true} \Rightarrow \text{false}$ ) because the predicate transformer model has a greater ability to represent “strange” processes, such as those displaying magical or angelic non-determinism. Further there is no danger this can lead to an incorrect implementation since it is not possible (assuming the principle of causality) to construct networks with such circular dependencies. The introduction of such dependencies leads the development process up a blind alley just as the laws of the program refinement calculus introduced in [25] allow the introduction of angelic and miraculous refinements which cannot be implemented. Morgan [25, §8] offers several arguments to justify this, but the primary justification is that simple, “safe” refinement laws, are preferable to complex, “sound” laws.

## Acknowledgments

The work reported in this paper has been supported by Australian Research Council grant number A4913006: *Formal methods for the specification and refinement of time-based systems and processes*.

## References

- [1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
- [2] M. Abadi and L. Lamport. Conjoining specifications. Research Report 118, Digital Equipment Corporation, Systems Research Center, 1993.

- [3] M. Abadi and G. D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114:3–30, 1993.
- [4] R. J. R. Back. A calculus of program derivations. *Acta Informatica*, 25:593–624, 1988.
- [5] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer Verlag, 1990.
- [6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, New Jersey, 1976.
- [7] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International, second edition, 1993.
- [8] E. C. R. Hehner. Abstractions of time. In A. W. Roscoe, editor, *A Classical Mind*, pages 191–210. Prentice Hall, 1994.
- [9] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10), October 1969.
- [10] C. A. R. Hoare. Specification-oriented semantics for communicating processes. In J. Diaz, editor, *Automata, Languages, and Programming*, 1983.
- [11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [12] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
- [13] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, 1986.
- [14] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. and Sys.*, 5(4):596–619, October 1983.
- [15] G. Kahn. The semantics of a simple language for parallel processing. In J. L. Rosenfeld, editor, *Information Processing '74*, pages 471–475. North-Holland, 1974.
- [16] R. Koymans. Specifying real-time properties with Metric Temporal Logic. *Real-Time Systems*, 2:255–299, 1990.

- [17] B. P. Mahony. Frame-based typing of predicate transformers. Department of Computer Science, University of Queensland, 1994.
- [18] B. P. Mahony. Predicate transformer semantics for network composition. Department of Computer Science, University of Queensland, 1994.
- [19] B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In *Proceedings of the 6th Australian Software Engineering Conference (ASWEC91)*, 1991.
- [20] B. P. Mahony, C. Millerchip, and I. J. Hayes. A boiler control system: A case study in timed refinement. In *International Invitational Workshop - Design and Review of Software Controlled Safety-Related Systems*, Ottawa, June 1993.
- [21] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [22] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.
- [23] C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, C.A.R. Hoare series editor. Prentice Hall, 1990.
- [24] C. C. Morgan, K. A. Robinson, and P. Gardiner. On the refinement calculus. Technical Monograph PRG-70, Oxford University Programming Research Laboratory, 1988.
- [25] C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), July 1988. Reprinted in [24, pp. 7–30].
- [26] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
- [27] B. Moszkowsky. *Executing temporal logic programs*. Cambridge University Press, 1986.
- [28] P. K. Pandya and M. Joseph. P-A logic – a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.
- [29] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.
- [30] E. W. Stark. A simple generalisation of Kahn’s Principle to indeterminate dataflow networks. In *Semantics for Concurrency*, pages 157–174. Springer-Verlag, July 1990.

- [31] C. Stirling. An introduction to modal and temporal logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, volume 491 of *Lecture Notes in Computer Science*, pages 2–20. Springer-Verlag, 1991.
- [32] J. von Wright. *A Lattice-theoretical Basis for Program Refinement*. PhD thesis, Åbo Akademi, 1991.