Composing Snippets

Igor Benko
University of Waterloo
Waterloo, ON N2L3G1, Canada
ibenko@mitra.com

Jo Ebergen
Sun Microsystems Laboratories
Palo Alto, CA 94303, USA
Jo.Ebergen@eng.sun.com

Abstract

A simple formal framework for representing safety and progress properties of concurrent systems is introduced. The framework is based on Enhanced Characteristic Functions (ECF), which lead to simple definitions of operations such as hiding and process product. Two distinct compositions are proposed: The network composition that models networks of devices, and the specification composition that enables a constraint-based approach to building specifications. A part-wise design and verification approach is proposed. This approach may avoid state explosion in the verification of implementations for constraint-based specifications.

1. Introduction

We propose a simple formal framework for the specification and implementation of concurrent systems. The formalism is based on Enhanced Characteristic Functions (ECFs)[15], and a process in this framework is called an ECF process. We show how to apply ECF processes to specifying asynchronous circuit components and how we can use ECF processes for verifying safety and progress properties of networks of circuit components. Besides presenting a simple formalism, our main contributions are a constraint-based approach to building specifications that include progress properties and a part-wise refinement method that allows us to avoid state explosion in the verification of implementations.

The constraint-based approach to building specifications can be useful when behaviors involve a large degree of concurrency. A similar approach has been applied in different formalisms: [3] discusses how a constraint-based specification approach is supported in LOTOS. [7] advocates the use of the "weave" operation for building specifications from constraints. The usage of the weave in [8] and [2] demonstrates how combining behavioral constraints can lead to concise specifications of non-trivial behaviors. [11] and [12] define the weave for Petri nets and demonstrate its application in specifying asynchronous circuits. In [14], a constraint-based specification takes advantage of chain constraints to introduce timing requirements into specifications.

Our ECF model is heavily inspired by the XDI model of Tom Verhoeff [15] and the X^{2}DI model of Mallon et al[10]. We use the same set of labels, and our interpretation of the labels is the same. There are, however, some differences between the two models. Most notably, we do not focus exclusively on delay-insensitive (XDI) processes or X^{2}DI processes. Rather, we consider a larger domain of processes. Although this larger domain of processes has its limitations [4], it allows for a simpler formalism. Furthermore, our model deals with hiding separately, has a different definition for the product of processes and differentiates between the network composition and the specification composition. Finally, in the ECF model we offer a formal definition of snippets [11, 12] and we propose a new part-wise design method that applies to specifications expressed as a specification composition of snippets.

Our progress criteria are closely related to finalization in Process Spaces from [13]. The Process Spaces model from [13], however, requires a separate specification for each correctness condition that is to be addressed. ECF processes, on the other hand, capture safety and progress concerns at the same time.

Due to space limitations we have omitted any proofs. Proofs and additional information can be found in [1].

2. Labels and processes

An ECF process P is a triple $\langle \mathbf{i}.P, \mathbf{o}.P, \mathbf{f}.P \rangle$, where $\mathbf{i}.P$ is an input alphabet, $\mathbf{o}.P$ is an output alphabet. We stipulate that $\mathbf{i}.P$ and $\mathbf{o}.P$ are disjoint. The alphabet of process P, denoted by $\mathbf{a}.P$ is $\mathbf{i}.P \cup \mathbf{o}.P$. An enhanced characteristic function (ECF) is a function $\mathbf{f}.P : (\mathbf{a}.P)^* \to \Lambda$, which maps traces to a set of labels Λ. By attaching a label to a trace we specify the progress and safety properties of a process after having executed the sequence of communication events represented by the trace. We borrow the set of labels from [15]:

$$\Lambda = \{\perp, \Delta, \Box, \nabla, \top\}$$

The interpretation of labels is as follows. After a process has executed a trace labeled with ∇, the process guarantees progress by eventually producing an output. In other words, a trace labeled with ∇ puts a process in a *transient* state, thus, traces labeled with ∇ are called transient traces. In a transient state, a process may be capable of receiving an input. A trace labeled with $\Box$ brings the process in an *indifferent* state, thus such a trace is called an indifferent trace. After having executed an indifferent trace, a process does not guarantee to produce an output, neither does the process demand an input from its environment. A trace labeled with Δ is called a *demanding* trace and brings a process to a demanding state. In a demanding state, a process demands an input but does not guarantee to produce an output, although producing an output may be possible. A trace labeled with $\perp$ is a *failure* trace and brings a process to a failure state. An output ending in a failure state indicates that the process can cause a failure by producing that output. Because failures must be avoided by the environment, an input ending in a failure state indicates that the environment must not provide that input. A trace labeled with $\top$ is a *miracle* trace and brings a process to a miracle state. An input ending in a miracle state indicates that the environment caused a miracle by producing that input. Because miracles cannot be performed by a process, an output ending in a miracle state indicates that the process cannot produce that output.

Traces labeled with either $\Box$, Δ, or ∇ are called *legal* traces, while traces labeled with either $\perp$ or $\top$ are called *illegal* traces. The set of legal traces for process P is denoted by $\mathbf{l}.P$. Illegal traces define the behaviors of a process where a failure or miracle has occurred. We require that all extensions of illegal traces are also illegal. More formally, an ECF for a process must be $\perp$-persistent and $\top$-persistent:

$$\mathbf{f}.P.t = \perp \;\Rightarrow\; (\forall u : u \in (\mathbf{a}.P)^* : \mathbf{f}.P.tu = \perp) \qquad (1)$$

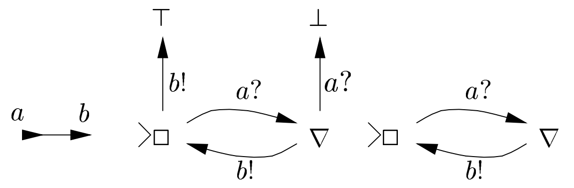
$$\mathbf{f}.P.t = \top \;\Rightarrow\; (\forall u : u \in (\mathbf{a}.P)^* : \mathbf{f}.P.tu = \top) \qquad (2)$$

We write $\mathcal{PROC}(I, O)$ to denote the set of all processes with input alphabet I and output alphabet O.

2.1. ECF processes and state graphs

We often depict ECF processes by means of state graphs. We start by creating an initial state, which is labeled with $\mathbf{f}.P.\varepsilon$. Next we create a distinct state for each one-symbol trace a and label the state with $\mathbf{f}.P.a$. We also create a transition for each symbol a, such that the transition leads from the initial state to the state corresponding to one-symbol trace a. Input symbols are postfixed with a question mark and output symbols are postfixed with an exclamation mark. We continue this process inductively, creating an infinite graph, which contains a distinct state for each trace. A graph can be reduced if subgraphs stemming from two states are the same. Such two states are equivalent, and can, consequently, be merged.

For example, all states labeled with $\perp$ are equivalent, because of $\perp$-persistence. For that reason, all transitions leaving states labeled with $\perp$ are self loops. A similar argument holds for $\top$ states. In order to reduce the clutter, we omit self loops on $\top$ and $\perp$ states when we depict processes with state graphs.



a) Schematic b) State graph c) Simplified graph

Figure 1. A specification of a WIRE

Figure 1b shows the state graph corresponding to a process that specifies a WIRE. Notice that the WIRE is initially in a $\Box$ state. That is, neither the WIRE nor its environment have any obligation for producing a communication action. Furthermore, after the WIRE has received an input, the WIRE is in a ∇ state, where it is obliged to produce an output.

In order to reduce clutter further, we may omit the states labeled with $\perp$ and $\top$ label, which results in the state graph of Figure 1c. When the $\top$ and the $\perp$ state are not shown, we stipulate that all missing transitions on output symbols lead to the $\top$ state and all missing transitions on input symbols lead to the $\perp$ state.

2.2. Safety and Progress Violations

Safety and progress violations can both be present in a process. A safety violation manifests itself by the presence of a failure state that can be reached by an output. A progress violation manifests itself by the presence of so-called *unhealthy states*. In an unhealthy state progress is required but cannot be satisfied. For example, a ∇ state requires progress by the process, but, because all outputs lead to the $\top$ state, no output can be produced. We call traces leading to such a state ∇-*unhealthy traces*, and we call a process that contains a ∇-unhealthy trace a ∇-*unhealthy* process [10].

A similar observation can be made for Δ states where progress is required from the environment but the environment cannot provide an input. A trace leading to such a state is called a Δ-*unhealthy trace* and a process containing a Δ-unhealthy trace is a Δ-unhealthy process. A process is unhealthy if it is either ∇ or Δ-unhealthy.

Safety and progress violations can arise in various ways: as a result of composing healthy processes or as a result of hiding output symbols from a healthy process. We have chosen to include processes with safety and progress violations in our domain of ECF processes. Excluding unhealthy processes and requiring that the process domain be closed under product and hiding would require more complicated definitions of these two operations. For similar reasons, [10] also includes unhealthy processes in the X^2DI domain. The XDI model [15] does not include unhealthy processes.

3. Refinement

The refinement relation defines whether one process implements another. If process P is refined by process Q, denoted by $P \sqsubseteq Q$, we say that process P is implemented by process Q. When $P \sqsubseteq Q$, then process Q is at least as safe as process P. Furthermore, process Q makes at most as many progress demands on its environment as process P, and process Q makes at least as many progress guarantees as process P.

Our refinement relation is based the order among trace labels [15]:

$$\bot \sqsubseteq \Delta \sqsubseteq \Box \sqsubseteq \nabla \sqsubseteq \top \qquad (3)$$

A justification for the order on labels is as follows: A failure trace is worse than a trace that does not fail. For that reason, $\bot$ is the least element in the partial order on labels. A trace that demands an input but does not guarantee an output is worse than a trace that does not demand an input nor guarantee an output, so $\Delta \sqsubseteq \Box$. A trace that makes no progress demands or guarantees is worse than a trace that guarantees output progress, hence $\Box \sqsubseteq \nabla$. Finally, producing a miracle is always better than anything else, thus $\top$ is the greatest of all the labels.

Definition 3.1 (Refinement)
Let P and Q be processes, such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P = \mathbf{o}.Q$. P is refined by Q, denoted by $P \sqsubseteq Q$, iff:

$$P \sqsubseteq Q \quad \equiv \quad \Big(\forall t : \, t \in (\mathbf{a}.P)^* : \, \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\Big) \qquad (4)$$

Theorem 3.2 (Properties of refinement)
The refinement relation $\sqsubseteq$ is a partial order on $\mathcal{PROC}(I,O)$. Furthermore, $(\mathcal{PROC}(I,O), \sqsubseteq)$ is a complete lattice.

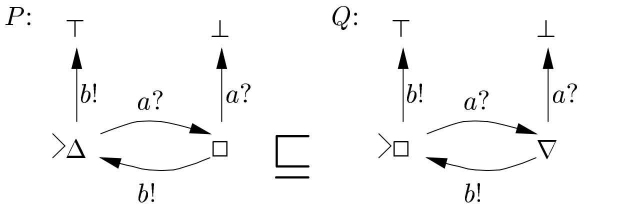


Figure 2. An example of refinement

3.1. A Refinement Example

Figure 2 shows an example of refinement. Process Q in Figure 2 specifies a "reliable" WIRE. Process P, on the other hand, specifies a "demanding, lazy" WIRE. In the initial state, process P demands from its environment an input on port a. After having received an input on port a, process P makes no progress guarantees. That is, process P may or may not produce an output on port b. Process Q implements process P, because in the initial state process Q does not demand an input from the environment. That is, process Q makes fewer progress demands on the environment than process P. Furthermore, after having received an input on port a, process Q guarantees that an output on port b is produced. This means that process Q guarantees progress where process P does not. Finally, we notice that the same traces that lead to the $\bot$ state in process P also lead to the $\bot$ state in process Q. Similarly, the same traces that lead to the $\top$ state in process P also lead to the $\top$ state in process Q. Hence, process Q refines process P.

4. Product

Process product is an operation that allows us to compose processes. Our definition of process product is based on the product of trace labels, defined in Table 1. The product table comes from Chapter 7 of [15].

$\times$	$\bot$	Δ	$\Box$	∇	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$
Δ	$\bot$	Δ	Δ	∇	$\top$
$\Box$	$\bot$	Δ	$\Box$	∇	$\top$
∇	$\bot$	∇	∇	∇	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 1. Product table

The product of labels captures the interaction between two processes. Consider, for example, the table entry $\Delta \times \Box = \Delta$. This entry describes a state in an interaction between two processes where one process demands an input and another process has no progress demands or guarantees. The result of the label product tells us that the network of two processes also demands an input from its environment. A similar argument explains why $\nabla \times \Box = \nabla$: If one process guarantees progress and the other process has no progress demands and guarantees, then the network of these two processes guarantees progress.

Because we give process obligations a priority over environment obligations, we have $\nabla \times \Delta = \nabla$. If one process is in a ∇ state, guaranteeing an output, and another process is in a Δ state, demanding an input, then the network of two processes will eventually produce an output.

We assume that, if one component performs a miracle, then the product performs a miracle. For this reason we have $\top \times \lambda = \top$ for any label λ. Finally, if one component fails, then the product fails as well. Thus, we have $\bot \times \lambda = \bot$ for any label λ other than $\top$. Notice that $\top \times \bot = \top$. That is, a miracle cures a failure.

The process product $P \times Q$ is defined by means of the trace-wise product of the traces in P and Q, unless the $\top$ or $\bot$ state has been reached. Once the product reaches the $\top$ or $\bot$ state, the product stays in that state. This property guarantees that the product of two processes is again $\top$ and $\bot$ persistent.

Definition 4.1 (Product)
The product of processes P and Q, denoted by $P \times Q$, is defined as

$$\begin{aligned}
\mathbf{i}.(P \times Q) &= (\mathbf{i}.P \cup \mathbf{i}.Q) - (\mathbf{o}.P \cup \mathbf{o}.Q)\\
\mathbf{o}.(P \times Q) &= \mathbf{o}.P \cup \mathbf{o}.Q\\
\mathbf{f}.(P \times Q).\varepsilon &= \mathbf{f}.P.\varepsilon \times \mathbf{f}.Q.\varepsilon\\
\mathbf{f}.(P \times Q).ta &= \begin{cases} \mathbf{f}.P.(ta \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(ta \downarrow \mathbf{a}.Q) \\ \qquad \textbf{if} \ \ \mathbf{f}.(P \times Q).t \notin \{\top, \bot\} \\ \mathbf{f}.(P \times Q).t \\ \qquad \textbf{otherwise} \end{cases}
\end{aligned}$$

With $s \downarrow A$ we denote trace s projected on alphabet A.

Theorem 4.2 (Product properties)
The product of processes is associative, commutative, idempotent, and monotonic with respect to refinement.

5. Hiding

Hiding conceals output ports of a process. Recall that our refinement relation requires that an implementation has the same input and output ports as a specification. This alphabet restriction may cause a problem if an implementation is expressed as a product of processes, where "internal" connections between these processes appear as output ports in the product. The output alphabet of the specification might not include internal ports of the implementation, thus we cannot compare the implementation and the specification. By means of hiding we can hide the internal ports and then verify whether the refinement holds.

The result of hiding is a process that tells us what kind of behavior the environment can expect from process P when ports in set A are concealed. Suppose that the environment of P has observed the trace of communication actions given by t. Let $X(t)$ be the set of all traces that process P could have executed in order for the environment to observe communication actions that amount to trace t.

$$X(t) = \{s \mid s \in (\mathbf{a}.P)^* \wedge s \downarrow (\mathbf{a}.P - A) = t\}$$

The environment does not know which trace from set $X(t)$ process P has executed. For this reason, after observing trace t, the environment does not know in what kind of a state process P is: $\bot$, demanding, indifferent, transient, or $\top$. In order for the environment to be able to cope with the worst possible outcome of events, the environment must assume that the state of process P is represented by the least label of all traces in $X(t)$.

Definition 5.1 (Hiding)
Let P be a process and let $A \subseteq \mathbf{o}.P$. The hiding of symbols from A in process P is denoted by $|[A :: P]|$ and defined by

$$\begin{aligned}
\mathbf{i}.|[A :: P]| &= \mathbf{i}.P\\
\mathbf{o}.|[A :: P]| &= \mathbf{o}.P - A\\
\mathbf{f}.|[A :: P]|.t &= \big(\sqcap s \ : \ s \in (\mathbf{a}.P)^* \\
&\qquad\qquad \wedge\, s \downarrow (\mathbf{a}.P - A) = t \\
&\qquad : \ \mathbf{f}.P.s\big)
\end{aligned} \tag{5}$$

The notation $\sqcap$ denotes the greatest lower bound. The idea for this definition of hiding was introduced to us by Willem Mallon [9].

Theorem 5.2 (Properties of hiding and product) *Hiding is monotonic, and hiding distributes over product. In formula, for processes P and Q and subset A of $\mathbf{o}.P$*

$$P \sqsubseteq Q \ \Rightarrow \ |[\, A :: P \,]| \sqsubseteq |[\, A :: Q \,]| \tag{6}$$

$$|[\, A :: P \,]| \ \times \ Q \ = \ |[\, A :: P \times Q \,]| \ \text{ if } \ A \cap \mathbf{a}.Q = \emptyset \tag{7}$$

(Substitution theorem)
Let P, Q, R, and S be processes. If $P \sqsubseteq |[\, A :: Q \times R \,]|$ and $R \sqsubseteq S$, then

$$P \sqsubseteq |[\, A :: Q \times S \,]| \tag{8}$$

The Substitution theorem is the cornerstone of hierarchical refinement, because it allows us to substitute a component in a network with an implementation of that component without violating any correctness conditions.

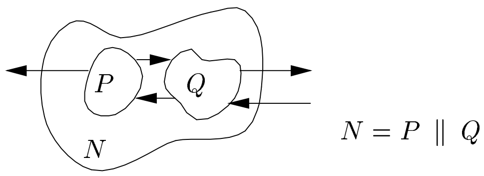


Figure 3. The network composition

6. Network composition

Network composition computes the joint behavior of a set of devices connected to form a network. For example, processes P and Q from Figure 3 each represent a physical device. Ports of P and Q with the same name are connected, meaning that communication between P and Q may take place. The network composition of P and Q, denoted by $P \parallel Q$, is a process that models the joint operation of processes P and Q. Formally, the network composition is nothing but the product of processes with one restriction: no connections between outputs may occur.

Definition 6.1 (Network composition)
The network composition *of P and Q, denoted by $P \parallel Q$, applies when $\mathbf{o}.P \cap \mathbf{o}.Q = \emptyset$, and is equal to $P \times Q$.*

Unlike [15], we make no restrictions on connections between inputs. That is, we can connect any number of input ports in order to create a "composite" input port of the network. If we are concerned about multiple input connections, we can connect input ports via explicit fork components.

Notice also that we leave internal network connections visible, unlike [10, 15]. More precisely, a connection between an input port and an output port within a network is seen as an output port of the network. An internal connection between two input ports is seen as an input port of the network. If we wish to make any of the output ports of the network invisible, we can apply the hiding operator. This approach gives us the freedom, first, to form a network composition of any desired number of network components and, second, to hide any set of outputs to study the properties of the remainder.

Because a process may contain both safety and progress violations, a network composition can tell us if the network satisfies all safety and progress properties. For example, if the process composition contains a Δ-unhealthy state, then deadlock may occur, because some component in the network demands progress, but no other component nor the environment guarantees that progress will be made. Even the presence of a $\Box$ state in a network gives us an indication that the network may, at some point, simply stop. Recall that in a $\Box$ state no device guarantees to provide an output and no device demands an input.

7. An Example

Consider the network composition of two WIREs show in Figure 4. One WIRE has input port a and output port m, and another has input port m and output port b. If the first wire receives an input on port a before the second WIRE has produced an output on port b, the network may fail. In the state graph of Figure 4, we can see this failure, because the trace $amam$ leads to the $\bot$-state. That is, the last symbol in trace $amam$ is an output symbol that leads from a legal state to the $\bot$ state. This indicates that we can detect safety violations by checking whether there is an output transition leading from a legal state to the $\bot$-state.

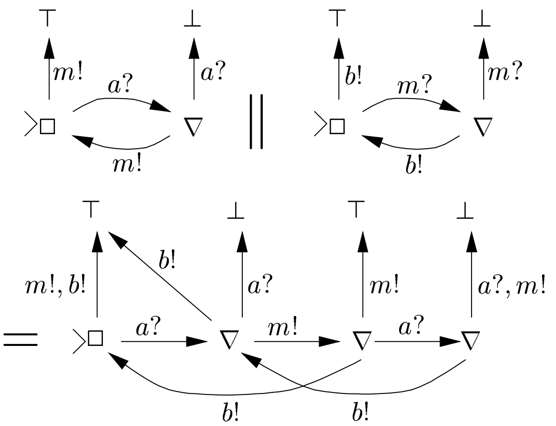


Figure 4. The network composition of two WIREs

The state labels in the graphs of Figure 4 contain information about progress properties of a network of two WIREs. For example, after the first WIRE has received an input on port a and has produced an output on port m, the product ends up in a state labeled with ∇, because the second WIRE guarantees that it will produce an output on port b.

Hiding takes into account progress and safety properties of a process. For example, consider process P and Q in Figure 5. The result of hiding output m is the specification of a WIRE, shown as process Q in Figure 5b. For example, take trace $t = aa$. Some traces in $X(aa)$ are aa, ama, and $amam$. Their corresponding labels in process P are $\mathbf{f}.P.aa = \bot$, $\mathbf{f}.P.ama = \nabla$, $\mathbf{f}.P.amam = \bot$. Because $\bot$ is the smallest of these labels, we have by definition of hiding $\mathbf{f}.|[m :: P]|.aa = \bot$, which agrees with $\mathbf{f}.Q.aa = \bot$. Because inputs leading to the $\bot$ state may not be produced by the environment, the result of hiding forbids the environment to provide an input that would cause an internal failure of a process.

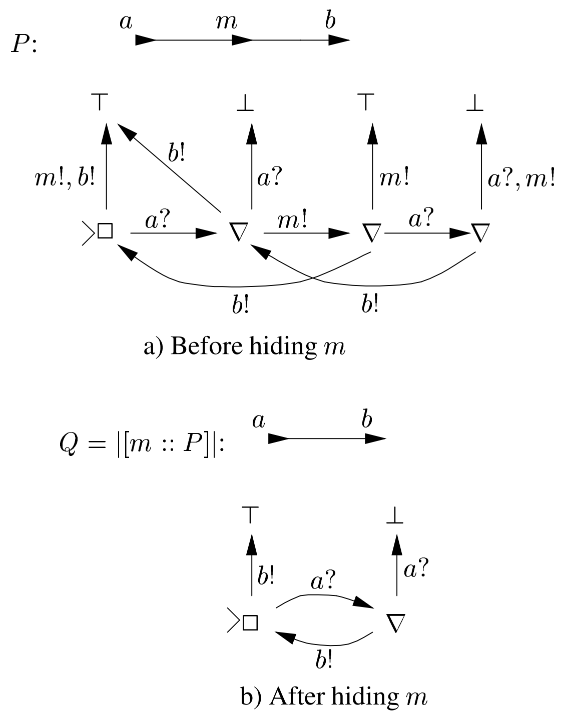


Figure 5. Hiding a connection between two WIREs

8. Specification composition

We use the specification composition to specify the communication behaviors of one component as a conjunction of constraints. Such specifications are called constraint-oriented specifications. As illustrated in Figure 6, a constraint can be seen as a restricted view of the communication behavior of the process. Constraints are expressed by special processes called *snippets*. Once we have collected a complete set of constraints, we combine the constraints with the specification composition, resulting in the complete specification of the process. Making sure that the set of collected constraints is complete presents challenges that are inherent to any method for producing a specification. Because the specification is the starting point of a design process, we cannot prove the correctness of a specification. The ability to focus on small aspects of the behavior, however, does help in gaining a better insight into the operation of the device.

In order to present some attractive properties involving the specification composition we introduce the definitions of rounded product, snippet, and output-persistent refinement. We first start with the definition of the *rounded product* of processes. By rounding we transform a potentially unhealthy process into a healthy one. More precisely, if $\mathbf{u}.P$ denotes the set of *unhealthy* traces of process P, then

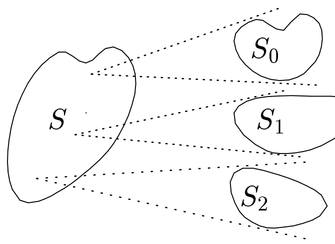


Figure 6. Specification composition

by rounding we transform each unhealthy trace in $\mathbf{u}.P$ into an indifferent trace. Notice that an indifferent trace cannot be unhealthy.

Definition 8.1 (Rounding) *Let P be a process. Rounding of process P is denoted by $[P]$ and is defined as follows: $\mathbf{i}.[P] = \mathbf{i}.P$, $\mathbf{o}.[P] = \mathbf{o}.P$, and*

$$\mathbf{f}.[P].t \quad = \quad \begin{cases} \mathbf{f}.P.t & \text{if } t \notin \mathbf{u}.P \\ \Box & \text{if } t \in \mathbf{u}.P \end{cases} \tag{9}$$

For processes P and Q we write $P \otimes Q$ to denote their rounded product $[P \times Q]$.

Our second definition concerns *snippets*. A snippet is a process that always guarantees progress, if an output is possible, and never demands progress from its environment.

Definition 8.2 (Snippet)
Process P is called a snippet *if*

1. *Process P must make progress iff an output is possible: For every legal trace t we have*

$$(\mathbf{f}.P.t = \nabla) \;\Leftrightarrow\; (\exists a :\; a \in \mathbf{o}.P : \mathbf{f}.P.ta \neq \top)$$

2. *The environment of P never guarantees progress and never performs a miracle: For all legal traces t we have*

$$\mathbf{f}.P.t \neq \Delta,\; \mathbf{f}.P.\varepsilon \neq \top, \qquad \text{and} \quad (\forall b : b \in (\mathbf{i}.P) : \mathbf{f}.P.tb \neq \top)$$

¿From the definition of a snippet it follows that a snippet is a healthy process.

Our third definition pertains to the *output-persistent* refinement denoted by $\sqsubseteq_o$. If $P \sqsubseteq_o Q$, then, on top of the normal progress and safety requirements, implementation Q must be capable of producing every output that can be produced by the specification P.

Definition 8.3 (Output-persistent refinement)
The output-persistent refinement of snippet P by snippet Q is denoted by $P \sqsubseteq_o Q$ and is defined as

$$P \sqsubseteq_o Q \;\equiv\; P \sqsubseteq Q \;\wedge\; P \preceq_o Q \tag{10}$$

where $P \preceq_o Q$ stands for process Q being output-persistent with respect to process P:

$$P \preceq_o Q \;\equiv\; \big(\forall t, a:\; t \in \mathbf{l}.P \wedge a \in \mathbf{o}.P \;:\; (\mathbf{f}.P.ta \neq \top \Rightarrow \mathbf{f}.Q.ta \neq \top)\big) \tag{11}$$

The following properties hold.

Theorem 8.4 (Properties)
1. $\sqsubseteq_o$ is a partial order on processes.
2. Snippets are closed under the rounded product.
3. The rounded product of snippets is idempotent, commutative, and associative.
4. The rounded product is monotonic with respect to output-persistent refinement, i.e.,

$$P \sqsubseteq_o Q \;\Rightarrow\; P \otimes R \sqsubseteq_o Q \otimes R$$

5. The product of snippets with disjoint alphabet is healthy: For snippets P and Q, where $\mathbf{o}.P \cap \mathbf{o}.Q = \emptyset$, we have $P \otimes Q = P \times Q$.

The reasons for the introduction of snippets and output-persistent refinement is that some properties above do not hold for processes and refinement in general. For example, the rounded product applied to processes is not associative in general and is also not monotonic with respect to refinement.

Our definition of the specification composition is based on the rounded product:

Definition 8.5 (Specification composition)
The specification composition *of P and Q applies when P and Q are snippets and when $\mathbf{o}.P \cap \mathbf{i}.Q = \emptyset$, and $\mathbf{o}.Q \cap \mathbf{i}.P = \emptyset$. The specification composition of P and Q is denoted by $P \,\&\, Q$ and defined by $P \otimes Q$.*

Although the formal definitions of the network composition and the specification composition are similar, the applications of the two operations are quite distinct. You use the network composition to model the joint behavior of a collection of devices. You use the specification composition to model the behavior of only one device by combining all constraints on the behavior of that device. Behavioral constraints of one device and network behaviors can both be captured by the same formal construct: ECF processes.

Snippets are closed under the specification composition, but not under hiding. This discrepancy does not cause any problems. In fact, we exploit this property, because almost all interesting ECF processes can be expressed by means of hiding symbols in a specification composition of snippets. The symbols that are hidden often serve as internal synchronization points among the snippets. The next sections illustrate the use of this specification method.

We use a textual notation called *commands* to represent snippets. A command is a regular expression for the legal trace set of a snippet. The basic building blocks are $a?$ and $a!$ denoting an input and an output action, respectively. If E and F are commands, then $E; F$ denotes their concatenation, $E \mid F$ denotes the union, and $*[E]$ denotes the Kleene closure.

The safety and progress properties of the traces of a command are defined as follows. First we stipulate that commands always express safe snippets. Thus, every illegal trace obtained from a legal trace by adding an output symbol is labeled with $\top$, and every illegal trace obtained from a legal trace by adding an input symbol is labeled with $\bot$. Secondly, from the definition of a snippet, it follows what the labels are for legal traces: if an output is possible after a legal trace, then that trace is labeled with ∇; otherwise, the legal trace is labeled with $\Box$.

Because a command defines a snippet and a snippet is a process, we can use commands to represent ECF processes. For example, a command for a WIRE is

$$\text{WIRE} = *[\, a?; b! \,]$$

We illustrate the use of commands and the specification composition in the specification of an initialized SEQUENCER. An initialized SEQUENCER is an arbiter that arbitrates among requests that can arrive concurrently from a number of different sources. The arbiter ensures that only one pending request is granted at any time.

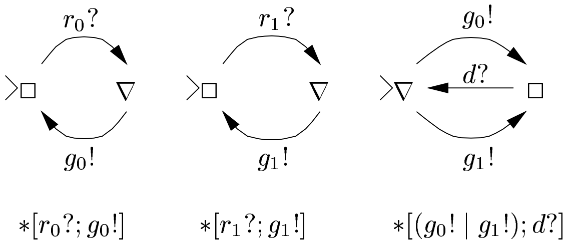
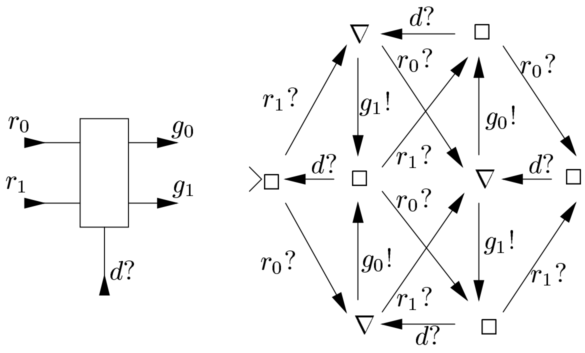


Figure 7. Snippets for the SEQUENCER

In Figure 7 we show three snippets that capture the behavior of the SEQUENCER. For each snippet we have given a state graph and a command. In order to reduce the clutter, we omitted the $\top$ and the $\bot$ state from the state graphs in Figure 7. By convention, all missing input transitions lead to the $\bot$ state and all missing output transitions lead to the $\top$ state.

The first snippet specifies that requests r_0 and grants g_0 must alternate and that, after receiving request r_0, the arbiter guarantees progress. The second snippet is similar to the first, except that it addresses the alternation of r_1 and g_1. Finally, the third snippet expresses the mutual exclusion between grants g_0 and g_1. This snippet also states that a grant and done signal must alternate. Moreover, by labeling the initial state of this snippet with ∇ we require that the SEQUENCER does not stop when it can issue a grant.

Figure 8b shows the result of the specification composition of the snippets of Figure 7. Notice that the initial state of the graph of Figure 8b carries label $\Box$, whereas the product of the labels of the initial states of the snippets from Figure 7 is ∇. This means that the label of the initial state of the specification of the SEQUENCER is rounded down to $\Box$, because the SEQUENCER initially cannot produce an output. Notice also that the arbiter is in a ∇ state only when it has received one or two requests *and* a done signal d. Initially, we assume that signal d has been received.



a) Schematic b) State graph

Figure 8. A specification of an arbiter

The specification of the SEQUENCER involves a significant amount of concurrency. Requests r_0 and r_1 can arrive independently of each other, which contributes to sequences of events that may not be easy to envision if we were to generate the state graph directly rather than by computing the specification composition of snippets. Thinking in terms of snippets allows us to focus on a few aspects of the behavior of the SEQUENCER, which combined lead to the final specification. It also allows us to give a concise specification: an n input SEQUENCER requires only $n + 1$ small snippets, whereas the complete state graph quickly becomes prohibitively large.

9. Part-wise design method

In this section we outline the part-wise design method, which assumes that we have expressed a specification S as a specification composition of snippets where the set A of "auxiliary internal" symbols is hidden.

$$S \;=\; |[\, A :: P \;\&\; Q \,]|$$

Notice that S belongs to a process domain that is larger than the domain of snippets, because snippets are not closed under hiding.

Our intention is to seek output-persistent refinements for snippets P and Q in isolation and then combine the two output-persistent refinements to obtain a refinement of specification S. We emphasize that verifying output-persistent refinements of P and Q in isolation tends to be significantly less complex than verifying the final refinement of specification S. The number of states in $P \& Q$ can be of the order of the product of the number of states of P and Q. Thus, by looking at snippets P and Q in isolation, we may avoid a state explosion.

Assume that snippet P can be refined by the network composition of snippets P_0 and P_1, and that snippet Q can be refined by the network composition of snippets Q_0 and Q_1. Assume, furthermore, that these refinements are output-persistent refinements:

$$\begin{array}{lcl} P & \sqsubseteq_o & P_0 \parallel P_1 \\ Q & \sqsubseteq_o & Q_0 \parallel Q_1 \end{array}$$

Because P_0 and P_1 have no common outputs, and neither do Q_0 and Q_1 have common outputs, we have, by Theorem 8.4 (5):

$$\begin{array}{lcl} P & \sqsubseteq_o & P_0 \otimes P_1 \\ Q & \sqsubseteq_o & Q_0 \otimes Q_1 \end{array}$$

Recall that snippets are closed under rounded product. Thus, $P_0 \otimes P_1$ and $Q_0 \otimes Q_1$ are snippets. Hence, we can apply monotonicity of rounded product with respect to output-persistent refinement and transitivity of output-persistent refinement, which leads to

$$P \otimes Q \;\sqsubseteq_o\; P_0 \otimes P_1 \otimes Q_0 \otimes Q_1$$

Next we group snippets that have common outputs. For example, assume that snippets P_0 and Q_0 have common outputs. Thus, we can write $P_0 \otimes Q_0 = P_0 \& Q_0$. We also take into account that $P \otimes Q = P \& Q$. Hence, we get

$$P \& Q \;\sqsubseteq_o\; (P_0 \& Q_0) \otimes P_1 \otimes Q_1$$

Finally, we recall that, by Theorem 8.4 (5), the rounded product of snippets with no common outputs is equal to their network composition:

$$P \& Q \;\sqsubseteq_o\; (P_0 \& Q_0) \parallel P_1 \parallel Q_1$$

Now we know that the specification composition of snippets P and Q can be refined with a network of three components. Recall that the output-persistent refinement implies the normal refinement. Furthermore, hiding is monotonic with respect to refinement. Thus, we get our final refinement for S

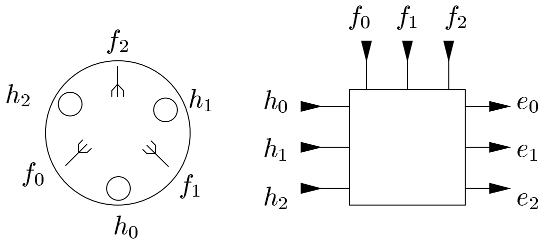
$$|[A :: P \& Q]| \;\sqsubseteq\; |[\, A :: (P_0 \& Q_0) \parallel P_1 \parallel Q_1 \,]|$$

Let us summarize the design approach presented in this section: If we are able to express a process as a specification composition of snippets with internal symbols hidden, we can look in isolation at refinements of individual snippets. Then, we group snippets with common outputs and each of these groups of snippets represents a component in an implementation of the original specification.

10. Dining philosophers: Specification

The dining philosophers is a canonical synchronization problem that was originally stated by Dijkstra [6]:

> Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having her own place at the table. Their only problem — besides those of philosophy — is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks at each plate, so that presents no difficulty: as a consequence, however, no two neighbors can be eating simultaneously.



a) dining table b) schematic of a scheduler

Figure 9. Dining philosophers

To keep the example simple, we consider the case with only three philosophers using the table shown in Figure 9a. We would like to design a device that schedules the meals for the philosophers. Each philosopher is engaged in an unbounded cycle of thinking followed by eating. When philosopher i gets hungry, she sends signal h_i to the scheduler, shown in Figure 9b. If both forks f_i and f_{i+1} are free, the scheduler sends signal e_i to philosopher i to inform her that she can eat. When philosopher i has finished a meal, she releases the forks. This release eventually leads to signals f_i and f_{i+1} to the scheduler, meaning that forks i and $i+1$ are no longer in use. The addition is modulo 3.

We specify the behavior of the scheduler as a specification composition of snippets. First we observe that, if a philosopher becomes hungry, she will eventually eat. That is, input h_i must be followed by output e_i. This observation leads to the following snippets for $i = 0, 1, 2$

$$*[\, h_i?; e_i! \,]$$

Next we turn to the use of the forks. Each fork can be accessed by two philosophers, but only one philosopher may use it at any time. Thus, for fork i, signal f_i must be followed by either signal e_i or e_{i-1}. We assume that initially all forks are on the table. These observations lead to the following snippet that describes the usage of fork i:

$$*[\, (e_i! \mid e_{i-1}!); f_i? \,]$$

The specification of the meal scheduler is the specification composition of the snippets introduced above:

$$\begin{array}{rcl} \text{SCH} & = & *[\, h_0?; e_0! \,] \\ & \& & *[\, h_1?; e_1! \,] \\ & \& & *[\, h_2?; e_2! \,] \\ & \& & *[\, (e_0! \mid e_2!); f_0? \,] \\ & \& & *[\, (e_0! \mid e_1!); f_1? \,] \\ & \& & *[\, (e_1! \mid e_2!); f_2? \,] \end{array}$$

The complete state graph for this specification contains 64 states and would be hard to find without using the specification composition.

11. Dining philosophers: Implementation

Figure 10 shows an implementation of the meal scheduler consisting of three SEQUENCERs. We verify this implementation in a few steps using the part-wise refinement.

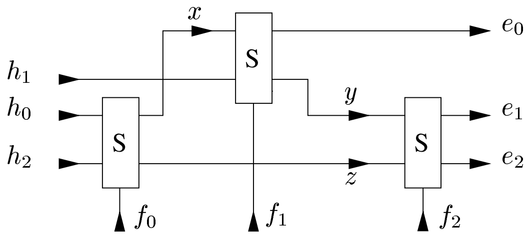


Figure 10. A solution to the dining philosophers problem

We first introduce the internal symbols x, y, and z in the specification of the scheduler:

$$\begin{array}{rcl} \text{SCH}_0 & = & *[\, h_0?; x!; e_0! \,] \\ & \& & *[\, h_1?; y!; e_1! \,] \\ & \& & *[\, h_2?; z!; e_2! \,] \\ & \& & *[\, (x!; e_0! \mid z!; e_2!); f_0? \,] \\ & \& & *[\, (e_0! \mid y!; e_1!); f_1? \,] \\ & \& & *[\, (e_1! \mid e_2!); f_2? \,] \end{array}$$

The introduction of symbols x, y and z is based on the schematic of the implementation in Figure 10 and effectively forces an order in the assignment of forks to philosophers. The original pecification SCH, on the other hand, does not stipulate in which order the forks are assigned. Without proof we state that specification SCH_0 stipulates that philosophers 0 and 1 first get assigned their left-hand forks and then their right-hand forks. Philosopher 2, on the other hand, first gets assigned her right-hand fork and then her left-hand fork. This approach follows Dijkstra's solution [6] of the dining philosophers problem, where such an asymmetry in the sequence of acquiring forks guarantees the absence of deadlock. This order of assignment can also be inferred from Figure 10. We have

$$\text{SCH} \;=\; |[\, x, y, z :: \;\text{SCH}_0 \,]| \tag{12}$$

If the internal symbols in SCH_0 would assign the forks of each philosopher in the same order, then, as demonstrated in [6], the resulting "solution" has a possibility of deadlock. Our refinement relation detects such deadlocks; thus equality 12 would not hold.

The snippets that form SCH_0 have the following simple refinements:

$$\begin{array}{rcl}
*[\, h_0?; x!; e_0! \,] & \sqsubseteq_o & *[\, h_0?; x! \,] \;\&\; *[\, x!; e_0! \,] \\
*[\, h_1?; y!; e_1! \,] & \sqsubseteq_o & *[\, h_1?; y! \,] \;\&\; *[\, y!; e_1! \,] \\
*[\, h_2?; z!; e_2! \,] & \sqsubseteq_o & *[\, h_2?; z! \,] \;\&\; *[\, z!; e_2! \,] \\
*[\, (x!; e_0! \mid z!; e_2!); f_0? \,] & \sqsubseteq_o & *[\, (x! \mid z!); f_0? \,] \\
 & & \&\; *[\, x!; e_0! \,] \\
 & & \&\; *[\, z!; e_2! \,] \\
*[\, (e_0! \mid y!; e_1!); f_1? \,] & \sqsubseteq_o & *[\, (e_0! \mid y!); f_1? \,] \\
 & & \&\; *[\, y!; e_1! \,] \\
*[\, (e_1! \mid e_2!); f_2? \,] & \sqsubseteq_o & *[\, (e_1! \mid e_2!); f_2? \,]
\end{array}$$

According to our part-wise refinement method we we must group snippets with common output symbols and take their specification composition. The network composition of these groupings forms a refinement of the scheduler. Thus, we get

$$\begin{array}{lll}
\text{SCH} & \sqsubseteq & |[\, x, y, z :: \\
 & & \quad (\quad *[\, h_0?; x! \,] \;\&\; *[\, h_2?; z! \,] \\
 & & \quad\;\; \&\; *[\, (x! \mid z!); f_0? \,] \,) \qquad \text{(SEQ)} \\
 & & \| \;\; (\quad *[\, h_1?; y! \,] \;\&\; *[\, x?; e_0! \,] \\
 & & \quad\;\; \&\; *[\, (e_0! \mid y!); f_1? \,] \,) \qquad \text{(SEQ)} \\
 & & \| \;\; (\quad *[\, y?; e_1! \,] \;\&\; *[\, z?; e_2! \,] \\
 & & \quad\;\; \&\; *[\, (e_1! \mid e_2!); f_2? \,] \,) \qquad \text{(SEQ)} \\
 & &]|
\end{array}$$

Consequently, the implementation of SCH consists of three SEQUENCERs connected as shown in Figure 10.

The solution to the dining philosophers problem presented in Figure 10 is just a hardware rendition of Dijkstra's solution from [6]: each SEQUENCER implements a semaphore that the philosophers use in order to get access to their forks.

The solution can easily be generalized to any number of philosophers. A flat verification of such an implementation quickly becomes impossible, even for a machine, because of the state explosion. Using part-wise refinement, however, our proof obligations increase only slightly and can simply be done by hand.

12. Conclusions

We have presented a simple formalism for the specification and implementation of asynchronous circuits. Three aspects of this formalism deserve special mention: the difference between the network composition and the specification composition, the formal definition of snippets, and the part-wise refinement method.

The network composition models the joint behavior of a network of devices, where each device is represented by a process. The specification composition, on the other hand, models the behavior of just one device as a combination of snippets, where each snippet represents an aspect of the device's behavior. In our previous formal models, there was no difference between the formal definitions of the network composition and the specification composition. Only when progress conditions come into play, this difference emerges.

The usefulness of the specification composition becomes apparent when we specify complex behaviors that involve a large degree of concurrency. State graphs for such specifications tend to grow quickly and may become difficult to keep track of. It has been our experience that individual snippets tend to remain small in size, and a list of snippets tends to be smaller and easier to keep track of than the state graph that represents the complete behavior. Furthermore, focusing on small individual aspects of a complex behavior allows us to gain a better insight into the operation of a device.

The part-wise refinement method allow us to avoid a state explosion in the verification: Instead of verifying a complete implementation, the part-wise refinement method allows us to verify the refinements of just the snippets of a process. The part-wise refinement can be used in combination with a stepwise refinement, also called hierarchical verification. Together they form powerful tools in design and verification.

13. Acknowledgments

Acknowledgments are due to Graham Birtwistle, John Brzozowski, and the reviewers for their comments on previous drafts of this paper. Sun Microsystems Laboratories is gratefully acknowledged for the financial support.

References

[1] I. Benko. *ECF Processes and Asynchronous Circuit Design*. PhD thesis, University of Waterloo, Department of Computer Science, 1999.

[2] I. Benko and J. Ebergen. Delay-insensitive solutions to the committee problem. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 228–237. IEEE Computer Society Press, Nov. 1994.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[4] M. E. Bush and M. B. Josephs. Some limitations to speed-independence in asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[5] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[6] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[7] J. C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 5(3):107–119, 1991.

[8] J. C. Ebergen. Arbiters: an exercise in specifying and decomposing asynchronously communicating components. *Science of Computer Programming*, 18(3):223–245, June 1992.

[9] W. Mallon. Personal communication, 1997.

[10] W. C. Mallon, J. T. Udding, and T. Verhoeff. Analysis and applications of the XDI model. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 231–242, Apr. 1999.

[11] C. Molnar, I. Jones, and I. Sutherland. A way to compose Petri nets, 1992. Sun Microsystems Laboratories Memo 92:0354.

[12] C. Molnar, I. Sutherland, and I. Jones. A Petri-net weave, 1992. Sun Microsystems Laboratories Memo 92:0357.

[13] R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, Dept. of Computer Science, Univ. of Waterloo, Canada, Aug. 1998.

[14] R. Negulescu and A. Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 159–170, 1998.

[15] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Eindhoven University of Techology, 1994.

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