ECF Processes and Asynchronous Circuit Design

by

Igor Benko

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1999

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Specifying hardware can present major challenges, especially when one is faced with a system that involves a large degree of concurrency. We propose a simple formal framework for representing concurrent systems. The framework is based on Enhanced Characteristic Functions (ECFs), which assign labels to sequences of communication actions. Labels represent properties of a process after a sequence of actions has been executed. A process in the framework is called an ECF process.

The general ECF model does not rely on a particular set of labels. Instead, the labels must satisfy a number of simple properties, which we use to define the operations on processes: refinement, hiding, and process product. The properties of labels, lifted to processes, lead to a number of desirable properties of these operations.

An instantiation of the general ECF model is obtained by choosing a particular set of labels. We study two instantiated ECF models, one addressing safety properties, and one addressing progress properties of processes. In each of the two models we give a correctness condition for a process and we show that the refinement relation has three equivalent characterizations. We also give alternative definitions of hiding for both the safety and the progress model. Finally, we show that ECF processes lend themselves well to a design technique based on the Factorization Theorem.

We define a specification composition, which combines behavioral constraints into one specification. Constraints are expressed by so-called snippets. We propose a part-wise design method that is applicable to specifications expressed in terms of snippets. This design method allows us to find in isolation implementations of the snippets. These partial implementations are then combined into an implementation of the original specification. We illustrate the part-wise design method on a number of examples.

Acknowledgements

This thesis would not have been written without the continuous support and encouragement from my wife and my best friend, Jasna Jurjovec. Even when times were really tough for her, she was always a source of warmth and courage. I am deeply grateful to her and I hope that I can, with time, return at least part of what I have received.

My parents were always unconditionally behind me, regardless of being an ocean away. Their love and confidence gave me strength when I needed it most.

My advisor, Jo Ebergen, managed to keep in regular contact despite of the geographic distance and the demands of his job. He fought patiently through a large number of reports I kept sending to him and he always returned them promptly, marked with good comments and suggestions. For their hard work, I thank the members of my thesis committee: Graham Birtwistle, John Brzozowski, Gord Cormack, and Bruno Preiss. In particular, John Brzozowski worked above and beyond the call of duty and provided many suggestions that lead to improvements in presentation of the material in the thesis.

My internships at Sun Microsystems Laboratories were times of joy and eye opening. Ivan Sutherland, Charlie Molnar, Bill Coates, Ian Jones, Jon Lexau, and Scott Fairbanks accepted me as one of their own and taught me far more than just circuit design.

I was very fortunate to connect with a number exceptional individuals who were a rich source of friendship and advice: Peter Buhr, Laurie Brown, Jean Duhamel, Wendy Hatch, Robert Kroeger, and Wendy Rush. I also thank Peter Buhr, Jean Duhamel, and fellow members of the Warriors squash team for the many games and the beatings that helped me keep the levels of stress and humility under control.

Finally, I would like to acknowledge sources of my financial support: Ontario Graduate Scholarship, Information Technology Research Centre of Ontario, National Sciences and Engineering Research Council of Canada, and Sun Microsystems.

Contents

1 Introduction **1**

1.1 Asynchronous circuits 2

1.2 Preview 6

1.3 Related work 13

1.3.1 Other formalisms 19

1.4 Contributions 22

1.5 Road map 25

1.6 Notation 26

2 ECF processes **29**

2.1 Labels 30

2.2 Processes 32

2.3 Refinement and reflection 35

2.4 Product 43

2.5 Correctness and testing 48

2.6 Hiding 50

2.7 Substitution theorem . . . 56
2.8 Summary . . . 58

3 ECF processes and safety **61**

3.1 Labels and processes . . . 63
3.2 Refinement . . . 65
3.2.1 Examples of refinement . . . 66
3.3 Product . . . 70
3.3.1 Product and legal traces . . . 74
3.4 Reflection . . . 75
3.5 Safety . . . 76
3.6 An alternative characterization of refinement . . . 78
3.7 Hiding . . . 80
3.7.1 Examples of hiding . . . 80
3.8 Properties of hiding . . . 85
3.9 Factorization Theorem . . . 89
3.10 Summary . . . 93

4 ECF treatment of progress **95**

4.1 Labels and processes . . . 96
4.1.1 Example processes . . . 97
4.2 Unhealthy processes and progress failures . . . 100
4.3 Refinement . . . 101
4.3.1 Examples of refinement . . . 102

4.4 Reflection 104

4.5 Product 106

4.6 Progress and refinement 110

4.7 Hiding 115

4.8 Hiding and unhealthy processes 117

4.9 Hiding and correctness 119

4.10 An alternate definition of hiding 121

4.11 Factorization Theorem 123

4.12 Summary 126

5 Specification composition **129**

5.1 The specification composition and the network composition in the safety model 132

5.2 Healthy and unhealthy processes 133

5.3 Rounding processes up and down 136

5.4 Snippets and output-persistent refinement 140

5.5 Properties 143

5.6 The network composition in the progress model 152

5.7 The specification composition in the progress model 154

5.8 Summary 158

6 Applications **161**

6.1 Commands 163

6.2 2-to-4 phase converter 171

6.3 Part-wise design method . . . 174

6.4 Micropipeline cell . . . 177

6.4.1 Micropipeline cell: Specification . . . 177

6.4.2 Micropipeline cell: Implementation . . . 179

6.5 FIFO . . . 181

6.5.1 FIFO: Specification . . . 182

6.5.2 FIFO: Implementation . . . 184

6.6 Dining philosophers . . . 187

6.6.1 Dining philosophers: specification . . . 187

6.6.2 Dining philosophers: Implementation . . . 189

6.6.3 An "implementation" with deadlock . . . 194

6.7 3-input SEQUENCER . . . 195

6.8 Summary . . . 200

7 Conclusions **201**

A Glossary of symbols **207**

B Ordered sets and lattices **209**

B.1 Relations . . . 209

B.2 Ordered sets and lattices . . . 210

C Proofs **213**

Bibliography **227**

List of Tables

3.1 Product of labels 70

3.2 Reflection on labels 75

4.1 Reflection on labels 105

4.2 Product table 106

List of Figures

1.1 C-element 7

1.2 C-element and JOIN 7

1.3 The JOIN with safety properties 9

1.4 The JOIN with progress properties 10

1.5 The WIRE 11

1.6 The network of the JOIN and the WIRE 11

1.7 The network of the JOIN and the WIRE after hiding output c 12

1.8 Two snippets for the JOIN 13

2.1 An example of a set of labels 31

2.2 Creating a state graph for a process 33

2.3 An example of a process represented by a state graph 34

2.4 Verifying process refinement on state graphs 36

2.5 Process reflection 42

2.6 Calculating the state graph for the process product 45

3.1 A specification of the WIRE 64

3.2 An implementation of the WIRE 66

3.3 Another implementation of the WIRE 67

3.4 A specification of the SELECTOR 68

3.5 An implementation of the SELECTOR 68

3.6 A bad implementation of the SELECTOR 69

3.7 Another bad implementation of the SELECTOR 69

3.8 The product of two WIREs 72

3.9 Process product illustration 73

3.10 The reflection of the WIRE 76

3.11 A safe and an unsafe process 77

3.12 A self-failing process 81

3.13 A process that fails after a "hidden" transition 81

3.14 Hiding transitions to $\top$ state 82

3.15 A process that aborts after receiving an input 82

3.16 A process that aborts after producing an output 83

3.17 (b) pruning transitions on port c, versus (c) hiding port c 84

3.18 The product of two WIREs 84

3.19 Hiding the connection between WIREs: Hidden transitions marked by ε . 85

3.20 Hiding the connection between two WIREs: Final result 85

3.21 Factorization Theorem 90

3.22 The MERGE 92

3.23 An application of the Factorization Theorem 93

3.24 A bad initial guess 93

4.1 A specification of a WIRE . . . 97
4.2 A specification of a JOIN . . . 98
4.3 A specification of an arbiter . . . 99
4.4 Unhealthy processes . . . 100
4.5 An example of refinement . . . 103
4.6 A violation of refinement of the WIRE . . . 103
4.7 An unhealthy implementation . . . 104
4.8 Refinement does not preserve nondeterminism . . . 105
4.9 A product of two WIREs . . . 108
4.10 Refinement requires more than safety . . . 111
4.11 A Δ state indicates lack of progress . . . 111
4.12 The progress condition alone is insufficient for the refinement . . . 112
4.13 Hiding a connection between two WIREs . . . 116
4.14 The ticker process . . . 117
4.15 A prospective implementation of the WIRE . . . 118
4.16 An unhealthy process reveals progress problems . . . 118
4.17 Hiding can mask incorrectness . . . 119
4.18 The MERGE . . . 125
4.19 An application of the Factorization Theorem . . . 125

5.1 The network composition . . . 129
5.2 The specification composition . . . 130
5.3 A specification of the JOIN . . . 132

5.4 Product is *not* a specification composition 134

5.5 A Δ-unhealthy process can be the result of a process product 136

5.6 Rounding up a Δ-unhealthy process 137

5.7 Rounding-down a ∇-unhealthy process 137

5.8 The rounded product of processes is not monotonic 139

5.9 Rounded product on snippets is not monotonic 141

5.10 Examples of output-persistent refinement 143

5.11 Rounded product is not associative for processes 147

5.12 Rounded product is not associative for processes that fail on a transition from a $\Box$-state 148

5.13 The monotonicity does not hold for the rounded product of processes . . . 151

5.14 The monotonicity does not hold if we allow input transitions from legal states to the $\top$-state 152

5.15 The network composition of two WIREs 154

5.16 The specification composition for the JOIN 156

5.17 A schematic for the SEQUENCER 156

5.18 Snippets for the SEQUENCER 157

5.19 A specification for the SEQUENCER 158

6.1 State graphs for atomic commands 166

6.2 2-by-1 JOIN and Arbitrating Test-and-Set 170

6.3 2-to-4 phase converter 171

6.4 A micropipeline cell 177

6.5 An implementation of a micropipeline cell 181

6.6 A schematic of the FIFO 182

6.7 Internal data movement in the FIFO 183

6.8 An implementation of the FIFO 186

6.9 Dining philosophers 187

6.10 A solution to the dining philosophers problem 193

6.11 3-input SEQUENCER 195

6.12 The topology of a token-ring implementation for a SEQUENCER 197

6.13 A token-ring implementation of a 3-input SEQUENCER 199

Chapter 1

Introduction

The first step in building software or hardware is producing a specification. This step can present major challenges, especially when one is faced with specifying a system that involves a significant degree of concurrency. One challenge is to choose a notation that is simple, yet expressive enough to capture the intended properties of a design. A formalism behind the notation should provide means for comparing an implementation and the specification, allowing one to check whether there is a match between what was intended and what was done.

We propose a simple formal framework, based on Enhanced Characteristic Functions (ECFs), that can be used for representing concurrent systems. 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 studying safety and progress properties of networks of circuit components.

1.1 Asynchronous circuits

In this section we introduce asynchronous circuits, which we use in examples of applications of ECF processes.

The vast majority of circuits designed today are *synchronous circuits*, built around an external synchronization signal called the clock. The role of the clock is similar to the role of the drummer on a galley who was setting pace to the oarsmen: At a tick of the clock, all components of a synchronous circuit must be ready to perform a subsequent operation. The reasons why clocked circuits are today's choice for the vast majority of circuit designers are manifold. First, the discreteness of time introduced by the clock simplifies many tasks in synchronous design. Second, in clocked circuits one does not have to control or remove hazards [Huffman, 1964], thus Boolean algebra can be applied to circuit design in a straightforward manner. Third, many design tools and large libraries of components are available for synchronous-circuit design. Fourth, inertia caused by industry and knowledge investments is unlikely to allow a rapid shift in a design paradigm.

An alternative to clocked circuits are *asynchronous* circuits, which are circuits without the clock. An asynchronous circuit consists of a network of components, each operating at its own speed. Global synchronization performed by the clock is replaced by local synchronizations between components. The local synchronization is typically achieved by means of request/acknowledge handshaking.

According to various models, we can classify asynchronous circuits into several categories. The most commonly used categories are as follows:

Self-timed circuits are described in [Seitz, 1980]. Their characteristic is that each circuit element is contained in a so-called equipotential region, where delays in wires are under control. It is left open whether any assumptions are made about delays on wires that connect circuit components.

Speed-independent circuits, defined by Muller and Bartky [Muller and Bartky, 1959], have delays associated with circuit components, but communication delays in interconnecting wires are negligible. The class of speed-independent circuits has some limitations as demonstrated in [Bush and Josephs, 1996]. Furthermore, the assumption of negligible wire delays can be unrealistic for large circuits. As shown in [Ebergen, 1991], communication delays can be modeled by inserting explicit WIRE components, allowing us to avoid unrealistic assumptions. The ECF model proposed in Chapter 2 applies to speed-independent circuits. With the ECF model we demonstrate that speed-independence allows for a simple formal model that can be easy to use.

Delay-insensitive circuits have no assumptions about delays in components and interconnecting wires. The concept of delay insensitivity grew from the work on Macromodules [Clark and Molnar, 1974], and was first formalized in [Udding, 1984]. If logic gates are used as basic components, the class of delay-insensitive circuits is very limited, as shown in [Brzozowski and Ebergen, 1992]. If more complex components are used, however, practical circuits can be built. For example, circuits produced by the Tangram compiler reported in [van Berkel, 1992] are delay-insensitive.

Quasi delay-insensitive circuits are delay-insensitive, except for so-called *isochronic forks*. The term *isochronic fork* refers to a forked wire where all branches have the same delay, or where the delay skew between the branches is bounded. Quasi delay-insensitive circuits are used, for example, by Martin [Martin, 1990, Martin, 1993]. Circuits produced by the Tangram compiler [van Berkel, 1992] after the optimization phase are also quasi delay-insensitive.

We are not aware of formal characterizations of self-timed circuits and of quasi delay-insensitive circuits. Also, speed-independence and delay-insensitivity were defined originally in different formal models. Consequently, a formal relationship between the models described above has not been established.

In the early days of computing, asynchronous circuits were recognized as an "awkward alternative" to clocked circuits [Turing, 1947]. Since then, researchers identified a number of benefits that could be offered by asynchronous circuits and interest in asynchronous circuits has been revived. Some of these benefits are

- **Low power:** The clock distribution contributes up to 50% of the power dissipation in a circuit [Badeau et al., 1992]. Because of the growing number of portable devices where low power consumption is critical, low-power design techniques have been gaining importance. Asynchronous circuits have a potential for low power, because there is no clock driver on a chip and the synchronizations among the components may be reduced to merely the essential ones. The Tangram compiler [van Berkel and Saeijs, 1988, van Berkel and Rem, 1995] has been used to pursue designing low-power asynchronous circuits. Examples of circuits designed are an Error Corrector for a DCC Player [van Berkel et al., 1994] and 80C51 microcontroller for a pager [Kessels and Marston, 1997, Gageldonk et al., 1998]. Pagers seem to be the first electronic products based on asynchronous circuits that are produced massively. Another prominent example of low-power asynchronous circuits are AMULET processors designed at the Manchester University [Furber et al., 1993, Furber et al., 1996, Furber et al., 1998].

- **Speed:** In synchronous circuits, the clock must be slow enough to accommodate the slowest component in the circuit. For this reason, the speed of synchronous circuits is limited by the slowest component. In an asynchronous circuit, each component operates at its own speed; thus, the performance may not be limited by the slowest component. Furthermore, the clock period must accommodate the worst possible case for data dependent operations. Completion-detection circuits used in asynchronous circuits allow for the average-case behavior, which may be significantly better than the worst-case behavior reflected by clocked circuits. The division circuit presented in [Williams, 1994] is an example of a completion-detection circuit that was used commercially, and was shown to perform faster than synchronous division circuits.

- **Robustness to metastability:** If a circuit enters a metastable state, it may remain there for an unbounded amount of time [Chaney and Molnar, 1973]. If the metastable behavior is not resolved within a clock period, a clocked circuit may exhibit erroneous behavior. Because chasing the clock is not an issue in asynchronous circuits, the circuit can wait until metastability has been resolved and then continue with its work, functionally unaffected by the unexpected delay.

- **Scalability:** Synchronous designs are scalable only to a limited extent, because delays in wires do not scale linearly with the characteristic dimensions of the circuit, including the clock period; see Section 4.13 of [Weste and Eshraghian, 1993] or Chapter 4 of [van de Snepscheut, 1985]. As a consequence, timing constraints may be violated in a scaled-down synchronous circuit and the circuit may not operate properly. Delay-insensitive circuits are designed without any assumptions about delays in responses of components and in interconnecting wires. Thus, delay-insensitive designs can be scaled safely. The ease of scalability is yet to be demonstrated as a practical benefit in designing asynchronous circuits. We believe that advances in silicon compilation and requirements for speeding up the design cycle will make the ease of scalability an important advantage of asynchronous circuits.

- **Modularity of design:** The clock is a global signal which must be distributed across the entire clocked circuit. The requirement that all circuit components must meet the timing constraints imposed by the clock is a global condition which does not naturally support modular design. Asynchronous circuits, on the other hand, can be easy to compose and thus lend themselves to a modular design. Ease of modular composition is also a reason why asynchronous circuits should be a good choice for designing at a high level. Silicon compilers such as Tangram [van Berkel and Saeijs, 1988, van Berkel and Rem, 1995] are early indicators that modularity of asynchronous circuits may help in speeding up the design cycle.

For comprehensive overviews of formalisms and methodologies for asynchronous circuit design we refer the reader to Chapter 15 of [Brzozowski and Seger, 1995] and to [Davis and Nowick, 1995].

1.2 Preview

In this section we give an intuitive preview of the theory of ECF processes developed in Chapter 2. We show some examples of ECF processes and we demonstrate a number of operations on ECF processes. ECF stands for "enhanced characteristic function", which assigns labels to traces. A trace label is used to indicate properties of a process that has executed that trace. To keep the presentation short, we postpone discussing previous work until Section 1.3.

We use ECF processes for modeling asynchronous circuit components. Figure 1.1 shows a schematic and a CMOS implementation of a common asynchronous circuit component called the C-element. To make modeling of a circuit manageable and applicable to large designs, we model circuits in a discrete binary domain. That is, we assume that

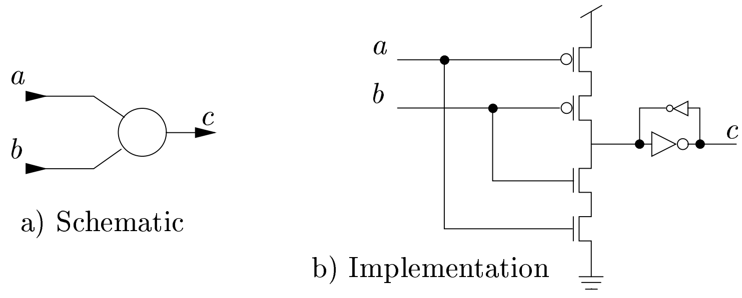


Figure 1.1: C-element

transistors operate as on/off switches, and we assume that signals can have one of the two possible values, either HIGH of LOW. Figure 1.2a shows a state graph that represents the behavior of the C-element, assuming that all terminals are initially LOW. The symbol $>$ marks the initial state of the graph, $\uparrow$ represents an up-transition, and $\downarrow$ represents a down-transition. In our specifications we assume that a component operates within some environment. The environment controls the values of the inputs to the component and the component controls the values of its outputs. For example, when both inputs to the C-element have the same value, the output of the C-element will assume the value of its inputs, otherwise the output of the C-element remains unchanged.

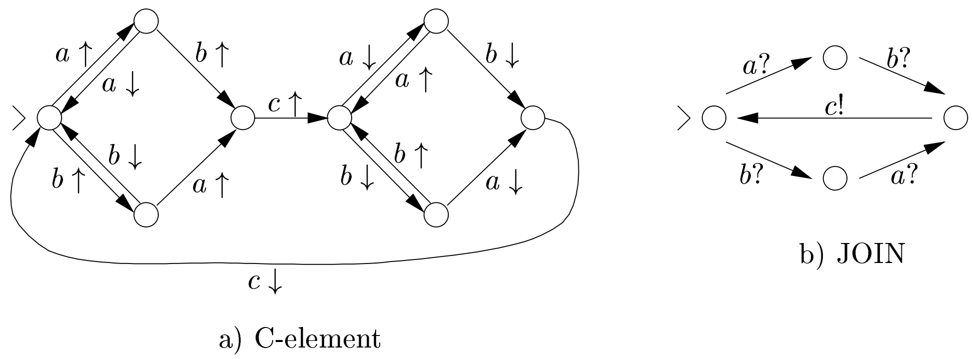


Figure 1.2: C-element and JOIN

The environment of the C-element from Figure 1.2a can “revoke” an input. For example, the environment can initially change the value of input a from LOW to HIGH, and then it can return the value of a back to LOW, without ever changing the value of input b. Revoking an input can create a race [Brzozowski and Seger, 1995]: Assume that input a has changed from LOW to HIGH. Now suppose that input a changes from HIGH to LOW and, at the same time, input b changes from LOW to HIGH. What is the value of output c? It is possible that input b wins the race, thus output c changes from LOW to HIGH. It is also possible that input a wins the race and the value of output c remains LOW. Looking at Figure 1.1, we observe that we could also get a short pulse on output c, if transistors connected to input a switch more slowly than transistors connected to input b. That is, the implementation of the C-element shown in Figure 1.1 has a static hazard [Stevens, 1994, Brzozowski and Seger, 1995]. Hazards are undesirable, because they can cause unpredictable behaviors.

In order to avoid the race, revoking of inputs of a C-element is often prohibited. A C-element, where revoking an input is not allowed, is called a JOIN. Figure 1.2b shows a state graph of a JOIN, ignoring differences between up-transitions and down-transitions. We use $a?$ and $b?$ to denote input transitions on ports a and b, respectively, and $c!$ to denote output transitions on port c. After transitions on both input a and input b have taken place, a transition on output c takes place. This behavior can repeat.

Figure 1.2b contains only information on what the JOIN and its environment can do, but it does not address the question of what the JOIN and its environment cannot or must not do. Figure 1.3 shows a model for the JOIN that characterizes explicitly sequences of communication actions (or traces) that the environment must avoid or that the JOIN guarantees to avoid. In order to obtain the state graph in Figure 1.3, we added two states to the graph from Figure 1.2b: the $\bot$ state, and the $\top$ state. $\bot$ is pronounced as "bottom" and $\top$ is pronounced as "top". All traces that cause a failure of the JOIN

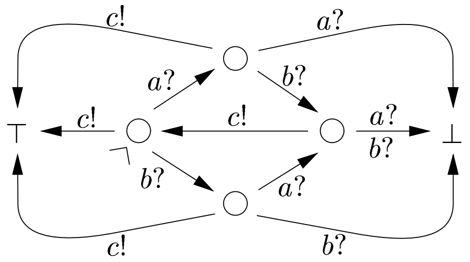


Figure 1.3: The JOIN with safety properties

lead to the $\perp$ state. The JOIN fails if the environment provides two transitions on the same input port without waiting for a transition on the output port. For this reason, trace aa leads to the $\perp$ state. The $\top$ state is the destination for the traces that cannot take place. For example, the JOIN cannot initially produce an output on port c, thus a transition on output c leads from the initial state to the $\top$ state.

The specification of the JOIN shown in Figure 1.3 is an example of a process from the *safety* model of Chapter 3. The safety model is an instantiation of the abstract model of ECF processes developed in Chapter 2. In the safety model we focus on specifying when a process is capable of receiving inputs and when a process is capable of producing outputs.

Another instantiation of the general ECF model is the progress model presented in Chapter 4. In the progress model we can specify when a component guarantees progress by providing an output. We can also specify when a component demands progress from its environment by demanding an input. Figure 1.4 shows a specification of the JOIN that contains progress information. We denote progress information by attaching labels to states: label $\square$ marks quiescent states where no progress is either required or demanded; label Δ marks "demanding" states where the component demands progress from its environment; label ∇ marks transient states where the component guarantees to make progress by producing an output. The symbolism behind these labels can be

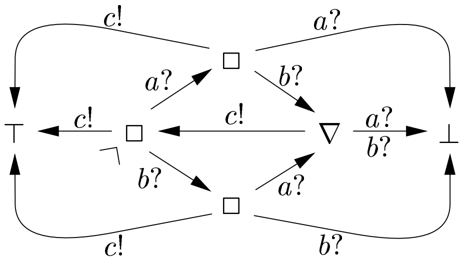


Figure 1.4: The JOIN with progress properties

memorized as follows. The transient triangle ∇ will eventually topple. The $\underline{\text{d}}$emanding Δ ($\underline{\text{d}}$elta) will not. The indifferent $\Box$ has a flat base like Δ, but is upside-down symmetric.

For example, in the initial state, the JOIN makes no progress guarantees nor does it demand an input from its environment. For that reason, the initial state is labeled with $\Box$. After having received both inputs, the JOIN is in a state labeled with ∇. In that state the JOIN guarantees to produce an output on port c.

The graph from Figure 1.4 contains the $\perp$ and the $\top$ state. The meanings of the $\perp$ state and the $\top$ state are the same as in the safety model illustrated in Figure 1.3. Thus, the progress model also addresses safety characteristics of a process.

We define a number of operations on ECF processes. One such operation is *network composition*, which we use to model networks of processes. Take, for example, a network consisting of the JOIN from Figure 1.4 and the WIRE from Figure 1.5. The WIRE is a component that can receive an input and then it guarantees to produce an output. After that, the behavior repeats. Figure 1.6 shows a state graph of a process that represents a network of the JOIN and the WIRE. All missing output transitions lead to the $\top$ state, which is omitted in order to reduce the clutter. The process is a result of the network composition of the JOIN and the WIRE. The result of the network composition contains information about the failures that can occur in the network. For example, if the environment provides the second input on port a before the JOIN has produced an

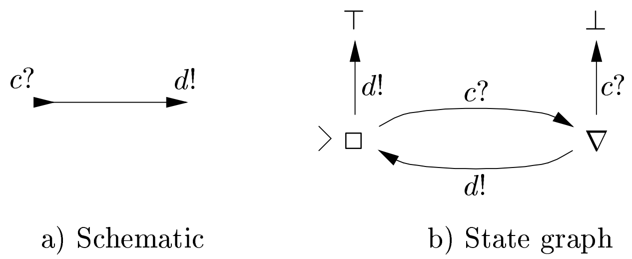


Figure 1.5: The WIRE

output, the network fails, which indicates a failure of the JOIN. Furthermore, suppose that the JOIN produces the second output on port c before the WIRE has produced an output on port d. In that case the network fails, indicating that the JOIN has produced an output when the WIRE was not able to receive an input. That is, failures of the network can represent safety problems that are internal to the network.

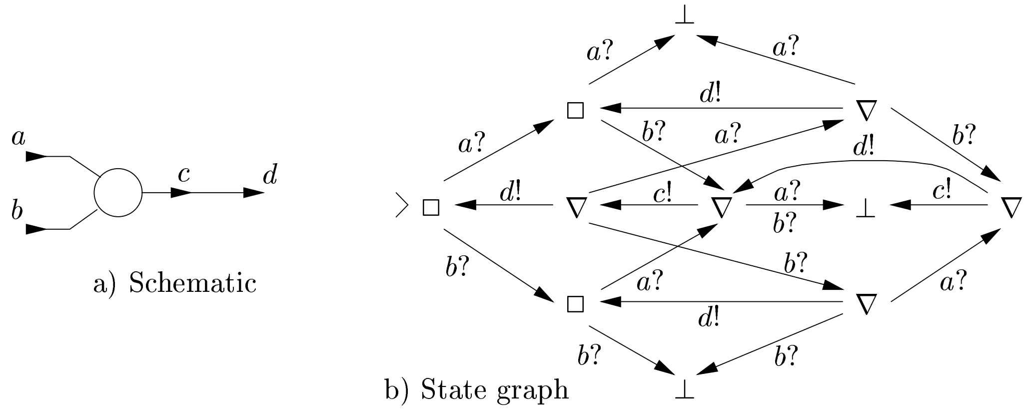


Figure 1.6: The network of the JOIN and the WIRE

After the network from Figure 1.6 has received inputs on ports a and b, it guarantees progress towards producing an output on port d. That is, the network is in a ∇ state. First, an output on port c is produced, indicating the communication between the JOIN and the WIRE. The network is now in another state labeled with ∇, from which an output on port d is guaranteed. Our network composition does not conceal connections between components within the network. Rather, the connections appear as output ports of a network.

The refinement relation on ECF processes determines whether an implementation satisfies a specification. Both the specification and the implementation are expressed as ECF processes. The refinement relation allows us to compare only processes that have the same input ports and the same output ports. If an implementation is a network of components, we apply an operation called *hiding* in order to conceal internal connections between components in the network. Figure 1.7 shows the result of hiding port c from the network of Figure 1.6. Port c represents the connection between the JOIN and the WIRE.

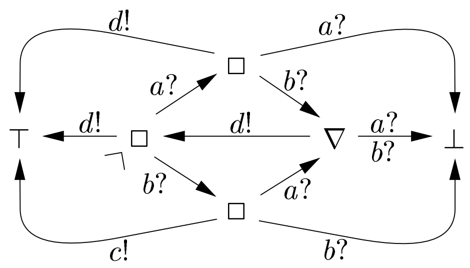
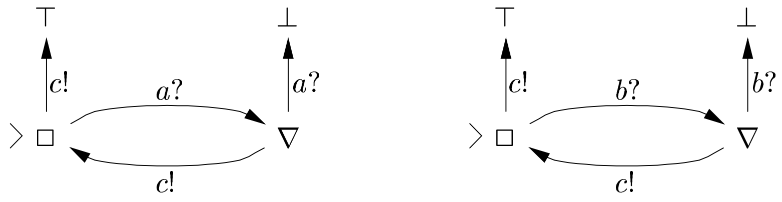


Figure 1.7: The network of the JOIN and the WIRE after hiding output c

Hiding takes into account progress and safety properties of a process. For example, the process of Figure 1.6 guarantees producing an output on port d after having received inputs on ports a and b. Similarly, the result of hiding port c, shown in Figure 1.7 is in a ∇ state after inputs on ports a and b have been received, guaranteeing that an output on port d will be produced.

Besides network composition, we also define a *specification composition* for ECF processes. Our application of the specification composition is as follows: We look at the behavior of a process in terms of constraints and we express each constraint as a so-called snippet process. The specification composition allows us to combine the snippets into



a) A snippet for a and c b) A snippet for b and c

Figure 1.8: Two snippets for the JOIN

a specification of a process. For example, the behavior of the JOIN can be seen as a "conjunction" of two constraints. The first constraint, shown in Figure 1.8a, states that inputs on port a and outputs on port c must alternate. Furthermore, after receiving an input on port a, the JOIN will produce an output on port c. The second constraint, shown in Figure 1.8b, takes into account ports b and c. The constraint states that inputs on port b and outputs on port c alternate and that, after having received an input on port b, the JOIN will produce an output on port c. The specification composition of the two constraints results in the specification of the JOIN shown in Figure 1.4. In Chapter 6 we propose a design method that one can apply to specifications expressed as specification compositions of constraints.

1.3 Related work

In this section we survey previous work that influenced most of the work leading to the ECF model proposed in this thesis.

A number of formalisms have been applied to asynchronous circuit design. Our ECF model belongs to the category of trace theory, introduced by Jan van de Snepscheut in [van de Snepscheut, 1985]. Snepscheut discusses how trace structures can be applied to modeling asynchronous circuits and stays short of defining a refinement relation, which would link a specification of a circuit to its implementation. Snepscheut, however, does define formally the concept of transmission interference, which is a part of safety correctness criteria in many of the later models, including our safety model from Chapter 3. Snepscheut defines three composition operators on trace structures, the weave, the blending, and the agglutination. All three operators were intended to represent the joint operation of two "mechanisms" described by trace structures. The only difference between the weave and the blending is that in blending the connections between devices are hidden. Agglutination, on the other hand, is a much more complicated operator that assumes arbitrary delays in connections between components. Snepscheut's weave is captured by the process product in the safety model in Chapter 3. Progress issues are not addressed in [van de Snepscheut, 1985].

Jan Tijmen Udding used trace theory in order to state so-called JTU-rules that characterize delay-insensitive circuits [Udding, 1984]. A refinement relation is not defined. Udding, however, does give a formal definition of the absence of computation interference. Together with the absence of transmission interference, the absence of computation interference forms the basis for our safety condition. Udding also proves that the network consisting of process P, its reflection $\sim P$, and the connection wires between the two is free of transmission and computation interference, if P is a so-called DI process.

This proof can be related to the work of Jo Ebergen, who proposed that the reflection of a specification can be used in testing whether an implementation satisfies a given specification [Ebergen, 1989, Ebergen, 1991]. That is, implementation I satisfies specification S if the network consisting of I and $\sim S$ satisfies some correctness criteria. The reflection of the specification acts as the environment in which the implementation must operate correctly. We show that the refinement relations in the safety model of Chapter 3 and in the progress model of Chapter 4 both can be expressed in terms of correctness of the network consisting of the implementation and the reflection of the specification. While Udding concentrated on delay-insensitive circuits, the speed-independence is fundamental to Ebergen's model. Ebergen showed that delay-insensitive circuits can be characterized with a speed-independent model by inserting explicit WIREs in connections between circuit components. Such an approach has an advantage, because it allows for a composition operator that is simpler than Snepscheut's agglutination. Ebergen does not define an explicit composition operator on processes, but he has a refinement relation called "decomposition". In the definition of decomposition, the weave is used implicitly for computing the joint behavior of a network of processes.

Among the correctness criteria for Ebergen's decomposition is "the completeness of network behaviors", which is intended as a progress condition. Namely, Ebergen requires that any implementation must be capable of exhibiting all sequences of communication actions that the specification can exhibit. [Peeters, 1990] shows that this progress condition allows some undesirable implementations; for example, deadlock is not captured by the completeness of network behaviors. Ebergen addressed the need for a better progress condition in his verification program VERDECT [Ebergen and Berks, 1995], where two additional progress conditions were implemented. One condition detects the presence of unbounded sequences of internal communication actions between components within an implementation, which may be an indication of a livelock. The other condition, called "the absence of illegal stops" checks whether an implementation can produce an output at any point where the specification can produce an output. These two conditions were described only informally and their properties have not been studied. The output-persistent refinement relation that we define in Chapter 5 captures the absence of illegal stops.

Ebergen recognized that the weave can be used not only for composing processes that represent circuit components, but also for combining behavioral constraints into a specification of one component. Such a constraint-oriented specification approach can be used for producing simple specifications of behaviors that involve a significant degree of concurrency. See [Ebergen, 1992], [Ebergen et al., 1993], and [Benko and Ebergen, 1994] for examples of such specifications. The Separation Theorem from [Ebergen, 1991] captures an elegant design technique that applies to specifications given as a weave of constraints. This design technique is illustrated in [Ebergen, 1992], [Ebergen et al., 1993], and [Benko, 1993]. In this thesis we pursue the idea of constructing specifications from behavioral constraints. We show that, when progress concerns are taken into account, the operand for combining behavioral constraints differs from the operand for computing the behavior of a network of independent components. In Chapter 6 we propose a "partwise" design method that is similar to an application of Ebergen's Separation Theorem, but now progress concerns are also taken into account.

David Dill's prefix-closed trace structures and complete trace structures [Dill, 1989] introduced a distinction between success traces and failure traces. Success traces correspond to legal traces in the ECF model, and failure traces corresponds to traces labeled with $\perp$. Dill introduces a composition operator on trace structures, but does not address the question of combining behavioral constraints into one specification. Dill also defines explicitly the hiding operation, used for hiding output symbols of a component. Hiding is based on the projection defined in [van de Snepscheut, 1985]. A result of Dill's hiding can be a process for which some trace is both a success trace and a failure trace. Such an ambiguity models situations where both success and failure are possible, depending on unknown factors. In our hiding operation we take a more pessimistic approach, maintaining the distinction between failure traces and success traces.

[Dill, 1989] introduces two separate models: Prefix-closed trace structures address the question of safety. Complete trace structures, intended for studying progress properties, are not prefix-closed and involve infinite traces. Infinite traces and the lack of prefix-closedness make operations on complete trace structures fairly complicated. The progress properties that can be captured by complete trace structures include liveness, which we study in Chapter 4, and fairness, which, as argued in [Black, 1986], is beyond the power of finite-trace models such as ECF processes.

The main inspiration for ECF processes comes from DI and XDI processes proposed by Tom Verhoeff [Verhoeff, 1994, Verhoeff, 1998a]. One major difference between our ECF model and Verhoeff's DI and XDI models is that our ECF model describes speed independent circuits, while the DI model and the XDI model apply to delay-insensitive circuits only. The DI model is a sister model to our safety model from Chapter 3 and addresses safety issues only. The XDI model, on the other hand, corresponds to our progress model from Chapter 4 and deals with safety and progress concerns.

[Verhoeff, 1994] introduced Enhanced Characteristic Functions (ECFs) that are used to attach labels to traces and has recognized that "ECFs are a neat way of describing processes and their cooperation in systems". On the other hand, Verhoeff did not base the DI and the XDI model on ECFs, because "the tediousness and sheer number of details concerning ECFs was disappointing". We find our ECF-based model that applies to speed-independent circuits to be quite simple. In our opinion, the source of complexity referred to by Verhoeff was delay-insensitivity. In particular, defining the process composition in Verhoeff's models was complicated: Verhoeff gives two definitions of the composition. In the first definition, the composition is defined indirectly by using a correctness condition. In the second definition, the process composition is defined as a fix-point of a sequence of transformations. In our ECF model, on the other hand, we were able to define process composition, or process product, as we call it, by simply lifting the product on labels to ECFs. Also, Verhoeff does not address the question of combining constraints into one specification. The only composition operator defined in [Verhoeff, 1994] corresponds to our network composition.

Verhoeff's composition hides connections between components. One consequence of that decision is that in DI and XDI processes we can have only point-to-point connections. In ECF processes, the composition does not hide connections between processes. Instead, we can hide connections by applying a separate hiding operation. In our opinion, separating hiding from process composition contributes to the simplicity of the model.

The testing paradigm [Nicola and Hennessy, 1983] is the basis for Verhoeff's refinement relation. That is, an implementation satisfies the specification if the implementation passes all the tests that are passed by the specification. Our refinement relation, on the other hand, is based on comparison of labels attached to traces. We prove that the refinement relations in the safety model from Chapter 3 and the progress model from Chapter 4 can also be characterized by means of the testing paradigm.

Willem Mallon extended the XDI model by allowing so-called unhealthy processes [Mallon et al., 1999]. In an unhealthy process, some trace indicates that a process guarantees progress after that trace. The process itself, on the other hand, is not capable of producing an output. Unhealthiness indicates a problem with the process specification; thus, Verhoeff excluded unhealthy processes from his XDI model. Mallon, on the other hand, argues that allowing unhealthy processes simplifies the model and leads to a number of desirable formal properties. Consequently, the X^2DI model presented in [Mallon et al., 1999] includes unhealthy processes. We agree with Mallon that unhealthy processes lead to a simpler formalism and we include unhealthy processes in the progress model of Chapter 4. The main focus of Mallon's work is a design technique based on the so-called Factorization Theorem. We prove that the Factorization Theorem also holds for ECF processes. Mallon does not address the distinction between the specification composition and the network composition.

1.3.1 Other formalisms

Besides different versions of trace theory, there are a number of other formalisms that relate to ECF processes. Here we discuss some of these models.

Mark Josephs introduced a model called "receptive process theory". The name "receptive" was borrowed from [Dill, 1989] and means that a process has no control over when its environment will provide an input. Receptive processes rework Hoare's theory of Communicating Sequential Processes [Hoare, 1985] under the assumption of receptiveness. ECF processes are also receptive.

Receptive processes capture progress concerns by using failure traces in order to express progress properties of a process: After a failure trace, a process cannot produce an output. This approach is very similar to the treatment of progress in Dill's complete trace structures. Receptive processes, however, have no means of expressing progress conditions for the environment of a process. Such conditions can be expressed in the ECF progress model. A network composition is defined for receptive processes, but the question of assembling specifications by combining behavioral constraints is not addressed. [Josephs, 1992] hints that a refinement relation could be defined for receptive processes by comparing failure sets of a specification and an implementation, but stays short of an explicit definition of refinement. Rather, the examples in [Josephs, 1992] require that the specification and the composition of components comprising an implementation are the same.

Milner's Calculus of Communicating Systems (CCS) [Milner, 1989] is another formalism that was applied to modeling asynchronous circuits. [Stevens et al., 1993] demonstrates how to express specifications of asynchronous circuits in terms of CCS. CCS specifications can be constructed as parallel compositions of CCS agents. The technique of composing agents is different from composing constraints with the weave, because the semantics of CCS parallel composition implies communication between agents. Thus, an agent does not represent a constraint on a behavior of a process. Rather, we can see agents as a means of breaking a specification in a hierarchical manner into smaller objects.

CCS itself does not address progress properties of processes. [Liu et al., 1993] and [Stevens, 1994] demonstrate how progress properties can be addressed by coupling CCS with the modal-μ calculus. More precisely, [Liu et al., 1993] and [Stevens, 1994] show how to construct tests for the absence of deadlock, fairness, safety and liveness. These tests are expressed as fixed points of μ-calculus formulae and their formulation can be quite involved. Fortunately, [Liu et al., 1993] and [Stevens, 1994] were able to take advantage of results from temporal logic, which made the tests easier to understand. A general purpose verification tool called Concurrency Workbench made it possible to automate progress-property tests on CCS agents.

The theory of labeled transition systems provides a number of equivalences on CCS agents. One can use these equivalences in order to compare a specification and its implementation. Ken Stevens [Stevens, 1994] argues that it is usually too tight a requirement that a specification and implementation have equivalent behaviors. Instead, Stevens proposes two refinement relations, called trace conformance and logic conformance. Trace conformance is similar to our output-persistent refinement defined in Chapter 5: Both relations require a match between outputs that can be produced by a specification and outputs that can be produced by an implementation. Unfortunately, trace conformance does not detect deadlocks. Stevens incorporated a number of aspects of bisimulation into a stronger relation, called logic conformance, which does detect deadlocks. The extensions of CCS developed in [Stevens, 1994] were incorporated into a verification tool called Analyze.

Radu Negulescu proposed process spaces [Negulescu, 1998], a very general formalism that can be applied to studying safety and progress properties of asynchronous circuits. Process spaces are built on top of a so-called set of abstract executions. A trace set is an example of a set of executions. The executions are split into three sets: goals represent a desirable course of actions; rejects are executions where the environment behaves improperly, violating behavioral requirements of a process; escapes are executions where a process behaves improperly, violating the requirements of its environment.

In process spaces, each property requires a separate specification. That is, if we were to study safety and progress properties of a circuit, we would produce two specifications, one for safety and one for progress properties. In our ECF model, we cover both safety and progress with one specification.

Process spaces have a composition operator called process product. The process product is similar to the product that we define for ECF processes and can be used as a network composition or as a specification composition when only safety issues are a concern [Negulescu and Peeters, 1998]. We show that, when progress issues are a concern, the process product cannot be used as a specification composition. A refinement relation is defined for process spaces and it was shown that the refinement relation conforms to the testing paradigm. [Negulescu, 1998] mentions a possibility of defining a hiding operation for process spaces, but does not give an explicit definition.

The safety condition on process spaces is expressed as so called robustness of a process. As opposed to the absence of computation interference, robustness does not require that the process has no input terminals. Rather, a robust process must be capable at any time of receiving an input on any of its input ports. A robust process never fails by itself.

The safety condition for ECF processes also does not require that a process has no input terminals and the condition demands that a process cannot fail by itself. On the other hand, our safety condition does not require that a process must always be able to receive an input on any of its input ports.

Negulescu informally introduces the concept of finalization, which captures the same progress condition as our refinement relation in the progress model of Chapter 4. While finalization uses finite traces, the concepts of liveness and progress in process spaces involve infinite traces. Liveness guarantees that desired events are not postponed forever and progress in process spaces guarantees that desired events are not postponed for an unbounded time. Negulescu's liveness and progress capture correctness conditions that are not addressed in ECF processes.

1.4 Contributions

The main contributions of this thesis are

- The abstract model of ECF processes is new. ECF processes rely on a number of simple properties that must be satisfied by some unknown set of labels. These properties are then lifted to ECF processes, which results in short and simple proofs of properties of ECF processes.

  ECF processes provide a unified framework that we can apply to studying various properties of processes and of networks of processes. In order to apply ECF processes, we instantiate the abstract ECF model with an appropriate set of labels. This gives us a "concrete" model where all the properties proven for ECF processes hold. In the thesis we discuss two instantiations of ECF processes: a model where we address the safety of processes and a model where we address progress of processes.

ECF processes model so-called isochronic communication. That is, communication between processes is assumed to be instantaneous. Isochrony allows us to lift the product on labels to the product on processes in a straightforward manner. The result is a simple definition of process product. In the instantiated models, the process product serves as the network composition for processes.

- The definition of hiding is new. We demonstrate that hiding applied to processes in the safety model takes into account safety properties of a process. In the progress model, hiding also takes into account progress properties of a process. This distinguishes hiding from a simple projection operation used in "standard" trace theory, where progress and safety properties are not taken into account. Hiding enjoys a number of desirable formal properties. Most notably, hiding is monotonic with respect to the refinement relation and hiding distributes over process product.

- One instantiation of ECF processes is the safety model of Chapter 3, where we illustrate operations on ECF processes. The safety correctness condition presented in Chapter 3 follows the same approach as in previous work with a minor change in the formulation. Namely, our safety condition can be applied to any process in contrast to the absence of computation interference, which is typically defined only for a closed network. Furthermore, we do not require that a safe process withstand misuse by the environment. This makes our safety condition less demanding than robustness from [Negulescu, 1998].

  We demonstrate that the process product defined for ECF processes in the safety model can be used for modeling networks of processes as well as for combining process snippets expressing behavioral constraints into a specification of one component.

- Another instantiation of ECF processes is the progress model of Chapter 4. Our progress condition is the same as the progress condition in XDI model proposed in [Verhoeff, 1994]. This means that we can detect deadlocks, but we do not address questions of fairness and starvation.

  We demonstrate that the process product can be used for modeling networks of processes in the progress model. On the other hand, the process product is *not* suitable for producing specifications by combining process snippets expressing behavioral constraints. This is a new observation, revealing that, when progress properties are a concern, there is a formal distinction between the network composition and the specification composition. In previous work as well as in our safety model, no distinction has been made between the network and the specification composition.

- The observation that the progress model demands a distinction between the network composition and the specification composition led us to defining a domain of *snippets*. Snippets are restricted processes that we use for expressing behavioral constraints. We define the specification composition on snippets, which we use for combining snippets into a specification of one process. We give a number of examples that illustrate how to specify circuit components in terms of behavioral constraints expressed by snippets.

- We propose a part-wise design method that can be applied to specifications constructed as a specification composition of snippets. The design method allows us to find in isolation an implementation for each snippet. We then combine these partial implementations into an implementation of the original specification. This design method may help us avoid a state explosion in verification of implementations that involve a large degree of concurrency. Instead of verifying the complete implementation, we only have to verify implementations of individual snippets. Because snippets tend to be small sequential processes, verifying their implementations is usually not a difficult task. The rest of the work is then done by applying a number of theorems, which leads to the final implementation. We illustrate this design approach on a number of examples.

- The refinement relation on ECF processes is defined in a simple manner by comparing labels of traces. For the safety and for the progress model we prove that the refinement relation can be characterized by applying a correctness condition to a network consisting of an "implementation" and the reflection of the "specification". Furthermore, we prove that the refinement relation can also be expressed in terms of the testing paradigm. That is, process I implements specification S if I passes all the tests that are passed by the specification. This means that our refinement relation has three equivalent characterizations.

- The testing paradigm allowed us to prove the Factorization Theorem in both the safety and the progress model. The Factorization Theorem provides a bound on a solution of the Design Equation [Verhoeff, 1994, Mallon et al., 1999], which is the cornerstone of a design method advocated in [Mallon et al., 1999]. Consequently, the design method based on solving the Design Equation is also applicable to ECF processes.

1.5 Road map

In Chapter 2 we propose the general model of ECF processes. We define the refinement relation, reflection, product and hiding, and we prove a number of properties that hold for these operations. Finally, we prove the Substitution Theorem that forms the basis for hierarchical verification and design.

In Chapter 3 we present a safety model, which is an instantiation of the general ECF model. We apply the safety model to studying safety properties of processes and networks of processes.

In Chapter 4 we present a progress model, which is another instantiation of the general ECF model. In the progress model we address progress properties of processes and of networks of processes. We illustrate how progress properties are reflected in all the operations on processes in the progress model.

In Chapter 5 we discuss the specification composition in the safety model and in the progress model. In progress model, we define the domain of snippets, which are processes that satisfy a number of conditions. We define the network composition for processes and the specification composition for snippets.

In Chapter 6 we propose a part-wise design method, which applies to specifications expressed as a specification composition of constraints. We illustrate the part-wise design method on a number of examples. We also demonstrate a step-wise design method, based on the application of the Substitution Theorem.

In Appendix B we briefly introduce the theory of ordered sets and lattices. Appendix C contains proofs of the lemmas used in the main body of the thesis.

1.6 Notation

Function application is denoted with an infix dot: $f.x$ is the image of x under application of f. We use a slightly unconventional notation for expressing quantifications. Universal quantification is denoted by

$$(\forall L : D : E)$$

where L is a list of bound variables, D is the domain predicate, and E is the quantified expression. Typically, D and E contain variables from L. For example, the following expression states that function f is monotonic:

$$(\forall x, y : x \leq y : f.x \leq f.y)$$

Existential quantification is denoted by $\exists$ and is expressed in the same format as the universal quantification. We write

$$\{L : D : E\}$$

to denote the set of all values E obtained by substituting values that satisfy predicate D for the variables in L. For example, if $\mathcal{N}$ denotes the set of natural numbers, then the set of all squares of natural numbers is denoted by

$$\{n : n \in \mathcal{N} : n^2\}$$

. Most proofs in the thesis have a special layout. For example, when we prove $P \Rightarrow R$ by first showing that $P \Rightarrow Q$ and then $Q \Leftrightarrow R$, we write

$$
\begin{array}{ll}
 & P \\
\Rightarrow & \{ \text{ hint why } P \Rightarrow Q \ \} \\
 & Q \\
\Leftrightarrow & \{ \text{ hint why } Q \Leftrightarrow R \ \} \\
 & R
\end{array}
$$

Chapter 2

ECF processes

In "standard" trace theory, such as the one of [van de Snepscheut, 1985] or [Ebergen, 1989], we specify the behavior of a device by listing all allowed sequences of interactions between the device and its environment. Such a list of sequences is called a trace set. Traditionally, traces that are not present in a trace set are forbidden and are considered to represent invalid sequences of communication actions. The presence of a trace in the trace set can be determined by a characteristic function, which distinguishes between valid and invalid sequences of communication actions. [Verhoeff, 1994] proposed a model in which a characteristic function can tell us more than whether some sequence of communication actions is valid or not. Such an *enhanced characteristic function* (ECF) assigns a label to a trace, and the label indicates some property of the trace. For example, the label can specify the safety status of a process after that trace has taken place. More precisely, a label can tell us whether a safety failure has occurred. In another application, an ECF can specify the progress guarantees and demands of a process after exhibiting the trace.

In this chapter we develop an abstract model where a process is defined by an input and an output alphabet and by an ECF. The ECF maps traces over the input and output alphabet to a set of labels Λ. We assume that Λ satisfies a number of conditions. On the basis of the properties of Λ, we define a refinement relation and a number of operations on processes. We emphasize that the properties of the operations and of the refinement relation on processes hold regardless of the particular choice of Λ, as long as Λ satisfies the basic requirements. Consequently, the abstract ECF model forms a framework for a number of different models that could be geared towards studying specific properties of processes and their networks.

In this chapter we refrain from discussing applications of the abstract model. Rather, we do the groundwork for subsequent chapters, where we pick two concrete sets of labels in order to study safety and progress properties of processes.

2.1 Labels

We require that a set of labels satisfies a number of conditions:

Definition 2.1.1 (Labels)
A set of symbols Λ is called a set of labels *if it satisfies the following conditions:*

1. *A partial order $\sqsubseteq$ is defined on Λ, and $(\Lambda, \sqsubseteq)$ is a complete lattice with least element $\bot$ and greatest element $\top$.*

2. *A product, denoted by $\times$, is defined on Λ, such that Λ is closed under product. The product is associative, commutative, idempotent, and monotonic with respect to the partial order $\sqsubseteq$. For labels λ and γ, we have*

$$\lambda \times \gamma \in \{\top, \bot\} \quad \Leftrightarrow \quad \lambda \in \{\top, \bot\} \ \vee \ \gamma \in \{\top, \bot\} \tag{2.1}$$

3. *A reflection* $\sim\lambda$ *is defined for each* $\lambda \in \Lambda$. *The reflection is its own inverse and it reverses the partial order:*

$$\lambda \sqsubseteq \gamma \quad \Rightarrow \quad \sim\gamma \sqsubseteq \sim\lambda \tag{2.2}$$

$$\sim(\sim\lambda) \quad = \quad \lambda \tag{2.3}$$

We use lower-case Greek letters to refer to members of Λ.

Equations 2.2 and 2.3 imply that $\sim\bot = \top$ and $\sim\top = \bot$. The proof goes as follows: Take any label λ. We know that $\bot \sqsubseteq \sim\lambda$. By Equations 2.2 and 2.3 we get for any label λ that $\lambda \sqsubseteq \sim\bot$. We conclude that $\sim\bot = \top$, because $\top$ is the greatest element in Λ. Furthermore, by Equation 2.3 we have $\sim\top = \bot$.

Labels: $\Lambda = \{\bot, \diamondsuit, \heartsuit, \top\}$

Partial order: $\bot \sqsubseteq \diamondsuit \sqsubseteq \heartsuit \sqsubseteq \top$

Product:

$\times$	$\bot$	$\diamondsuit$	$\heartsuit$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\diamondsuit$	$\bot$	$\diamondsuit$	$\diamondsuit$	$\top$
$\heartsuit$	$\bot$	$\diamondsuit$	$\heartsuit$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

Reflection:

λ	$\bot$	$\diamondsuit$	$\heartsuit$	$\top$
$\sim\lambda$	$\top$	$\heartsuit$	$\diamondsuit$	$\bot$

Figure 2.1: An example of a set of labels

Figure 2.1 shows an example of a set of labels, and a partial order, a product, and a reflection defined for that set of labels. One can verify by inspection that the set Λ from Figure 2.1 satisfies the requirements from Definition 2.1.1. We have no intuitive interpretation for the set of labels used in this chapter. On the other hand, we do provide intuitive explanations for labels used in Chapters 3 and 4.

2.2 Processes

A process P is a triple $\langle \mathbf{i}.P, \mathbf{o}.P, \mathbf{f}.P \rangle$, where $\mathbf{i}.P$ is the input alphabet of P, $\mathbf{o}.P$ is the output alphabet of P, and $\mathbf{f}.P$ is the enhanced characteristic function for P. We stipulate that $\mathbf{i}.P$ and $\mathbf{o}.P$ are disjoint. The alphabet of process P, denoted by $\mathbf{a}.P$, is $\mathbf{i}.P \cup \mathbf{o}.P$. The function $\mathbf{f}.P$ assigns labels to traces: $\mathbf{f}.P : (\mathbf{a}.P)^* \to \Lambda$.

We assume that, once something bad happens to a process, it cannot recover. That is, once trace t is labeled with $\perp$, every extension tu of t is labeled with $\perp$. We call this property $\perp$ persistence:

$$\mathbf{f}.P.t = \perp \;\; \Rightarrow \;\; (\forall u : \; u \in (\mathbf{a}.P)^* : \; \mathbf{f}.P.tu = \perp) \tag{2.4}$$

Similarly, once trace t is labeled with $\top$, every extension tu of t is labeled with $\top$. This property is called $\top$ persistence.

$$\mathbf{f}.P.t = \top \;\; \Rightarrow \;\; (\forall u : \; (\mathbf{a}.P)^* : \; \mathbf{f}.P.tu = \top) \tag{2.5}$$

Definition 2.2.1 (Process)
A process P is a triple $P = \langle \mathbf{i}.P, \mathbf{o}.P, \mathbf{f}.P \rangle$, satisfying the following conditions:

1. $\mathbf{i}.P \cap \mathbf{o}.P = \emptyset$.

2. $\mathbf{f}.P$ *is* $\top$ *and* $\perp$ *persistent.*

The set $\mathbf{i}.P$ *is the input alphabet of* P*,* $\mathbf{o}.P$ *is the output alphabet of* P*, and* $\mathbf{f}.P : (\mathbf{a}.P)^* \to \Lambda$ *is the enhanced characteristic function.*

For process P, the set of *legal* traces, denoted by $\mathbf{l}.P$, is the set of all traces that are *not* labeled with either $\top$ or $\bot$:

$$\mathbf{l}.P = \{t \,:\, t \in (\mathbf{a}.P)^* \ \wedge\ \mathbf{f}.P.t \not\in \{\top, \bot\} \,:\, t\} \tag{2.6}$$

Traces that are not legal are called *illegal* traces. That is, illegal traces are labeled with either $\top$ or $\bot$. The name illegal suggest that we intend to use labels $\top$ and $\bot$ in order to represent undesired or erroneous events in the behavior of a device.

To each process one can naturally attach a directed state graph as follows: 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_i and label the state with $\mathbf{f}.P.a_i$. We also create a transition for each symbol a_i, such thet the transition leads from the initial state to the state corresponding to one-symbol trace a_i. To distinguish input symbols from output symbols, we attach the question mark to input symbols and the exclamation mark to output symbols. We continue this process inductively, creating an infinite graph where each trace in process P has its own distinct state. The state graph is deterministic from the automaton point of view.

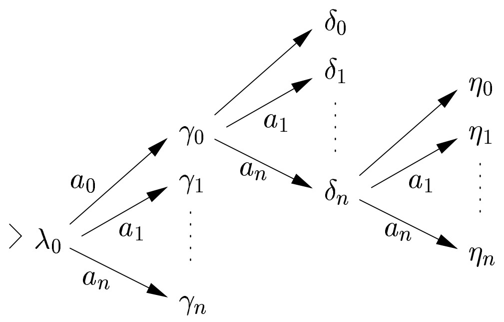


Figure 2.2: Creating a state graph for a process

Figure 2.2 illustrates how one can create a state graph for an arbitrary process P. Such a graph is infinite, because it contains a distinct state for each trace. A general 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 equivalenty because of $\perp$-persistence: All states in a subgraph stemming from a $\perp$ state are labeled with $\perp$. For that reason, all transitions leaving states labeled with $\top$ are self loops. A similar argument holds for $\top$ states. In order to reduce the clutter, we do not depict self loops on $\top$ and $\perp$ states when we depict processes with state graphs.

Processes that we use in our examples have simple, finite state graph representations. For example, the state graph in Figure 2.3 shows process P where traces are labeled with labels from Figure 2.1. The process is defined as follows: $\mathbf{i}.P = \{a\}$, $\mathbf{o}.P = \{b\}$, $\mathbf{f}.P.\varepsilon = \diamondsuit$, $\mathbf{f}.P.a = \heartsuit$, $\mathbf{f}.P.t = \perp$ for $t \in (b + ab)(a + b)^*$, and $\mathbf{f}.P.t = \top$ for $t \in aa(a + b)^*$.

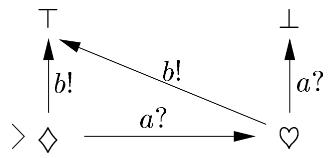


Figure 2.3: An example of a process represented by a state graph

What if the initial state (or the empty trace) is labeled with $\top$ or with $\perp$? Then we get two special processes. When the initial state of process P is labeled with $\top$, then $\mathbf{f}.P.\varepsilon = \top$, which means that all traces in this process are labeled with $\top$. We name this process $\mathsf{MIRACLE}(I, O)$, where I is the input alphabet and O is the output alphabet. The name MIRACLE is borrowed from [Verhoeff, 1994] and it suggests that MIRACLE is "the best possible" process, which acts as a miraculous panacea. If the initial state of process P is labeled with $\perp$, then $\mathbf{f}.P.\varepsilon = \perp$. That is, all traces in such a process are labeled with $\perp$. We call such a process $\mathsf{ABORT}(I, O)$, where I and O are, respectively, the input and the output alphabet of the process. The name ABORT comes from [Verhoeff, 1994] and it suggests that ABORT is "the worst possible" process.

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

2.3 Refinement and reflection

Recall that $\sqsubseteq$ denotes the partial order on labels. When we compare two processes, we compare labels of their traces. The processes refinement is defined for processes that have the same alphabets. The partial-order relation is called a refinement. The purpose of refinement is to determine whether one process implements another.

Definition 2.3.1 (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:

$$P \sqsubseteq Q \quad \equiv \quad \Big(\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\Big) \tag{2.7}$$

When $P \sqsubseteq Q$, we often refer to process P as the specification and to process Q as the implementation. That is, $P \sqsubseteq Q$ means that process Q is an implementation of a specification expressed by process P.

If processes P and Q are represented by state graphs, we can verify the refinement relation directly on the state graphs: First, we check whether the input and the output alphabets of the two processes are the same. Then we compute a direct product of two state graphs by the following procedure: The set of states in the direct product is the Cartesian product of the sets of states from graphs for processes P and Q. That is, if p is a state in the graph representing process P, and if q is the state in a graph representing process Q, then (p, q) is a state in the direct product of the two graphs. The initial state in the direct product is state (p_0, q_0), such that p_0 and q_0 are initial states in the state graphs for processes P and Q, respectively. A transition on port a from state (p, q) to

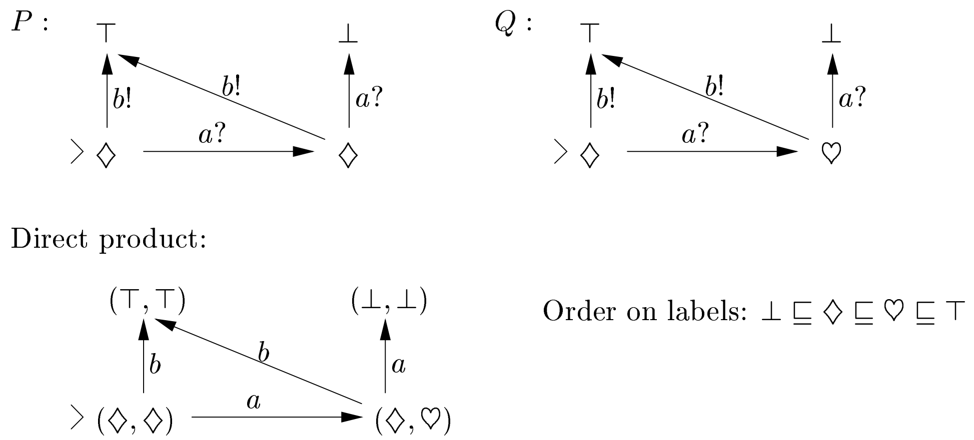


Figure 2.4: Verifying process refinement on state graphs

state (p', q') exists if there exists a transition of port a from state p to state p' in the state graph for process P and if there exists a transition from state q to state q' in the state graph for process Q. State (p, q) in the direct product has a pair of labels, (λ, γ), such that λ is the label of state p and γ is the label of state q. Once we have computed the direct product of the two state graphs, we can verify the refinement: For each reachable state (p, q) in the direct product we check whether the label of state p in the graph for process P is smaller than the label of state q in the graph for process Q.

Figure 2.4 shows state graphs for two processes, P and Q, and reachable states of the direct product of the two graphs. Traces in processes P and Q are labeled with labels from Figure 2.1. We can see that $\lambda \sqsubseteq \gamma$ for each pair of labels (λ, γ) in the reachable states of the direct product. Consequently, $P \sqsubseteq Q$.

Notice that in the refinement relation we simply compare labels of identical traces. We know that the refinement on labels is a partial order. As a consequence, we have

Theorem 2.3.2 (Partial order)

Refinement $\sqsubseteq$ *is a partial order on* $\mathcal{PROC}(I, O)$.

Proof: A relation is a partial order when it is reflexive, antisymmetric, and transitive. It is easy to see that $\sqsubseteq$ is reflexive:

$$\Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \sqsubseteq \mathbf{f}.P.t\Big) \;\;\Rightarrow\;\; P \sqsubseteq P$$

Assume that P and Q are processes where $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P = \mathbf{o}.Q$. Antisymmetry, $P \sqsubseteq Q \,\wedge\, Q \sqsubseteq P \;\;\Rightarrow\;\; P = Q$, can be seen as follows.

$$\begin{array}{ll}
 & P \sqsubseteq Q \;\;\wedge\;\; Q \sqsubseteq P \\
\equiv & \quad \{\text{ Definition 2.3.1 }\} \\
 & \Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\Big) \;\;\wedge\;\; \Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.Q.t \sqsubseteq \mathbf{f}.P.t\Big) \\
= & \quad \{\; \mathbf{a}.P = \mathbf{a}.Q, \text{ calculus } \} \\
 & \Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t \;\;\wedge\;\; \mathbf{f}.Q.t \sqsubseteq \mathbf{f}.P.t\Big) \\
\Rightarrow & \quad \{ \sqsubseteq \text{ is antisymmetric for labels } \} \\
 & \Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t = \mathbf{f}.Q.t\Big) \\
\Rightarrow & \quad \{\; \mathbf{i}.P = \mathbf{i}.Q,\; \mathbf{o}.P = \mathbf{o}.Q, \text{ and } \mathbf{f}.P = \mathbf{f}.Q \;\} \\
 & P = Q
\end{array}$$

Finally, we prove that $\sqsubseteq$ is transitive:

$$\begin{array}{ll}
 & P \sqsubseteq Q \;\;\wedge\;\; Q \sqsubseteq R \\
\equiv & \quad \{\text{ Definition 2.3.1 }\} \\
 & \Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\Big) \;\;\wedge\;\; \Big(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.Q.t \sqsubseteq \mathbf{f}.R.t\Big) \\
= & \quad \{\text{ Calculus, } \mathbf{a}.P = \mathbf{a}.Q \;\} \\
 & \Big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t \;\;\wedge\;\; \mathbf{f}.Q.t \sqsubseteq \mathbf{f}.R.t\Big)
\end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{ \sqsubseteq \text{ is transitive for labels } \} \\
& \left(\forall t:\ t \in (\mathbf{a}.P)^*:\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.R.t\right) \\
\Rightarrow & \{ \mathbf{a}.P = \mathbf{a}.R, \text{ Definition 2.3.1 } \} \\
& P \sqsubseteq R
\end{array}
$$

□

Recall that $(\Lambda, \sqsubseteq)$ is a complete lattice. Because the refinement relation on processes simply lifts the partial order from labels to sets of traces, we can prove the following:

Theorem 2.3.3 (Complete lattice)
$(\mathcal{PROC}(I,O), \sqsubseteq)$ *is a complete lattice.*

Proof: In this proof we rely on the following property of complete lattices:

Let A be a non-empty set and let $\sqsubseteq$ be a partial order on A. $(A, \sqsubseteq)$ is a complete lattice if and only if $(P, \sqsubseteq)$ has a maximum element and the greatest lower bound $\sqcap B$ exists for every non-empty subset B of A.

See Theorem 2.16 in [Davey and Priestley, 1990] for the proof of the property above.

Since $\mathcal{PROC}(I,O)$ has a maximum element $\mathsf{MIRACLE}(I,O)$, we have to prove that the greatest lower bound exists for any $S \subseteq \mathcal{PROC}(I,O)$. Let P be the following system:

$$
\begin{array}{rcl}
\mathbf{i}.P & = & I \\
\mathbf{o}.P & = & O \\
\mathbf{f}.P.t & = & (\sqcap R:\ R \in S:\ \mathbf{f}.R.t)
\end{array}
$$

P is the greatest lower bound on S. We still have to prove that P is a process. More precisely, we need to prove that P satisfies Equations 2.5 and 2.4, which means that P is $\top$ and $\bot$ persistent.

Take trace t such that $\mathbf{f}.P.t = \top$. Then, for all processes $R \in S$, we have $\mathbf{f}.R.t = \top$. Furthermore, for any trace u we have, for all processes $R \in S$, $\mathbf{f}.R.tu = \top$, because each process R is $\top$-persistent. Consequently, for any trace u, we have $\mathbf{f}.P.tu = \top$. That is, P satisfies Equation 2.5.

Now take trace t such that $\mathbf{f}.P.t = \bot$. Then there exists a process $R \in S$ such that $\mathbf{f}.R.t = \bot$. Furthermore, for any trace u, we have $\mathbf{f}.R.tu = \bot$, because process R is $\bot$-persistent. Consequently, for any trace u, we have $\mathbf{f}.P.tu = \bot$. Equation 2.4 is satisfied by P, hence P is a process. □

In our definition of refinement, we say that $P \sqsubseteq Q$ if the label of each trace in process P is at most the label of the same trace in process Q. The following theorem allows us to narrow the set of traces for which we have to compare labels in order to determine whether one process refines another. The theorem states that it is sufficient to compare the labels of the the empty trace and of all one-symbol extensions of legal traces of process P.

Theorem 2.3.4

For processes P and Q, where $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P = \mathbf{o}.Q$, the following holds:

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \big(\forall t :\ t \in (\mathbf{l}.P)(\mathbf{a}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\big) \tag{2.8}$$

Proof: We first recall the definition of refinement:

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

Obviously, we have

$$P \sqsubseteq Q \quad \Rightarrow \quad \Big(\forall t :\ t \in (\mathbf{l}.P)(\mathbf{a}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\Big)$$

Next we assume

$$\big(\forall t :\ t \in (\mathbf{l}.P)(\mathbf{a}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\,\big) \tag{2.9}$$

and we prove that $P \sqsubseteq Q$ holds.

First take $\mathbf{l}.P = \emptyset$. Then, either $\mathbf{f}.P.\varepsilon = \bot$, or $\mathbf{f}.P.\varepsilon = \top$. If $\mathbf{f}.P.\varepsilon = \bot$, then, by $\bot$-persistence, $\mathbf{f}.P.t = \bot$ for every $t \in (\mathbf{a}.P)^*$. Because $\bot$ is the least label, $\mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t$. If $\mathbf{f}.P.\varepsilon = \top$, then $\mathbf{f}.Q.\varepsilon = \top$, because $\mathbf{f}.P.\varepsilon \sqsubseteq \mathbf{f}.Q.\varepsilon$ by Equation 2.9, and because $\top$ is the greatest label. By $\top$-persistence, we have $\mathbf{f}.P.t = \mathbf{f}.Q.t = \top$ for every $t \in (\mathbf{a}.P)^*$. Hence, $\mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t$.

Now assume that $\mathbf{l}.P \neq \emptyset$. This means that $\varepsilon \in \mathbf{l}.P$. Furthermore, by Equation 2.9 we can see that $\mathbf{f}.P.a \sqsubseteq \mathbf{f}.Q.a$ for every $a \in \mathbf{a}.P$. Consequently, Equation 2.9 implies

$$\mathbf{f}.P.\varepsilon \sqsubseteq \mathbf{f}.Q.\varepsilon\ \wedge\ \big(\forall t, a :\ t \in \mathbf{l}.P \wedge a \in \mathbf{a}.P :\ \mathbf{f}.P.ta \sqsubseteq \mathbf{f}.Q.ta\big) \tag{2.10}$$

Notice that in Equation 2.10 we compare all traces in $(\mathbf{a}.P)^*$ other than traces in $((\mathbf{l}.P)(\mathbf{a}.P) - \mathbf{l}.P)(\mathbf{a}.P)^+$. Thus, we must prove that Equation 2.10 implies:

$$(\forall t :\ t \in ((\mathbf{l}.P)(\mathbf{a}.P) - \mathbf{l}.P)(\mathbf{a}.P)^+ :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\,)$$

Take trace $t \in (\mathbf{l}.P)(\mathbf{a}.P) - \mathbf{l}.P$ and trace $u \in (\mathbf{a}.P)^+$. That is, trace t is a one-symbol extension of a legal trace, but trace t is not a legal trace itself. This means that $\mathbf{f}.P.t \in \{\bot, \top\}$. By $\top$ and $\bot$-persistence we know that $\mathbf{f}.P.t = \mathbf{f}.P.tu$. Therefore, $\mathbf{f}.P.tu \in \{\bot, \top\}$.

We have to prove that $\mathbf{f}.P.tu \sqsubseteq \mathbf{f}.Q.tu$. If $\mathbf{f}.P.tu = \bot$, then, because $\bot$ is the least label, we have $\mathbf{f}.P.tu \sqsubseteq \mathbf{f}.Q.tu$. If, on the other hand, $\mathbf{f}.P.tu = \top$, we get the following:

$$\mathbf{f}.P.tu = \top$$

$\Rightarrow \quad \{ \mathbf{f}.P.tu = \mathbf{f}.P.t \}$

$\mathbf{f}.P.t = \top$

$\Rightarrow \quad \{ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t$, by Equation 2.9 $\}$

$\mathbf{f}.Q.t = \top$

$\Rightarrow \quad \{ \top$-persistence of Equation 2.5 $\}$

$\mathbf{f}.Q.tu = \top$

$\Rightarrow \quad \{$ assumption: $\mathbf{f}.P.tu = \top \}$

$\mathbf{f}.P.tu \sqsubseteq \mathbf{f}.Q.tu$

□

Next we define reflection. When taking reflection we swap the input and the output alphabet of a process, and for each label we calculate its reflection. As proposed in [Ebergen, 1989], the reflection swaps the roles of a process and its environment. A specification can be seen as a contract that defines the obligations of the process and of its environment. For example, a specification can define when the environment is allowed to produce an input and when the process guarantees to produce an output. The reflection of P, denoted by $\sim P$, is a process that assumes the obligations of the environment of process P. Furthermore, the environment of $\sim P$ assumes the obligations of process P.

Definition 2.3.5 (Reflection)

Reflection of process P, denoted by $\sim P$, is defined as

$$\begin{aligned} \mathbf{i}.{\sim}P &= \mathbf{o}.P \\ \mathbf{o}.{\sim}P &= \mathbf{i}.P \\ \mathbf{f}.{\sim}P.t &= {\sim}\mathbf{f}.P.t \end{aligned}$$

When process P is represented by a state graph, we can compute the reflection of the process directly on the state graph: We take the graph for process P and we replace each state label with its reflection. Furthermore, all question marks in the original graph are replaced by exclamation marks and all exclamation marks from the graph for process P are replaced with question marks. Figure 2.5 shows an example of a reflection of a process where traces are labeled with labels from Figure 2.1.

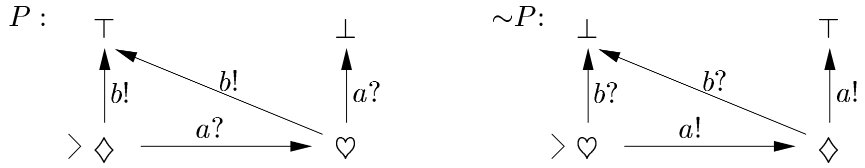


Figure 2.5: Process reflection

We have

$$\mathsf{MIRACLE}(I,O) = {\sim}\mathsf{ABORT}(O,I) \qquad \mathsf{ABORT}(I,O) = {\sim}\mathsf{MIRACLE}(O,I)$$

Reflection on processes reverses the partial order:

Property 2.3.6

The following holds for processes P and Q:

$$P \sqsubseteq Q \;\Leftrightarrow\; {\sim}Q \sqsubseteq {\sim}P$$

Proof:

$$
\begin{array}{ll}
 & P \sqsubseteq Q \\
\equiv & \{ \text{ Definition 2.3.1 } \} \\
 & \left(\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t\right) \\
\Leftrightarrow & \{ \text{ Equations 2.2 and 2.3 } \} \\
 & \left(\forall t :\ t \in (\mathbf{a}.P)^* :\ {\sim}(\mathbf{f}.Q.t) \sqsubseteq {\sim}(\mathbf{f}.P.t)\right) \\
\Leftrightarrow & \{ \text{ Definition 2.3.5 } \} \\
 & \left(\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.{\sim}Q.t \sqsubseteq \mathbf{f}.{\sim}P.t\right)
\end{array}
$$

$\Leftrightarrow$ { Definition 2.3.1 }

$\sim Q \sqsubseteq \sim P$

$\square$

2.4 Product

Recall that we require that a product is defined on Λ and that the product on labels is idempotent, commutative, associative, and that $\top$ and $\perp$ are absorbing elements. We define the process product on the basis of the product on Λ.. When we choose a concrete set of labels, the process product models the joint operation of processes.

In the definition below we use $t \downarrow A$ to denote trace t projected on alphabet A. That is, $t \downarrow A$ denotes trace t from which all symbols not in A have been deleted.

In the first attempt at defining the ECF for the process product we might attempt to compute the label as the product of labels in individual processes:

$$\mathbf{f}'.(P \times Q).t = \mathbf{f}.P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \tag{2.11}$$

In Chapter 3 we show that processes are not closed under the product defined by Equation 2.11, because the result may not be $\perp$ persistent. For this reason, our definition of process product takes into account the $\top$ and the $\perp$ persistence: That is, until the product reaches $\top$ or $\perp$, the label of trace in the product is calculated as the product of labels in individual processes. Once either $\top$ or $\perp$ is reached, the label does not change.

Definition 2.4.1 (Product)

The product of processes P and Q, denoted by $P \times Q$, is defined as

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

The labels of the empty trace, of all legal traces, and of all one-symbol extensions of legal traces in $P \times Q$ are calculated by Equation 2.11. If a one-symbol extension ta of legal trace t is not a legal trace, then the label of all traces tau is determined by the label of trace ta. For this reason, most of our proofs need to take into account only labels of one-symbol extensions of legal traces. For example, Theorem 2.3.4 allows us to check process refinement only by looking at one-symbol extensions of legal traces. Consequently, the case-based definition of the process product does not introduce additional complications.

If processes P and Q are represented by state graphs, we can compute a state graph for $P \times Q$ directly from graphs for processes P and Q. The state graph for $P \times Q$ is based on the direct product of graphs for P and Q. Because the process product requires that the two state graphs have the same alphabets, we first add self-loops to state graphs for processes P and Q. The self loops are labeled with symbols that are present in one process but not in the other. We mark new symbols as inputs. Next, we compute the direct product of the state graphs as described in the procedure for verifying the refinement on state graphs. To obtain the state graph for $P \times Q$, we take the direct product and we modify the graph as follows: Each label (λ, γ) is replaced by label $\lambda \times \gamma$. All states labeled with $\top$ are merged into one state, and all states labeled with $\bot$ are also merged. All transitions leaving states labeled with either $\top$ or $\bot$ are deleted and

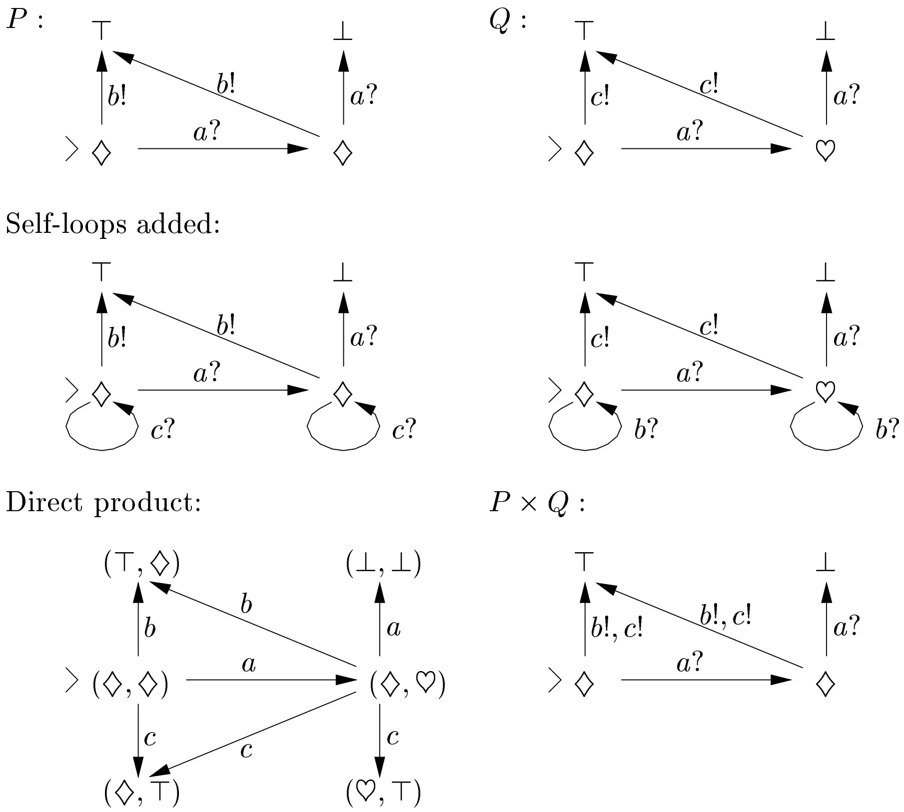


Figure 2.6: Calculating the state graph for the process product

replaced by self loops. Recall that in our depicting of state graphs we omit self-loops of $\top$ and $\bot$ states. Finally, each symbol associated with a transition is postfixed with the question mark if the symbol belongs to the input alphabets of both process P and process Q. Otherwise, the symbol is postfixed by the exclamation mark. Figure 2.6 shows an example of computing the state graph for process $P \times Q$ on the basis of state graphs for processes P and Q. Traces in processes P and Q are labeled with labels from Figure 2.1.

The properties of the process product follow from the properties of the product on labels:

Theorem 2.4.2 (Properties of product)

The product on processes is idempotent, associative, commutative, and monotonic.

Proof: Idempotence and commutativity of process product follow directly from idempotence and commutativity of label product, thus the proofs are trivial and we omit them for the sake of brevity.

Associativity: Proving that $\mathbf{i}.(P \times (Q \times R)) = \mathbf{i}.((P \times Q) \times R)$ and $\mathbf{o}.(P \times (Q \times R)) = \mathbf{o}.((P \times Q) \times R)$ amounts to simple set manipulation. For the sake of brevity, we omit this part of the proof.

We prove $\mathbf{f}.(P \times (Q \times R)).t = \mathbf{f}.((P \times Q) \times R).t$ by induction on the length of the string. First take $t = \varepsilon$:

$$\begin{array}{ll}
 & \mathbf{f}.(P \times (Q \times R)).\varepsilon \\
= & \quad \{ \text{ Definition 2.4.1, twice } \} \\
 & \mathbf{f}.P.\varepsilon \times (\mathbf{f}.Q.\varepsilon \times \mathbf{f}.R.\varepsilon) \\
= & \quad \{ \times \text{ is associative on labels } \} \\
 & (\mathbf{f}.P.\varepsilon \times \mathbf{f}.Q.\varepsilon) \times \mathbf{f}.R.\varepsilon \\
= & \quad \{ \text{ Definition 2.4.1, twice } \} \\
 & \mathbf{f}.((P \times Q) \times R).\varepsilon
\end{array}$$

Now assume that, for any trace t of length less or equal to n, $\mathbf{f}.(P \times (Q \times R)).t = \mathbf{f}.((P \times Q) \times R).t$. Consider trace tx, where $x \in \mathbf{a}.P \cup \mathbf{a}.R \cup \mathbf{a}.R$. If $\mathbf{f}.(P \times (Q \times R)).t \in \{\bot, \top\}$, then, by $\bot$ and $\top$ persistence, $\mathbf{f}.(P \times (Q \times R)).tx = \mathbf{f}.(P \times (Q \times R)).t$ and $\mathbf{f}.((P \times Q) \times R).tx = \mathbf{f}.((P \times Q) \times R).t$. Hence, $\mathbf{f}.(P \times (Q \times R)).tx = \mathbf{f}.((P \times Q) \times R).tx$.

If $\mathbf{f}.(P \times (Q \times R)).t \notin \{\bot, \top\}$, we have:

$$\begin{array}{ll}
 & \mathbf{f}.(P \times (Q \times R)).tx \\
= & \quad \{ \text{ Definition 2.4.1, Equation 2.1 } \}
\end{array}$$

$$
\begin{array}{ll}
& \mathbf{f}.P.(tx \downarrow \mathbf{a}.P) \times (\mathbf{f}.Q.(tx \downarrow \mathbf{a}.Q) \times \mathbf{f}.R.(tx \downarrow \mathbf{a}.R)) \\
= & \quad \{ \times \text{ is associative on labels } \} \\
& (\mathbf{f}.P.(tx \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(tx \downarrow \mathbf{a}.Q)) \times \mathbf{f}.R.(tx \downarrow \mathbf{a}.R) \\
= & \quad \{ \text{ Definition 2.4.1, Equation 2.1 } \} \\
& \mathbf{f}.((P \times Q) \times R).tx
\end{array}
$$

Monotonicity: Assume that $P \sqsubseteq Q$. Because $\mathbf{i}.P = \mathbf{i}.Q$, we know that $\mathbf{i}.(P \times R) = \mathbf{i}.(Q \times R)$. Similarly, from $\mathbf{o}.P = \mathbf{o}.Q$ we conclude that $\mathbf{o}.(P \times R) = \mathbf{o}.(Q \times R)$.

In order to show the monotonicity for ECFs, we first show, by induction on the length of the string, that

$$\mathbf{f}.(P \times R).t \sqsubseteq \mathbf{f}.(Q \times R).t$$

holds for all traces t in $\mathbf{a}.(P \times R)^*$. First let trace t be an empty trace. Then, the label of trace t is computed according to Equation 2.11. This leads to the following:

$$
\begin{array}{ll}
& \mathbf{f}.(P \times R).\varepsilon \\
= & \quad \{ \text{ Definition 2.4.1 } \} \\
& \mathbf{f}.P.\varepsilon \times \mathbf{f}.R.\varepsilon \\
\sqsubseteq & \quad \{ \times \text{ monotonic on labels } \} \\
& \mathbf{f}.Q.\varepsilon \times \mathbf{f}.R.\varepsilon \\
\sqsubseteq & \quad \{ \text{ Definition 2.4.1 } \} \\
& \mathbf{f}.(Q \times R).\varepsilon
\end{array}
$$

Now assume that

$$\mathbf{f}.(P \times R).t \sqsubseteq \mathbf{f}.(Q \times R).t$$

holds for all traces t such that $|t| \leq n$, where $n \geq 0$. We have three cases to consider. First we assume that trace t is a legal trace. Then we have, for some symbol x:

$$
\begin{array}{cl}
 & \mathbf{f}.(P \times R).tx \\
= & \quad \{ \text{ Definition 2.4.1 } \} \\
 & \mathbf{f}.P.(tx \downarrow \mathbf{a}.P) \times \mathbf{f}.R.(tx \downarrow \mathbf{a}.R) \\
\sqsubseteq & \quad \{ \times \text{ monotonic on labels, } \mathbf{a}.P = \mathbf{a}.Q \ \} \\
 & \mathbf{f}.Q.(tx \downarrow \mathbf{a}.Q) \times \mathbf{f}.R.(tx \downarrow \mathbf{a}.R) \\
\sqsubseteq & \quad \{ \text{ Definition 2.4.1 } \} \\
 & \mathbf{f}.(Q \times R).tx
\end{array}
$$

Now assume that trace t is not a legal trace in $P \times R$. If $\mathbf{f}.(P \times R).t = \bot$, then it must be the case that $\mathbf{f}.(P \times R).tx \sqsubseteq \mathbf{f}.(Q \times R).tx$, because $\mathbf{f}.(P \times R).tx = \mathbf{f}.(P \times R).t = \bot$.

If $\mathbf{f}.(P \times R).t = \top$, then it must be the case that $\mathbf{f}.(Q \times R).t = \top$, because $\mathbf{f}.(P \times R).t \sqsubseteq \mathbf{f}.(Q \times R).t$. Furthermore, for any symbol x, $\mathbf{f}.(Q \times R).tx = \top$, hence $\mathbf{f}.(P \times R).tx \sqsubseteq \mathbf{f}.(Q \times R).tx$.

□

2.5 Correctness and testing

When we study process networks, it is useful to define the concept of network correctness. In [Verhoeff, 1994], the notion of correctness was used as a basis for introducing the refinement relation, which was defined in terms of the testing paradigm [Nicola and Hennessy, 1983]. In this section we show that the ECF model allows for the same view of the refinement relation.

In the ECF model, a process network is just another process. We assume that we have defined a predicate **correct**.P, which determines whether process P is "correct".

The definition of **correct**.P depends on the concrete instantiation of the set of labels. We require, though, that the notion of correctness be related to the refinement relation as follows:

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \textbf{correct}.(\sim P \times Q) \tag{2.12}$$

When we choose the particular set of labels and define **correct**.P, we have to prove that Equation 2.12 is satisfied.

We can prove, in general, that a notion of correctness leads to the characterization of the refinement relation in terms of the testing paradigm. That is, if $P \sqsubseteq Q$, then process Q will pass any test that process P has passed. In our context, a test is a process that operates in conjunction with processes P or Q. The relationship between the refinement relation and the notion of correctness as expressed in Theorem 2.5.1 was first established in the model of [Verhoeff, 1994] and later in [Negulescu, 1998].

Theorem 2.5.1 (Testing)

Let P and Q be processes such that $\mathbf{i}.P = \mathbf{i}.Q$ *and* $\mathbf{o}.Q = \mathbf{o}.P$. *Then*

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \big(\forall R :\ R \in \mathcal{PROC}(\mathbf{o}.P, \mathbf{i}.P) :\ \textbf{\textit{correct}}.(P \times R) \Rightarrow \textbf{\textit{correct}}.(Q \times R)\big)$$

Proof: We first tackle the implication from left to right. For all $R \in \mathcal{PROC}(\mathbf{o}.P, \mathbf{i}.P)$, we have

$$\begin{array}{ll}
 & P \sqsubseteq Q \ \wedge\ \textbf{correct}.(P \times R) \\
\Rightarrow & \{ \text{ Commutativity of } \times, \text{ Equation 2.12 } \} \\
 & \sim R \sqsubseteq P \ \wedge\ P \sqsubseteq Q \\
\Rightarrow & \{ \sqsubseteq \text{ is transitive } \} \\
 & \sim R \sqsubseteq Q
\end{array}$$

$\Rightarrow \quad \{$ Equation 2.12, $\sim\sim R = R$ $\}$

$\quad \mathbf{correct}.(Q \times R)$

The implication from right to left can be proven as follows:

$$\big(\forall R:\ R \in \mathcal{PROC}(\mathbf{o}.P, \mathbf{i}.P):\ \mathbf{correct}.(P \times R) \Rightarrow \mathbf{correct}.(Q \times R)\big)$$

$\Rightarrow \quad \{$ Let $R = \sim P$ $\}$

$\quad \mathbf{correct}.(P \times \sim P) \Rightarrow \mathbf{correct}.(Q \times \sim P)$

$\Rightarrow \quad \{$ Equation 2.12 $\}$

$\quad P \sqsubseteq P \Rightarrow P \sqsubseteq Q$

$\Rightarrow \quad \{$ $\sqsubseteq$ is transitive $\}$

$\quad P \sqsubseteq Q$

□

If Equation 2.12 holds, then we have *three* equivalent definitions of refinement. All three definitions have different interpretations. In the first definition, the refinement is based on the comparison of trace labels. In the second definition, the refinement used the reflection of the specification as an environment in which an implementation must operate correctly. In the third definition, the refinement is characterized with the testing paradigm. That is, an implementation must pass all the tests that are passed by the specification.

2.6 Hiding

The purpose of the hiding operation is to hide some of the "internal" ports of a process. Recall that, when we compute the product of two processes, the connections between the processes become output ports of the product. Because "internal" ports are always output ports, we consider hiding output ports only.

The idea for the definition below comes from Willem Mallon [Mallon, 1997] and it captures the following argument. We assume that a label that is larger according to the partial order represents a status of a process that is "better" than the status of the process represented by a smaller label. When we hide some internal ports of a process, the environment does not know in what state the process is, because the environment cannot know what internal actions have been performed by the process. That is, a process can be in one of several different states, where each state could carry a different label. Because the environment of the process should be prepared for the "worst" possible behavior of the process, the result of hiding represents the "worst" possible state in which a process can be after executing a sequence of communications actions, where not all the actions may be visible to the environment.

Because $(\Lambda, \sqsubseteq)$ is a complete lattice, the least upper bound is defined for any $\Gamma \subseteq \Lambda$. Hence, the definition below is valid for all processes.

Definition 2.6.1 (Hiding)

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

$$\begin{array}{lcl} \mathbf{i}.|[A :: P]| & = & \mathbf{i}.P \\ \mathbf{o}.|[A :: P]| & = & \mathbf{o}.P - A \\ \mathbf{f}.|[A :: P]|.t & = & \big(\sqcap s :\ s \in (\mathbf{a}.P)^* \ \wedge\ s \downarrow (\mathbf{a}.P - A) = t :\ \mathbf{f}.P.s\big) \end{array} \tag{2.13}$$

Theorem 2.6.2 (Hiding yields a process)

Let P be be a process and let $A \subseteq \mathbf{o}.P$. Then, $|[A :: P]|$ is a process.

Proof: Because P is a process, $\mathbf{i}.P \cap \mathbf{o}.P = \emptyset$. Consequently, $\mathbf{i}.P \cap (\mathbf{o}.P - A) = \emptyset$. That is, the input and the output alphabets of $|[A :: P]|$ are disjoint.

Next we show that $\mathbf{f}.|[A :: P]|$ is $\perp$-persistent. Take trace t, such that $\mathbf{f}.|[A :: P]|.t = \perp$.

Then, there exists trace s, such that $s \downarrow (\mathbf{a}.P - A) = t$ and $\mathbf{f}.P.s = \bot$. Because $\mathbf{f}.P$ is $\bot$-persistent, $\mathbf{f}.P.su = \bot$ for any trace u. Consequently, $\mathbf{f}.|[A :: P]|.tv = \bot$ for any trace v.

To show $\top$-persistence of $\mathbf{f}.|[A :: P]|$, we take trace t, such that $\mathbf{f}.|[A :: P]|.t = \top$. Then, for every trace s, such that $s \downarrow (\mathbf{a}.P - A) = t$, $\mathbf{f}.P.s = \top$. Because $\mathbf{f}.P$ is $\top$-persistent, $\mathbf{f}.P.su = \top$ for any trace u. Consequently, $\mathbf{f}.|[A :: P]|.tv = \top$ for any trace v. □

We provide an intuitive explanation of hiding and a number of examples of hiding in Chapters 3 and in 4, where we discuss particular instantiations of the ECF model. In Chapters 3 and 4 we also provide alternative definitions of hiding.

Below we prove a number of properties of hiding. In particular, we can hide ports in an arbitrary order, hiding is monotonic, and hiding distributes over product.

Theorem 2.6.3 *Let P be a process and let A and B be sets of symbols such that $A \cup B \subseteq \mathbf{o}.P$. Then the following holds:*

$$|[A :: |[B :: P]|\,]| \;=\; |[A \cup B :: P]| \tag{2.14}$$

Proof: First we check the alphabets:

$$\begin{array}{lllll} \mathbf{i}.|[A :: |[B :: P]| & = & \mathbf{i}.|[A \cup B :: P]| & = & \mathbf{i}.P \\ \mathbf{o}.|[A :: |[B :: P]| & = & \mathbf{o}.|[A \cup B :: P]| & = & \mathbf{o}.P - A - B \end{array}$$

Next we compare the ECFs:

$$\begin{array}{ll} & \mathbf{f}.|[A :: |[B :: P]|\,]|.t \\ = & \{\ \text{Definition 2.6.1}\ \} \\ & \big(\sqcap s :\ s \in (\mathbf{a}.|[B :: P]|)^* \ \wedge\ s \downarrow (\mathbf{a}.|[B :: P]| - A) = t :\ \mathbf{f}.|[B :: P]|.s\big) \end{array}$$

$$
\begin{aligned}
= & \quad \{ \text{ Definition 2.6.1 } \} \\
& \big(\sqcap s : \ s \in (\mathbf{a}.P - B)^* \ \wedge \ s \downarrow (\mathbf{a}.P - B - A) = t : \\
& \qquad\qquad\qquad (\sqcap u : \ u \in (\mathbf{a}.P)^* \ \wedge \ u \downarrow (\mathbf{a}.P - B) = s : \ \mathbf{f}.P.u) \big) \\
= & \quad \{ \text{ Join the domains } \} \\
& \big(\sqcap s, u : \ s \in (\mathbf{a}.P - B)^* \ \wedge \ s \downarrow (\mathbf{a}.P - B - A) = t \ \wedge \\
& \qquad\qquad\qquad u \in (\mathbf{a}.P)^* \ \wedge \ u \downarrow (\mathbf{a}.P - B) = s : \ \mathbf{f}.P.u \big) \\
= & \quad \{ \ u \downarrow (\mathbf{a}.P - B) = s \ \wedge \ s \downarrow (\mathbf{a}.P - B - A) = t \ \Rightarrow \ u \downarrow (\mathbf{a}.P - B - A) = t \ \} \\
& \big(\sqcap u : \ u \in (\mathbf{a}.P)^* \ \wedge \ u \downarrow (\mathbf{a}.P - B - A) = t : \ \mathbf{f}.P.u \big) \\
= & \quad \{ \text{ Definition 2.6.1 } \} \\
& \mathbf{f}.|[A \cup B :: P]|.t
\end{aligned}
$$

□

Theorem 2.6.4 (Monotonicity of hiding)

Hiding is monotonic with respect to refinement. Let $A \subseteq \mathbf{o}.P$. Then:

$$P \sqsubseteq Q \quad \Rightarrow \quad |[A :: P]| \sqsubseteq |[A :: Q]|$$

Proof: Let $P \sqsubseteq Q$. Then, $\mathbf{i}.P = \mathbf{i}.Q$ and, consequently, $\mathbf{i}.|[A :: P]| = \mathbf{i}.|[A :: Q]|$. Furthermore, $\mathbf{o}.P = \mathbf{o}.Q$, which implies $\mathbf{o}.|[A :: P]| = \mathbf{o}.|[A :: Q]| = \mathbf{o}.P - A$.

We have to prove $\mathbf{f}.|[A :: P]|.t \sqsubseteq \mathbf{f}.|[A :: Q]|.t$ for any trace t:

$$
\begin{aligned}
& \mathbf{f}.|[A :: P]|.t \\
= & \quad \{ \text{ Definition 2.6.1 } \} \\
& \big(\sqcap s : \ s \in (\mathbf{a}.P)^* \wedge s \downarrow (\mathbf{a}.P - A) = t : \ \mathbf{f}.P.s \big) \\
\sqsubseteq & \quad \{ \ \mathbf{f}.P.s \sqsubseteq \mathbf{f}.Q.s, \ \mathbf{a}.P = \mathbf{a}.Q \ \} \\
& \big(\sqcap s : \ s \in (\mathbf{a}.Q)^* \wedge s \downarrow (\mathbf{a}.Q - A) = t : \ \mathbf{f}.Q.s \big)
\end{aligned}
$$

$=$ { Definition 2.6.1 }

$\mathbf{f}.|[A :: Q]|.t$

$\square$

Hiding distributes over product if certain conditions are satisfied. More precisely, if the symbols of A do not occur in Q, then first hiding A in P and then composing P with Q is the same as first composing P with Q and then hiding A.

Theorem 2.6.5 (Distributivity of hiding)

For processes P and Q, such that $A \cap \mathbf{a}.Q = \emptyset$, we have

$$|[A :: P]| \times Q = |[A :: P \times Q]|$$

Proof: First we check the alphabets:

$$\mathbf{i}.(|[A :: P]| \times Q) = \mathbf{i}.(P \times Q) = \mathbf{i}.|[A :: P \times Q]|$$

$$\mathbf{o}.(|[A :: P]| \times Q) = \mathbf{o}.|[A :: P \times Q]| = (\mathbf{o}.P \cup \mathbf{o}.Q) - A$$

Next we prove that $\mathbf{f}.(|[A :: P]| \times Q).t = \mathbf{f}.|[A :: P \times Q]|.t)$. Because processes are closed under hiding and product, and because of $\top$ and $\perp$-persistence, we only need to consider t being either an empty trace or a one-symbol extension of a legal trace.

$\mathbf{f}.(|[A :: P]| \times Q).t$

$=$ { definition of product }

$\mathbf{f}.|[A :: P]|.(t \downarrow (\mathbf{a}.P - A)) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)$

$=$ { Definition 2.6.1 }

$\left(\sqcap v \ : \ v \in (\mathbf{a}.P)^* \wedge v \downarrow (\mathbf{a}.P - A) = t \downarrow (\mathbf{a}.P - A) \ : \ \mathbf{f}.P.v \right) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)$

$$
\begin{array}{ll}
= & \{\ \times \text{ is monotonic on labels, so } \times \text{ distributes over } \sqcap\ \} \\
& \big(\ \sqcap v\ ::\ v \in (\mathbf{a}.P)^* \wedge v \downarrow (\mathbf{a}.P - A) = t \downarrow (\mathbf{a}.P - A)\ :\ \mathbf{f}.P.v \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)\,\big) \\
= & \{\ A \cap \mathbf{a}.Q = \emptyset;\ \text{take } v = w \downarrow \mathbf{a}.P;\ \text{Lemma 2.6.6}\ \} \\
& \big(\ \sqcap w\ :\ w \in (\mathbf{a}.P \cup \mathbf{a}.Q)^* \wedge\ w \downarrow ((\mathbf{a}.P - A) \cup \mathbf{a}.Q) = t \downarrow ((\mathbf{a}.P - A) \cup \mathbf{a}.Q)\ : \\
& \mathbf{f}.P.(w \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)\,\big) \\
= & \{\ A \cap \mathbf{a}.Q = \emptyset,\ t \downarrow (\mathbf{a}.P - A) \cup \mathbf{a}.Q) = t\ \ \} \\
& \big(\sqcap w : w \in (\mathbf{a}.P \cup \mathbf{a}.Q)^* \wedge\ w \downarrow ((\mathbf{a}.P \cup \mathbf{a}.Q) - A) = t : \mathbf{f}.P.(w \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)\,\big) \\
= & \{\ w \downarrow \mathbf{a}.Q = t \downarrow \mathbf{a}.Q \text{ since } A \cap \mathbf{a}.Q = \emptyset\ \} \\
& \big(\sqcap w : w \in (\mathbf{a}.P \cup \mathbf{a}.Q)^* \wedge w \downarrow ((\mathbf{a}.P \cup \mathbf{a}.Q) - A) = t : \mathbf{f}.P.(w \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(w \downarrow \mathbf{a}.Q)\,\big) \\
= & \{\ \text{Definition 2.4.1}\ \} \\
& \big(\ \sqcap w : w \in (\mathbf{a}.P \times \mathbf{a}.Q)^* \wedge w \downarrow ((\mathbf{a}.P \cup \mathbf{a}.Q) - A) = t : \mathbf{f}.(P \times Q).w\,\big) \\
= & \{\ \text{Definition 2.6.1}\ \} \\
& \mathbf{f}.|[A :: P \times Q]|.t
\end{array}
$$

$\square$

In the proof above we made use of the following lemma.

Lemma 2.6.6

Let P and Q be processes and let A be a set of symbols such that $A \cap \mathbf{a}.Q = \emptyset$. For t, a trace over alphabet $(\mathbf{a}.P - A) \cup \mathbf{a}.Q$, we have

$$
\begin{array}{ll}
& \{v\ :\ v \in (\mathbf{a}.P)^* \wedge v \downarrow (\mathbf{a}.P - A) = t \downarrow (\mathbf{a}.P - A)\ :\ v\} \\
= & \{\ A \cap \mathbf{a}.Q = \emptyset,\ v = w \downarrow \mathbf{a}.P,\ t \in ((\mathbf{a}.P - A) \cup \mathbf{a}.Q)^*\ \ \} \\
& \{w\ :\ w \in (\mathbf{a}.P \cup \mathbf{a}.Q)^* \wedge\ w \downarrow ((\mathbf{a}.P - A) \cup \mathbf{a}.Q) = t\ :\ w \downarrow \mathbf{a}.P\}
\end{array}
$$

A corollary of the above theorem is

Corollary 2.6.7

Hiding distributes over product. If $A \cap \mathbf{a}.Q = \emptyset$ *and* $B \cap \mathbf{a}.P = \emptyset$*, then*

$$|[A :: P]| \ \times \ |[B :: Q]| = |[A \cup B :: P \times Q]|$$

Proof:

$$\begin{array}{ll}
 & |[A :: P]| \ \times \ |[B :: Q]| \\
= & \quad \{ \ A \cap \mathbf{a}.Q = \emptyset, \text{ Theorem 2.6.5 } \ \} \\
 & |[A :: P \times |[B :: Q]|\,]| \\
= & \quad \{ \ B \cap \mathbf{a}.P = \emptyset, \text{ Theorem 2.6.5 } \ \} \\
 & |[A :: |[B :: P \times Q]|\,]| \\
= & \quad \{ \text{ Theorem 2.6.3 } \ \} \\
 & |[A \cup B :: P \times Q]|
\end{array}$$

□

2.7 Substitution theorem

In this section we prove the Substitution Theorem, which is the cornerstone of the top-down approach to design and verification. The Substitution Theorem allows us to substitute a component in any network with an implementation of that component. We leave examples of applications of Substitution Theorem for Chapter 6 we discuss a design methods based on ECF processes.

Theorem 2.7.1 (Substitution theorem)

Let P*,* Q*,* R*, and* S *be processes. If*

$$P \sqsubseteq |[A :: Q \times R]| \quad \text{and} \quad R \sqsubseteq |[B :: S]|$$

then

$$P \sqsubseteq |[A :: Q \times |[B :: S]|\ ]|$$

Proof:

$$\begin{array}{ll}
& R \ \sqsubseteq \ |[\, B :: S \,]| \\
\Rightarrow & \{ \text{ Theorem 2.4.2 – monotonicity of } \times \ \} \\
& Q \times R \ \sqsubseteq \ Q \times |[\, B :: S \,]| \\
\Rightarrow & \{ \text{ Theorem 2.6.4 – monotonicity of hiding } \} \\
& |[\, A :: Q \times R \,]| \ \sqsubseteq \ |[\, A :: Q \times |[\, B :: S \,]|\ ]| \\
\Rightarrow & \{ \ P \sqsubseteq |[A :: Q \times R]|, \text{ Theorem 2.3.2 – transitivity of } \sqsubseteq \ \} \\
& P \sqsubseteq |[\, A :: Q \times |[\, B :: S \,]|\ ]|
\end{array}$$

□

Thus, the scope of the local symbols B in the component S only pertains to S, and we don't have to invent fresh names for each new refinement. Local symbols can be reused, just like in programming languages.

If we impose some alphabet restrictions, we can rephrase the Substitution Theorem as follows.

Theorem 2.7.2 (Substitution Theorem, rephrased)

Let $B \cap \mathbf{a}.Q = \emptyset$. *If*

$$P \sqsubseteq |[A :: Q \times R]| \quad \text{and} \quad R \sqsubseteq |[B :: S]|$$

then

$$P \sqsubseteq |[A \cup B :: Q \times S]|$$

Proof:

$$
\begin{array}{ll}
& P \\
\sqsubseteq & \{ \text{ Substitution Theorem } \} \\
& |[\, A :: Q \;\times\; |[B :: S\,]|\,]| \\
\sqsubseteq & \{ \; B \cap \mathbf{a}.Q = \emptyset, \text{ Theorem 2.6.5 -- hiding distributes over product } \} \\
& |[\, A :: |[B :: Q \times S\,]|\,]| \\
\sqsubseteq & \{ \text{ Theorem 2.6.3 } \} \\
& |[\, A \cup B :: Q \times S\,]|\,]|
\end{array}
$$

□

The condition $B \cap \mathbf{a}.Q = \emptyset$ can always be satisfied by a renaming the local symbols B in S.

2.8 Summary

In this chapter we developed a formal model, called the ECF model, where a process is defined with its enhanced characteristic function (ECF). ECFs were introduced in [Verhoeff, 1994]. However, the development of a formal model on the basis of ECFs is a new result. An enhanced characteristic function maps traces to a set of labels. We refrained from a concrete instantiation of the label set. Instead, we required a number of simple properties to hold for the set of labels: the partial order on labels induces a complete lattice, the reflection on labels turns the partial order upside down, and the product on labels must be idempotent, commutative, associative, and monotonic. If the set of labels is small, these properties are easy to check by inspection.

On the basis of the operations on labels, we defined the refinement relation on processes and a number of operations on processes. We emphasize that all these properties hold in any concrete instantiation of the ECF model.

The following table summarizes the operations defined and the properties proven. In the table, P, Q, and R denote processes, and A and B denote sets of symbols.

operation	notation	property
refinement	$\sqsubseteq$	$(\mathcal{PROC}(I, O), \sqsubseteq)$ is a complete lattice.
		environment: $P \sqsubseteq Q \;\Leftrightarrow \textbf{correct}.({\sim}P \times Q)$
		testing: $P \sqsubseteq Q \;\Leftrightarrow\; (\textbf{correct}.(P \times R) \;\Rightarrow\; \textbf{correct}.(Q \times R))$
reflection	${\sim}P$	turns refinement upside-down: $P \sqsubseteq Q \;\Rightarrow\; {\sim}Q \sqsubseteq {\sim}P$
product	$\times$	idempotence: $P \times P = P$
		associativity: $(P \times Q) \times R = (P \times (Q \times R)$
		commutativity: $P \times Q = Q \times P$
		monotonicity: $P \sqsubseteq Q \;\Rightarrow\; P \times R \;\sqsubseteq\; Q \times R$
hiding	$\lvert[A :: P]\rvert$	arbitrary order: $\lvert[\, A :: \lvert[\, B :: P\,]\rvert\,]\rvert = \lvert[\, A \cup B :: P\,]\rvert$
		monotonicity: $P \sqsubseteq Q \;\Rightarrow\; \lvert[\, A :: P\,]\rvert \;\sqsubseteq\; \lvert[\, A :: Q\,]\rvert$
		distributivity: $\lvert[\, A :: P\,]\rvert \times Q = \lvert[\, A :: P \times Q\,]\rvert$, if $\mathbf{a}.Q \cap A = \emptyset$

Finally, we proved the Substitution Theorem, which is a foundation for hierarchical design and verification:

$$P \sqsubseteq \lvert[\, A :: P \times R\,]\rvert \;\wedge\; R \sqsubseteq \lvert[\, B :: S\,]\rvert \;\;\Rightarrow\;\; P \sqsubseteq \lvert[\, A :: P \times \lvert[\, B :: S\,]\rvert\,]\rvert$$

Chapter 3

ECF processes and safety

In this chapter we apply the ECF process model to studying safety of networks of devices. By a network of devices we mean a set of devices where connections are established between ports that carry the same name. We model a device by an ECF process, and we say that a process network is safe when each output that a process can produce can also be accepted by all receiving processes. Previous studies of network safety have been extensive and include [Udding, 1984], [van de Snepscheut, 1985], [Ebergen, 1991], [Dill, 1989], [Verhoeff, 1994], and [Negulescu, 1998]. A safe network is also referred to as a network with absence of computation interference [Udding, 1984, van de Snepscheut, 1985, Ebergen, 1991] or with absence of choking [Dill, 1989].

There are some differences between our approach to safety and the approaches taken in previous work. For example, in [Verhoeff, 1994], [Ebergen, 1991], and [Udding, 1984] failures are represented by traces that are absent from the trace set of a specification. We model failures explicitly by assigning the $\perp$ label to traces that represent failures. In [Ebergen, 1991], the safety condition is defined only for closed networks. Just as [Verhoeff, 1994] does, we define safety as a stand-alone concept without any alphabet restrictions, and we prove that safety is closely related to our refinement relation.

The safety condition on process spaces [Negulescu, 1998] is expressed as so-called robustness of a process. As opposed to the absence of computation interference, robustness does not require that the process have no input terminals. Rather, a robust process must be capable of receiving an input at any time on any of its input ports. A robust process never fails by itself. The safety condition for ECF processes also does not require that a process have no input terminals and the condition demands that a process cannot fail by itself. On the other hand, our safety condition does not require that a process must always be able to receive an input on any of its input ports. If a process has no input ports, then our safety condition is equivalent to the robustness from [Negulescu, 1998].

Hiding, used to conceal a subset of output ports of a process, is an operation that distinguishes our model from the previous work. Hiding from [Dill, 1989] and projection from [Ebergen, 1991] simply delete the hidden symbols from traces without taking into account safety properties of a process. Our hiding operation takes the safety issues into account. [Verhoeff, 1994] does not define an explicit hiding operation. Rather, hiding is implicitly included in the process composition, which contributes to the complexity of computing the composition of processes in the DI model of [Verhoeff, 1994].

In order to address safety properties, we instantiate the model from Chapter 2. We first choose a set of labels and we demonstrate how our labels reflect safety of a process Because the labels satisfy the requirements of the general ECF model, all properties proven in Chapter 2 also hold for the instantiated model.

Besides illustrating the properties and operations defined in Chapter 2, we prove some additional properties that pertain to the safety model. In particular, we give a formal characterization of safety and show how this characterization relates to the refinement relation. We also prove the Factorization Theorem, which provides a solution to the

Design Equation from [Verhoeff, 1994].

3.1 Labels and processes

When we talk about safety of networks of processes, we are interested whether one process can produce an output that some other process in the network is unable to receive. We describe the behavior of a device by characterizing all sequences of possible communication actions that take place between the device and its environment. Following the general ECF model, we assign a label to each sequence, indicating some property of that sequence. In the safety model, our only concern is to know which sequences of communication actions are allowed to take place during the operation of a device. We call such traces legal traces and we assign to them the label $\bigcirc$. According to the ECF model, our set of labels must also include $\top$ and $\perp$; thus, the set of labels we use is

$$\Lambda = \{\perp, \bigcirc, \top\}$$

The meaning of the labels is as follows. For label $\bigcirc$, we already know that it marks sequences of allowed communication actions in the behavior of a device. Traces labeled with either $\perp$ or $\top$ can be divided into four categories:

- Trace ta is labeled with $\perp$ and a is an input symbol. That is, a transition on input symbol a leads to the $\perp$ state. Such a trace denotes a requirement that the environment of the process is not supposed to provide an input on port a after trace t. If the environment provides an input on port a, the process fails.

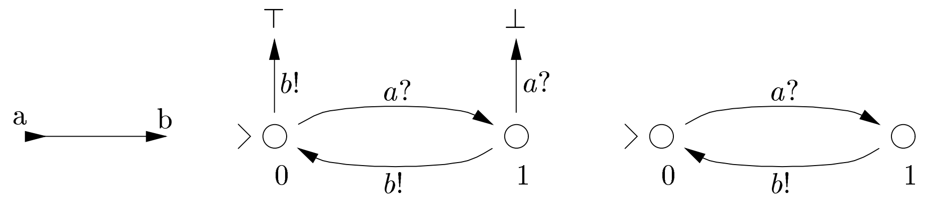
- Trace ta is labeled with $\perp$ and a is an output symbol. That is, a transition on output symbol a leads to the $\perp$ state. Such a trace denotes that, after exhibiting trace t, the process fails while producing an output on port a.

- Trace ta is labeled with $\top$ and a is an output symbol. That is, a transition on output symbol a leads to the $\top$ state. Such a trace denotes a requirement that, after exhibiting trace t, the process is not supposed to produce an output on port a.
- Trace ta is labeled with $\top$ and a is an input symbol. That is, a transition on input symbol a leads to the $\top$ state. Such a trace denotes that, after exhibiting trace t, the environment of the process fails while providing an input on port a.

We denote the set of legal traces of process P by $\mathbf{l}.P$. In the context of the model presented in this chapter, we can rewrite Definition 2.6 as follows:

$$\mathbf{l}.P = \{t \ : \ t \in (\mathbf{a}.P)^* \ \wedge \ \mathbf{f}.P.t = \bigcirc \ : \ t\} \tag{3.1}$$

Recall that labels of traces can be interpreted as labels of states in a state graph depicting a process. All traces that lead to the same state have the same label. In Figure 3.1 we show a process that specifies the behavior of a WIRE with the input port a and the output port b. In the state graph we postfix inputs with a question mark and outputs with an exclamation mark.



a) WIRE schematic b) WIRE specification b) simplified graph

Figure 3.1: A specification of the WIRE

The WIRE may first receive an input on port a and then it may produce an output on port b. The cycle then repeats. In the initial state, the transition on output b leads to the $\top$ state. This means that, initially, the WIRE cannot produce an output on port b. Furthermore, the transition on port a starting in state 1, leads to the $\perp$ state. This means that the WIRE, after having received an input on port a, cannot receive another input on the same port before producing an output.

In Figure 3.1c we show the "simplified" state graph that describes the behavior of the WIRE. In the simplified graph we remove the $\top$ and the $\perp$ state and the transitions leading to these two states. We can use simplified graphs in the case where only input transitions lead from a legal state to the $\perp$ state and only output transitions lead from a legal state to the $\top$ state.

3.2 Refinement

The ECF model requires that we define a partial order among labels, such that $\perp$ is the least element and $\top$ is the greatest element. We define the partial order on Λ as

$$\perp \sqsubseteq \bigcirc \sqsubseteq \top \tag{3.2}$$

The partial order defined above is a total order. Thus, we can immediately conclude that $(\Lambda, \sqsubseteq)$ is a complete lattice as required by the ECF model.

In terms of safety considerations, our intuition behind the partial order is as follows. If $\lambda \sqsubseteq \gamma$, for labels λ and γ, we say than a trace labeled with λ is less safe that a trace labeled with γ. In particular, a failure trace, labeled with $\perp$, is less safe than a legal trace labeled with $\bigcirc$. A trace labeled with $\top$ is considered a miracle trace that cannot occur and can cause no damage. Thus, a trace labeled with $\top$ is safer than any trace that can occur.

Following the ECF model, we extend the refinement on labels to the refinement on processes:

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

With the examples below we demonstrate that $P \sqsubseteq Q$ means that process Q is "at least as safe" as process P. By "at least as safe" we mean that process Q can accept an input in all situations where process P can accept an input. That is, the environment can violate the specification of process Q in fewer situations than it can violate the specification of process P. Furthermore, when $P \sqsubseteq Q$, process Q can produce an output in at most as many situations as process P can. This means that process P has more opportunities to violate processes in its environment than process Q.

Recall that $(\mathcal{PROC}(I,O), \sqsubseteq)$ is a complete lattice. $\mathsf{ABORT}(I,O)$ is the smallest process, thus, we consider $\mathsf{ABORT}(I,O)$ to be the least safe process in $\mathcal{PROC}(I,O)$. $\mathsf{MIRACLE}(I,O)$ is the largest process in $\mathcal{PROC}(I,O)$, thus we consider it to be the safest.

3.2.1 Examples of refinement

Let us examine the refinement relation more closely. Recall that, when $P \sqsubseteq Q$, we refer to process P as the specification, and we call process Q an implementation.

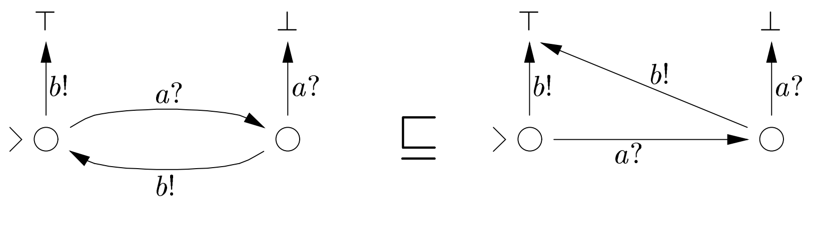


Figure 3.2: An implementation of the WIRE

Figure 3.2 shows an implementation of the WIRE. This implementation can initially receive an input on port a, but will never produce an output on port b. We verify the refinement by comparing the labels of the states in the two state graphs. The initial states have the same labels, as do the states to which traces a and b lead. Trace b leads to the $\top$ state, which means that, by $\top$-persistence, the refinement holds for all extensions of trace b. In the WIRE, trace ab leads to a state labeled with $\bigcirc$, but in the implementation, trace ab leads to the $\top$ state. Because $\top$ is the greatest of all labels, the refinement holds for trace ab. Furthermore, by $\top$-persistence we know that the refinement holds for all extensions of trace ab. Finally, we observe trace aa. In the WIRE and in the implementation trace aa leads to the $\perp$ state. By $\perp$-persistence it follows that the refinement holds for all extensions of trace aa. We conclude that the refinement holds for the labels of all traces, thus the refinement between two processes of Figure 3.2 indeed holds.

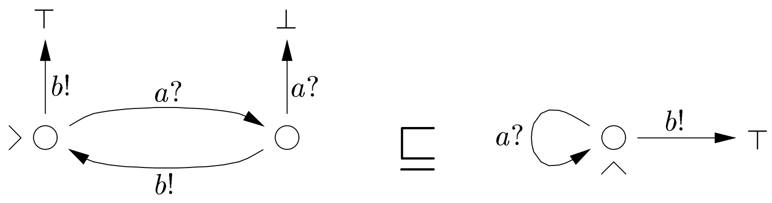


Figure 3.3: Another implementation of the WIRE

In the previous example we demonstrated that an implementation does not have to be able to produce an output when a specification can. In the example of Figure 3.3 we show that an implementation can receive an input when the specification cannot: Consider trace aa. In the WIRE process, trace aa leads to the $\perp$ state, which means that the WIRE cannot receive two inputs in a row on port a. In the implementation, two inputs in a row are possible, because trace aa leads to the state labeled with $\bigcirc$. One can check that the refinement relation in Figure 3.3 holds by computing the direct product of the two state graphs, as suggested in Chapter 2.

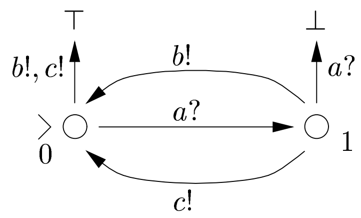


Figure 3.4: A specification of the SELECTOR

In the next example we consider an output-nondeterministic specification, and we check whether the nondeterminism must be preserved in an implementation. In Figure 3.4 we show a specification of the SELECTOR. The SELECTOR has one input port, a, and two output ports, b and c. Initially, the SELECTOR is ready for receiving an input on port a. After having received that input, the SELECTOR has a choice of producing an output on either port b or port c. After an output has been produced, the cycle repeats.

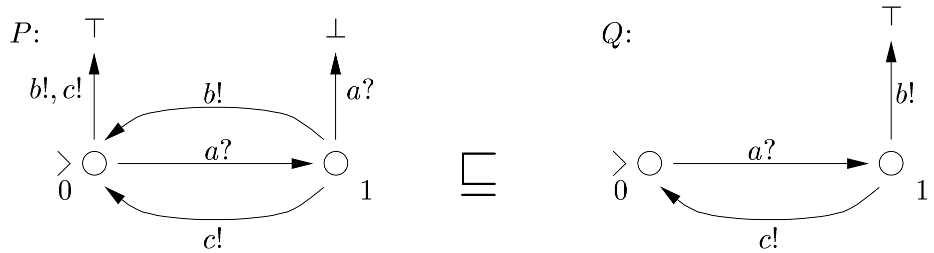


Figure 3.5: An implementation of the SELECTOR

State 1 is the point where the SELECTOR makes a nondeterministic choice between producing either output b or output c. The SELECTOR can be implemented by process Q of Figure 3.5. Notice that process Q makes no choice between producing output b or output c. That is, Q can only produce an output on port c, while port b always stays inactive. This example shows that in our model nondeterminism needs not be preserved in an implementation.

The next two examples illustrate violations of the refinement relation. Figure 3.6

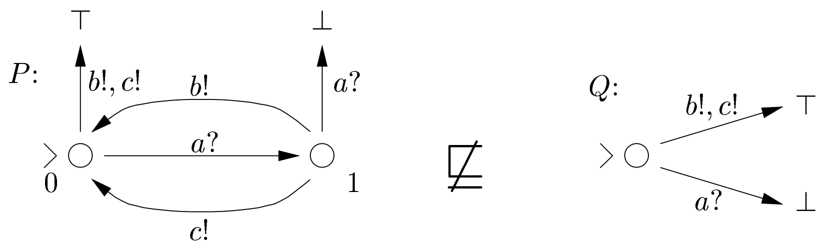


Figure 3.6: A bad implementation of the SELECTOR

shows a proposed implementation of the SELECTOR, where the implementation initially cannot receive an input on port a. Notice that trace a in the SELECTOR leads to the state labeled with $\bigcirc$, and, in process Q, trace a leads to the $\perp$ state. Because $\bigcirc \not\sqsubseteq \perp$, process Q of Figure 3.6 violates the refinement relation. This example illustrates that an implementation must be capable of receiving at least the inputs that can be received by the specification.

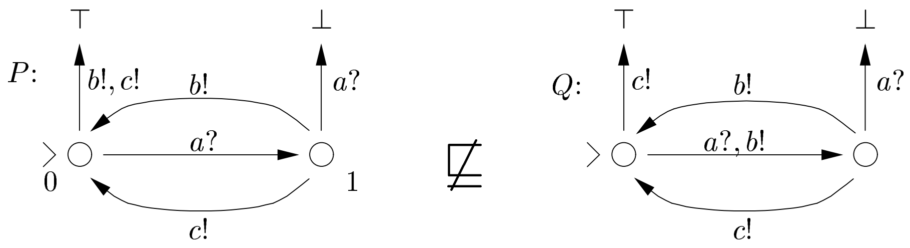


Figure 3.7: Another bad implementation of the SELECTOR

In our last example we consider an "implementation" that can produce an output in a state where the specification cannot produce an output. Process Q of Figure 3.7 can produce an output on port b in its initial state. The SELECTOR can only receive an input on port a in the initial state, but can produce no outputs. Thus, the label of trace b in the SELECTOR is $\top$, and in process Q the label of trace b is $\bigcirc$. Because $\top \not\sqsubseteq \bigcirc$, the refinement relation between the SELECTOR and process Q of Figure 3.7 does not hold. The example illustrates that an implementation cannot produce more outputs than what is allowed by the specification.

3.3 Product

In order for the properties from Chapter 2 to hold, the product on labels must be associative, commutative, idempotent, and monotonic with respect to $\sqsubseteq$. We define the product of the safety labels in Table 3.1. One can verify by inspection that all the properties listed above indeed hold.

$\times$	$\bot$	$\bigcirc$	$\top$
$\bot$	$\bot$	$\bot$	$\top$
$\bigcirc$	$\bot$	$\bigcirc$	$\top$
$\top$	$\top$	$\top$	$\top$

Table 3.1: Product of labels

How do we interpret the entries in Table 3.1? The result of the product reflects the interaction between two processes. If one process exhibits a trace with label λ and another process exhibits a trace with label γ, then the network of the two processes exhibits the trace with label $\lambda \times \gamma$. This point of view is at the heart of the definition of the process product.

Recall that label $\bigcirc$ indicates that the trace is in agreement with how the process should behave. For this reason, we have $\bigcirc \times \bigcirc = \bigcirc$. That is, if a trace causes no problems in either process, then the network of processes operates without problems for that trace. Label $\top$ marks traces that cannot take place. For this reason, $\top \times \lambda = \top$ for any label λ. That is, if a trace cannot take place in one of the processes, then it also cannot take place in the network of processes. Finally, recall that label $\bot$ represents a failure. We stipulate that, if one of the processes in a network fails, then the network itself fails. For this reason, $\perp \times \lambda = \perp$ for λ being any label but $\top$. If $\lambda = \top$, then the trace cannot take place and the network does not fail.

Let us recall the definition of the process product. The product of processes P and Q is a process, denoted by $P \times Q$, and is defined as

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

In the definition above, t is an arbitrary trace and a is a symbol.

We use the process product in order to model a network of processes. In the network we assume that we have connections between ports with the same name. We also assume that the communication between different processes takes place instantaneously. On the other hand, a process can take an arbitrarily long time before producing an output. These assumptions match a so-called speed-independent model, where arbitrary delays can occur within components, but the communication is instantaneous.

The process product enjoys the properties we proved in Chapter 2: it is idempotent, associative, commutative, and monotonic with respect to process refinement.

Let us examine the process product through an example. Figure 3.8 shows the result of the product of two processes each specifying a WIRE. Process P corresponds to a WIRE with input port a and output port m. Process Q corresponds to a WIRE with input port m and output port b. Figure 3.8 also shows process $P \times Q$, which represents a network of two connected WIRES. The two $\perp$ states in the state graph of process $P \times Q$

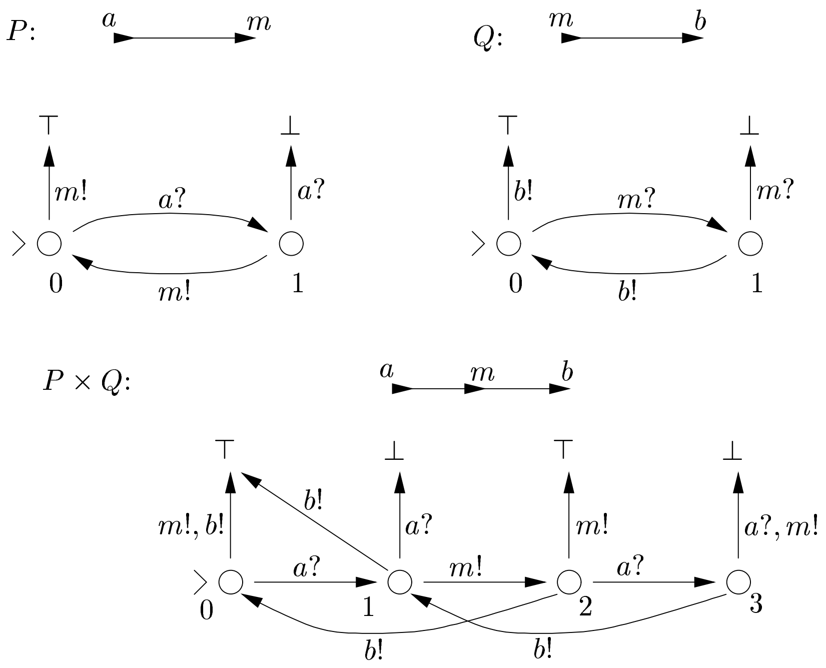


Figure 3.8: The product of two WIREs

are really the same state, which we depicted twice in order to reduce the clutter. The same holds for the two $\top$ states. Notice that this network may fail. If the environment produces an input on port a immediately after the first WIRE has produced output m, then the first WIRE can produce another output on port m before the second wire has produced an output on port b. The transition on output m from state 3 to the $\perp$-state represents the possibility of this failure. This example demonstrates that the product of two processes contains information on failures that can occur because one of the network components can violate the specification of another network component.

In Chapter 2 we questioned why the label of the trace in the process product cannot simply be calculated as

$$\mathbf{f}'.(P \times Q).t = \mathbf{f}.P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \tag{3.3}$$

Processes P and Q and their product shown in Figure 3.9 demonstrate why we were not able to simplify our definition.

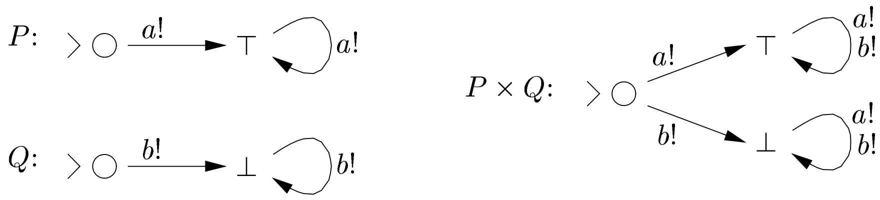


Figure 3.9: Process product illustration

Assume that the enhanced characteristic function of the process product was defined by Equation 3.3. Let us apply this function to some traces in processes P and Q and see whether the results conflict with the product depicted in Figure 3.9.

$$
\begin{array}{lcccccl}
\mathbf{f}'.(P \times Q).\varepsilon & = & \bigcirc \times \bigcirc & = & \bigcirc & = & \mathbf{f}.(P \times Q).\varepsilon \\
\mathbf{f}'.(P \times Q).a & = & \top \times \bigcirc & = & \top & = & \mathbf{f}.(P \times Q).a \\
\mathbf{f}'.(P \times Q).b & = & \bigcirc \times \bot & = & \bot & = & \mathbf{f}.(P \times Q).b \\
\mathbf{f}'.(P \times Q).ba & = & \bot \times \top & = & \top & \neq & \mathbf{f}.(P \times Q).ba
\end{array}
$$

For traces of length zero and one, $\mathbf{f}'.(P \times Q)$ agrees with $\mathbf{f}.(P \times Q)$. The difference appears in a trace of length two: $\mathbf{f}'.(P \times Q).ba = \top$, but, because of $\bot$-persistence, $\mathbf{f}.(P \times Q).ba$ should be equal to $\bot$. This means that using Equation 3.3 for calculating the labels of traces in the process product would violate the $\bot$-persistence requirement for the processes.

For the product $P \times Q$, the labels of all legal traces and of all one-symbol extensions of legal traces are calculated by Equation 3.3. These are the traces that play an important role in our treatment of safety, as we are interested in whether a one-symbol extension of a legal trace may carry the $\bot$ label. Because we can use Equation 3.3 for all one-symbol extensions of legal traces, our proofs are not additionally complicated by the case-based definition of the process product.

3.3.1 Product and legal traces

The process product models networks of processes. How can we characterize legal traces in a network of processes? Recall that legal traces capture "allowed" sequences of communication actions in a process. The theorem below states that a legal trace in a network of processes is a legal trace with respect to each process in the network. That is, a legal trace in a process product must be a result of a product of legal traces in individual processes.

Theorem 3.3.1

The following holds for processes P and Q and any trace $t \in (\mathbf{a}.(P \times Q))^*$:

$$t \in \mathbf{l}.(P \times Q) \quad \Leftrightarrow \quad (t \downarrow \mathbf{a}.P) \in \mathbf{l}.P \ \wedge \ (t \downarrow \mathbf{a}.Q) \in \mathbf{l}.Q \tag{3.4}$$

Proof:

$t \in \mathbf{l}.(P \times Q)$

$\Leftrightarrow$ { Equation 2.6 — legal traces }

$\mathbf{f}.(P \times Q).t \notin \{\top, \bot\}$

$\Leftrightarrow$ { Definition 2.4.1 — product }

$\mathbf{f}.P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \notin \{\bot, \top\}$

$\Leftrightarrow$ { Table 3.1 }

$\mathbf{f}.P.(t \downarrow \mathbf{a}.P) \notin \{\bot, \top\} \ \wedge \ \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \notin \{\bot, \top\}$

$\Leftrightarrow$ { Equation 2.6 — legal traces }

$(t \downarrow \mathbf{a}.P) \in \mathbf{l}.P \ \wedge \ (t \downarrow \mathbf{a}.Q) \in \mathbf{l}.Q$

□

A simple corollary of the theorem above is that an illegal trace in a process product must be illegal in at least one of the individual processes:

$$t \notin \mathbf{l}.(P \times Q) \quad \Leftrightarrow \quad (t \downarrow \mathbf{a}.P) \notin \mathbf{l}.P \ \vee \ (t \downarrow \mathbf{a}.Q) \notin \mathbf{l}.Q \tag{3.5}$$

3.4 Reflection

Reflection is a known operator in trace theory [Ebergen, 1989]. The reflection of a process exchanges the roles of input and output ports of the process. We use the reflection as an environment for a prospective implementation of a process.

λ	$\bot$	$\bigcirc$	$\top$
$\sim\lambda$	$\top$	$\bigcirc$	$\bot$

Table 3.2: Reflection on labels

We define the reflection on labels in Table 3.2. By inspection we can verify that the reflection on labels satisfies the requirements from Chapter 2. Namely,

$$\lambda \sqsubseteq \gamma \ \Rightarrow \ \sim\gamma \sqsubseteq \sim\lambda \qquad \text{and} \qquad \sim(\sim\lambda) = \lambda$$

Now we can follow Chapter 2 further and extend the reflection to processes. That is, when taking reflection we swap the input and the output alphabet of a process, and for each trace we take the reflection of its label:

$$\mathbf{i}.(\sim P) = \mathbf{o}.P \qquad \mathbf{o}.(\sim P) = \mathbf{i}.P \qquad \mathbf{f}.(\sim P).t = \sim(\mathbf{f}.P.t)$$

Recall that $\mathsf{ABORT}(I, O)$ and $\mathsf{MIRACLE}(I, O)$ are reflections of each other. Furthermore, taking the reflection of processes reverses the partial order:

$$P \sqsubseteq Q \;\Rightarrow\; {\sim}Q \sqsubseteq {\sim}P$$

In Figure 3.10 we illustrate the reflection of the WIRE. By means of reflection we swapped inputs and outputs, and we exchanged the $\top$ and the $\perp$ states. Observe that the WIRE and the reflection of the WIRE form a safe network. That is, whenever the WIRE is ready to produce an output, its reflection is ready to receive that output and, vice versa, whenever the reflection of the WIRE is ready to produce an output, the WIRE is ready to receive it.

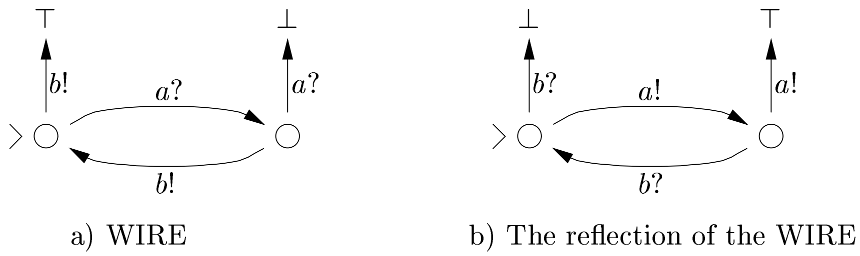


Figure 3.10: The reflection of the WIRE

3.5 Safety

So far we made a number of informal remarks about process safety, but we have not defined the concept formally. In this section we revisit the concept of safety, we give a formal definition, and we establish a connection between the refinement relation and safety.

Informally, we say that a process network is safe when each output that can be produced also can be accepted by all receivers. Recall that a process product representing a network contains information on failures that may occur. More precisely, a transition on an output port leading from a legal state to the $\perp$ state points to a safety violation. Consequently, we check the safety of a process by searching for output transitions leading from a legal state to the $\perp$ state.

In the definition below, juxtaposition denotes set concatenation.

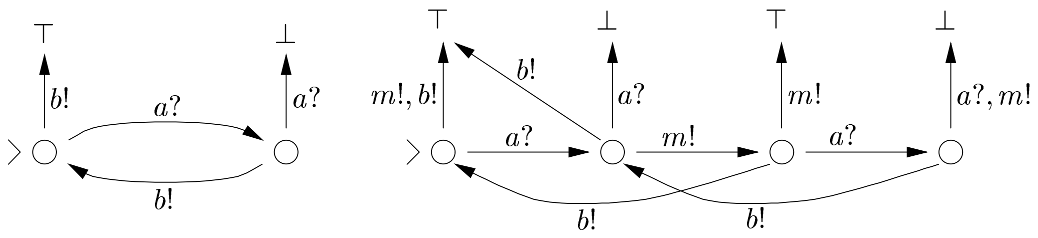
Definition 3.5.1 (Safety)

Process P is safe, denoted by $\boldsymbol{safe}.P$, if the following holds:

$$\boldsymbol{safe}.P \equiv \Big(\forall t :\ t \in (\mathbf{l}.P)(\mathbf{o}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.t \neq \perp\Big)$$

Notice that $\mathsf{ABORT}(I, O)$ is not safe, but $\mathsf{MIRACLE}(I, O)$ is a safe process.

The process of Figure 3.11a is safe, but the process of Figure 3.11b is not safe, because a legal trace can lead to a failure: Trace $a\,m\,a\,m$ leads to the $\perp$ state, because one WIRE inside the network can produce an output while another WIRE cannot receive that input.



a) WIRE b) A network of two WIRES

Figure 3.11: A safe and an unsafe process

A safe process is not immune to misuse by its environment. A malign environment can cause a failure in a safe process by providing an input when the process is not ready to receive that input. On the other hand, if the environment never violates the specification of the safe process, the process never fails by itself. For example, if a safe process represents a network of processes, then no process in the network produces an output while some other process cannot receive that output.

3.6 An alternative characterization of refinement

Next we address the connection between the refinement relation and the definition of safety. The theorem we prove below provides the following view of the refinement relation: If $P \sqsubseteq Q$, then we call process P the specification and process Q an implementation. The reflection of the specification gives us the environment in which an implementation must be able to operate safely. The idea for using the reflection of a specification as an environment for an implementation comes from [Ebergen, 1989, Ebergen, 1991] and was also applied in [Dill, 1989], [Verhoeff, 1994] and [Negulescu, 1998].

Theorem 3.6.1 (Refinement and safety)
Let P and Q be processes such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P = \mathbf{o}.Q$. Then

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \boldsymbol{safe}.({\sim}P \times Q)$$

Proof: We have to prove for processes P and Q, such that $\mathbf{i}.P = \mathbf{i}.Q$, and $\mathbf{o}.P = \mathbf{o}.Q$, $P \sqsubseteq Q$ is equivalent to $\mathbf{safe}.({\sim}P \times Q)$.

$$\begin{array}{ll} & \mathbf{safe}.({\sim}P \times Q) \\ \Leftrightarrow & \{ \text{ Lemma 3.6.3, } \mathbf{a}.({\sim}P \times Q) = \mathbf{o}.({\sim}P \times Q) \ \} \\ & (\forall t : \ t \in (\mathbf{a}.P)^* : \ \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \bigcirc) \\ \Leftrightarrow & \{ \ \mathbf{i}.P = \mathbf{i}.Q, \ \mathbf{o}.P = \mathbf{o}.Q, \text{ Lemma 3.6.4 } \} \\ & (\forall t : \ t \in (\mathbf{a}.P)^* : \ {\sim}\mathbf{f}.P.t \times \mathbf{f}.Q.t \sqsupseteq \bigcirc) \\ \Leftrightarrow & \{ \text{ Lemma 3.6.2 } \} \\ & (\forall t : \ t \in (\mathbf{a}.P)^* : \ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t) \end{array}$$

$\Leftrightarrow \quad \{$ Definition 2.3.1 — process refinement $\}$

$\quad P \sqsubseteq Q$

$\square$

We used the following three lemmas in the proof of Theorem 3.6.1. See Appendix C for proofs.

Lemma 3.6.2

The following holds for labels λ and γ:

$$(\sim\lambda \times \gamma \sqsupseteq \bigcirc) \Leftrightarrow (\lambda \sqsubseteq \gamma)$$

Lemma 3.6.3 *For process P, where $\mathbf{a}.P = \mathbf{o}.P$, we have:*

$$\boldsymbol{safe}.P \Leftrightarrow (\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \sqsupseteq \bigcirc)$$

Lemma 3.6.4

Let P and Q be processes, such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P \subseteq \mathbf{o}.Q$. Then,

$$(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.(\sim P \times Q).t \sqsupseteq \bigcirc) \Leftrightarrow (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.\sim P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \bigcirc)$$

Theorem 3.6.1 implies that the definition of safety satisfies Equation 2.12:

$$P \sqsubseteq Q \Leftrightarrow \mathbf{correct}.(\sim P \times Q)$$

That is, the definition of safety can indeed be used as a correctness condition in our model. This means, for example, that we can apply Theorem 2.5.1 and look at the refinement relation in the light of testing:

$$P \sqsubseteq Q \;\Leftrightarrow\; \big(\forall R:\; R \in \mathcal{PROC}(\mathbf{o}.P, \mathbf{i}.P) :\; \mathbf{safe}.(P \times R) \Rightarrow \mathbf{safe}.(Q \times R)\big)$$

The equivalence above states that, if $P \sqsubseteq Q$, then process Q operates safely in any environment R in which process P operates safely.

3.7 Hiding

Hiding is the operator for concealing a set of output ports of a process. In particular, when we want to determine whether a network is an implementation of a specification, we first compute the product of all processes in the network. Then we hide "internal ports" that are not present in the specification. The result is a process with the same port structure as the specification. Thus, we can then check whether the refinement relation holds.

Recall from Chapter 2 that we defined hiding for process P and for set of output symbols A:

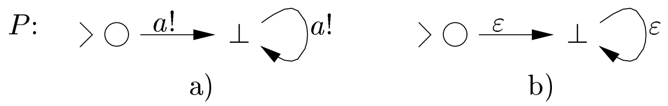
$$\begin{array}{lcl}
\mathbf{i}.|[A :: P|] & = & \mathbf{i}.P \\
\mathbf{o}.|[A :: P|] & = & \mathbf{o}.P - A \\
\mathbf{f}.|[A :: P]|.t & = & \big(\sqcap\, s :\; s \in (\mathbf{a}.P)^* \;\wedge\; s \downarrow (\mathbf{a}.P - A) = t :\; \mathbf{f}.P.s\big)
\end{array}$$

3.7.1 Examples of hiding

Below we provide a number of small examples that illustrate various aspects of hiding. Situations highlighted by our examples can arise as part of hiding output ports in a more complex process. In our examples we often mark transitions on hidden ports as

ε-transitions. We find such a transformation helpful in computing the result of hiding according to Definition 2.6.1.

We start with the process that fails on any output it produces. Figure 3.12a shows such a process. If we hide the output port of this process, then the result of hiding is ABORT.

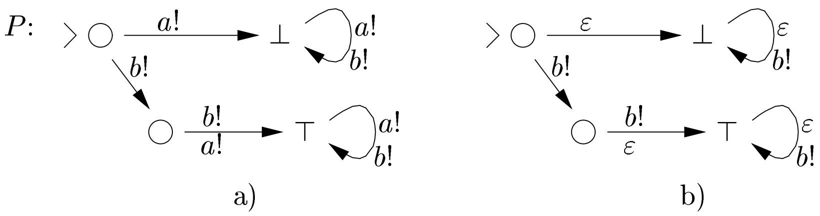


$|[a :: P]| = \mathsf{ABORT} = \; > \perp$

c)

Figure 3.12: A self-failing process

We can explain this result by means of the original definition of hiding. Figure 3.12b shows process P after hidden transitions have been labeled as epsilon transitions. Notice that we can reach the $\perp$ state via a path of ε-transitions starting in the initial state. Thus, $\mathbf{f}.|[a :: P]|.\varepsilon = \perp$ and $|[a :: P]| = \mathsf{ABORT}$.



$|[a :: P]| = \mathsf{ABORT}(\emptyset, \{b\})$

c)

Figure 3.13: A process that fails after a "hidden" transition

Process P shown in Figure 3.13a can reach a failure by producing a transition on port a, but will not fail on producing a transition on port b. What is the result of hiding port a? In Figure 3.13b we show process P, after we have labeled transitions on port a as ε transitions. There is an ε-path from the initial state to the $\perp$ state. Consequently,

$\mathbf{f}.|[a :: P]|.\varepsilon = \bot$, and the result of hiding is ABORT.

In the next example we consider a process that has a legal initial state, but all transitions lead to the $\top$ state. Figure 3.14a shows such a process P. In Figure 3.14b we show process P after all hidden transitions have been marked as ε-transitions. Notice that $\mathbf{f}.P.\varepsilon = \bigcirc$ and the only transition leaving the initial state is a transition on port a that leads to the $\top$ state. Therefore, $\mathbf{f}.|[A :: P]|.\varepsilon = \bigcirc \sqcap \top = \bigcirc$, as shown in Figure 3.14c.

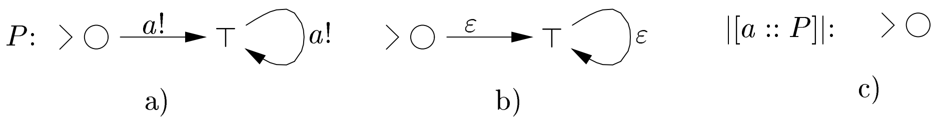


Figure 3.14: Hiding transitions to $\top$ state

This example indicates that the result of hiding can only be $\mathsf{MIRACLE}(I, O - A)$ when we start with process $\mathsf{MIRACLE}(I, O)$ and hide ports in set A.

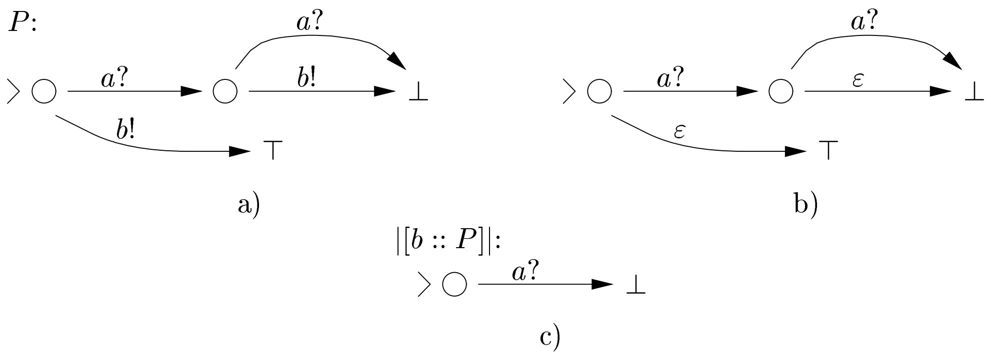


Figure 3.15: A process that aborts after receiving an input

Figure 3.15a shows a process that can fail on producing an output on port b after receiving an input on port a. In Figure 3.15b we show the process after the transitions on port b have been marked as ε transitions. Notice that the initial state is labeled with $\bigcirc$, and there is no ε-path that brings us from the initial state to the $\bot$ state in Figure

3.15b. Thus, $\mathbf{f}.|[b :: P]|.\varepsilon = \bigcirc$. Furthermore, after receiving an input on port a, we have an ε-transition leading to the $\perp$ state. Thus, $\mathbf{f}.|[b :: P]|.a = \perp$, as shown in Figure 3.15c.

The example from Figure 3.15 demonstrates how hiding can bring a failure to the surface: Process P fails after two inputs on port a have been received. Process $|[A :: P]|$, on the other hand, fails immediately after receiving one input on port a, because of an "internal" transition leading to the $\perp$ state.

In Figure 3.16a we show a process that fails after producing an output on port b after the process has produced an output on port a. Figure 3.16b shows the state graph after b-transitions have been marked as ε-transitions. Notice that, following an output on port a, there is an ε-transition leading to the $\perp$ state. Thus, $\mathbf{f}.|[A :: P]|.a = \perp$, as shown in Figure 3.16c.

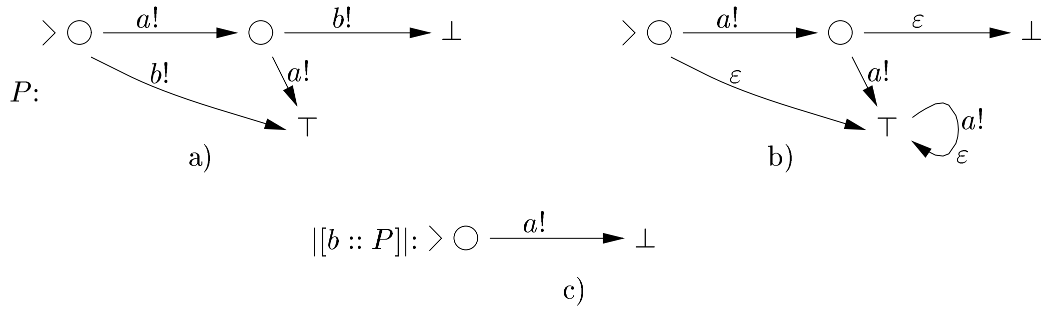


Figure 3.16: A process that aborts after producing an output

The following example shows a difference between our hiding and the implementation of hiding in Dill's verification tool [Dill, 1989]. Dill's implementation of hiding is often called "pruning". When pruning a state graph, we look for "hidden" failures and then we backtrack along the trace leading to the failure until we find the last input action. This input action is then forbidden by labeling the trace leading to the input action a failure. For more detail we refer the reader to [Dill, 1989], pages 60-61, and 104-105.

Figure 3.17a shows a process where a failure on port c follows an output on port b.

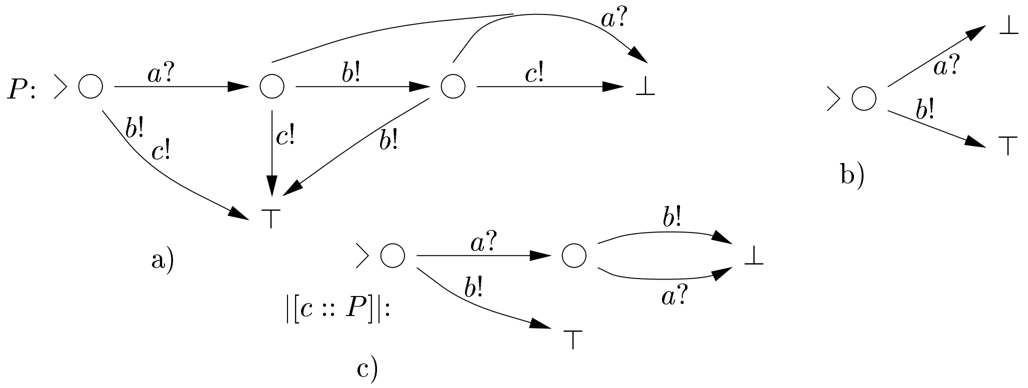


Figure 3.17: (b) pruning transitions on port c, versus (c) hiding port c

The output on port b follows an input on port a. Our intention is to hide port c. If we follow the strategy of pruning, we obtain the process of Figure 3.17b. Notice that we forbid the input on port a, thus we avoid the path that may lead to a failure.

Figure 3.17c shows the effect of hiding port c. We allow the input on port a, but the subsequent output on port b is considered a failure. Recall that in our partial order we have $\perp \sqsubseteq \bigcirc$, thus the result of hiding is larger than the result of pruning.

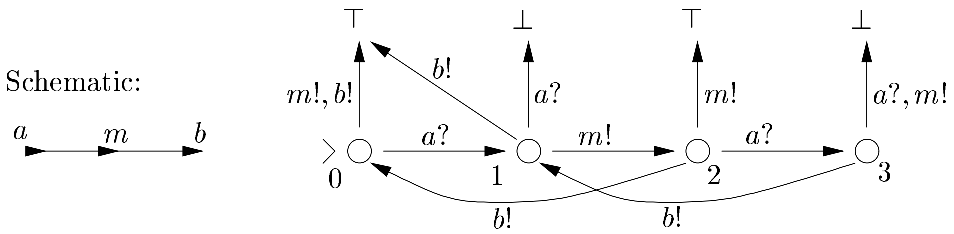


Figure 3.18: The product of two WIREs

In the next example we take a product of two WIREs and show how to hide the connection between them. Figure 3.18 shows the state graph of the product of two WIREs. One WIRE has input port a and output port m, and the other WIRE has input port m and output port b. The graph from Figure 3.18 is the same as the graph in Figure 3.8.

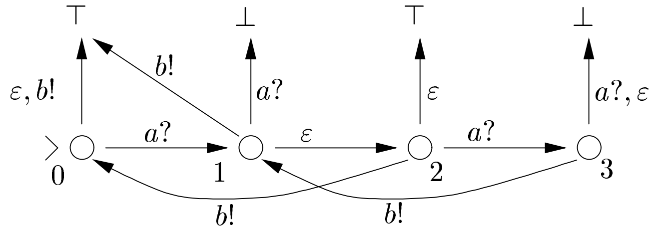


Figure 3.19: Hiding the connection between WIREs: Hidden transitions marked by ε

Figure 3.19 shows the state graph where we converted all transitions on m into ε-transitions. Notice that an ε-transition leads from state 3 to the $\perp$ state. Furthermore, we can only reach state 3 by an input transition on port a, which leads from state 2 to state 3. We learned in the example of Figure 3.15 that such an input transition is effectively disabled, because it leads to the $\perp$ state after hiding. This means that the product of two WIREs, with the connection between the WIREs hidden, cannot accept two inputs on port a in a row. Rather, the environment must wait for an output on port b before providing the next input on port a.

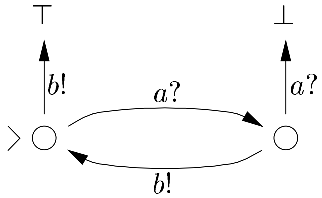


Figure 3.20: Hiding the connection between two WIREs: Final result

Figure 3.20 shows the result of hiding the connection between two WIREs. Not surprisingly, the result is a specification of a WIRE, which we first gave in Figure 3.1.

3.8 Properties of hiding

The following theorem provides an alternative characterization of hiding and helps us with an intuitive understanding of hiding:

Theorem 3.8.1

Let P be a process and let $A \subseteq \mathbf{o}.P$. Then,

$$|[A :: P]| = \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge \boldsymbol{safe}.(\sim R \times P) : R \big) \tag{3.6}$$

In words, $|[A :: P]|$ is the greatest process R with the input alphabet $\mathbf{i}.P$ and with the output alphabet $\mathbf{o}.P - A$, such that the reflection of R operates safely with P. Notice that the safety of $\sim R \times P$ must hold even though process $\sim R$ does not interact with any of the outputs that process P produces on ports from A.

A proof of Theorem 3.8.1 is as follows:

Proof:

$$\begin{array}{ll}
 & \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge \mathbf{safe}.(\sim R \times P) : R \big) \\
= & \quad \{ \text{ Lemma 3.8.2 } \} \\
 & \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge (\, R \sqsubseteq |[\, A :: P \,]| \,) : R \big) \\
= & \quad \{ \text{ Calculus, Theorem 2.3.3 -- } (\mathcal{PROC}(\mathbf{i}.P, (\mathbf{o}.P - A)), \sqsubseteq) \text{ is a complete lattice } \} \\
 & |[\, A :: P \,]|
\end{array}$$

$\square$

In the proof of Theorem 3.8.1 we used the following lemma. See Appendix C for the proof.

Lemma 3.8.2

Let P and R be processes such that $\mathbf{i}.R = \mathbf{i}.P$ and $\mathbf{o}.R = \mathbf{o}.P - A$ for some set A. Then

$$\boldsymbol{safe}.(\sim R \times P) \;\; \Leftrightarrow \;\; R \sqsubseteq |[\, A :: P \,]|$$

By applying Property 2.3.6, we can derive yet another characterization of hiding:

$$
\begin{array}{ll}
& |[\, A :: \; P \,]| \\
= & \{ \text{ Theorem 3.8.1 } \} \\
& \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = \mathbf{o}.P - A \wedge \mathbf{safe}.({\sim}R \times P) : R \big) \\
= & \{ \text{ Write } S = {\sim}R \text{ } \} \\
& \big(\sqcup S : \mathbf{i}.S = \mathbf{o}.S - A \wedge \mathbf{o}.S = \mathbf{i}.P \wedge \mathbf{safe}.(S \times P) : {\sim}S \big) \\
= & \{ \text{ Lemma 3.8.3 } \} \\
& {\sim}\big(\sqcap S : \mathbf{i}.S = \mathbf{o}.S - A \wedge \mathbf{o}.S = \mathbf{i}.P \wedge \mathbf{safe}.(S \times P) : S \big)
\end{array}
$$

In the result of hiding ports A in process P, we really are looking for an environment in which process P operates safely. The environment is blind to the events on the ports in set A. The reflection of the least, or the "least safe" such environment is the result of hiding:

$$
|[A :: P]| = {\sim}\big(\sqcap R : \mathbf{i}.R = \mathbf{o}.P - A \wedge \mathbf{o}.R = \mathbf{i}.P \wedge \mathbf{safe}.(R \times P) : R \big) \tag{3.7}
$$

In the proof of Equation 3.7, we used the following lemma.

Lemma 3.8.3

Let $\mathcal{A}$ be a set of processes. Then,

$$
\sqcup\{P :\ P \in \mathcal{A} :\ P\} \quad = \quad {\sim}\sqcap\{P :\ P \in \mathcal{A} :\ {\sim}P\}
$$

See Appendix C for the proof. Without proof we mention that the dual to the property expressed in Lemma 3.8.3 also holds:

$$
\sqcap\{P :\ P \in \mathcal{A} :\ P\} \quad = \quad {\sim}\sqcup\{P :\ P \in \mathcal{A} :\ {\sim}P\}
$$

Hiding enjoys all the properties we proved in Chapter 2: Ports can be hidden in an arbitrary order, hiding is monotonic and hiding distributes over the product. These properties hold, because the safety labels satisfy the properties required by the general ECF model.

Hiding does not affect the safety of a process. More precisely, if we start with a safe process P and we hide output symbols from set A, then the result of hiding, $|[A :: P]|$, is also a safe process.

Theorem 3.8.4

For process P and set $A \subseteq \mathbf{o}.P$, we have

$$\boldsymbol{safe}.P \quad \Leftrightarrow \quad \boldsymbol{safe}.|[A :: P]|$$

Proof: We assume $\mathbf{safe}.P$ and we prove, for trace $t \in \mathbf{l}.|[A :: P]|.\mathbf{o}.|[A :: P]| \cup \{\varepsilon\}$, that $\mathbf{f}.|[A :: P]|.t \neq \bot$.

First assume $t = \varepsilon$. Then, $\mathbf{f}.|[A :: P]|.t = (\sqcap s : s \in A^* : \mathbf{f}.P.s)$. From $\mathbf{safe}.P$ it follows that, for every $s \in (\mathbf{o}.P)^*$, $\mathbf{f}.P.s \neq \bot$. Furthermore, as $A \subseteq \mathbf{o}.P$, we conclude $\mathbf{f}.|[A :: P]|.\varepsilon \neq \bot$.

Now consider $\mathbf{f}.|[A :: P]|.t$, where $t = t'a$, such that $t' \in \mathbf{l}.|[A :: P]|$ and $a \in \mathbf{o}.P - A$. Then, $\mathbf{f}.|[A :: P]|.t'a = (\sqcap s', s'' : s'' \in A^* \wedge s'as'' \downarrow (\mathbf{a}.P - A) = t'a : \mathbf{f}.P.s'as'')$. What can we say about $\mathbf{f}.P.s'as''$ from the equation above? First we observe that $s' \downarrow (\mathbf{a}.P - A) = t'$. By assumption, $t' \in \mathbf{l}.|[A :: P]|$, thus $\mathbf{f}.P.s' \neq \bot$. Now we know that $s'a \in (\mathbf{l}.P)(\mathbf{o}.P)$ and from $\mathbf{safe}.P$ it follows that $\mathbf{f}.P.s'a \neq \bot$. Thus $s'a \in \mathbf{l}.P$. Because $s'' \in (\mathbf{o}.P)^*$, we can see that $s'as'' \in (\mathbf{l}.P)(\mathbf{o}.P)$. From $\mathbf{safe}.P$ it follows that that $\mathbf{f}.P.s'as'' \neq \bot$. Hence, $\mathbf{f}.|[A :: P]|.t'a \neq \bot$.

Now we prove the implication $\mathbf{safe}.|[A :: P]| \Rightarrow \mathbf{safe}.P$:

$$
\begin{aligned}
& \textbf{safe}.|[A :: P]| \\
\Rightarrow \quad & \{ \text{ Definition 3.5.1 — safety } \} \\
& \big(\forall t : t \in \mathbf{l}.|[A :: P]|\mathbf{o}.|[A :: P]| \cup \{\varepsilon\} : \mathbf{f}.|[A :: P]|.t \neq \bot \big) \\
\Rightarrow \quad & \{ \text{ Definition 2.6.1 — hiding } \} \\
& \big(\forall t : t \in \mathbf{l}.|[A :: P]|\mathbf{o}.|[A :: P]| \cup \{\varepsilon\} \\
& \qquad : (\sqcap s : s \in \mathbf{a}.P^* \wedge s \downarrow (\mathbf{a}.P - A) = t : \mathbf{f}.P.s) \neq \bot \big) \\
\Rightarrow \quad & \{ \text{ Calculus } \} \\
& \big(\forall t : t \in \mathbf{l}.|[A :: P]|\mathbf{o}.|[A :: P]| \cup \{\varepsilon\} \\
& \qquad : (\forall s : s \in \mathbf{a}.P^* \wedge s \downarrow (\mathbf{a}.P - A) = t : \mathbf{f}.P.s \neq \bot) \big) \\
\Rightarrow \quad & \{ A \subseteq \mathbf{o}.P; \text{ Calculus } \} \\
& (\forall s : s \in (\mathbf{l}.P)(\mathbf{o}.P) \cup \{\varepsilon\} : \mathbf{f}.P.s \neq \bot) \\
\Rightarrow \quad & \{ \text{ Definition 3.5.1 — safety } \} \\
& \textbf{safe}.P
\end{aligned}
$$

□

3.9 Factorization Theorem

Besides the testing view of the refinement relation, the notion of correctness also allows us to apply ECF processes to the following design problem from [Verhoeff, 1994]. When we are looking for an implementation of a given specification P, we can often guess at least one process, call it Q, that could be a part of an implementation. The question remains, given specification P and process Q, can we find process R such that

$$P \sqsubseteq Q \times R$$

This inequality, illustrated in Figure 3.21 is known as the Design Equation [Verhoeff, 1994, Mallon et al., 1999]: The Factorization Theorem proven below, gives a bound for process R, as shown in Figure 3.21.

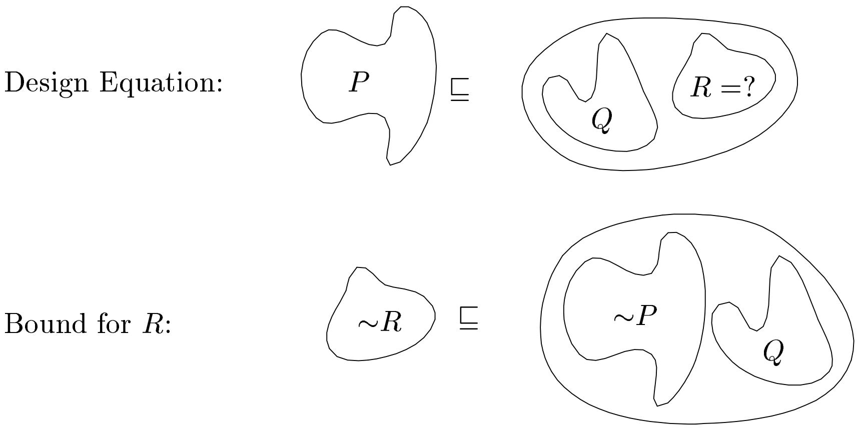


Figure 3.21: Factorization Theorem

Our process product leaves the internal connections visible. For this reason, our Design Equation and our Factorization Theorem take advantage of hiding in order to conceal the internal connections between the processes in the implementation. More precisely, if set A contains internal connections between processes Q and R, the Design Equation becomes

$$P \sqsubseteq |[A :: Q \times R]| \tag{3.8}$$

The parallel composition of [Verhoeff, 1994] and [Mallon et al., 1999] hides internal connections in a network. Thus, their formulation of Design Equation does not include hiding.

Theorem 3.9.1 (Factorization Theorem)

Let P, Q, and R be processes and let A and B be sets of symbols such that $A \cap \mathbf{a}.P = \emptyset$, *and* $B \cap \mathbf{a}.R = \emptyset$. *Then,*

$$P \sqsubseteq |[A :: Q \times R|] \;\;\Leftrightarrow\;\; \sim|[B :: (\sim P \times Q)]| \sqsubseteq R$$

Proof:

$P \sqsubseteq |[A :: Q \times R|]$

$\Leftrightarrow$ { Theorem 3.6.1 }

$\mathbf{safe}.(\sim P \times |[A :: Q \times R]|)$

$\Leftrightarrow$ { $A \cap \mathbf{a}.P = \emptyset$, Theorem 2.6.5 }

$\mathbf{safe}.|[A :: \sim P \times Q \times R]|$

$\Leftrightarrow$ { Theorem 3.8.4 }

$\mathbf{safe}.(\sim P \times Q \times R)$

$\Leftrightarrow$ { Theorem 3.8.4 }

$\mathbf{safe}.|[B :: \sim P \times Q \times R]|$

$\Leftrightarrow$ { $B \cap \mathbf{a}.R = \emptyset$, Theorem 2.6.5 }

$\mathbf{safe}.(|[B :: \sim P \times Q]| \times R)$

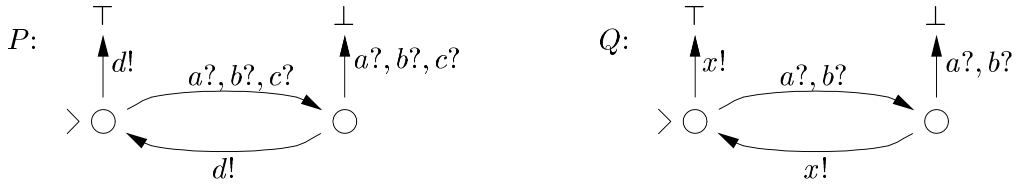
$\Leftrightarrow$ { Theorem 3.6.1 }

$\sim|[B :: \sim P \times Q]| \sqsubseteq R$

□

Sets A and B in the theorem above represent "internal" symbols. Namely, set A represents internal connections between Q and R, and set B represents internal connections between $\sim P$ and Q.

Let us illustrate an application of the Factorization theorem through an example. Let P be a three-input MERGE with inputs a, b, and c and with the output d. A MERGE is a common asynchronous component that waits for an input on any of its input ports and then produces an output on its output port. One restriction on the behavior of the MERGE is that receiving an input and producing an output must alternate. That is, an environment of the MERGE must not provide the second input before the MERGE has responded to the first input. Figure 3.22a shows a specification of a three-input MERGE.



a) three-input MEGRE b) a guessed part of an implementation for P

Figure 3.22: The MERGE

We would like to implement a three-input MERGE with a two-input MERGE and with another, yet unknown, component. Following the Factorization theorem we set process Q to be a specification of a two-input MERGE shown in Figure 3.22b. Notice that process Q has inputs a and b and output x, which connects process Q to process R. Now we apply the Factorization Theorem. Figure 3.23a shows the result of computing $\sim P \times Q$. Symbols a and b represent internal connections between processes $\sim P$ and Q, thus we hide them. Figure 3.23b shows the result of hiding. Notice that the lower bound for process R is exactly a specification of a MERGE with inputs x and c, and output d. To reduce the clutter, the state graph in Figure 3.23a does not contain the $\top$ state: All missing output transitions lead to the $\top$ state and all missing input transitions lead to the $\perp$ state.

Similar applications of the Factorization Theorem are discussed in [Verhoeff, 1994] and [Mallon et al., 1999]. They demonstrate that making a "good" guess for component Q is vital for a successful application of the Factorization Theorem in a design process. For example, when we make a bad guess for process Q, the lower bound for process R may become equal to MIRACLE. Process Q of Figure 3.24 is an example of a bad initial guess

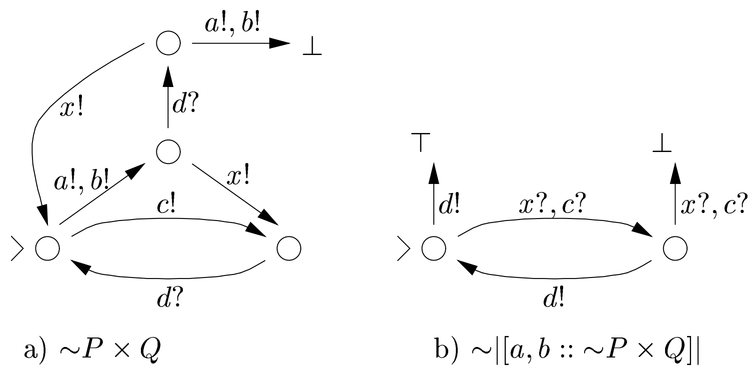


Figure 3.23: An application of the Factorization Theorem

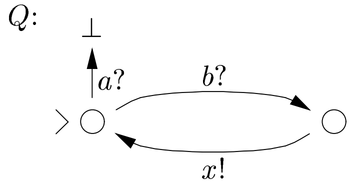


Figure 3.24: A bad initial guess

for a part of an implementation of a three-input MERGE shown in Figure 3.22a. Notice that process Q initially cannot receive an input on port a. It turns out that the lower bound given by the Factorization Theorem gives us for process R is equal to $\mathsf{MIRACLE}$. Because process $\mathsf{MIRACLE}$ has no known physical implementation, we have to restart the design process and look for a better initial guess.

3.10 Summary

In this chapter we instantiated the general ECF model with the set of labels that reflect safety of a process. In Chapter 2 all operations on processes were defined in an abstract manner. In this chapter, we were able to develop a concrete understanding of these operations by working through a number of examples.

We defined formally a safety correctness condition. This condition allowed us to give three equivalent definitions of the refinement relation: One, the refinement can be defined by comparing labels of traces. Two, the refinement can be defined by correctness of the network consisting from an implementation and the reflection of the specification, following the approach from [Ebergen, 1991]. Three, the refinement can be seen through the testing paradigm, following the approach from [Verhoeff, 1994].

Through examples we illustrated the hiding operation that allows us to conceal output ports of a process. Our operation differs from the previously used pruning [Dill, 1989] and projection [Ebergen, 1991]. In particular, hiding does not just remove the hidden actions, but it also takes into account potential safety problems that might occur when we hide ports. We provided an alternative definition of hiding, which allows us to build the intuitive understanding of the operation.

Finally, we proved the Factorization Theorem that provides an upper bound to the Design Equation from [Verhoeff, 1994] and [Mallon et al., 1999]. Our reformulated Design Equation takes advantage of hiding internal connections between processes. We demonstrated how the Factorization Theorem can be applied in a design technique.

Chapter 4

ECF treatment of progress

In Chapter 2 we developed a framework for modeling various properties of a process by attaching a label to each sequence of communication actions. In Chapter 3 we demonstrated how to apply the framework to studying safety of a process. In this chapter we take an application of the framework a step further, and we focus on progress properties of processes. Our progress condition detects possible deadlocks in a network of processes.

The model presented in this chapter is closely related to the XDI model of Tom Verhoeff [Verhoeff, 1994]. We use the same set of labels, and our interpretation of the labels is the same as in [Verhoeff, 1994]. There are, however, a number of important differences between our model and the model of [Verhoeff, 1994]. Most notably, we do not focus exclusively on delay-insensitive processes. Rather, we consider a larger domain of processes, the so-called speed-independent processes. Although the domain of speed-independent processes has its limitations [Bush and Josephs, 1996], it allows for a simpler formalism than the XDI model of [Verhoeff, 1994]. Another major difference is in the definition of process product, which models the joint operation of processes. In our model, we define the process product directly via the product of labels, while Verhoeff defines the product indirectly, either by using his correctness conditions, or by using the fix-point of a sequence of transformations.

Our progress criteria are closely related to finalization from [Negulescu, 1998]. There are, however, differences between the two models. Most notably, the process spaces model from [Negulescu, 1998] requires a separate specification of a process for each correctness condition that is to be addressed. For example, if we want to study safety and finalization, we are faced with producing two specifications, one addressing safety concerns and another dealing with finalization. ECF processes, on the other hand, capture safety and progress concerns at the same time.

Below, we introduce the new labels and we show a few examples of specifications of basic asynchronous components. On the basis of results from Chapter 2 we discuss, through examples, the refinement relation, the process product and the hiding operation.

4.1 Labels and processes

We use the set of labels from [Verhoeff, 1994]:

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

We use the same interpretation of the labels as [Verhoeff, 1994]: After a process has executed a trace that is labeled with ∇, the process guarantees progress by eventually producing an output on one of its ports. In other words, a trace that is labeled with ∇ puts the process to a *transient* state, which the process guarantees to leave. For this reason, 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 a process has executed an indifferent trace, the process does not guarantee that it will produce any output. The process also does not demand that its environment provide an input.

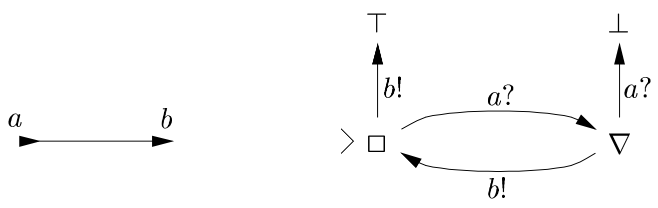
A trace labeled with Δ is called a *demanding* trace. After having executed such a trace, a process demands an input from its environment. A demanding trace puts the process to a demanding state. A process that is in a demanding state does not guarantee that it will produce an output, but it may be capable of producing an output.

Label $\perp$ indicates a failure. In a trace labeled with $\perp$ either the environment produced an input when the process was not ready, or the process failed by itself. The $\perp$ label indicates safety violations.

Finally, label $\top$ indicates traces that lead to a miracle. For example, if a process is not able to produce an output on some port, the trace that ends with that output is labeled with $\top$. Similarly, if the environment fails by itself while producing an input, the trace that ends with such an input is labeled by $\top$. One could say that label $\top$ indicates traces that can be executed only if a miracle happens.

4.1.1 Example processes

In order to illustrate our model, we give a number of specifications of asynchronous circuit components. We start with a specification of a WIRE, shown in Figure 4.1.



a) Schematic b) State graph

Figure 4.1: A specification of a WIRE

Initially, the WIRE is in a quiescent state labeled with $\Box$. In a quiescent state there are no progress obligations either for the WIRE or for the environment of the WIRE. Thus, nothing happens until an input on port a is received. After that, the WIRE moves to a transient state labeled with ∇. The WIRE cannot remain in a transient state, thus the WIRE will produce an output on port b. After the WIRE has produced an output on port b, the cycle resumes.

In the initial state, the WIRE cannot produce an output on port b. This is indicated by the transition from the initial state to the state labeled with $\top$. Once we reach the $\top$ state, there is no escape, because of the $\top$-persistence of processes. After receiving one input on port a, the WIRE cannot receive another input until it has produced an output on port b. That is, trace aa leads from the initial state to the $\perp$ state, indicating that the WIRE fails on that trace. Consequently, the environment is not allowed to provide another input until the WIRE has responded to the previous input. Because of the $\perp$-persistence of processes, there is no escape from the $\perp$ state.

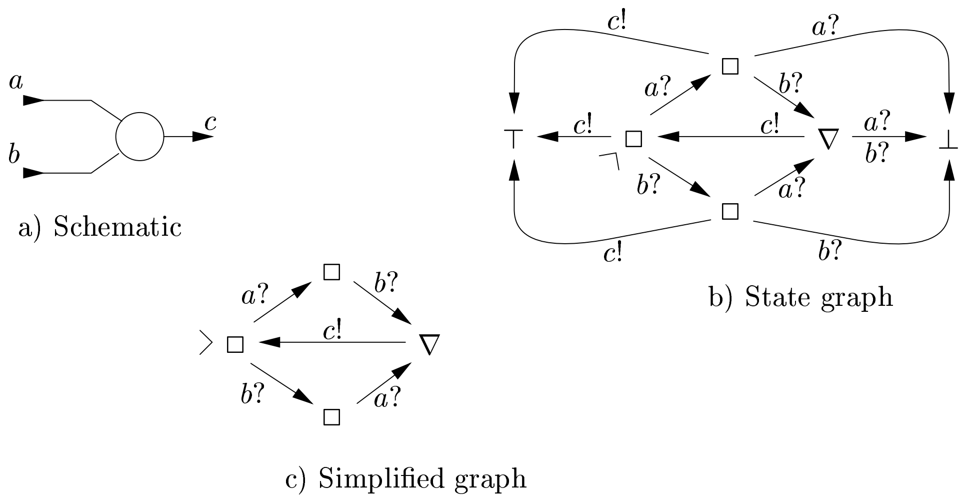
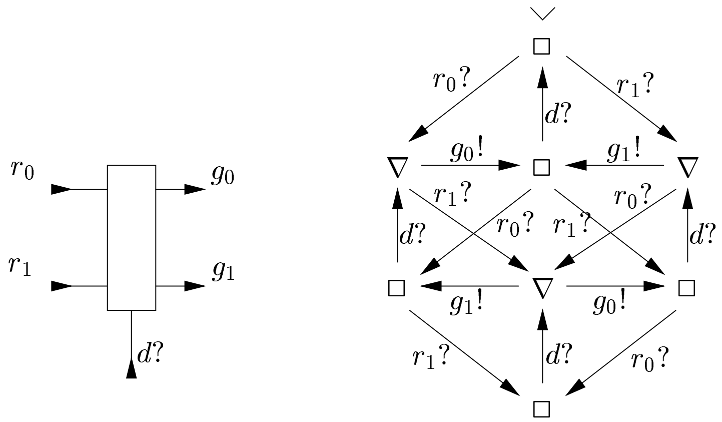


Figure 4.2: A specification of a JOIN

The JOIN is another common component. Figure 4.2 shows a schematic of the JOIN and a state-graph specification. In order to reduce clutter, we can omit the states labeled with $\perp$ and $\top$ label, which results in the state graph of Figure 4.2c. When the $\top$ and the $\perp$ state are not shown, we assume that all missing transitions on output symbols lead to the $\top$ state and all missing transitions on input symbols lead to the $\perp$ state.

Legal cyclic behavior of the JOIN consists of receiving an input on each of the two input ports, followed by producing an output on the output port. Notice that the JOIN is initially in a $\Box$ state. That is, neither the JOIN nor its environment have any obligation for producing a communication action. Furthermore, after the JOIN has received one of the inputs, the progress requirements do not change, because the JOIN is still in a state labeled with $\Box$. Only after *both* inputs have been received, the JOIN is in a ∇ state, where it is obliged to produce an output on port c.



a) Schematic b) State graph

Figure 4.3: A specification of an arbiter

Figure 4.3 shows a specification of an initialized SEQUENCER [Ebergen, 1991]. This component arbitrates between two requests, r_0 and r_1. The result of the arbitration is announced by either producing output g_0, which grants request r_0, or by producing output g_1, which grants request r_1. The arbiter can grant a subsequent request only after receiving a "done" input d. Notice that the arbiter is in a ∇ state only when it has received a request (or both of them) *and* a done signal d. Otherwise, the arbiter is in one of the states labeled with $\Box$, indicating that no action is either required or guaranteed.

4.2 Unhealthy processes and progress failures

The ∇ label indicates that a process guarantees progress after exhibiting a trace carrying this label. What if no outputs are possible after a trace labeled with ∇? Figure 4.4a shows such a process. All output transitions from the state labeled with ∇ lead to the

a) no outputs from the ∇ state b) no inputs from the Δ state

Figure 4.4: Unhealthy processes

$\top$ state. That is, the process cannot produce an output in the ∇ state. In other words, although the ∇ state guarantees progress, no progress can be made. We call such a process an *unhealthy* process [Mallon et al., 1999]. Unhealthiness of a process indicates a progress problem with the process specification.

A similar observation can be made for Δ states where a process cannot receive an input. The process demands from its environment to provide an input and, at the same time, the process is not capable of receiving any input from the environment. Such a process is shown in Figure 4.4b. Again, we call such a process unhealthy.

Just like unsafe processes are part of our process domain, so are unhealthy processes. Safety failures manifest themselves by the presence of legal traces followed by an output to the $\bot$ state. Progress failures, on the other hand, manifest themselves by means of the presence of unhealthy states. Including unhealthy processes in the process domain is in a contrast to [Verhoeff, 1994], where only healthy processes were considered.

Unhealthy processes can arise in various ways. We demonstrate below that an unhealthy process can arise as a result of the product of healthy processes, or as a result of hiding output symbols from a healthy process. Excluding unhealthy processes from our process domain and requiring that the process domain be closed under product and hiding would require more complicated definitions of these two operations. [Mallon et al., 1999] also includes unhealthy processes in an extension of the XDI model from [Verhoeff, 1994].

4.3 Refinement

We define a refinement relation on our new process domain by following the framework from Chapter 2. First, we require a partial order on labels, which we borrow from [Verhoeff, 1994]:

$$\bot \sqsubseteq \Delta \sqsubseteq \Box \sqsubseteq \nabla \sqsubseteq \top \tag{4.1}$$

Notice that the partial order above is a total order. This means that $\sqsubseteq$ induces a complete lattice on Λ. $\top$ is the greatest element and $\bot$ is the least element in the lattice. These are exactly the requirements that the framework from Chapter 2 sets for the partial order on labels. Consequently, we refer to Chapter 2 for the definition of a refinement relation on processes with the same alphabets:

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

From Chapter 2 we know that the refinement on processes is a partial order on

$\mathcal{PROC}(I,O)$ and that $(\mathcal{PROC}(I,O), \sqsubseteq)$ is a complete lattice with the least element $\mathsf{ABORT}(I,O)$ and with the greatest element $\mathsf{MIRACLE}(I,O)$.

The refinement relation tests whether one process implements another. That is, if $P \sqsubseteq Q$, we say that process Q implements process P. Later in this chapter we show formally that the refinement relation captures both safety and progress concerns. That is, if $P \sqsubseteq Q$, then process Q is at least as safe as process P in terms of safety considerations discussed in Chapter 3. Furthermore, if $P \sqsubseteq Q$, then 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. For this reason, the refinement relation used in this chapter is called the "progress refinement".

One justification for the partial order on labels is as follows: A failure trace can be implemented by a trace that does not fail. For that reason, $\perp$ is the least element in the partial order on labels. A trace that demands an input can be implemented by a trace that does not demand an input, but can still receive it. For that reason we have $\Delta \sqsubseteq \Box$. A trace that makes no progress requirements can be implemented by a trace that guarantees progress, hence $\Box \sqsubseteq \nabla$. Finally, producing a miracle is always better than anything else, thus $\top$ is the greatest of all the labels.

4.3.1 Examples of refinement

Below we give a number of examples that illustrate the progress refinement. We start with the processes of Figure 4.5. Process Q in Figure 4.5 specifies a WIRE. Process P, on the other hand, has different progress properties than the 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

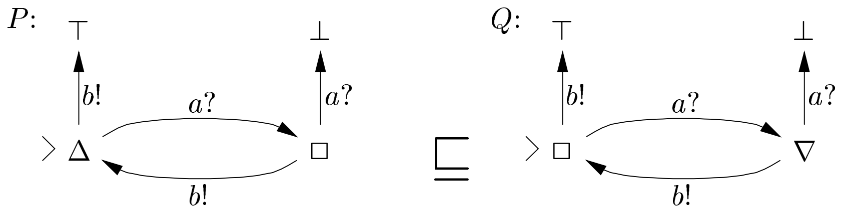


Figure 4.5: An example of refinement

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 $\perp$ state in process P also lead to the $\perp$ 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.

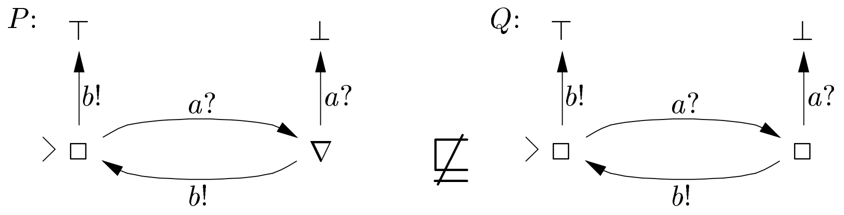


Figure 4.6: A violation of refinement of the WIRE

When progress is guaranteed by the specification and the proposed implementation does not guarantee progress, the refinement relation is violated. Process P of Figure 4.6 repeats the specification of the WIRE. Recall that WIRE guarantees that, after receiving an input on port a, an output on port b is produced. Process Q of Figure 4.6, however, cannot produce an output on port b after receiving an input on port b. That is, after receiving an input on port a, process P guarantees progress, but process Q does not, thus the refinement relation does not hold.

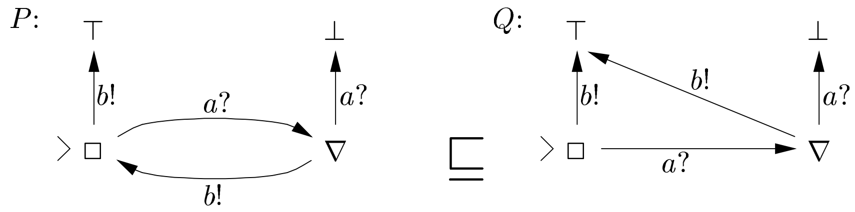


Figure 4.7: An unhealthy implementation

In Figure 4.7 we demonstrate that an unhealthy process can implement a healthy process. Process P of Figure 4.7 specifies a WIRE that guarantees to produce an output on port b after having received an input on port a. Receiving an input on port a, on the other hand, puts process Q to a ∇ state, where it cannot produce any outputs. Still, one can check by inspection that for any trace t, $\mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t$, thus $P \sqsubseteq Q$. This example indicates that our model allows that a healthy specification has an unhealthy implementation. Refining a healthy process by an unhealthy process is not really a problem, because in all practical examples we have considered, both the specification and the implementation were healthy processes.

Process P of Figure 4.8 is a specification of the SELECTOR. The SELECTOR is a component that, after receiving an input on port a, makes a nondeterministic choice between producing an output on port b or on port c. Process Q of Figure 4.8 is an implementation of the SELECTOR. Notice that process Q always produces an output on port b after having received an input on port a. This example shows that a nondeterministic choice can be implemented with a deterministic choice.

4.4 Reflection

When taking reflection we swap the input and the output alphabet of a process, and for each label we calculate its reflection according to Table 4.1, which we borrowed from

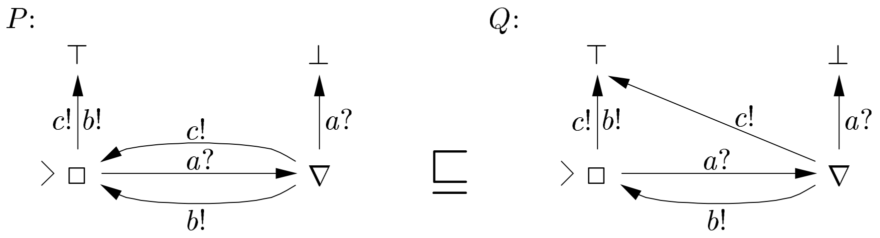


Figure 4.8: Refinement does not preserve nondeterminism

[Verhoeff, 1994].

λ	$\bot$	Δ	$\Box$	∇	$\top$
$\sim\lambda$	$\top$	∇	$\Box$	Δ	$\bot$

Table 4.1: Reflection on labels

The reflection is used to interchange the role of a process and the process's environment. Consider, for example, the reflection of ∇. Recall that ∇ denotes a state of a process in which it guarantees to produce an output. The environment of such a process can demand an input in that state, thus $\sim\nabla = \Delta$.

Table 4.1 satisfies the requirements from Chapter 2. Namely, reflection on labels reverses the partial order and reflection is its own inverse. Consequently, we use the definition of the process reflection as given for ECF processes in Chapter 2:

$$\mathbf{i}.{\sim}P = \mathbf{o}.P \qquad \mathbf{o}.{\sim}P = \mathbf{i}.P \qquad \mathbf{f}.{\sim}P.t = \sim(\mathbf{f}.P.t)$$

From Chapter 2 we know that the least and the greatest element of $\mathcal{PROC}(I, O)$ are reflections of each other.

$$\mathsf{MIRACLE}(I, O) = \sim\mathsf{ABORT}(O, I) \qquad \mathsf{ABORT}(I, O) = \sim\mathsf{MIRACLE}(O, I)$$

As we showed in Chapter 2, the reflection on processes turns the partial order on processes upside down:

$$P \sqsubseteq Q \;\Rightarrow\; \sim\!Q \sqsubseteq \sim\!P$$

Without proof we mention that both healthy and unhealthy processes are closed under reflection.

Our main application of reflection comes when we give an alternative characterization of the refinement relation. In order to reach that point we must first define the product of processes.

4.5 Product

$\times$	$\bot$	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 4.2: Product table

Table 4.2 defines the product on labels. We borrowed the product table from Chapter 7 of [Verhoeff, 1994], where it is shown that the product of labels is commutative, associative, idempotent, and monotonic with respect to the partial order. These are exactly the properties that we demanded in Chapter 2 for the product of labels.

We follow the definition of the process product given in Chapter 2, which simply lifts the product of labels to the product of processes:

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

The label of a state in $P \times Q$ is calculated as the product of state labels from processes P and Q, unless the $\top$ or the $\bot$ state has been reached. Once the product reaches the $\top$ or the $\bot$ state, the product stays in that state.

All properties of process product that hold for the abstract ECF model also hold for the progress model. Namely, the product of processes is associative, commutative, idempotent, and monotonic.

How can we interpret the entries in Table 4.2 and what is the intuition behind the product of processes? We take the same approach we took in Chapter 3: 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 requirements or obligations. The result of the label product tells us that the network of two processes also demands an input from its environment.

If one process is in ∇ 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. Because we give process obligations a priority over environment obligations, we have $\nabla \times \Delta = \nabla$. A similar argument explains why $\nabla \times \Box = \nabla$: If one process guarantees progress and the other process has no progress requirements and guarantees, then the network of these two processes guarantees progress.

We assume that, if one network component is in the $\top$ state, then the whole network is in the $\top$ state. For this reason we have $\top \times \lambda = \top$ for any label λ. Finally, if one network component has failed, which means that this component is in a $\perp$ state, then the network has failed as well. Thus, we have $\perp \times \lambda = \perp$ for any label λ other than $\top$. Notice that $\top \times \perp = \top$. That is, no failure can take place in the $\top$ state.

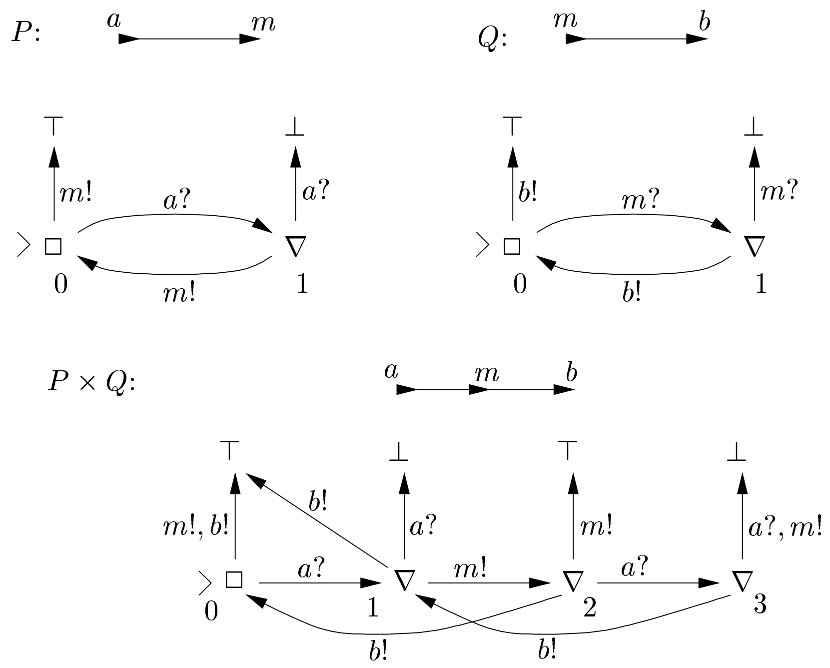


Figure 4.9: A product of two WIREs

Figure 4.9 shows a product of two WIREs. One WIRE has input port a and output port m, and the other WIRE has input port m the output port b. The structure of this state graph is the same as the structure of the product of WIREs in Chapter 3. The state labels in the graphs of Figure 4.9, however, tell us something 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.

The result of process product indicates that the network of two WIREs can fail if its environment provides two inputs on port a in a row without waiting for an output on port b. Notice that trace $amam$ leads to the $\perp$-state via an output transition from a legal state. This indicates that we can detect safety violations with the same strategy as in Chapter 3: We simply have to check whether there is an output transition leading from a legal state to the $\perp$-state.

The process product models a network of processes. The theorem below characterizes legal traces of a network: A legal trace for the network must be legal with respect to all individual components in the network. That is, a legal trace in a process product is a result of a product of legal traces in individual processes:

Theorem 4.5.1

The following holds for processes P and Q and any trace $t \in (\mathbf{a}.(P \times Q))^*$:

$$t \in \mathbf{l}.(P \times Q) \quad \Leftrightarrow \quad (t \downarrow \mathbf{a}.P) \in \mathbf{l}.P \ \wedge \ (t \downarrow \mathbf{a}.Q) \in \mathbf{l}.Q \tag{4.2}$$

Proof: Take $t \in (\mathbf{a}.(P \times Q))^*$.

$$
\begin{array}{ll}
 & t \in \mathbf{l}.(P \times Q) \\
\Leftrightarrow & \{ \text{ Equation 2.6 — legal traces } \} \\
 & \mathbf{f}.(P \times Q).t \notin \{\top, \perp\} \\
\Leftrightarrow & \{ \text{ Definition 2.4.1 — product } \} \\
 & (\mathbf{f}.P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)) \notin \{\perp, \top\} \\
\Leftrightarrow & \{ \text{ Table 4.2 } \} \\
 & \mathbf{f}.P.(t \downarrow \mathbf{a}.P) \notin \{\perp, \top\} \ \wedge \ \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \notin \{\perp, \top\} \\
\Leftrightarrow & \{ \text{ Equation 2.6 — legal traces } \}
\end{array}
$$

$(t \downarrow \mathbf{a}.P) \in \mathbf{l}.P \ \wedge \ (t \downarrow \mathbf{a}.Q) \in \mathbf{l}.Q$

□

A simple corollary of the theorem above is that an illegal trace in a process product must be illegal in at least one of the individual processes:

$$t \notin \mathbf{l}.(P \times Q) \quad \Leftrightarrow \quad (t \downarrow \mathbf{a}.P) \notin \mathbf{l}.P \ \vee \ (t \downarrow \mathbf{a}.Q) \notin \mathbf{l}.Q \tag{4.3}$$

4.6 Progress and refinement

In this section we define our progress condition and we show how the progress condition and safety conditions are related to progress refinement. We demonstrated in Figure 4.9 that in the progress model we can approach safety in the same way as we did in Chapter 3. Consequently, the definition of safety in the progress model is the same as the definition of safety in Chapter 3.

Definition 4.6.1 (Safety)
*We say that process P is safe, denoted by **safe**.P, if the following holds:*

$$\boldsymbol{safe}.P \equiv \Big(\forall t :\ t \in (\mathbf{l}.P)(\mathbf{o}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.(t) \neq \bot\Big) \tag{4.4}$$

In the safety model of Chapter 3 we established a simple relation between the refinement relation and the safety condition:

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \mathbf{safe}.({\sim}P \times Q)$$

In the progress model, this equivalence does not hold. Consider, for example, the processes of Figure 4.10. Let P be the specification of the WIRE in Figure 4.10a and let Q be a process of Figure 4.10b. We can see immediately that **safe**.($\sim P \times Q$) holds. However, $\mathbf{f}.P.a = \nabla$ and $\mathbf{f}.Q.a = \Box$; therefore, we know that Q does not refine P. The example demonstrates that safety of $\sim P \times Q$ alone is not sufficient for the progress refinement $P \sqsubseteq Q$ to hold.

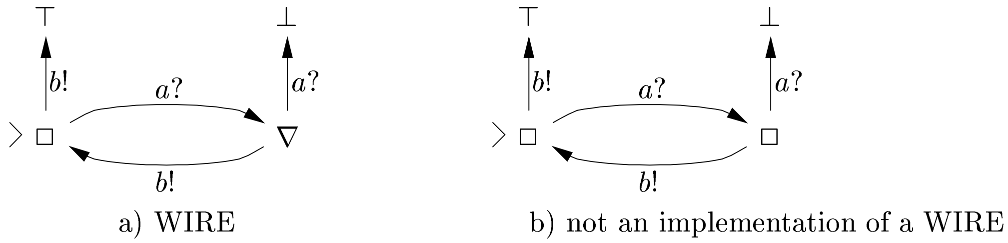


Figure 4.10: Refinement requires more than safety

What went wrong in the example above? Figure 4.11 shows the product $\sim P \times Q$. Notice that $\sim P \times Q$ has an empty set of input symbols, which means that the network consisting of processes $\sim P$ and Q is *closed.* The product $\sim P \times Q$, however, has a state labeled with Δ, which indicates that the network demands an input from its environment. That is, process $\sim P \times Q$ is unhealthy, demanding progress from its environment, while no environment can satisfy such a demand.

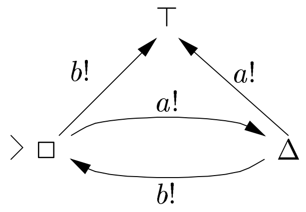


Figure 4.11: A Δ state indicates lack of progress

The example above illustrates a violation of our progress condition: There should be no traces labeled with Δ in $\sim P \times Q$. This condition is the same as the progress condition from [Verhoeff, 1994].

Definition 4.6.2 (Progress)

Process P satisfies the progress requirements, denoted by $\boldsymbol{prog}.P$*, if the following holds:*

$$\boldsymbol{prog}.P \;\equiv\; \big(\forall t :\; t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \neq \Delta\big)$$

The progress condition is intended to be used on processes with no input ports. If such a process fails to satisfy the progress condition, the process is unhealthy, which is an indication of a progress failure.

We emphasize that, when we apply the progress condition to a process that has input ports, a failure to meet the condition still implies a progress violation, *if the process is unhealthy.* If the process is healthy, the failure to meet the progress condition indicates that, at some point, the process may demand progress from its environment. We stipulate that demanding progress from an environment is too strong a requirement, thus such a process does not meet our progress condition.

In the example above we demonstrated that the safety condition alone does not suffice for an alternative characterization of the refinement relation. The progress condition alone is insufficient as well. One can check easily that process P from Figure 4.12 is not refined by process Q from the same figure: $\mathbf{f}.P.a = \top$, $\mathbf{f}.Q.a = \bot$, hence $\mathbf{f}.P.a \not\sqsubseteq \mathbf{f}.Q.a$ and $P \not\sqsubseteq Q$. On the other hand, product $\sim\!P \times Q$, shown in Figure 4.12, *does satisfy* the progress condition, because there is no Δ state in the graph for $\sim\!P \times Q$.

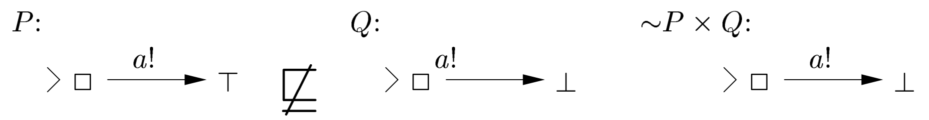


Figure 4.12: The progress condition alone is insufficient for the refinement

Notice that the product $\sim\!P \times Q$ does not satisfy our safety condition. This indicates that the safety and the progress condition together are sufficient to characterize the refinement relation.

Theorem 4.6.3 (Refinement characterization)

For processes P and Q we have

$$P \sqsubseteq Q \;\Leftrightarrow\; \boldsymbol{safe}.(\sim P \times Q) \wedge \boldsymbol{prog}.(\sim P \times Q)$$

Proof:

$$\begin{array}{ll}
 & \mathbf{safe}.(\sim P \times Q) \;\wedge \mathbf{prog}.(\sim P \times Q) \\
\equiv & \{\ \mathbf{i}.(\sim P \times Q) = \emptyset,\ \text{Lemma 4.6.4}\ \} \\
 & \big(\forall t :\ t \in (\mathbf{a}.(\sim P \times Q))^* :\ \mathbf{f}.(\sim P \times Q).t \sqsupseteq \square\big) \\
\Leftrightarrow & \{\ \mathbf{i}.P = \mathbf{i}.Q,\ \mathbf{o}.P = \mathbf{o}.Q,\ \text{Lemma 4.6.6}\ \} \\
 & (\forall t :\ t \in (\mathbf{a}.P)^* :\ \sim\mathbf{f}.P.t \times \mathbf{f}.Q.t \sqsupseteq \square) \\
\Leftrightarrow & \{\ \text{Lemma 4.6.5}\ \} \\
 & (\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t) \\
\Leftrightarrow & \{\ \text{Definition 2.3.1 — process refinement}\ \} \\
 & P \sqsubseteq Q
\end{array}$$

$\square$

In the proof of the theorem above we used the lemmas listed below. All the lemmas are proven in Appendix C.

Lemma 4.6.4

For process P, where $\mathbf{a}.P = \mathbf{o}.P$, we have:

$$\boldsymbol{safe}.P \;\wedge\; \boldsymbol{prog}.P \;\;\Leftrightarrow\;\; (\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \sqsupseteq \square\,)$$

Lemma 4.6.5

The following holds for labels λ and γ:

$$\lambda \sqsubseteq \gamma \;\;\Leftrightarrow\;\; ({\sim}\lambda \times \gamma) \sqsupseteq \Box$$

Lemma 4.6.6

Let P and Q be processes, such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P \subseteq \mathbf{o}.Q$. Then,

$$(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \Box) \;\;\Leftrightarrow\;\; (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \Box)$$

Theorem 4.6.3 implies that the conjunction of safety and progress conditions satisfies Equation 2.12. That is, the conjunction of safety and progress conditions can be used as a correctness criterion, as introduced in Section 2.5:

$$\mathbf{correct}.P \;\;\equiv\;\; \mathbf{safe}.P \,\wedge\, \mathbf{prog}.P \tag{4.5}$$

Now we can rewrite Theorem 4.6.3 as

$$P \sqsubseteq Q \;\;\Leftrightarrow\;\; \mathbf{correct}.({\sim}P \times Q) \tag{4.6}$$

We can apply Theorem 2.5.1 and look at the refinement relation in the light of testing:

$$P \sqsubseteq Q \;\;\Leftrightarrow\;\; \big(\forall R :\; R \in \mathcal{PROC}(\mathbf{o}.P, \mathbf{i}.P) \,:\, \mathbf{correct}.(P \times R) \Rightarrow \mathbf{correct}.(Q \times R)\big)$$

The equation above tells us that, if $P \sqsubseteq Q$, then process Q operates correctly in any environment where process P operates correctly. Notice that we restricted the alphabet of process R, which represents the environment, to match the alphabet of processes P

and Q.

Now we have three equivalent definitions of the refinement: One definition comes from the abstract ECF model; in another definition we take approach from [Ebergen, 1991], where we require that an implementation operates correctly together with the reflection of the specification; finally, the refinement can be seen from the testing point of view, which was the approach taken in [Verhoeff, 1994]. [Negulescu, 1998] also shows that the refinement relation in process spaces conforms to the testing paradigm.

4.7 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 a number of processes, where "internal" connections between these processes appear as output ports in the product. The output alphabet of the specification may 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 model presented in this chapter is an instantiation of the ECF model of Chapter 2. Thus, we take the definition of hiding directly from the ECF model. For process P and set $A \subseteq \mathbf{o}.P$, hiding output symbols from set A in process P is denoted by $|[A :: P]|$ and defined as

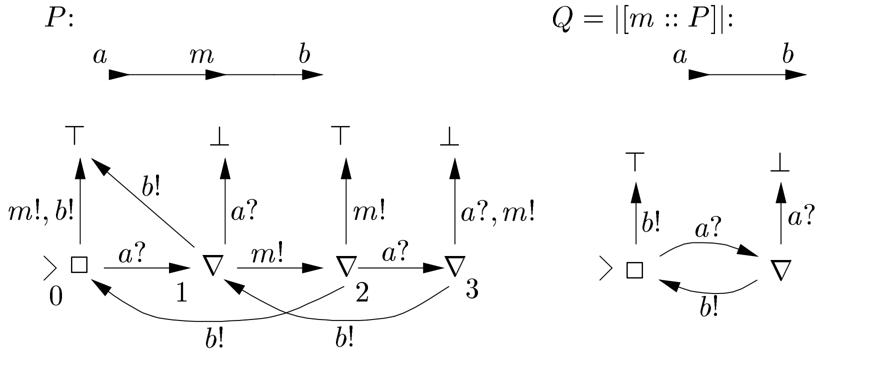
$$\begin{array}{lcl}
\mathbf{i}.|[A :: P]| & = & \mathbf{i}.P \\
\mathbf{o}.|[A :: P]| & = & \mathbf{o}.P - A \\
\mathbf{f}.|[A :: P]|.t & = & \big(\sqcap\, s :\ s \in (\mathbf{a}.P)^* \ \wedge\ s \downarrow (\mathbf{a}.P - A) = t :\ \mathbf{f}.P.s\big)
\end{array}$$

The result of hiding tells us what kind of behavior the environment can expect from process P when ports in set A are concealed. Suppose that process P has executed trace s. The environment of P has seen only communication actions in trace $t = s \downarrow (\mathbf{a}.P - A)$. Let

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

$X(t)$ is 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. 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 which 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 the events, the environment must assume that the state of process P is represented by the least label of all traces in $X(t)$. Consequently,

$$\mathbf{f}.|[A :: P]|.t = (\sqcap s : \ s \in X(t) : \ \mathbf{f}.P.s)$$



a) Before hiding m b) After hiding m

Figure 4.13: Hiding a connection between two WIREs

The definition of hiding is the same as in the safety model of Chapter 3, but hiding now also takes into account progress properties of a process. For example, consider the product of two WIREs, shown as process P in Figure 4.13a. The result of hiding output m is the specification of the WIRE, shown as process Q in Figure 4.13b. One can verify, by inspection, that for every trace $t \in (\mathbf{a}.Q)^*$ we have $\mathbf{f}.Q.t = \sqcap\{s : \ s \in X(t) : \mathbf{f}.P.s\}$. 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 $\mathbf{f}.P.aa = \bot$, we know that $\mathbf{f}.|[m :: P]|.aa = \bot$, which agrees with $\mathbf{f}.Q.aa = \bot$.

4.8 Hiding and unhealthy processes

The result of hiding can be an unhealthy process. Consider the "ticker" process of Figure 4.14a. After having received an input on port a, the ticker guarantees that it will keep producing outputs on port b.

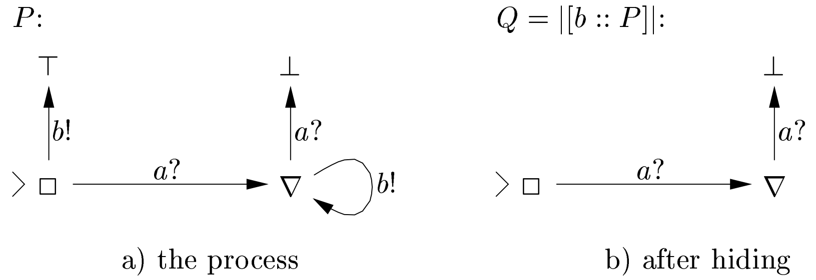


Figure 4.14: The ticker process

The result of hiding output b is unhealthy process Q shown in Figure 4.14b. After having received an input on port a, process Q is in a ∇ state, but no output is possible. Notice that the ∇ state appears in the result of hiding, because, after receiving an input on port a, the ticker is trapped in a cycle of "internal" symbols passing only through ∇ states. Without proof we mention that hiding outputs in a healthy process yields a healthy process if there is no ∇-cycle of "internal" symbols. This observation is in agreement with the interpretation of unhealthiness adopted in [Mallon et al., 1999].

The process of Figure 4.14b is unhealthy, because it has a ∇ state with no outputs possible. Can the result of hiding be a state graph that has a Δ state with no inputs possible? The answer is no, because we are hiding output ports only, thus all input transitions from the original state graph remain after hiding.

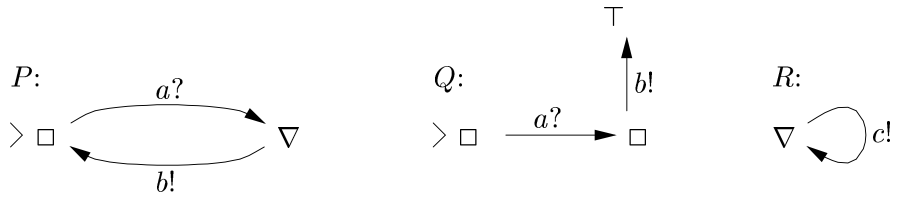


Figure 4.15: A prospective implementation of the WIRE

The following example illustrates how the result of hiding can indicate a progress problem with a network that is supposed to implement a specification. Let process P from Figure 4.15 be a specification that we are trying to implement with a network consisting of processes Q and R from Figure 4.15. Process P specifies a WIRE, process Q can receive an input on port a, but will never produce an output on port b, and process R keeps producing outputs on port c. Process R can be seen as a producer of an infinite sequence of ticks, which may occur at irregular intervals.

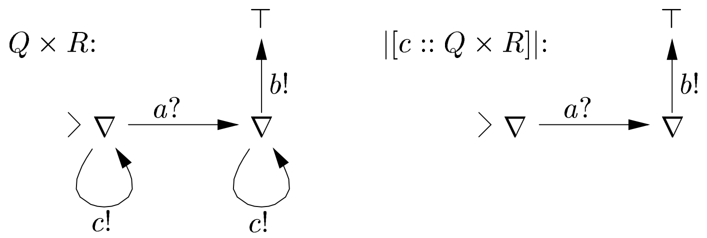


Figure 4.16: An unhealthy process reveals progress problems

We ask whether $P \sqsubseteq |[c :: Q \times R]|$ holds. Figure 4.16 shows process $|[c :: Q \times R]|$.

By inspection we can verify that process P can indeed be refined by $|[c :: Q \times R]|$. That is, a WIRE can be implemented with a process that never produces an output and a process that produces hidden ticks. [Negulescu and Brzozowski, 1995] uses this example in order to demonstrate that the progress condition from [Verhoeff, 1994] does not catch transient cycles of internal symbols. The same critique applies to our progress condition. One can observe, however, that process $|[c :: Q \times R]|$ is not healthy, although process $Q \times R$ is healthy. As observed above, unhealthiness introduced by hiding is an indication of an internal ∇-cycle of internal symbols. Consequently, we can use unhealthiness as an indication of a progress problem with our implementation.

4.9 Hiding and correctness

Recall that in the safety model hiding does not affect the correctness of a process. We proved in Theorem 3.8.4 that hiding output ports yields a safe process if and only if we started with a safe process: **safe**.$P \Leftrightarrow$ **safe**.$|[A :: P]|$. In the progress model, however, this equivalence does not hold in general. Process P of Figure 4.17 demonstrates that, in the progress model, **correct**.$|[A :: P]|$ does not imply **correct**.P. Notice that, for process P of Figure 4.17, **correct**.P does not hold, because the state graph corresponding to P contains a Δ state. On the other hand, **correct**.$|[b :: P]|$ holds. Without proof we mention that **correct**.P implies **correct**.$|[A :: P]|$ for any process P.

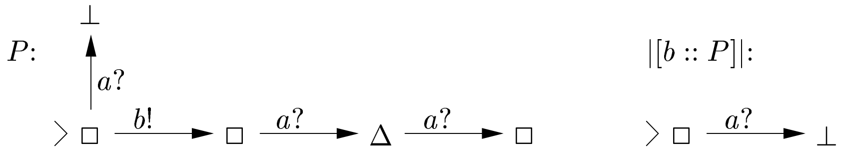


Figure 4.17: Hiding can mask incorrectness

Our main application of correctness is limited, however, to processes with no input ports, which we call "closed" processes. The name closed suggests that a process represents a network where every input port is connected to some output port. For any closed process P, **correct**.P is equivalent to **correct**.$|[A :: P]|$.

Theorem 4.9.1

For process P, such that $\mathbf{i}.P = \emptyset$*, and set* $A \subseteq \mathbf{o}.P$*, we have*

$$\boldsymbol{correct}.P \;\Leftrightarrow\; \boldsymbol{correct}.|[A :: P]|$$

Proof: We first recall Lemma 4.6.4, which tells us that, for a closed process P, one can simplify the definition of correctness:

$$\mathbf{correct}.P \;\Leftrightarrow\; \big(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \sqsupseteq \Box\big)$$

$$
\begin{array}{ll}
 & \mathbf{correct}.|[A :: P]| \\
\Leftrightarrow & \{\ \mathbf{i}.P = \emptyset,\ \text{Lemma 4.6.4}\ \} \\
 & \big(\forall t : t \in (\mathbf{a}.|[A :: P]|)^* : \mathbf{f}.|[A :: P]|.t \sqsupseteq \Box\big) \\
\Leftrightarrow & \{\ \text{Definition 2.6.1}\ \} \\
 & \big(\forall t : t \in (\mathbf{a}.|[A :: P]|)^* : (\sqcap s : s \in (\mathbf{a}.P)^* \wedge s \downarrow (\mathbf{a}.P - A) = t : \mathbf{f}.P.s) \sqsupseteq \Box\big) \\
\Leftrightarrow & \{\ \text{Calculus, } \sqsubseteq \text{ is a total order on } \Lambda\ \} \\
 & \big(\forall t : t \in (\mathbf{a}.|[A :: P]|)^* : (\forall s : s \in (\mathbf{a}.P)^* \wedge s \downarrow (\mathbf{a}.P - A) = t : \mathbf{f}.P.s) \sqsupseteq \Box\big) \\
\Leftrightarrow & \{\ \text{Calculus}\ \} \\
 & \big(\forall s : s \in (\mathbf{a}.P)^* : \mathbf{f}.P.s \sqsupseteq \Box)\big) \\
\Leftrightarrow & \{\ \text{Lemma 4.6.4},\ \mathbf{i}.P = \emptyset\ \} \\
 & \mathbf{correct}.P
\end{array}
$$

$\Box$

4.10 An alternate definition of hiding

The following theorem provides an alternative view of hiding. When we compute the result of hiding $|[A :: P]|$, we look for the greatest process R with input alphabet $\mathbf{i}.P$ and t output alphabet $\mathbf{o}.P - A$, such that the reflection of R operates correctly in the presence of process P. The correctness for $\sim R \times P$ must be guaranteed although process R does not have access to all ports of process P.

Theorem 4.10.1

Let P be a process and let $A \subseteq \mathbf{o}.P$. Then,

$$|[A :: P]| = \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge \boldsymbol{correct}.(\sim R \times P) : R \big) \tag{4.7}$$

We remark that Theorem 3.8.1 offers exactly the same characterization of hiding for the safety model of Chapter 3. The proofs of the two theorems follow the same strategy, with the only differences coming from different sets of labels used in the two models and from different definitions of correctness.

Proof:

$$\begin{aligned}
& \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge \mathbf{correct}.(\sim R \times P) : R \big) \\
= \quad & \{ \text{ Lemma 4.10.2 } \} \\
& \big(\sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge (R \sqsubseteq |[\, A :: P \,]| \,) : R \big) \\
= \quad & \{ \text{ Calculus, Theorem 2.3.3 -- } (\mathcal{PROC}(\mathbf{i}.P, (\mathbf{o}.P - A)), \sqsubseteq) \text{ is a complete lattice } \} \\
& |[\, A :: P \,]|
\end{aligned}$$

□

In the proof of Theorem 4.10.1 we used the following lemma. See Appendix C for the proof.

Lemma 4.10.2

Let P and R be processes such that $\mathbf{i}.R = \mathbf{i}.P$ *and* $\mathbf{o}.R = \mathbf{o}.P - A$ *for some set* A*. Then*

$$\boldsymbol{correct}.(\sim R \times P) \quad \Leftrightarrow \quad R \sqsubseteq |[\, A :: P \,]|$$

By applying Property 2.3.6, Theorem 4.10.1 can be rewritten as

$$\begin{array}{ll}
 & |[\, A ::\ P \,]| \\
= & \quad \{ \text{ Theorem 4.10.1 } \} \\
 & \big(\ \sqcup R : \mathbf{i}.R = \mathbf{i}.P \wedge \mathbf{o}.R = (\mathbf{o}.P - A) \wedge \mathbf{correct}.(\sim R \times P) : R \,\big) \\
= & \quad \{ \text{ Write } S = \sim R \ \} \\
 & \big(\ \sqcup S : \mathbf{i}.S = (\mathbf{o}.S - A) \wedge \mathbf{o}.S = \mathbf{i}.P \wedge \mathbf{correct}.(S \times P) : \sim S \,\big) \\
= & \quad \{ \text{ Lemma 3.8.3 } \} \\
 & \sim\!\big(\ \sqcap S : \mathbf{i}.S = (\mathbf{o}.S - A) \wedge \mathbf{o}.S = \mathbf{i}.P \wedge \mathbf{correct}.(S \times P) : S \,\big)
\end{array}$$

The derivation above leads to yet another characterization of hiding:

$$|[\, A :: P \,]| \quad = \quad \sim\!\big(\ \sqcap R : \mathbf{i}.R = (\mathbf{o}.P - A) \wedge \mathbf{o}.R = \mathbf{i}.P \wedge \mathbf{correct}.(R \times P) : R \,\big) \tag{4.8}$$

This equation provides a connection between the process product and the composition operation from [Verhoeff, 1994]. Verhoeff's composition implicitly hides internal connections between network processes. Furthermore, Verhoeff's composition is defined as the reflection of the least environment in which the network can operate without violating any of the correctness concerns. If we take processes P and Q and we assume that symbols in set A represent connections between P and Q, then $|[A :: P \times Q]|$ amounts to

$$\sim\!\big(\ \sqcap R : \mathbf{i}.R = (\mathbf{o}.(P \times Q) - A) \wedge \mathbf{o}.R = \mathbf{i}.(P \times Q) \wedge \mathbf{correct}.(R \times P \times Q) : R \,\big)$$

That is, $|[A :: P \times Q]|$ is the reflection of the least environment in which $P \times Q$ operates correctly, which matches the definition of composition from [Verhoeff, 1994].

4.11 Factorization Theorem

In Section 3.9 we found a lower bound for solutions of the Design Equation [Verhoeff, 1994]. Here we demonstrate that, in the progress model, a lower bound to the solution of the Design Equation can be calculated by the same expression as in the safety model.

In the Design Equation we start with process P and we guess that process Q will be a part of an implementation of process P:

$$P \sqsubseteq Q \times R$$

We are asked to find process R on the basis of processes P and Q, such that the Design Equation above is satisfied.

The parallel composition of [Verhoeff, 1994] and [Mallon et al., 1999] implicitly hides internal connections in a network. Our process product, on the other hand, leaves the internal connections visible. For this reason, our Design Equation and our Factorization Theorem include an explicit application of hiding. That is, we assume that set A contains the ports that form internal connections between processes Q and R. Our formulation of the Design Equation then becomes

$$P \sqsubseteq |[A :: Q \times R]|$$

Theorem 4.11.1 (Factorization Theorem)

Let P, Q, and R be processes and let A and B be sets of symbols such that $A \cap \mathbf{a}.P = \emptyset$, *and* $B \cap \mathbf{a}.R = \emptyset$. *Then,*

$$P \sqsubseteq |[A :: Q \times R|] \quad \Leftrightarrow \quad \sim|[B :: (\sim P \times Q)]| \sqsubseteq R$$

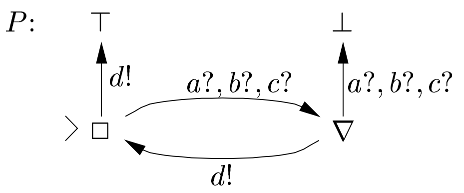
Proof:

$$\begin{array}{ll}
 & P \sqsubseteq |[A :: Q \times R|] \\
\Leftrightarrow & \quad \{ \text{ Equation 4.6 } \} \\
 & \mathbf{correct}.(\sim P \times |[A :: Q \times R]|) \\
\Leftrightarrow & \quad \{ A \cap \mathbf{a}.P = \emptyset, \text{ Theorem 2.6.5 } \} \\
 & \mathbf{correct}.|[A :: \sim P \times Q \times R]| \\
\Leftrightarrow & \quad \{ \sim P \times Q \times R \text{ is closed, Theorem 4.9.1 } \} \\
 & \mathbf{correct}.(\sim P \times Q \times R) \\
\Leftrightarrow & \quad \{ \sim P \times Q \times R \text{ is closed, Theorem 4.9.1 } \} \\
 & \mathbf{correct}.|[B :: \sim P \times Q \times R]| \\
\Leftrightarrow & \quad \{ B \cap \mathbf{a}.R = \emptyset, \text{ Theorem 2.6.5 } \} \\
 & \mathbf{correct}.(|[B :: \sim P \times Q]| \times R) \\
\Leftrightarrow & \quad \{ \text{ Equation 4.6 } \} \\
 & \sim|[B :: \sim P \times Q]| \sqsubseteq R
\end{array}$$

□

Sets A and B in the theorem above represent "internal" symbols. Namely, set A represents internal connections between Q and R, and set B represents internal connections between $\sim P$ and Q. Because progress refinement requires equal alphabets, sets A and B are defined implicitly. That is, $\mathbf{o}.P = (\mathbf{o}.Q \cup \mathbf{o}.R) - A$ and $\mathbf{i}.R = (\mathbf{i}.P \cup \mathbf{o}.Q) - B$.

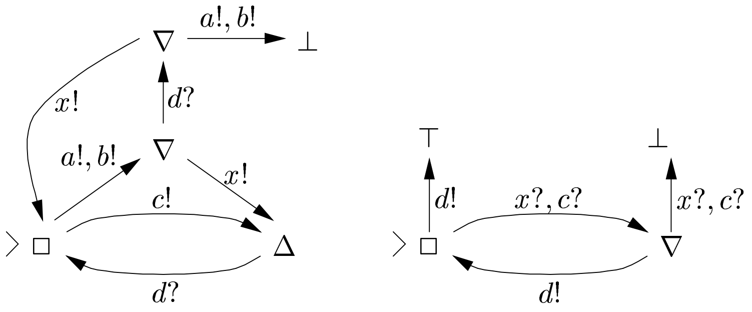
We demonstrate the application of the Factorization Theorem with the same example as in Section 3.9. The difference is that now the design equation also takes progress properties into account. Figure 4.18a shows process P, which is a specification of a three-input MERGE. Process Q of Figure 4.18b is a specification of a two-input MERGE. We would like to implement a three-input MERGE with the two-input MERGE, and we would like to find a process R that would complete the implementation.



a) three-input MERGE b) a guessed part of an implementation for P

Figure 4.18: The MERGE

Figure 4.19b shows the lower bound for process R as given by the Factorization Theorem. The lower bound for process R is a specification of another two-input MERGE. Just as we learned in Section 3.9 we can see that a three-input MERGE can be implemented with two two-input MERGEs, but now we also considered progress concerns on top of the safety concerns.



a) $\sim P \times Q$ b) $\sim|[a, b :: \sim P \times Q]|$

Figure 4.19: An application of the Factorization Theorem

4.12 Summary

In this chapter we instantiated the ECF model of Chapter 2 with the set of labels that describe safety and progress properties of a process. The set of labels we use was taken from [Verhoeff, 1994].

We illustrated how the refinement relation and operations on processes address progress properties of a process. The correctness condition we defined for the progress model consists of the progress condition and the safety condition. Our progress condition is exactly the same as the progress condition in [Verhoeff, 1994], which means that we are able to detect deadlocks in implementations of a given specification.

Similarly to the safety model, the refinement relation in the safety model has three equivalent definitions: One definition comes straight from the ECF model; in one definition we require correctness of an implementation operating together with the reflection of the specification, which is the approach proposed in [Ebergen, 1991]; finally, the refinement relation can also be defined via the testing paradigm. Such a testing approach to the refinement relation was taken in [Verhoeff, 1994] and was also addressed in [Negulescu, 1998].

We see the existence of three characterizations of refinement as a justification for the definition of refinement in ECF processes and as a justification for the definition of our correctness condition.

The hiding operation allows us to conceal output ports of a process. We gave a number of examples of hiding and showed that hiding and process product are closely connected to the composition from [Verhoeff, 1994]. We also showed that the result of hiding can be an unhealthy process, which may indicate the presence of internal transient cycles in a network.

Finally, we proved the Factorization Theorem and demonstrated how this theorem can be applied in a design process.

Chapter 5

Specification composition

In this chapter we discuss two composition operations on ECF processes, the network composition and the specification composition. We use the network composition to compute the joint behavior of a set of devices connected to form a network. For example,

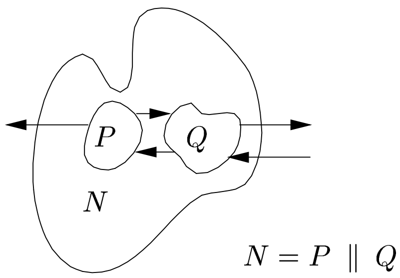
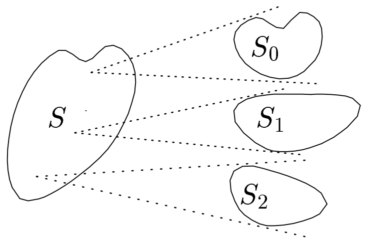


Figure 5.1: The network composition

processes P and Q from Figure 5.1 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.

The specification composition is another operation that we define for ECF processes.

We use the specification composition to arrive at processes that specify communication behaviors of individual devices. Such specifications are called constraint-oriented specifications. When producing a constraint-oriented specification, we find constraints that determine the communication behavior of a process. A constraint can be seen as a restricted view of the communication behavior of the process. For example, Figure 5.2 shows constraints S_0, S_1, and S_2, which represent restricted views at the behavior of process S. Once we have collected a set of constraints that completely determine the behavior of a process, we combine the constraints with the specification composition. The result of the specification composition of constraints is a process. For example, the specification composition of constraints from Figure 5.2, denoted by $S_0 \,\&\, S_1 \,\&\, S_2$, yields process S.



$S = S_0 \,\&\, S_1 \,\&\, S_2$

Figure 5.2: The specification composition

It turns out that constraints themselves are processes. Thus, a process can be used for two different purposes. One, a process can model the behavior of one physical device or of a network of devices; two, a process can model a constraint on the behavior of one device.

The constraint-based approach to building specifications can be very useful, especially when behaviors involve a large degree of concurrency. This approach to constructing specifications has been applied in different formalisms that have found applications in different areas: [Bolognesi and Brinksma, 1987] discusses how a constraint-based specification approach is supported in LOTOS. [Ebergen, 1991] advocates that the weave, which is a composition operation in "standard" trace theory, can be used for building specifications from constraints. The usage of the weave in [Ebergen, 1992] and [Benko and Ebergen, 1994] demonstrates how combining behavioral constraints can lead to concise specifications of non-trivial behaviors that involve a large degree of concurrency. [Molnar et al., 1992a] and [Molnar et al., 1992b] define the weave for Petri nets and demonstrate its application in specifying asynchronous-circuit components. In [Negulescu and Peeters, 1998], the constraint-oriented specification approach takes advantage of chain constraints to introduce timing requirements into specifications.

In the remainder of this chapter we study the two compositions in both the safety and the progress model. To the best of our knowledge, applications of the constraint-based specification approach were limited to situations when only safety was a concern. For the safety model, we demonstrate that from the formal point of view both the specification composition and the network composition correspond to the process product. That is, there is no formal difference between the specification composition and the network composition. Such a view has been adopted in [Jong and Lin, 1994], [Ebergen et al., 1995], [Pena and Cortadella, 1996], and [Negulescu, 1998].

For the progress model, on the other hand, we show that the process product cannot be applied for combining constraints into one specification. We define the specification composition in the progress model for the domain of snippets, which are ECF processes obeying a number of restrictions. We prove a number of properties of the specification composition and we provide a few examples of building specifications by composing snippets.

5.1 The specification composition and the network composition in the safety model

In the safety model, process product has, in addition to modeling process networks, another application. The product allows us to express a specification as a conjunction of constraints, where each constraint can be expressed as a process. Because the process product "synchronizes" common outputs and inputs of the constraint processes, the product exhibits exactly all the behaviors that are in accordance with each of the constraint processes. Such a "constraint-oriented" specification is a powerful tool that can greatly simplify producing a specification of a process, especially when the specification involves a large degree of concurrency.

For example, let us show how to apply the product of constraints in specifying the behavior of the JOIN. Recall that the JOIN has two input ports, a and b, and output port c. Initially, the JOIN waits for both inputs a and b. After the inputs have arrived, the JOIN produces an output on port c, and the cycle repeats.

The behavior of the JOIN can be seen as a product of two constraints. We first focus only on the behavior of the JOIN with respect to the input port a and the output port c. Input actions on port a and output actions on port c alternate. The JOIN can only produce an output after receiving an input, and we do not allow a further input until the output is produced. We express this alternation with the constraint shown in Figure 5.3a.

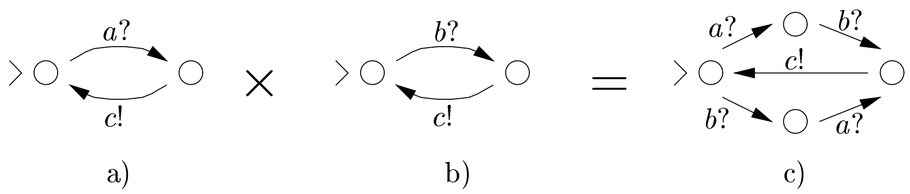


Figure 5.3: A specification of the JOIN

The inputs on port b and the outputs on port c also alternate, which we express as the process of Figure 5.3b. Finally, Figure 5.3c shows the product of the two constraints. The result is a specification of the JOIN.

We demonstrated in Chapter 3 that process product can also be used for modeling networks of independent devices modeled by processes. Thus, in the safety model, process product serves as a network composition and as the specification composition. When we apply process product as a network composition, denoted by $P \parallel Q$ for processes P and Q, we forbid connections between output ports of P and Q. This restriction comes from physical properties of electronic circuits: When we connect electronic circuits, we should avoid connections between output terminals, because we could create damaging short circuits.

On the other hand, when we use process product as the specification composition, denoted by $P \,\&\, Q$ for processes P and Q, we forbid connections between input ports and output ports of processes P and Q. We introduce this restriction because each of the processes captures an aspect of the behavior of the *same* device. Thus, each port is either an input or an output in all constraints describing the device.

5.2 Healthy and unhealthy processes

The process product cannot be used as a specification composition in the progress model. In order to demonstrate why this is the case we first recall the concept of unhealthiness, introduced in Chapter 4. A process that guarantees progress after some trace t should be capable of producing an output after exhibiting trace t. Such a process is called a healthy process. More precisely, healthy processes are processes where in each ∇ state at least one output can be produced, and in each Δ state at least one input can be received.

Formally, processes P is healthy if the following holds for every trace t:

$$\begin{aligned} &(\mathbf{f}.P.t = \nabla \Rightarrow (\exists a : a \in \mathbf{o}.P : \mathbf{f}.P.ta \neq \top)) \\ \wedge\ &(\mathbf{f}.P.t = \Delta \Rightarrow (\exists a : a \in \mathbf{i}.P : \mathbf{f}.P.ta \neq \perp)) \end{aligned}$$

Healthy processes are not closed under the product of ECF processes. This is the reason why the process product is not suitable for composing behavioral constraints. In order to illustrate this unsuitability we attempt to specify the JOIN with inputs a and b and with output c as a product of two constraints. One constraint specifies that output c must be produced only after input a has been received, and another constraint states that output c must be produced only after input b has been received. Figure 5.4 shows the product of the two constraints. In the state graphs of Figure 5.4 we omitted the $\top$ and the $\perp$ states to reduce the clutter; all missing output transitions lead to the $\top$ state and all missing input transitions lead to the $\perp$ state. One can see that the product of the

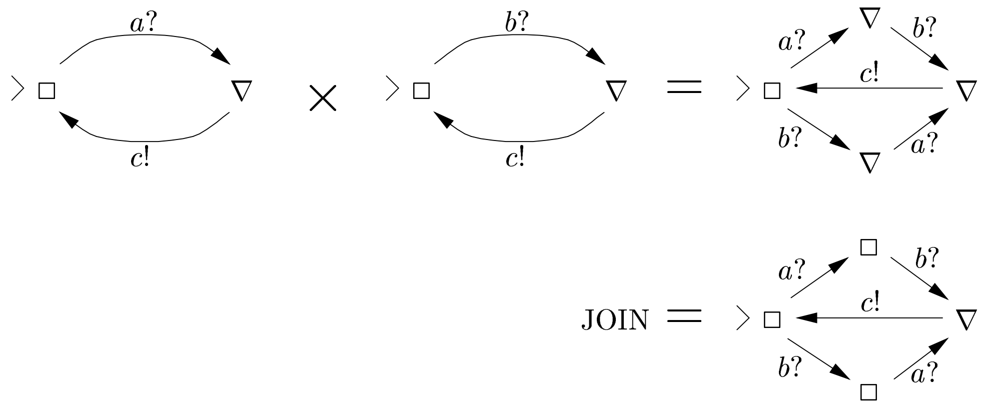


Figure 5.4: Product is *not* a specification composition

two constraints is an unhealthy process that differs from the specification of the JOIN. Namely, after receiving an input on port a, the product is in a ∇ state, but no output can immediately be produced.

The example above emphasizes two problems with using process product as the specification composition. One problem is that the result of the process product is different from what we would like to have as a specification. The second problem is that the result of the process product may be an unhealthy process and we see no practical use for an unhealthy specification.

In order to improve our understanding of the problems described above, we take a closer look at unhealthy processes. The healthiness of a process can be violated in two ways. We can have a ∇ trace after which no outputs can be produced. An example of such a process is the result of the process product shown in Figure 5.4. The set of processes that violate healthiness in a ∇ state are called ∇-unhealthy processes.

We find it convenient to characterize the set of traces that are labeled with ∇ and that cannot be followed by a legal output. For process P, we denote such a set with $\mathbf{u}_\nabla.P$, and we call it the set of ∇-unhealthy traces. We define $\mathbf{u}_\nabla.P$ as follows:

$$\mathbf{u}_\nabla.P = \{t : \mathbf{f}.P.t = \nabla \wedge (\forall a : a \in \mathbf{o}.P : \mathbf{f}.P.ta = \top) : t\} \tag{5.1}$$

For a healthy process P we have $\mathbf{u}_\nabla.P = \emptyset$.

Another way to violate healthiness is to have a Δ trace after which the process cannot receive any inputs. Following the pattern from above, we first characterize Δ-unhealthy traces. The Δ-unhealthy traces are labeled with Δ, but a process cannot receive an input after such a trace. The set of Δ-unhealthy traces of process P is denoted by $\mathbf{u}_\Delta.P$ and is defined as

$$\mathbf{u}_\Delta.P = \{t : \mathbf{f}.P.t = \Delta \wedge (\forall a : a \in \mathbf{i}.P : \mathbf{f}.P.ta = \bot) : t\} \tag{5.2}$$

For a healthy process P, we have $\mathbf{u}_\Delta.P = \emptyset$. In Figure 5.4 we demonstrated that a

∇-unhealthy process can be a result of the product of two healthy processes. Figure 5.5 shows that a Δ-unhealthy process can be the result of the product of two healthy processes.

$$\rangle\, \Box \xrightarrow{a?} \bot \qquad \times \qquad \rangle\, \Delta \xrightarrow{a?} \Box \xrightarrow{a?} \bot \qquad = \qquad \rangle\, \Delta \xrightarrow{a?} \bot$$

Figure 5.5: A Δ-unhealthy process can be the result of a process product

For process P, the set of unhealthy traces $\mathbf{u}.P$ is the union of $\mathbf{u}_\Delta.P$ and $\mathbf{u}_\nabla.P$. For a healthy process P, $\mathbf{u}.P$ is empty.

5.3 Rounding processes up and down

As the example of Figure 5.4 demonstrates, the unhealthiness of a process product prevents us from using the process product as the specification composition. In order to resolve this problem, one could identify all the states that cause unhealthiness and change their labels to $\Box$. This procedure can be seen as rounding the process up or down, depending on the label of the state that causes unhealthiness.

First we take a look at rounding up. Process P, rounded up, is denoted by $\lceil P \rceil$, and is defined as

$$\begin{aligned} \mathbf{i}.\lceil P \rceil &= \mathbf{i}.P \\ \mathbf{o}.\lceil P \rceil &= \mathbf{o}.P \\ \mathbf{f}.\lceil P \rceil.t &= \begin{cases} \mathbf{f}.P.t & \text{if } t \notin \mathbf{u}_\Delta.P \\ \Box & \text{if } t \in \mathbf{u}_\Delta.P \end{cases} \end{aligned} \tag{5.3}$$

The operation defined above is called rounding up, because labels of some traces may change from Δ to $\Box$, and $\Delta \sqsubseteq \Box$ in the partial order on labels. Thus, the result of rounding up is a process that is potentially larger than the process we started with: $P \sqsubseteq \lceil P \rceil$.

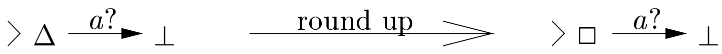


Figure 5.6: Rounding up a Δ-unhealthy process

Figure 5.5 shows that the process product can yield a Δ-unhealthy process. Such a process can be rounded-up and the result is a healthy process, as shown in Figure 5.6.

We define rounding down in a similar way. Process P, rounded down, is denoted by $\lfloor P \rfloor$, and is defined as

$$
\begin{aligned}
\mathbf{i}.\lfloor P \rfloor &= \mathbf{i}.P \\
\mathbf{o}.\lfloor P \rfloor &= \mathbf{o}.P \\
\mathbf{f}.\lfloor P \rfloor.t &= \begin{cases} \mathbf{f}.P.t & \text{if } t \notin \mathbf{u}_{\nabla}.P \\ \Box & \text{if } t \in \mathbf{u}_{\nabla}.P \end{cases}
\end{aligned}
\tag{5.4}
$$

When we round a process down, the label of some traces might change from ∇ to $\Box$. In the partial order of labels we have $\Box \sqsubseteq \nabla$, thus the result of rounding down is a process that may be smaller than the process we started with: $\lfloor P \rfloor \sqsubseteq P$.

We can, for example, take the unhealthy process of Figure 5.4 and round it down in order to obtain a healthy process, as shown in Figure 5.7.

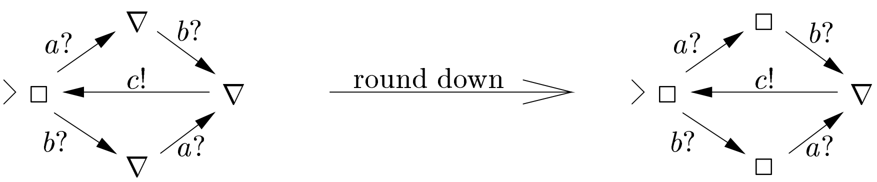


Figure 5.7: Rounding-down a ∇-unhealthy process

Of course, an unhealthy process can be both ∇-unhealthy and Δ-unhealthy. In order to make such a process healthy, we round it up and down. We call such an operation rounding, and we denote it by $[P]$ for process P. We define rounding as follows:

$$\begin{aligned}
\mathbf{i}.[P] &= \mathbf{i}.P \\
\mathbf{o}.[P] &= \mathbf{o}.P \\
\mathbf{f}.[P].t &= \begin{cases} \mathbf{f}.P.t & \text{if } t \notin \mathbf{u}.P \\ \Box & \text{if } t \in \mathbf{u}.P \end{cases}
\end{aligned} \tag{5.5}$$

Rounding a process up does not affect the ∇-unhealthy traces and rounding a process down does not affect the Δ-unhealthy traces. For this reason, rounding is a combination of rounding up and rounding down:

$$[P] = \lceil \lfloor P \rfloor \rceil = \lfloor \lceil P \rceil \rfloor \tag{5.6}$$

We use $\otimes$ to denote the rounded product. For processes P and Q we have

$$P \otimes Q \quad = \quad [P \times Q] \tag{5.7}$$

In the remainder of this chapter we demonstrate that, with some additional restrictions, the rounded product can be used as the specification composition. The restrictions stem from the formal properties that we would like to hold for the specification composition. One of these properties is monotonicity, which does not hold for the rounded product of processes as we demonstrate in the example of Figure 5.8. Process P from Figure 5.8 is refined by process Q. On the other hand, $P \otimes R$ is *not* refined by $Q \times R$, because $\mathbf{f}.(P \otimes R).\varepsilon = \Box$, $\mathbf{f}.(Q \otimes R).\varepsilon = \Delta$, but $\Box \not\sqsubseteq \Delta$.

A justification for the use of the rounded product as the specification composition

$$
\begin{array}{lclcl}
P: & & Q: & & R: \\
> \square \xrightarrow{a?} \bot & \sqsubseteq & > \square \xrightarrow{a?} \square \xrightarrow{a?} \bot & & > \Delta \xrightarrow{a?} \square \xrightarrow{a?} \bot \\
P \otimes R: & & Q \otimes R & & \\
> \square \xrightarrow{a?} \bot & \not\sqsubseteq & > \Delta \xrightarrow{a?} \square \xrightarrow{a?} \bot & &
\end{array}
$$

Figure 5.8: The rounded product of processes is not monotonic

is as follows. The product of two healthy processes yields an unhealthy state in one of the following two ways: A ∇-unhealthy state is a result of the direct product of a state labeled with $\square$ and a state labeled with ∇. A Δ-unhealthy state, on the other hand, is a a result of the direct product of a state labeled with $\square$ and a state labeled with Δ. When we defined the product on labels, we chose to give the progress obligations expressed by ∇ and Δ a priority over the lack of progress requirements expressed by $\square$. For that reason, $\nabla \times \square = \nabla$ and $\Delta \times \square = \Delta$. In the specification composition, however, we have to take into account that it should be possible to satisfy all progress requirements present in a specification. For that reason, we remove the requirements that cannot be satisfied either by a process or by its environment. This removal of unsatisfiable progress requirements is achieved by rounding a process down, which removes the requirement that a process produce an output when an output cannot be produced, and by rounding a process up, which removes a requirement that the environment provide an input when a process is unable of receiving any input.

In the following sections we restrict the domain for the specification composition and we define a stricter refinement relation. The combination of the restricted domain and the stricter refinement relation allows us to use the rounded product as the specification composition and, at the same time, we get all the formal properties that we would like to hold.

5.4 Snippets and output-persistent refinement

Snippets are processes that belong to the restricted domain defined below.

Definition 5.4.1 (Snippet)

Process P is called a snippet *if it satisfies the following restrictions:*

1. *P is ∇-healthy*
2. *P never demands an input:* $(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.P.t \neq \Delta)$
3. *P has no optional outputs:* $(\forall t, a : t \in (\mathbf{a}.P)^* \wedge a \in \mathbf{o}.P : \mathbf{f}.P.t = \Box \Rightarrow \mathbf{f}.P.ta = \top)$
4. *Each input transition leads either to a legal state or to the failure state:* $(\forall t : t \in (\mathbf{l}.P)(\mathbf{i}.P) \cup \{\varepsilon\} : \mathbf{f}.P.t \neq \top)$

A snippet is Δ-healthy, because a state graph for a snippet contains no Δ states. Because, by definition, a snippet is also ∇-healthy, we can see that snippet is a healthy process. Consequently, a product of snippets rounded down is the same as a rounded product of snippets, because of the absence of Δ traces.

Let us take a closer look at the requirements that a process must satisfy in order to be a snippet. Condition 1 demands that a snippet be a ∇-healthy process. That is, a snippet does not exhibit any progress violations. Safety violations, however, are allowed in a snippet. It has been our experience that most practical examples we have considered satisfy this restriction. Furthermore, we demonstrate below that the presence of ∇-unhealthy traces breaks associativity of the rounded product on snippets.

Condition 2 excludes Δ-traces from a snippet. We need the absence of Δ traces in order to prove that the specification composition is monotonic with respect to the output-persistent refinement that we define below.

Condition 3 tells us that a snippet cannot produce an output after a $\Box$-trace. This means that a snippet can make progress by producing an output only after a ∇-trace, but no progress by producing an output is possible after a $\Box$-trace. We show that, if a snippet can produce a legal output from a $\Box$ state, the rounded product of snippets is not associative.

Condition 4 forbids input transitions from a legal state to the $\top$ state. We show that without this requirement the rounded product is not monotonic with respect to the output-persistent refinement.

Notice that MIRACLE is not a snippet, because it does not satisfy condition 4. ABORT, on the other hand, is a snippet.

The example of Figure 5.8 demonstrates that the rounded product of processes is not monotonic. Unfortunately, when we restrict our domain to snippets, the rounded product still lacks monotonicity, as we demonstrate in the example of Figure 5.9.

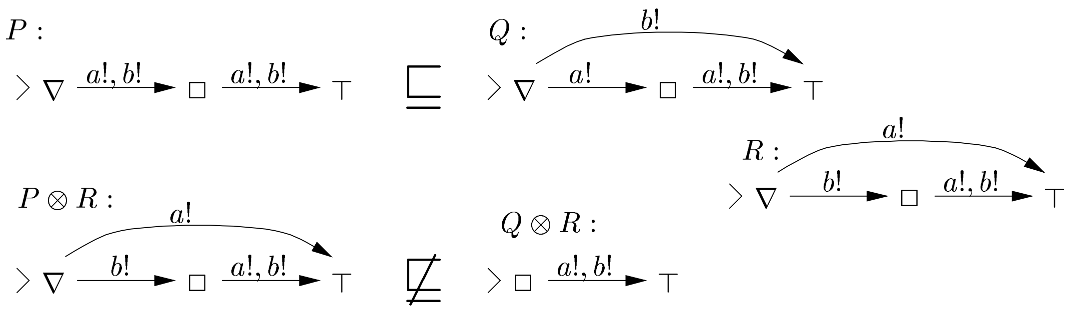


Figure 5.9: Rounded product on snippets is not monotonic

In order to regain the monotonicity of rounded product of snippets, we define a stricter refinement relation, called *output-persistent refinement*: In the output-persistent refinement we require that output transitions are preserved in a refinement. This requirement is called "output persistence".

Definition 5.4.2 (Output persistence) *Let P and Q be snippets, such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P = \mathbf{o}.Q$. Process Q is output-persistent with respect to process P, denoted by $P \preceq_o Q$, if the following holds*

$$P \preceq_o Q \quad \equiv \quad \big(\forall t, a :\ t \in \mathbf{l}.P \wedge a \in \mathbf{o}.P :\ (\mathbf{f}.P.ta \neq \top \Rightarrow \mathbf{f}.Q.ta \neq \top)\big) \tag{5.8}$$

The output-persistence captures the requirement that, if after legal trace t process P can produce output a, then process Q can also produce output a. Notice that, after producing an output on port a, processes P and Q can be in a state labeled by any label other than $\top$ and that there is no relationship between labels of the two target states.

In the output-persistent refinement we strengthen the progress refinement by the output persistence.

Definition 5.4.3 (Output-persistent refinement)
The output-persistent refinement of snippet P by snippet Q is denoted by $P \sqsubseteq_o Q$ and is defined as

$$P \sqsubseteq_o Q \quad \equiv \quad P \sqsubseteq Q \ \wedge \ P \preceq_o Q$$

In Figure 5.10 we illustrate the output-persistent refinement. Figure 5.10a shows that the output-persistent refinement allows us to refine a process failure with a legal output transition. The example of Figure 5.10b demonstrates that the output-persistent

a) $$> \triangledown \xrightarrow{a!} \bot \qquad \sqsubseteq_O \qquad > \triangledown \xrightarrow{a!} \square \xrightarrow{a!} \top$$

b) $$> \square \xrightarrow{a?} \bot \qquad \sqsubseteq_O \qquad > \square \xrightarrow{a?} \square \xrightarrow{a?} \bot$$

c) $$\begin{tikzcd} > \triangledown \arrow[r, "{a!, b!}"] & \square \arrow[r, "{a!, b!}"] & \top & \not\sqsubseteq_O & > \triangledown \arrow[r, "a!"] \arrow[rr, bend left, "b!"] & \square \arrow[r, "{a!, b!}"] & \top \end{tikzcd}$$

Figure 5.10: Examples of output-persistent refinement

refinement allows us to refine an illegal input transition with a legal input transition. That is, more inputs can be allowed on the right-hand side of the refinement relation. Finally, Figure 5.10c shows that the output-persistent refinement relation requires that all legal outputs that are present in the left-hand side process must also be present in the right-hand side process, but the outputs do not have to lead to the same state.

5.5 Properties

In this section we prove a number of properties of the rounded product of snippets and of the output-persistent refinement. Obviously, the output-persistent refinement implies the refinement for any processes P and Q:

$$P \sqsubseteq_o Q \;\Rightarrow\; P \sqsubseteq Q \tag{5.9}$$

Next we prove that $\sqsubseteq_o$ is a partial order.

Theorem 5.5.1 (Partial order)

$\sqsubseteq_o$ is a partial order on processes.

Proof: By Theorem 2.3.2 we know that $\sqsubseteq$ is a partial order. That is, $\sqsubseteq$ is reflexive, transitive, an antisymmetric.

Output persistence is reflexive. That is, $P \preceq_o P$, because

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

We can see that $\sqsubseteq_o$ is reflexive, because $\sqsubseteq$ is reflexive and $\preceq_o$ is reflexive.

Next we look at transitivity of $\sqsubseteq_o$. Take processes P, Q, and R, such that $P \sqsubseteq_o Q$ and $Q \sqsubseteq_o R$. Because output persistent refinement implies "vanilla" refinement, we know that $P \sqsubseteq Q$, $Q \sqsubseteq R$, and $P \sqsubseteq R$.

We have to show that $P \preceq_o R$. By assumption, we have $P \preceq_o Q$ and $Q \preceq_o R$:

$$\big(\forall t, a:\ t \in \mathbf{l}.P \wedge a \in \mathbf{o}.P:\ (\mathbf{f}.P.ta \neq \top \Rightarrow \mathbf{f}.Q.ta \neq \top)\big) \tag{5.10}$$

$$\big(\forall t, a:\ t \in \mathbf{l}.Q \wedge a \in \mathbf{o}.Q:\ (\mathbf{f}.Q.ta \neq \top \Rightarrow \mathbf{f}.R.ta \neq \top)\big) \tag{5.11}$$

Take trace $t \in \mathbf{l}.P$ and symbol $a \in \mathbf{o}.P$, such that $\mathbf{f}.P.ta \neq \top$. The following argument shows that $t \in \mathbf{l}.Q$: Because $P \sqsubseteq Q$ and $t \in \mathbf{l}.P$, we know that $\mathbf{f}.Q.t \neq \bot$. Furthermore, by Equation 5.10, $\mathbf{f}.Q.ta \neq \top$, and, by $\top$-persistence, $\mathbf{f}.Q.t \neq \top$. Because $\mathbf{f}.Q.t \neq \bot$ and $\mathbf{f}.Q.t \neq \top$, we have $t \in \mathbf{l}.Q$.

By Equation 5.11, we now get

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

Because $P \sqsubseteq R$ and $P \preceq_o R$, we have $P \sqsubseteq_o R$. That is, $\sqsubseteq_o$ is transitive.

Finally, we prove that $\sqsubseteq_o$ is antisymmetric:

$$\begin{array}{ll} & P \sqsubseteq_o Q \ \wedge \ Q \sqsubseteq_o P \\ \Rightarrow & \quad \{ \text{ By Equation 5.9 } \} \end{array}$$

$P \sqsubseteq Q \ \wedge \ Q \sqsubseteq P$

$\Rightarrow \quad \{ \sqsubseteq \text{ is antisymmetric } \}$

$P = Q$

$\square$

Next we turn to properties that involve the rounded product.

Theorem 5.5.2

Snippets are closed under $\otimes$. For snippets P and Q, we have

$$P \otimes Q = \lfloor P \times Q \rfloor$$

Proof: We prove in turn each of the requirements in the definition of a snippet (Definition 5.4.1).

Because of rounding, $P \otimes Q = [P \times Q]$ is healthy.

No Δ-trace can appear in $[P \times Q]$, because there are no Δ-traces in snippets P or Q. Furthermore, from Equation 5.5 it follows that there are no unhealthy traces in $[P \times Q]$.

Because no legal outputs are possible after $\square$-traces in either P or Q, no legal outputs are possible after $\square$-traces in $P \times Q$. Consequently, no legal outputs are possible after $\square$-traces in $[P \times Q]$.

Finally, in both snippet P and snippet Q, input transitions from a legal state can lead only to a legal state or to a $\perp$ state. From the product table for labels (Table 4.2), we can see that this property also holds for $P \times Q$. Furthermore, rounding does not affect the legality of a state, neither does it affect the $\perp$ state. Consequently, in $P \otimes Q$, input transitions from a legal state can lead only to a legal state or to a $\perp$ state.

We conclude that $[P \times Q]$ satisfies all requirements for a snippet.

Recall that, for snippet S, $\lceil S \rceil = \lfloor S \rfloor$, because there are no Δ-traces, thus, no rounding up takes place:

$$
\begin{array}{ll}
 & P \otimes Q \\
= & \quad \{ \text{ By definition } \} \\
 & \lceil P \times Q \rceil \\
= & \quad \{ \ P \times Q \text{ contains no } \Delta\text{-traces } \} \\
 & \lfloor P \times Q \rfloor
\end{array}
$$

□

Theorem 5.5.3

The rounded product is idempotent, commutative, and associative in the snippet domain.

Proof: Idempotence and commutativity follow directly from idempotence and commutativity of process product. We omit the proofs for the sake of brevity.

Associativity:

$$
\begin{array}{ll}
 & (P \otimes Q) \otimes R \\
= & \quad \{ \text{ Theorem 5.5.2 } \} \\
 & \lfloor \lfloor P \times Q \rfloor \times R \rfloor \\
= & \quad \{ \text{ Lemma 5.5.4 } \} \\
 & \lfloor P \times Q \times R \rfloor \\
= & \quad \{ \ \times \text{ is commutative and associative } \} \\
 & \lfloor Q \times R \times P \rfloor \\
= & \quad \{ \text{ Lemma 5.5.4 } \} \\
 & \lfloor \lfloor Q \times R \rfloor \times P \rfloor \\
= & \quad \{ \ \times \text{ is commutative } \}
\end{array}
$$

$$\lfloor P \times \lfloor Q \times R \rfloor \rfloor$$

$$= \quad \{ \text{ Theorem 5.5.2 } \}$$

$$P \otimes (Q \otimes R)$$

□

We used the following lemma when we proved associativity of $\otimes$ on snippets. See Appendix C for the proof.

Lemma 5.5.4

For snippets P, Q, and R, we have

$$\lfloor \lfloor P \times Q \rfloor \times R \rfloor = \lfloor P \times Q \times R \rfloor$$

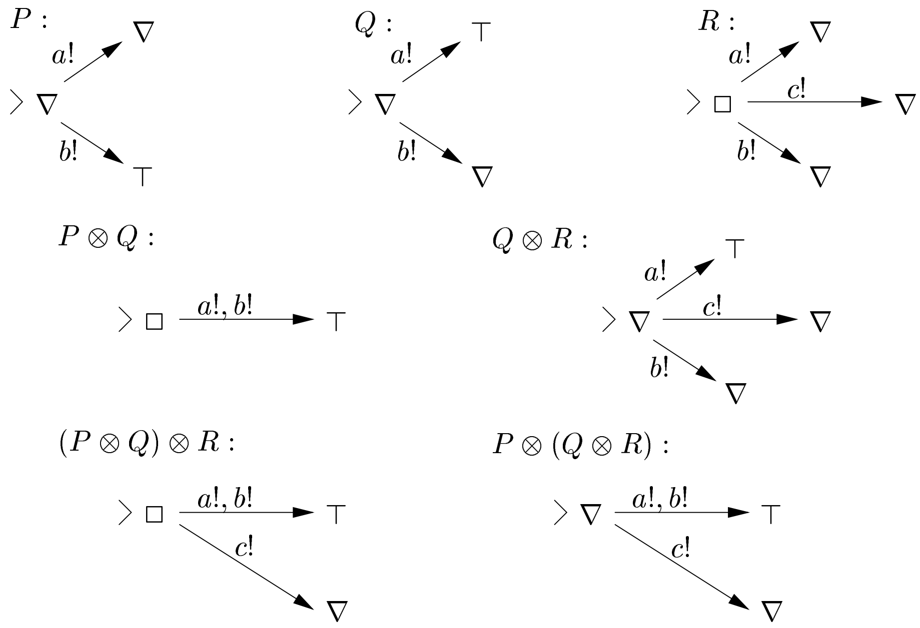


Figure 5.11: Rounded product is not associative for processes

In the example of Figure 5.11 we illustrate that the rounded product is *not* associative for processes. Notice that process R of Figure 5.11 is not a snippet, because it has an output transition on port c leaving the $\square$ state and ending in a ∇ state, which violates condition 3 from Definition 5.4.1.

The example of Figure 5.12 demonstrates that, if we allow that a process fails on a transition from a $\square$-state, then the rounded product is not associative.

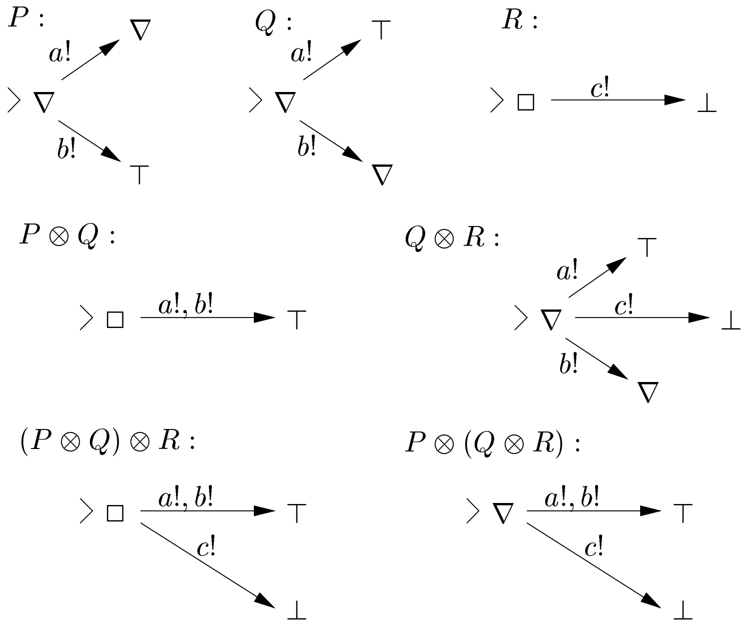


Figure 5.12: Rounded product is not associative for processes that fail on a transition from a $\square$-state

The following property establishes a relationship between the process product and the rounded product of snippets.

Theorem 5.5.5

For snippets P and Q where $\mathbf{o}.P \cap \mathbf{o}.Q = \emptyset$, we have $P \otimes Q = P \times Q$.

Proof: Recall that $P \otimes Q = \lfloor P \times Q \rfloor$ for snippets P and Q. In order to prove $P \otimes Q = P \times Q$, we show that no rounding-down takes place. That is, we show that $\mathbf{u}_{\nabla}.(P \times Q) = \emptyset$.

Recall that $\mathbf{u}_\nabla.P$ is defined as

$$\mathbf{u}_\nabla.P = \{t : \mathbf{f}.P.t = \nabla \ \wedge \ (\forall a : a \in \mathbf{o}.P : \mathbf{f}.P.ta = \top) : t\}$$

We know that $\mathbf{u}_\nabla.P = \mathbf{u}_\nabla.Q = \emptyset$, because P and Q are snippets.

Now take trace $t \in (\mathbf{a}.(P\times Q))^*$, such that $\mathbf{f}.(P\times Q).t = \nabla$. Recall from the definition of process product (Definition 2.4.1) that for a legal trace t we have

$$\mathbf{f}.(P \times Q).t = \mathbf{f}.P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q)$$

From the product table for labels we can see that we have two cases for $\mathbf{f}.(P \times Q).t = \nabla$. In the first case, $\mathbf{f}.P.(t \downarrow \mathbf{a}.P) = \nabla$ and $\mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \in \{\nabla, \Box\}$. The second case is symmetric: $\mathbf{f}.Q.(t \downarrow \mathbf{a}.P) = \nabla$ and $\mathbf{f}.P.(t \downarrow \mathbf{a}.P) \in \{\nabla, \Box\}$. We only need to consider the first case, because of symmetry.

$$\begin{array}{ll}
& \mathbf{f}.P.(t \downarrow \mathbf{a}.P) = \nabla \\
\Rightarrow & \{\ P \text{ is a snippet, thus } \mathbf{u}_\nabla.P = \emptyset\ \} \\
& (\exists a : a \in \mathbf{o}.P : \mathbf{f}.P.(ta \downarrow \mathbf{a}.P) \in \{\nabla, \Box\}) \\
\Rightarrow & \{\ \text{Assumptions: } a \in \mathbf{o}.P,\ \mathbf{o}.P \cap \mathbf{o}.Q = \emptyset,\ \mathbf{f}.Q.(t \downarrow \mathbf{a}.Q) \in \{\nabla, \Box\}, \text{ def. of } \times\ \} \\
& (\exists a : a \in \mathbf{o}.P : \mathbf{f}.(P \times Q).ta \in \{\nabla, \Box\})
\end{array}$$

In other words, trace t is not a ∇-unhealthy trace. Because t is an arbitrary trace from $P \times Q$, we have

$$\mathbf{u}_\nabla.(P \times Q) = \emptyset$$

$\Box$

Theorem 5.5.6 below addresses the monotonicity of the rounded product.

Theorem 5.5.6 (Monotonicity)

For snippets P, Q, and R, we have

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

Proof:

$$\begin{array}{ll}
 & P \sqsubseteq_o Q \\
\Rightarrow & \quad \{ \text{ Lemma 5.5.7 } \} \\
 & P \times R \sqsubseteq_o Q \times R \\
\Rightarrow & \quad \{ \text{ Lemma 5.5.8 } \} \\
 & \lfloor P \times R \rfloor \sqsubseteq_o \lfloor Q \times R \rfloor \\
\Rightarrow & \quad \{ \text{ Theorem 5.5.2; } P,\ Q,\ R \text{ are snippets } \} \\
 & P \otimes R \sqsubseteq_o Q \otimes R
\end{array}$$

$\square$

Two of the restrictions in our definition of a snippet were introduced because we wanted Theorem 5.5.6 to hold. In the following two examples we illustrate each of the two restrictions and we show why the theorem breaks down if the restrictions are not in place.

The example of Figure 5.13 demonstrates that the monotonicity with respect to the output-persistent refinement does not hold if there is a Δ-trace in a snippet.

Figure 5.14 shows an example that illustrates how the monotonicity of the rounded product breaks if we allow an input transition from a legal state to the $\top$ state. Process $P \otimes R$ of Figure 5.14 guarantees to produce an output on port b after having received an input on port a. Process $Q \otimes R$, on the other hand, cannot produce an output on

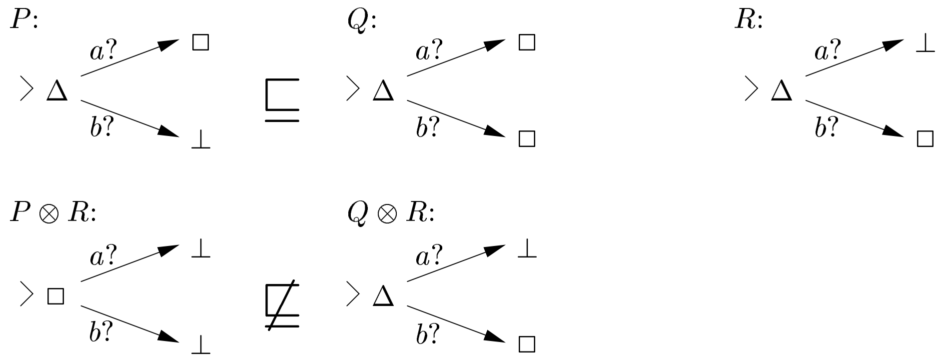


Figure 5.13: The monotonicity does not hold for the rounded product of processes

port b after having received an input on port a. Because not all outputs from $P \otimes R$ are preserved by $Q \otimes R$, $P \otimes R \not\sqsubseteq_o Q \otimes R$.

In order to prove Theorem 5.5.6, we needed the two lemmas shown below. See Appendix C for the proofs.

The first lemma tells us that the product of snippets is monotonic with respect to the output-persistent refinement.

Lemma 5.5.7

For snippets P, Q, and R, we have

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

The second lemma tells us that rounding down is monotonic with respect to output-persistent refinement.

Lemma 5.5.8

For processes P, Q, and R,

$$P \sqsubseteq_o Q \;\Rightarrow\; \lfloor P \rfloor \sqsubseteq_o \lfloor Q \rfloor \tag{5.12}$$

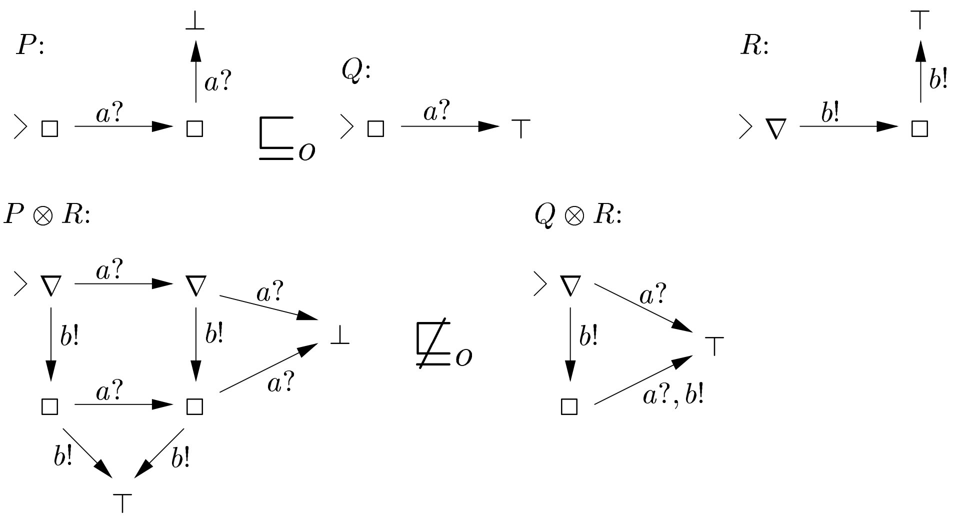


Figure 5.14: The monotonicity does not hold if we allow input transitions from legal states to the $\top$-state

5.6 The network composition in the progress model

We reached the point where we can make a distinction between the *network composition* and *specification composition*. What are the two compositions? When we talk about network composition, we talk about describing a network of devices. We represent each device as a process and we are interested in the behavior of the whole network as a collection of devices. In our framework, we describe the network as another process, which we compute by taking a network composition of processes representing individual devices.

Formally, the network composition is nothing but the product of processes with an additional restriction imposed. The restriction stems from physical properties of electronic circuits, which is the main target application of our formalism. Namely, when we connect electronic circuits, we should avoid connections between output terminals, because we could create damaging short circuits. For this reason, we define the network composition as the process product, where there are no connections between output ports:

Definition 5.6.1 (Network composition)

The product of processes P and Q, where $\mathbf{o}.P \cap \mathbf{o}.Q = \emptyset$, is called the network composition *and is denoted by $P \parallel Q$.*

We observe that, by Theorem 5.5.5, $P \parallel Q = P \otimes Q$, for snippets P and Q.

We would like to make a number of additional comments on the network composition. Notice that, unlike [Verhoeff, 1994], 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. We allow such connections, although we realize that connecting a large number of input ports can create a large fan-in which slows down the operation of a circuit. If we are concerned about creating large fan-ins, we can connect input ports via explicit fork components that may provide necessary amplification.

Notice also that we leave internal network connections visible, unlike [Verhoeff, 1994]. 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. The hiding operator as defined in Chapter 2 is used only for hiding output ports, but not for hiding input ports.

The network composition explicitly reveals safety failures of the network. Consider, for example, the network composition of two WIREs, one with input a and output m, and another with input m and output b. Figure 5.15 shows the network composition of the two WIREs. Notice that, 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 5.15, we can see this failure, because the trace $amam$ leads to the $\perp$-state. That is, the last symbol in trace *amam* is an output symbol that leads from a legal state to the $\perp$ state. In contrast to the above, the models of [Verhoeff, 1994] and [Mallon et al., 1999] exclude processes that have an output transition leading to the $\perp$ state.

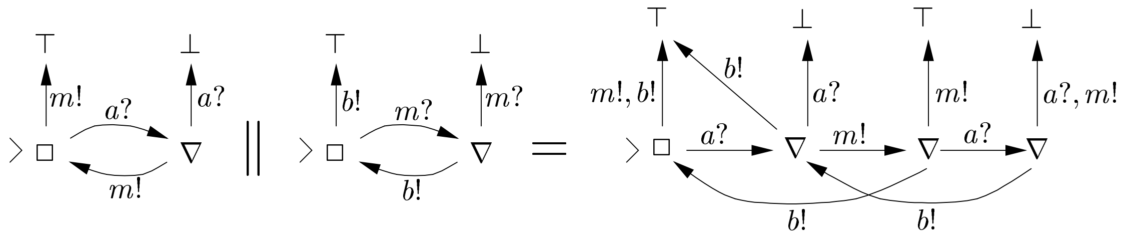


Figure 5.15: The network composition of two WIREs

The network composition tells us what the progress properties of the network behavior are. For example, if a process corresponding to a closed network, which is a network with no input ports, contains a Δ state, then we know that this network can run into progress problems. More precisely, a Δ state in this case indicates a deadlock where some device in the network demands progress, but no other device in the network 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.

5.7 The specification composition in the progress model

The specification composition is an operation that we *do not* use for computing the behavior of a network of components. Instead, we use the specification composition to construct a specification of *one* device. We build a specification by looking at various constraints on the behavior of a component. We express each constraint as a snippet. Once we have collected all the relevant constraints, we compose all the snippets in a specification of the device's behavior. The operation we use for combining the snippets into one specification is the rounded product. We do, however, impose one alphabet restriction on the specification composition. We prohibit connections between input and output ports of different snippets. Our rationale behind this restriction is as follows. Each snippet captures an aspect of the behavior of the *same* device. Thus, each port is either an input or an output in *all* snippets describing the device.

Definition 5.7.1 (Specification composition)
The rounded product of snippets P and Q, where $\mathbf{o}.P \cap \mathbf{i}.Q = \emptyset$, and $\mathbf{o}.Q \cap \mathbf{i}.P = \emptyset$ is called the specification composition *and is denoted by $P \,\&\, Q$.*

By Theorem 5.5.2 we know that snippets are closed under the specification composition. Does this mean that by using the specification composition we limit ourselves to only processes from the domain of snippets? As examples in Chapter 6 demonstrate, we often build specifications by composing a number of snippets. In these snippets we introduce "auxiliary internal" symbols, which serve as internal synchronization points. After we have composed the snippets, we hide these internal symbols. Because snippets are not closed under hiding, the final specification belongs to a domain that is larger than the domain of snippets. We do not have a characterization of the domain of processes obtained by hiding symbols from snippets.

As an example of the use of the specification composition we return to the specification of the JOIN. In Figure 5.4 we show two snippets that each capture an aspect of the behavior of the JOIN. One snippet tells us that the JOIN will produce an output on port c after receiving an input on port a. The other snippet tells us that the JOIN will produce an output on port c after receiving an input on port b. Thus, both snippets tell us that the JOIN will not produce an output on port c until having received inputs on both port a *and* b. Figure 5.16 shows the result of the specification composition of the two snippets. In contrast to the process product, the specification composition results in the correct specification of the JOIN.

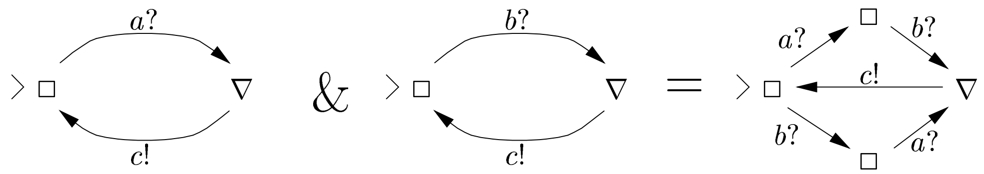


Figure 5.16: The specification composition for the JOIN

In the next example we use specification composition in order to arrive at a specification of an arbiter. An arbiter is a device that arbitrates among requests that can concurrently arrive from a number of different sources. The arbiter ensures that only one pending request is granted at any time.

The arbiter we specify here is called an initialized SEQUENCER [Ebergen, 1991], which we already considered in Chapter 4. Figure 5.17 shows a schematic for the SEQUENCER. Observe that the SEQUENCER has two input ports for receiving requests, namely ports r_0 and r_1. Ports g_0 and g_1 are used for granting requests r_0 and r_1, respectively. Finally, the input port d is called the "done" port. Receiving an input on port d tells the arbiter that the last granted request has been served, thus the arbiter can grant a subsequent request.

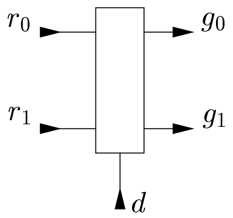


Figure 5.17: A schematic for the SEQUENCER

In Figure 5.18 we show three snippets that capture the behavior of the SEQUENCER. The first snippet indicates that a request r_0 is granted by grant g_0. The snippet also states 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.

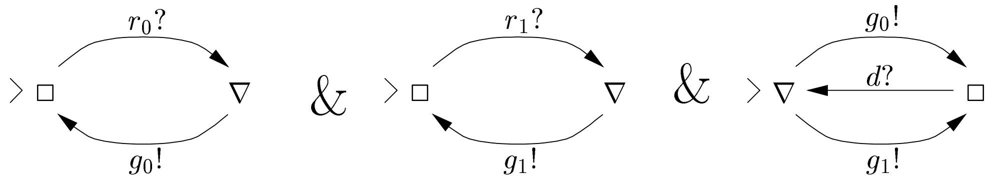


Figure 5.18: Snippets for the SEQUENCER

Figure 5.19 shows the result of the specification composition of the snippets of Figure 5.18. In order to reduce the clutter, we omitted the $\top$ and the $\perp$ state from the state graph in Figure 5.19: All missing input transitions lead to the $\perp$ state and all missing output transitions lead to the $\top$ state. Notice that the initial state of the graph of Figure 5.19 carries label $\Box$, whereas the product of the labels of the initial states of the snippets from Figure 5.18 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.

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.

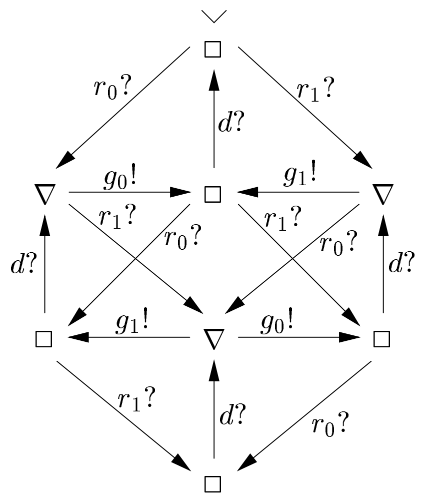


Figure 5.19: A specification for the SEQUENCER

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 rather small in size, such as the snippets of Figure 5.18. 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 the device.

5.8 Summary

In this chapter we open the path for constructing concise specifications by means of specification composition of snippets. We first demonstrate that process product is an appropriate operation for the specification composition in the safety model. In the progress model, on the other hand, process product cannot be used as the specification composition. Instead, we introduce the rounded product and introduce a restricted process domain called snippets. We prove that snippets are closed under the rounded product and that the rounded product on snippets is idempotent, commutative and associative. We also introduce a stricter refinement relation, called output-persistent refinement, which requires that all legal outputs from a specification must be preserved in an implementation. We demonstrate that we need this stricter refinement relation in order to prove that the rounded product on snippets is monotonic with respect to the output-persistent refinement. We also demonstrate that each of the restrictions for snippets is needed in order to prove the associativity and the monotonicity of the rounded product, which provides a justification for our definition of snippets. Finally, we show how the process product can be used to describe the behavior of a network of devices and how the rounded product can be used to specify a concurrent behavior as a specification composition of simple snippets.

Chapter 6

Applications

In this chapter we present a number of examples that demonstrate applications of the theory developed in previous chapters. The examples are worked out in the progress model, but they could be easily transcribed into the safety model.

We begin by introducing “commands”, a textual notation for representing snippets. Our commands are an adaptation of the commands used in [van de Snepscheut, 1985] and [Ebergen, 1991]. Having a simple textual notation allows us to tackle examples where state graphs could become rather large.

In Section 6.2 we demonstrate how to apply the Substitution Theorem in a design of a 2-to-4 phase protocol converter. The Substitution Theorem (Theorem 2.7.1) is the cornerstone of hierarchical design and verification. The theorem allows us to do “stepwise refinement” of a specification. First we refine the specification by a network of potentially complex components. For each of these components we can find its own implementation, possibly consisting of a network of simpler components. The Substitution Theorem allows us to use these implementations as a substitute for the components in the implementation of the original specification.

In Section 6.3 we outline a "part-wise" design method. In this method we assume that we have a specification expressed as a specification composition of a number of snippets. For each of these snippets we can find its own implementation. We show how to apply the theory of ECF processes to transform the implementations of individual snippets to an implementation of the original specification.

Just like the step-wise refinement, the part-wise design method can help us avoid the state explosion in verifying the refinement relation. We avoid the state explosion, because the design method only requires that we verify refinements of individual snippets. One snippet typically represents a simple sequential behavior that can be expressed in a small state graph. Thus, verifying a refinement of a snippet tends to be a manageable task. On the other hand, the number of states in a state graph representing the specification composition of snippets corresponds roughly to the product of the number of states of the state graphs in individual snippets. For this reason, verification of a decomposition of the complete specification may not be feasible. The part-wise design method, however, allows us to perform just a number of "small" verification tasks. We then derive an implementation for the complete specification by applying a number of properties proven in previous chapters.

The design method presented in Section 6.3 proposes a design approach that is similar to an application of the Separation Theorem from [Ebergen, 1989]; see [Ebergen, 1992] and [Benko, 1993] for examples of applications of the Separation Theorem. The design method from Section 6.3, however, applies to a model that addresses safety and progress concerns, while the Separation Theorem from [Ebergen, 1989] applies to a model where progress concerns were not the major focus [Peeters, 1990].

It is also possible to undertake a similar design approach using the theory of Process Spaces of [Negulescu, 1998]. The composition operator used in Process Spaces corresponds to our process product. In Chapter 4 we demonstrated that the process product cannot be used as a specification composition in a model that addresses both safety and progress concerns. For this reason, we believe that the application of the part-wise design method in Process Spaces is limited to safety models. The step-wise design approach based on the Substitution Theorem, however, is applicable in Process Spaces.

In Sections 6.4 — 6.7 we apply the design method from Section 6.3 to a number of examples. In Section 6.4 we design a micropipeline cell [Sutherland, 1989], in Section 6.5 we design a FIFO buffer, in Section 6.6 we show how to solve the dining-philosophers problem [Dijkstra, 1971], and in Section 6.7 we show how to derive an implementation of a 3-input SEQUENCER. With these examples we demonstrate that the ECF snippets are flexible and expressive enough to handle a wide variety of design problems.

6.1 Commands

In previous chapters we represented processes by state graphs. In this section we represent snippets in a concise textual notation called "commands", which were inspired by Dijkstra's "guarded commands". The commands were introduced in [Rem et al., 1983] and were further refined in [van de Snepscheut, 1985] and [Ebergen, 1991]. Below we introduce commands and we show how a command defines a snippet.

Definition 6.1.1 (Commands)

Let A be a set of symbols and let $a \in A$. Commands on A are defined recursively as follows:

1. *$a?$ and $a!$ are commands*

2. *If E and F are commands then $E; F$, $E \mid F$, and $*[E]$ are commands*

For command E, the set of symbols postfixed with ? is called the input alphabet of the command and is denoted by $\mathbf{i}.E$*. The set of symbols postfixed with ! is called the set of output symbols and is, for command E, denoted by* $\mathbf{o}.E$*. We stipulate that* $\mathbf{i}.E \cap \mathbf{o}.E = \emptyset$*.*

Commands $a?$ and $a!$ are often called atomic commands.

A command is a regular expression [Hopcroft and Ullman, 1979], just that we use a non-standard notation: $E; F$ denotes concatenation, $E \mid F$ denotes union, and $*[E]$ denotes Kleene closure. We stipulate the following priority for the operations on commands: repetition has the highest priority, followed by concatenation, and choice.

A language for command E is denoted by $\mathbf{lang}.E$, and is defined in the following way:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbf{lang}.a? & = & \{a?\} & \mathbf{lang}.(E;F) & = & (\mathbf{lang}.E)(\mathbf{lang}.F) \\
\mathbf{lang}.a! & = & \{a!\} & \mathbf{lang}.(E|F) & = & \mathbf{lang}.E \cup \mathbf{lang}.F \\
 & & & \mathbf{lang}.(*[E]) & = & (\mathbf{lang}.E)^*
\end{array}$$

Notice that $a?$ and $a!$ are treated as single symbols.

A language of a command is a regular set, because a command is a regular expression. We recall that regular sets are closed under prefix-closure. That is, if L is a regular language, then the prefix-closure of L, denoted by $\mathbf{pref}\,.L$ is also a regular language.

The prefix-closure of the regular set corresponding to a regular expression obtained from a command is called the trace set of the command:

Definition 6.1.2 (The trace set of a command)
The trace set of command E is denoted by $\mathbf{t}.E$ *and is defined as*

$$\mathbf{t}.E = \mathbf{pref}\,.\mathbf{lang}.E$$

Because a command defines a regular expression, we can follow the standard algorithm from [Hopcroft and Ullman, 1979] and obtain a minimized deterministic automaton that accepts the language of the command. Then, we remove the "dead" state and make all states accepting. The result is a state graph for the trace set of a command.

Now we show how a snippet can be uniquely associated with the state graph for the trace set of a command. In the algorithm described below, transitions labeled with symbols from $\mathbf{i}.E$ are called input transitions and transitions labeled with symbols from $\mathbf{o}.E$ are called output transitions.

1. Add two states, one labeled with $\top$ and the other labeled with $\bot$.
2. Make the state graph complete: All missing transitions on output symbols lead to the $\top$ state; all missing transitions on input symbols lead to the $\bot$ state. All transitions leaving the $\top$ and the $\bot$ state are self-loops.
3. All states that have an outgoing legal output transition (not leading to the $\top$ or $\bot$ state) are labeled with ∇.
4. All states that have not been labeled yet are labeled with $\Box$.

By $\mathbf{proc}.E$ we denote the triple $P = \langle \mathbf{i}.E, \mathbf{o}.E, \mathbf{f}.E \rangle$, where $\mathbf{f}.E$ is the enhanced characteristic function. We compute $\mathbf{f}.E.t$ by following trace t in the labeled state graph obtained from command E. The theorem below characterizes $\mathbf{proc}.E$ in the context of ECF processes:

Theorem 6.1.3 (Command defines a snippet)

Let E be a command. Then, $\mathbf{proc}.E$ is a safe snippet.

Proof: First we prove that $P = \mathbf{proc}.E$ is a process. Notice that $\mathbf{i}.P = \mathbf{i}.E$ and $\mathbf{o}.P = \mathbf{o}.E$. Because E is a command, we have $\mathbf{i}.E \cap \mathbf{o}.E = \emptyset$. Consequently, $\mathbf{i}.P \cap \mathbf{o}.P = \emptyset$.

In the state graph for P, all transitions leaving the $\top$ and $\perp$ state are self-loops. This means that $\mathbf{f}.E$ is $\perp$ and $\top$ persistent. Hence, $P = \mathbf{proc}.E$ is a process.

Next we prove that P is a snippet by proving that P satisfies all four conditions from Definition 5.4.1: P is ∇-healthy by construction of the state graph for P, because only states with an outgoing legal output transition are labeled with ∇; thus, condition 1 is satisfied. By construction, the state graph for P does not contain a Δ state, which satisfies condition 2. By construction, all states with an outgoing legal output transition are labeled with ∇, satisfying condition 3. By construction, all illegal input transitions lead to the $\perp$ state, which satisfies condition 4.

Finally, we prove that P is safe: The safety condition from Definition 4.6.1 is satisfied, because in the state graph for P, no output transition leads from a legal state to the $\perp$ state. $\square$

Without proof we mention that the trace set of command E is equal to the legal set of the snippets defined by command E:

$$\mathbf{t}.E \;=\; \mathbf{l}.\mathbf{proc}.E$$

Figure 6.1a shows the state graph for the snippet corresponding to command $a?$. This snippet may receive one input on port a Command $a!$, on the other hand, denotes a snippet that guarantees to produce one output on port a. Figure 6.1b shows the state graph for the snippet corresponding to command $a!$.

$$> \square \xrightarrow{a?} \square \xrightarrow{a?} \perp$$

a) The snippet for command $a?$

$$> \nabla \xrightarrow{a!} \square \xrightarrow{a!} \top$$

b) The snippet for command $a!$

Figure 6.1: State graphs for atomic commands

In Theorem 6.1.3 we showed that a command defines a snippet. In the rest of this chapter we slightly abuse the notation and use commands as a textual representation of snippets. Although commands do not contain any explicit information about progress properties, commands establish implicitly "default" progress properties that are reflected in snippets corresponding to commands. Design methods for ECF processes applied to specifications expressed in terms of commands guarantee that implicit progress properties included in commands are satisfied.

In specifications expressed as commands, we use concatenation to denote that behavior expressed in one command follows the behavior expressed in another command. Repetition tells us that some behavior will repeat, and we use the choice to express several alternatives that the behavior can follow.

We illustrate the operations on commands by giving specifications of a number of asynchronous circuit components. We can specify a WIRE with input a and output b as

$$\mathrm{WIRE} \quad = \quad *[a?; b!]$$

In this command, $a?; b!$ specifies that an input on port a will be followed by an output on port b. The repetition expresses that the WIRE may repeat its behavior after producing an output on port b.

Next we specify a MERGE with inputs a and b and output c. Initially, the MERGE can receive an input on one of its two input ports, which we express by $a? \mid b?$. After having received an input, the MERGE will produce an output on port c. The behavior then repeats.

$$\mathrm{MERGE} \quad = \quad *[(a? \mid b?); c!]$$

INVERSE TOGGLE also has two input ports, a and b, and one output port c. The behavior of an INVERSE TOGGLE is similar to the behavior of the MERGE, just that INVERSE TOGGLE requires that inputs on ports a and b alternate. That is, initially the INVERSE TOGGLE can receive an input on port a, which is followed by an output on port c. Then, the INVERSE TOGGLE can receive an input on port b, which is followed by an output on port c. After that, the behavior repeats, which we express as

$$\text{INVERSE TOGGLE} \quad = \quad *[a?; c!; b?; c!]$$

The next component we specify is the TOGGLE, which has one input port, a, and two output ports, b and c. The TOGGLE can first receive an input on port a, which is followed by an output on port b. The next input on port a causes the TOGGLE to produce an output on ports c. Then the behavior repeats. That is, outputs on ports b and c are produced in an alternating manner. A specification of the TOGGLE is

$$\text{TOGGLE} \quad = \quad *[a?; b!; a?; c!]$$

The specification of a JOIN with inputs a and b and output c can be composed from two snippets as shown in Figure 5.16. Here, we use the notation of commands in order to denote the same snippets that we used in Figure 5.16. One snippet states that an output on port c is produced only after receiving an input on port a, and the other snippets tells us that the JOIN will produce an output only after receiving an input on port b. Thus, the JOIN can be specified as

$$\text{JOIN} \quad = \quad *[a?; c!] \ \& \ *[b?; c!]$$

Next we introduce two more components that we are going to use in our examples later in the chapter. The first component is a 2-by-1 JOIN, shown in Figure 6.2a. The 2-by-1 JOIN is a generalization of the JOIN, performing synchronization between inputs a_0 and n or a_1 and n. Either a_0 and n are received in an arbitrary order and output b_0 is produced, or a_1 and n are received and output b_1 is produced. The environment is restricted to produce only one of the inputs a_0 and a_1 in each cycle. We can express the behavior of the 2-by-1 JOIN with two constraints. First, either input a_0 is followed by output b_0, or input a_1 is followed by output b_1. The snippet capturing this constraint is

$$*[\, a_0?; b_0! \mid a_1?; b_1! \,]$$

The second constraint on the behavior of the 2-by-1 JOIN states that input n is followed by either output b_0 or b_1. The snippet expressing this constraint is

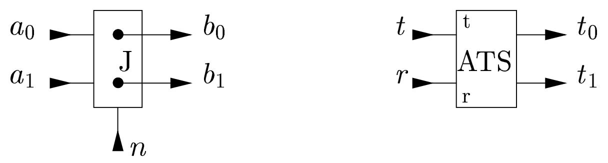
$$*[\, n?; (b_0! \mid b_1!) \,]$$

Finally, a specification of the 2-by-1 JOIN is the specification composition of the two snippets capturing the constraints:

$$\text{2-by-1 JOIN} \;=\; *[\, a_0?; b_0! \mid a_1?; b_1! \,] \;\;\&\;\; *[\, n?; (b_0! \mid b_1!) \,]$$

The $2 - by - 1$ JOIN is also referred to as the DECISION-WAIT [Verhoeff, 1998b].

Figure 6.2b shows the Arbitrating-Test-and-Set (ATS) component that was introduced by Keller [Keller, 1974]. The ATS can be seen as a one-bit memory cell with built-in test-and-set operations: During its operation the ATS may be either "set" or "reset", with "set" being the initial state. Upon receiving a "test-and-set" input t, the component responds with either t_0, if it is "reset", or with t_1 if it is "set". Furthermore, an input



a) 2-by-1 JOIN b) ATS

Figure 6.2: 2-by-1 JOIN and Arbitrating Test-and-Set

t always causes the component to become "set". After receiving "reset" input r, the component becomes "reset" and remains in this state until a "test-and-set" input t causes t_0 to be produced.

Inputs t and r may arrive simultaneously. In such a situation arbitration takes place and the ATS component chooses whether the "test-and-set" input t will be served before the reseting of the ATS component takes place.

The behavior described above can be captured with two constraints. First, after receiving a "test" input t, the ATS component responds either by t_0 or t_1:

$$*[\,t?;(t_0! \mid t_1!)\,]$$

Second, after receiving a "reset" input r, the ATS component will respond by producing an output t_0:

$$*[\,r?;t_0!\,]$$

The specification of the ATS component is the specification composition of the snippets representing the constraints:

$$\text{ATS} \;=\; *[\,t?;(t_0! \mid t_1!)\,] \;\&\; *[\,r?;t_0!\,]$$

6.2 2-to-4 phase converter

This section demonstrates an application of the Substitution Theorem (Theorem 2.7.1). In an application of the Substitution Theorem, we start with a network that implements some specification. The theorem allows us to replace a component in the network by any implementation of that component.

Consider a 2-to-4 phase converter, shown in Figure 6.3a. The 2-to-4 phase converter transforms the 2-phase protocol on terminals r_0 and a_0 to the 4-phase protocol on terminals r_1 and a_1. The 2-phase protocol, initiated by the environment, is a sequence of two communication actions $r_1!a_1?$. The 2-phase protocol is commonly used in so-called transition signaling. The 4-phase protocol, initiated by the converter, is a sequence of four communication actions $r_0?a_0!r_0?a_0!$. One can find the 4-phase protocol in interfaces of many data latches [Day and Woods, 1995, Furber and Day, 1996]. If we interpret communication actions as voltage transitions, we can see that, after one cycle of the 4-phase protocol, the voltage levels on all ports are the same as in the beginning of the cycle. In the 2-phase protocol, the voltage levels are changed after each cycle.

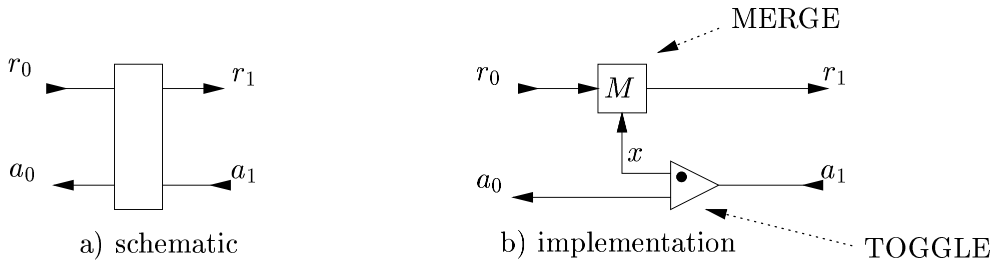


Figure 6.3: 2-to-4 phase converter

A specification of the 2-to-4 phase converter is

$$P = *[r_0?; r_1!; a_1?; r1!; a1?; a0!]$$

The first occurrence of r_0 triggers one cycle of the 4-phase protocol. After one cycle of the 4-phase protocol has been completed, the cycle of the 2-phase protocol comes to an end as well.

Notice that the first input on port a_1 triggers output r_1 and the second input on a_1 triggers output a_0. This suggests that a TOGGLE might be used in an implementation. Furthermore, outputs on port r_1 are triggered by alternating inputs r_0 and a_1. This suggests that an INVERSE TOGGLE might appear in an implementation. Port r_1 cannot appear as an output of both TOGGLE and INVERSE TOGGLE. For that reason, we introduce an "internal" symbol x in the specification of the converter. Our intention is that x serve as a connection between the TOGGLE and the INVERSE TOGGLE:

$$P' = *[r_0?; r_1!; a_1?; x!; r_1!; a_1?; a_0!]$$

We introduced symbol x in such a way that it cannot prevent P' from receiving any input that component P can receive, neither can internal symbol x prevent P' from producing any output that component P can produce. For that reason, we have $P = |[x :: P']|$.

In general, we can always introduce "internal" output symbols in a sequential command between an input and an output. Hiding those "internal" symbols gives back the original command. We apply this heuristic in a number of examples in this chapter.

Below we show that P' is refined by $R \parallel S$, where R is an INVERSE TOGGLE and S is a TOGGLE:

$$\begin{aligned} R &= *[r_0?; r_1!; x?; r_1!] \\ S &= *[a_1?; x!; a_1?; a_0!] \end{aligned}$$

We have $P = |[x :: P']|$, and, $P' \sqsubseteq R \parallel S$. By the Substitution Theorem, we get

$P \sqsubseteq |[x :: R \parallel S]|$.

Let us provide a few observations that indicate that the refinement $P \sqsubseteq R \parallel S$ indeed holds. We have to verify that the network consisting of $\sim P$, R, and S is safe and that the network satisfies the progress condition from Definition 4.6.2. In other words, we have to argue that network $R \parallel S$ can receive an input at any point where process P can receive an input. Furthermore, wherever P guarantees progress, progress must also be guaranteed by $R \parallel S$.

Process P can initially receive an input on port r_0, but there are no progress obligations on either P nor on its environment. The same holds for the product $R \parallel S$: process R can initially receive an input on port r_0 and there are no progress obligations imposed. After receiving the input on r_0, process P guarantees an output on port r_1, which also holds for process R in the implementation and thus for $R \parallel S$. After having produced an output on port r_1, process P engages in the remaining part of the 4-phase protocol. The same holds for the implementation: Process S can receive an input on port a_1. Then process S guarantees an output on port x, which can be received by process R. Furthermore, process R then guarantees producing an output on port r_1. At that point, process S is waiting on an input on port a_1, which completes the 4-phase protocol. After the 4-phase protocol is completed, the converter P guarantees to complete the 2-phase protocol by producing an output on port a_0. Similarly, process S in the implementation also guarantees to produce an output on port a_0 after having received the second input on port a_1. When both protocols have been completed, the converter returns to the initial state, waiting for an input on port a_0. Similarly, both processes in the implementation also return to their respective initial states, thus a new cycle can begin.

An INVERSE TOGGLE can be refined further by a MERGE:

$$*[r_0?; r_1!; x?; r_1!] \;\sqsubseteq\; *[(r_0?|x?); r_1!]$$

The MERGE does not specify the order in which the inputs have to arrive while the INVERSE TOGGLE does. Let T denote the MERGE component used in the refinement above: $T = *[(r_0?|x?); r_1!]$. When we apply the Substitution Theorem again and replace INVERSE TOGGLE R by MERGE T, we get

$$P \sqsubseteq |[x :: T \parallel S]|$$

Figure 6.3b shows the final implementation of the 2-to-4 phase converter.

6.3 Part-wise design method

In this section we outline the part-wise design method.

Assume that we are given an informal specification of the behavior of a process. We produce a formal specification by collecting snippets that represent partial views of the behavior of the process. Once we have collected the set of snippets that specify completely the behavior of the process, we combine the snippets with the specification composition and we hide the set A of "auxiliary internal" symbols that we have introduced. The result of this process is a specification S, expressed in the following form:

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

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

It is a non-trivial task to determine whether a set of snippets specifies completely some communication behavior: There can be no general algorithm for verifying whether our initial specification is correct, because we have no reference point to which we could compare our specification. We can, however, build confidence in our specification by performing simulations and comparing the behavioral patterns that result from the simulations to our informal specification. We remark that he simulations do not require building a complete state graph for $P \,\&\, Q$, thus we can avoid state explosion.

Our intention is to seek refinements for snippets P and Q in isolation and then combine the two refinements in a refinement of specification S. We emphasize that verifying refinements of P and Q in isolation tends to be significantly less complex than verifying an implementation of the original 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:

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

By Theorem 5.5.5 we know that the network composition of snippets that have no outputs in common is the same as the rounded product of those snippets. Hence, we can write

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

By Theorem 5.5.2, snippets are closed under rounded product. This indicates that $P_0 \otimes P_1$ and $Q_0 \otimes Q_1$ are snippets. Hence, we can apply monotonicity of rounded product with respect to output-persistent refinement (Theorem 5.5.6) and transitivity of output-persistent refinement (Theorem 5.5.1), 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 $P_0 \otimes Q_0 = P_0 \,\&\, Q_0$. From the specification S we also know 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 5.5.5, 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 "vanilla" refinement. Furthermore, Theorem 2.6.4 tells us that hiding is monotonic with respect to refinement. Thus, we get

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

Finally, we recall that the refinement is transitive by Theorem 2.3.2. Consequently, specification S can be refined as

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

Let us summarize the design approach presented in this section: If we are able to express a specification as a specification composition of snippets, 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. The subsequent sections illustrate this design method on a number of examples.

6.4 Micropipeline cell

A micropipeline [Sutherland, 1989] consists of a series of cells shown in Figure 6.4. In this section we consider the design of one micropipeline cell. We give a specification of the cell, and we apply the part-wise design method from Section 6.3 in order to arrive at an implementation of the cell.

6.4.1 Micropipeline cell: Specification

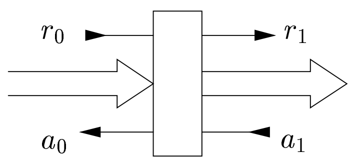


Figure 6.4: A micropipeline cell

A micropipeline cell receives a datum from the left and passes on the datum to the right. The operation of the micropipeline cell can be seen as a series of interactions with its left environment and its right environment. Both environments operate concurrently and independently of each other. The ports of the micropipeline cell can be explained as follows. Ports r_0 and r_1 represent requests. A signal on r_0 indicates that the left environment is ready to pass a datum to the cell. When the cell produces a signal on r_1, the cell indicates that the right environment can take a datum from the cell. Ports a_0 and a_1 represent acknowledgments. A signal on a_0 indicates that the cell has received a datum from the left environment. A signal on a_1, on the other hand, indicates, that the right environment has taken a datum from the cell.

A micropipeline cell interacts concurrently with its left and right environment. We assume that initially the cell is empty and that the right hand side environment is ready for receiving data. We can describe interaction with each of the two environments with a snippet. The output x in the snippets below tells us that the data has been latched by the cell. Signal x is not a part of the communication between the cell and its environment. Rather, we use signal x for internal synchronization between the interactions of the cell with its left and right hand side environments. In the specification of the cell, we will hide signal x.

Let us begin with the left environment:

$$P = *[r_0?; x!; a0!]$$

Snippet P tells us that, after receiving input r_0, the data is first latched, denoted with output x, and then the cell sends acknowledgment a_0.

The snippet for the right-hand side is as follows:

$$Q = *[x!; r_1!; a_1?]$$

Snippet Q tells us that the cell will first latch a datum, which we indicate by output x. The cell then produces output r_1, informing the right environment that there is a new datum available. After the right environment provides input a_1, telling the cell that the datum has been taken, the cycle repeats, because the cell is ready to capture a new datum.

The specification composition of snippets P and Q combines the interactions with the left and the right environment into a specification of a micropipeline cell. Both, left and right-hand side interactions "synchronize" on latching a datum. Because the interface of the micropipeline cell does not include output x, a specification of a micropipeline cell is the specification composition of snippets P and Q where we hide output x:

$$\mathrm{CELL} = |[\, x :: P \;\&\; Q \,]|$$

6.4.2 Micropipeline cell: Implementation

We expressed the specification of the micropipeline cell as a specification composition of snippets; thus, we can follow the design approach outlined in Section 6.3.

Each of the snippets can be refined in a straightforward manner:

$$P = *[r_0?; x!; a0!] \quad \sqsubseteq_o \quad *[r_0?; x!] \;\|\; *[x?; a0!]$$

$$Q = *[x!; r_1!; a1?] \quad \sqsubseteq_o \quad *[x!; a_1?] \;\|\; *[x?; r_1!]$$

By Theorem 5.5.5, $E \;\|\; F = E \otimes F$ for snippets E and F that have no outputs in common.

Hence, we have

$$\begin{array}{lcl} P & \sqsubseteq_o & *[r_0?; x!] \otimes *[x?; a0!] \\ Q & \sqsubseteq_o & *[x!; a_1?] \otimes *[x?; r_1!] \end{array}$$

By Theorem 5.5.6, the rounded product is monotonic with respect to output-persistent refinement. Furthermore, the output persistent refinement is transitive by Theorem 5.5.1. These two properties lead to

$$P \otimes Q \quad \sqsubseteq_o \quad *[r_0?; x!] \otimes *[x?; a0!] \otimes *[x!; a_1?] \otimes *[x?; r_1!]$$

After grouping the processes with common outputs we get:

$$P \otimes Q \quad \sqsubseteq_o \quad \big(*[r_0?; x!] \otimes *[x!; a_1?] \big) \otimes *[x?; r_1!] \otimes *[x?; a0!]$$

In Theorem 5.5.5 we proved that the rounded product and the product are the same when snippets have no outputs in common. Hence, we have

$$P \otimes Q \quad \sqsubseteq_o \quad \big(*[r_0?; x!] \otimes *[x!; a_1?; x!] \big) \parallel *[x?; r_1!] \parallel *[x?; a0!]$$

By the definition of specification composition, Definition 5.7.1, we can see that the two remaining rounded products can be seen as specification compositions. Furthermore, we recall that the output-persistent refinement implies the "regular" refinement:

$$P \,\&\, Q \quad \sqsubseteq \quad \big(*[r_0?; x!] \,\&\, *[x!; a_1?] \big) \parallel *[x?; r_1!] \parallel *[x?; a0!]$$

By Theorem 2.6.4, we have that hiding is monotonic with respect to the refinement, which leads to

$$
\begin{array}{lll}
\text{CELL} & \sqsubseteq & |[\ x :: \\
& & \quad (\ *[r_0?; x!]\ \ \&\ *[x!; a_1?]\ ) \\
& & \|\ \ *[x?; r_1!] \\
& & \|\ \ *[x?; a0!] \\
& &]|
\end{array}
$$

The first group of snippets is a specification of a JOIN with inputs r_0 and a_1 and output x. Input a_1 to the JOIN is initialized, thus, in the first cycle, the JOIN only waits for an input on port r_0. The remaining two processes each represent a WIRE. Figure 6.5 shows a schematic of the implementation we derived. The implementation derived is the standard implementation of the control circuit in a micropipeline cell. Figure 6.5 does not show the data latch that is activated by the output of the JOIN.

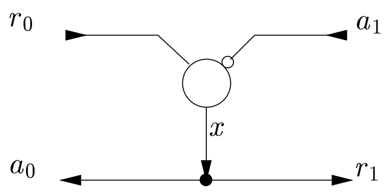


Figure 6.5: An implementation of a micropipeline cell

6.5 FIFO

In this section we follow the design approach from Section 6.3 and we derive an implementation for an asynchronous FIFO. A FIFO is a device that can store a number of datums that can later be retrieved in a First-In-First-Out fashion. Figure 6.6 shows a schematic of the FIFO. Signal *req_put* tells the FIFO that its environment wants to put a datum in the FIFO. The FIFO acknowledges the receipt of the datum by producing signal *ack_put*. On the right-hand side, the FIFO produces signal *req_get* when the environment can retrieve a datum from the FIFO. The environment acknowledges that it has taken a datum from the FIFO by providing a signal on port *ack_get*. In a synchronous FIFO, the left and the right interface of the FIFO would operate in a lock-step. In an asynchronous FIFO, on the other hand, datums can be put in the FIFO and taken out of the FIFO concurrently.

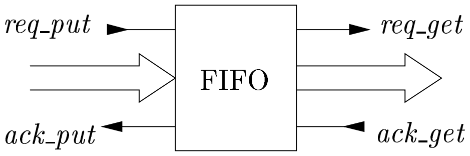


Figure 6.6: A schematic of the FIFO

6.5.1 FIFO: Specification

A FIFO can be built from a linear sequence of cells, each capable of storing one datum. A datum enters the left-most cell in the FIFO and then moves towards the right-most cell where the datum waits until it is taken out of the FIFO. In a synchronous FIFO, datums would move in lock-step, while in an asynchronous FIFO each datum can move at its own pace. Figure 6.7 shows a three-place FIFO, consisting of three cells connected in a sequence. The picture also contains "internal" signals that indicate data movement. Namely, signal *put* indicates that the first cell has taken a datum from the environment. Signal m_1 indicates that a datum has moved from cell 1 to cell 2. Signal m_2 indicates that a datum has moved from cell 2 to cell 3. Finally, signal *get* indicates that the environment has taken a datum from cell 3.

There are many configurations in which we can connect cells in order to build a

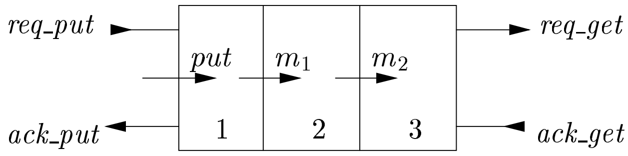


Figure 6.7: Internal data movement in the FIFO

FIFO. For example, cells can be connected into a tree [Brunvand, 1995] or in a square block [Molnar, 1993]. The architecture of a FIFO implementation is determined by how we introduce internal symbols in the specification.

In order to specify the behavior of a three-place FIFO, we produce snippets describing data movement between cells. The first snippet describes the interactions between cell 1 and the environment: The environment first produces signal *req_put* and informs the FIFO that it would like to put a datum in a cell. After the datum has been moved to cell 1, indicated by signal *put*, the FIFO produces signal *ack_put*, telling the environment that the datum has been captured. This behavior can then repeat:

$$A = *[\,req\_put?;\ put!;\ ack\_put!\,]$$

Snippet B describes filling and emptying of cell 1. The cell must first receive a datum, indicated by signal *put*, and then it can pass the datum to the next cell, which is indicated by signal m_1. Because a cell can contain at most one datum, filling and emptying of a cell must alternate.

$$B = *[\,put!;\ m_1!\,]$$

After cell 2 has received a datum, which is indicated by signal m_1, it can pass the datum to cell 3, which is indicated by signal m_2. Filling and emptying of cell 2 is described by snippet C:

$$C = *[\, m_1!;\ m_2! \,]$$

Finally, snippet D describes filling and emptying of cell 3.. First, the last cell must receive a datum, which is indicated by signal m_2. The cell then produces signal *req_get*, telling the environment that there is a datum available. After the environment has taken the datum from the FIFO, it acknowledges the receipt of the datum by providing signal *ack_get*:

$$D = *[\, m_2!;\ req\_get!;\ ack\_get? \,]$$

We build the specification of a three-place FIFO as a specification composition of snippets A, B, C, and D.

$$\text{FIFO} = |[\, put, m_1, m_2 :: A \ \&\ B \ \&\ C \ \&\ D \,]|$$

The specification above involves a significant degree of concurrency: several datums can travel along the FIFO at the same time. In spite of the degree of concurrency involved, the specification above is easy to extend to a FIFO of a larger size. For each additional cell we simply add a snippet of the form $*[m_i!;\ m_{i+1}!]$. A specification of a 10-stage FIFO would, thus, require 11 snippets.

6.5.2 FIFO: Implementation

In order to obtain an implementation for the three-stage FIFO, we find refinements for each of the snippets. All refinements are simple and, for the sake of brevity, we omit the correctness arguments.

$$
\begin{array}{lll}
A & \sqsubseteq_o & \underbrace{*[\,req\_put?;\; put!\,]}_{E_1} \;\|\; \underbrace{*[\,put?;\; ack\_put!\,]}_{E_2} \\
B & \sqsubseteq_o & \underbrace{*[\,put!;\; m_1?\,]}_{F_1} \;\|\; \underbrace{*[\,put?;\; m_1!\,]}_{F_2} \\
C & \sqsubseteq_o & \underbrace{*[\,m_1!;\; m_2?\,]}_{G_1} \;\|\; \underbrace{*[\,m_1?;\; m2!\,]}_{G_2} \\
D & \sqsubseteq_o & \underbrace{*[\,m_2?; req\_get!\,]}_{H_1} \;\|\; \underbrace{*[\,m_2!; ack\_get?\,]}_{H_2}
\end{array}
$$

Because, for snippets, the network composition is the same as the rounded product (Theorem 5.5.5), we can replace all occurrences of $\|$ in the refinements above with $\otimes$. Furthermore, by monotonicity of rounded product (Theorem 5.5.6) and by transitivity of $\sqsubseteq_o$ (Theorem 5.5.1), we get

$$A \otimes B \otimes C \otimes D \sqsubseteq_o (E_1 \otimes F_1) \otimes E_2 \otimes (F_2 \otimes G_1) \otimes (G_2 \otimes H_2) \otimes H_1$$

Now we group the snippets with common outputs and apply the definition of specification composition:

$$A \,\&\, B \,\&\, C \,\&\, D \;\sqsubseteq_o\; (E_1 \,\&\, F_1) \otimes E_2 \otimes (F_2 \,\&\, G_1) \otimes (G_2 \,\&\, H_2) \otimes H_1$$

Finally, we recall that for snippets with no common outputs the rounded product and the network composition are the same:

$$A \,\&\, B \,\&\, C \,\&\, D \;\sqsubseteq_o\; (E_1 \,\&\, F_1) \,\|\, E_2 \,\|\, (F_2 \,\&\, G_1) \,\|\, (G_2 \,\&\, H_2) \,\|\, H_1$$

In order to obtain an implementation of the FIFO, we only have to hide internal symbols. We take into account that output-persistent refinement implies "vanilla" refinement. Furthermore, hiding is monotonic by Theorem 2.6.4, thus we have

$$\text{FIFO} \sqsubseteq |[put, m_1, m_2 :: (E_1 \,\&\, F_1) \parallel E_2 \parallel (F_2 \,\&\, G_1) \parallel (G_2 \,\&\, H_2) \parallel H_1]|$$

Below we examine the specifications of the components that implement a three-place FIFO. The components we find are JOINs with one initialized input, and WIREs.

$$\begin{array}{lcll}
E_1 \;\&\; F_1 & = & *[\, req_put?;\; put!\,] \;\&\; *[\, put!;\; m_1?\,] & \text{(JOIN)} \\
E_2 & = & *[\, put?;\; ack_put!\,] & \text{(WIRE)} \\
F_2 \;\&\; G_1 & = & *[\, put?;\; m_1!\,] \;\&\; *[\, m_1!;\; m_2?\,] & \text{(JOIN)} \\
G_2 \;\&\; H_2 & = & *[\, m_1?;\; m_2!\,] \;\&\; *[\, m_2!; ack_get?\,] & \text{(JOIN)} \\
H_1 & = & *[\, m_2?; req_get!\,] & \text{(WIRE)}
\end{array}$$

Figure 6.8 shows the schematic of the implementation of a three-place FIFO. The implementation consists of three micropipeline stages discussed in Section 6.4. We conclude that a micropipeline implements a FIFO, satisfying both progress and safety requirements. We also remark that the derivation above can be generalized in a straightforward manner to apply to an implementation of a FIFO of any size.

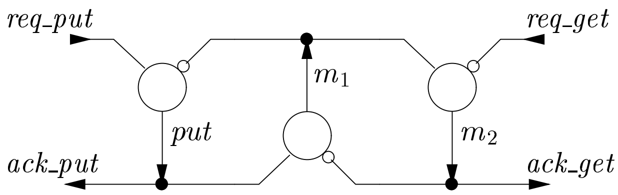
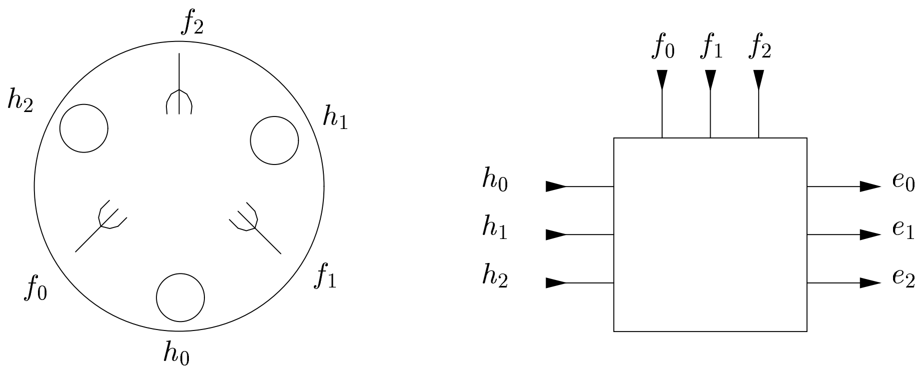


Figure 6.8: An implementation of the FIFO

6.6 Dining philosophers

The dining philosophers is a canonical synchronization problem that was originally stated by Dijkstra [Dijkstra, 1971]:

> 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

Figure 6.9: Dining philosophers

6.6.1 Dining philosophers: specification

In order to keep the example simple, we consider the case with only three philosophers. The table is illustrated in Figure 6.9a. We would like to design a device that schedules the meals for the philosophers. Figure 6.9b shows the schematic of the scheduler. The operation of the scheduler is as follows. When philosopher i is hungry, she sends signal h_i to the scheduler. The scheduler informs philosopher i that she can eat by producing signal e_i. When philosopher i has finished a meal, she produces signals f_i and f_{i-1} in order to let the scheduler know that forks i and $i-1$ are no longer in use. The subtraction is modulo 3.

Each fork can be accessed by two philosophers, but only one philosopher may use it at a time. Signal e_i indicates that philosopher i is eating, using forks i and $i-1$. These two forks cannot be used again until philosopher i puts them back on the table. Thus, for fork i, signals e_i and e_{i-1} must alternate with signal f_i in order to guarantee that fork i is used by only one philosopher at a time. We assume that initially all forks are on the table.

We specify the behavior of the scheduler as a specification composition of snippets. First we observe that only a hungry philosopher can eat. That is, signal e_i must be preceded by h_i. This observation leads to the following three snippets:

$$*[\,h_0?;e_0!\,]\ \ \&\ \ *[\,h_1?;e_1!\,]\ \ \&\ \ *[\,h_2?;e_2!\,]$$

Next we turn to the use of the forks. Fork 0, for example, can be used by philosophers 0 and 2. Furthermore, two philosophers cannot use the same fork at the same time. This means that a fork must be put on the table before it is used again. For example, if philosopher 0 is eating, indicated by signal e_0, fork 0 cannot be used again until signal f_0 arrives, telling the scheduler that fork 0 has been released. That is, neither philosopher 0 nor philosopher 2 can eat until fork 0 has been put on the table, thus signals e_0 and e_2 must alternate with signal f_0. These observations lead to the following snippet that describes the usage of fork 0:

$$*[\,(e_0! \mid e_2!); f_0?\,]$$

The snippet above implies that fork 0 is initially on the table. Snippets that describe the use of forks 1 and 2 are similar: $*[\,(e_0! \mid e_1!); f_1?\,]$ describes the use of fork 1 and $*[\,(e_1! \mid e_2!); f_2?\,]$ describes the use of fork 2.

The specification of the meal scheduler is the specification composition of all snippets:

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

6.6.2 Dining philosophers: Implementation

In order to apply our part-wise design method, we introduce a number of internal symbols in the specification of the scheduler:

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

In specification SCH, the receipt of signal h_0 indicates that philosopher 0 wants to use forks 0 and 2, but the specification does not say whether the forks are taken in some particular order. In SCH_0 we assume that philosophers take forks one after another. More precisely, philosopher 0 first takes fork 0, which is indicated by internal signal x.

Signal e_0, which indicates that philosopher 0 has taken both fork 0 and fork 1 can only take place after signal x. This means that philosopher 0 takes fork 1 only after she has already captured fork 0. We can make similar observations about the order in which philosophers 1 and 2 take their forks. Philosopher 1 first takes fork 1, which the scheduler indicates with internal signal y. After that, philosopher 1 takes fork 2 and is able to eat, indicated by signal e_1. Philosopher 2 first takes fork 0 indicated with internal signal z. When philosopher 2 has also obtained fork 2, the scheduler lets the philosopher eat by producing signal e_2.

SCH_0 assumes that philosophers 0 and 1 first take their left-hand forks and then their right-hand forks. Philosopher 2, on the other hand, first takes her right-hand fork and then her left-hand fork. This approach follows Dijkstra's solution [Dijkstra, 1971] of the dining philosophers problem, where such an asymmetry in the sequence of acquiring forks guarantees the absence of deadlock. Internal symbols x, y, and z only impose the order in which philosophers take their fork. The internal symbols do not, however, prevent SCH_0 from producing any input that SCH can produce, neither can internal symbols prevent SCH_0 from receiving any input that SCH can receive. For that reason we have:

$$\mathrm{SCH} \ \sqsubseteq \ |[\, x, y, z :: \ \mathrm{SCH}_0 \,]|$$

In order to obtain a refinement for SCH_0, we apply the following heuristic to snippets that form SCH_0: Suppose that we have a snippet of the following form

$$E = *[a?; b!; c!]$$

Snippet E can be rewritten as follows

$$*[a?;b!;c!] \quad = \quad *[a?;b!] \;\; \& \;\; *[b!;c!] \;\; \& \;\; *[a?;c!]$$

Snippet $*[a?;c!]$ in the specification composition above ensures that E cannot receive an input on port a before an output on port c has been produced. The output-persistent refinement, however, allows us to relax such a requirement by allowing an implementation to receive an input when the specification cannot. Thus, snippet E can be refined as:

$$*[a?;b!;c!] \quad \sqsubseteq_o \quad *[a?;b!] \;\; \& \;\; *[b!;c!]$$

We can apply this heuristic and obtain the following refinements for the snippets that form SCH_0:

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

Let SCH_1 denote the specification composition of the refinements above:

$$\begin{array}{rcll}
\mathrm{SCH}_1 & = & *[\,h_0?;x!\,] & \text{(snippet } A\text{)} \\
& \& & *[\,h_1?;y!\,] & \text{(snippet } B\text{)} \\
& \& & *[\,h_2?;z!\,] & \text{(snippet } C\text{)} \\
& \& & *[\,(x! \mid z!);f_0?\,] \;\&\; *[\,x!;e_0!\,] \;\&\; *[\,z!;e_2!\,] & \text{(snippet } D\text{)} \\
& \& & *[\,(e_0! \mid y!);f_1?\,] \;\&\; *[\,y!;e_1!]] & \text{(snippet } E\text{)} \\
& \& & *[\,(e_1! \mid e_2!);f_2?\,] & \text{(snippet } F\text{)}
\end{array}$$

By monotonicity of rounded product (Theorem 5.5.6) we know that

$$\mathrm{SCH}_0 \;\sqsubseteq_o\; \mathrm{SCH}_1$$

Furthermore, because the output-persistent refinement implies the vanilla refinement, we can apply the Substitution Theorem and get

$$\mathrm{SCH} \sqsubseteq |[\,x,y,z :\ \mathrm{SCH}_1\,]|$$

We follow the part-wise specification method, thus we first find refinements for each of the snippets in SCH_1. Because snippets are very simple, we quickly find the following:

$$\begin{array}{lcl}
A & \sqsubseteq_o & *[\,h_0?;x!\,] \\
B & \sqsubseteq_o & *[\,h_1?;y!\,] \\
C & \sqsubseteq_o & *[\,h_2?;z!\,] \\
D & \sqsubseteq_o & *[\,(x! \mid z!);f_0?\,] \;\|\; *[\,x?;e_0!\,] \;\|\; *[\,z?;e_2!\,] \\
E & \sqsubseteq_o & *[\,(e_0! \mid y!);f_1?\,] \;\|\; *[\,y?;e_1!\,] \\
F & \sqsubseteq_o & *[\,(e_1! \mid e_2!);f_2?\,]
\end{array}$$

All the refinements above are easy to verify. In the refinements of snippets A, B, C, and D, the specification and the implementation are the same. In the refinements of snippets D and F we simply took the original snippets and we assigned the input and output alphabets such that the conditions for the network composition are satisfied.

After we group snippets with common output symbols, we get

$$\begin{array}{lllr} \mathrm{SCH}_1 & \sqsubseteq & (\,*[\,h_0?;x!\,] \;\&\; *[\,h_2?;z!\,] \;\&\; *[\,(x! \mid z!);f_0?\,]\,) & \text{(SEQUENCER)} \\ & \| & (\,*[\,h_1?;y!\,] \;\&\; *[\,x?;e_0!\,] \;\&\; *[\,(e_0! \mid y!);f_1?\,]\,) & \text{(SEQUENCER)} \\ & \| & (\,*[\,y?;e_1!\,] \;\&\; *[\,z?;e_2!\,] \;\&\; *[\,(e_1! \mid e_2!);f_2?\,]\,) & \text{(SEQUENCER)} \end{array}$$

The implementation of SCH_1 consists of three SEQUENCERs connected as shown in Figure 6.10. Consequently, the meal scheduler is implemented by the same network where we hide internal symbols x, y, and z.

The solution to the dining philosophers problem presented in Figure 6.10 is just a hardware rendition of Dijkstra's solution from [Dijkstra, 1971]: The SEQUENCERs implement semaphores that philosophers use in order to get access to their forks.

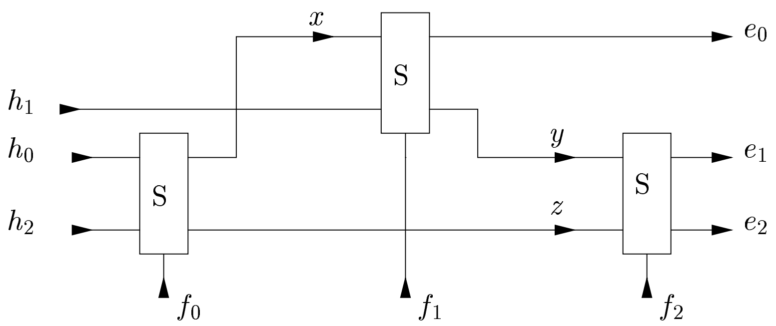


Figure 6.10: A solution to the dining philosophers problem

6.6.3 An "implementation" with deadlock

What would happen if all three philosophers took their left-hand forks first and then their hight-hand forks? Such an approach would lead to the following specification of the scheduler:

$$\begin{array}{rcll} \mathrm{SCH}_2 & = & *[\,h_0?; x!; e_0!\,] & \text{(snippet } A) \\ & \& & *[\,h_1?; y!; e_1!\,] & \text{(snippet } B) \\ & \& & *[\,h_2?; z!; e_2!\,] & \text{(snippet } C) \\ & \& & *[\,(x!; e_0! \mid e_2!); f_0?\,] & \text{(snippet } D) \\ & \& & *[\,(y!; e_1! \mid e_0!); f_1?\,] & \text{(snippet } E) \\ & \& & *[\,(z!; e_2! \mid e_1!); f_2?\,] & \text{(snippet } F) \end{array}$$

In SCH_2, philosopher 0 first captures fork 0, which is indicated by signal x. Only then signal e_0 can take place, indicating that philosopher 0 has also captured fork 1. We can make similar observations for philosophers 1 and 2. For philosopher 1, signal y indicates that first fork 1 is picked up and only then philosopher 1 can pick up fork 2. For philosopher 2, signal z indicates that first fork 2 is picked up and only then philosopher 1 can pick up fork 0.

Observe that $\mathbf{f}.\mathrm{SCH}_2.h_0xh_1yh_2z = \Box$, because of rounding down no output is possible after this trace: Snippets A and D allow producing output e_0, which is blocked by snippet E. Snippets B and E allow producing output e_1, which is blocked by snippet F. Finally, snippets C and F allow producing output e_2, but snippet D is blocking that output.

Following the definition of hiding, we can compute that $\mathbf{f}.|[x, y, z :: \mathrm{SCH}_2]|.h_0h_1h_2 = \Box$. On the other hand $\mathbf{f}.\mathrm{SCH}.h_0h_1h_2 = \nabla$, because the scheduler guarantees producing either e_0, e_1 or e_2 after receiving inputs h_0, h_1, and h_2. This means that $\mathrm{SCH} \not\sqsubseteq |[x, y, z :: \mathrm{SCH}_2]|$, thus we cannot apply our part-wise design method to SCH_2.

From [Dijkstra, 1971] we know that the meal scheduler specified by SCH_2 leads to a deadlock. Because $\mathrm{SCH} \not\sqsubseteq |[x, y, z :: \mathrm{SCH}_2]|$, we can see that the deadlock was caught by our progress condition. From the observations above we know that $\mathbf{f}.(\sim\mathrm{SCH}).h_0h_1h_2 = \Delta$ and that $\mathbf{f}.|[x, y, z :: \mathrm{SCH}_2]|.h_0h_1h_2 = \Box$. Thus, $\mathbf{f}.(\sim\mathrm{SCH} \times |[x, y, z :: \mathrm{SCH}_2]|).h_0h_1h_2 = \Delta$, which means that $\mathbf{prog}.(\sim\mathrm{SCH} \times |[x, y, z :: \mathrm{SCH}_2]|)$ does not hold. Notice that the network consisting of $\sim$SCH and $|[x, y, z :: \mathrm{SCH}_2]|$ is closed. The presence of a Δ state in a closed network indicates that some component in the network demands an input, but no component is obliged to guarantee any progress. That is, the presence of a reachable Δ state in the state graph of a closed network indicates a possibility of deadlock.

6.7 3-input SEQUENCER

In this section we address an implementation of a 3-input SEQUENCER. While a "regular" SEQUENCER arbitrates between two requests, a 3-input SEQUENCER arbitrates between three requests. Figure 6.11 shows a schematic of a 3-input SEQUENCER, where r_0, r_1, and r_2 denote the three request inputs, g_0, g_1 and g_2 indicate the outputs that grant requests, and input d denotes the "done" signal. The "done" signal indicates that the last granted request has been served and that a new request can be granted.

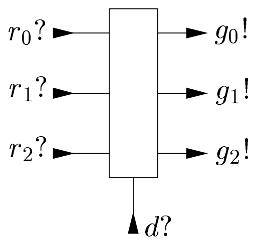


Figure 6.11: 3-input SEQUENCER

The specification of the 3-input SEQUENCER follows the same pattern as the specification of the 2-input SEQUENCER in Section 5.7:

$$
\begin{array}{rcl}
\mathrm{SEQ} & = & *[\,r_0?; g_0!\,] \\
 & \& & *[\,r_1?; g_1!\,] \\
 & \& & *[\,r_2?; g_2!\,] \\
 & \& & *[\,(g_0! \mid g_1! \mid g_2!); d?]
\end{array}
$$

Below we derive an implementation of the 3-input SEQUENCER. The implementation employs a token ring in order to poll requests.

In the token-ring implementation we would like to poll requests in order to see whether there is any pending request that the SEQUENCER should grant. Known token-ring implementations of an arbiter include [Martin, 1985] and [Ebergen, 1992]. [Martin, 1985] proposes a circuit that allows the token to circulate only when an arbiter has received a request signal. Thus, the arbiter from [Martin, 1985] reduces the number of communication actions that take place before the arbiter responds to a request signal. In our implementation, on the other hand, we allow the token to circulate even when the arbiter has not received a request. [Ebergen, 1992] derives a token-ring implementation of an arbiter by applying the Separation Theorem from [Ebergen, 1991]. Ebergen's Separation Theorem is similar to our part-wise design method, but it does not address progress concerns.

Figure 6.12 shows the topology of a token-ring implementation that we are aiming for. Signals t, t_0, t_1, and t_2 represent the propagation of the token around the ring. Notice that the token can be inserted into the ring by the "done" signal. Granting a request takes the token out of the ring.

We introduce internal symbols into the specification of the SEQUENCER such that

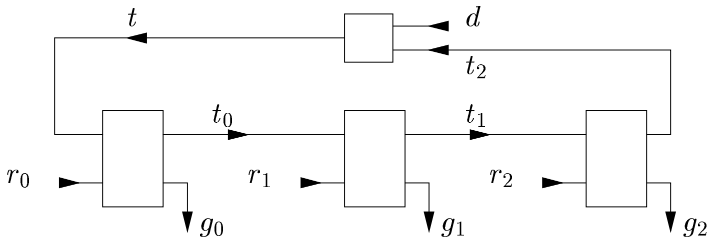


Figure 6.12: The topology of a token-ring implementation for a SEQUENCER

the internal symbols reflect the operation of the token ring:

$$
\begin{array}{llll}
\mathrm{SEQ}_0 & = & *[r0?;g_0!] & \text{(snippet } A\text{)} \\
& \& & *[r1?;g_1!] & \text{(snippet } B\text{)} \\
& \& & *[r2?;g_2!] & \text{(snippet } C\text{)} \\
& \& & *[t!;(g_0!;d?|t_0!;(g_1!;d?|t_1!;(g_2!;d?|t_2!)))] & \text{(snippet } D\text{)}
\end{array}
$$

Snippets A, B, and C come straight from the specification SEQ. Snippet D, on the other hand, introduces the token ring. One trip around the ring consists of the sequence $t\,t_0\,t_1\,t_2$. Initially, the token starts traveling around the ring, which is indicated by symbol t. Once the token comes around the ring, signal t_2 is produced. At that point, the token is re-sent around the ring. The token passes through three stations on its way around the ring. At each station a pending request can be granted or the token gets passed along the ring. At the first station, grant $g_0!$ can be produced or the token can be sent along, which is indicated by symbol t_0. The token is consumed upon granting a request and a new token is generated when the done signal d is received.

SEQ_0 can receive inputs at any time that specification SEQ can receive inputs. Namely, inputs r_0, r_1, and r_2 appear only in snippets A, B, and C, respectively, and these three snippets are present in SEQ. Furthermore, SEQ can receive a "done" input

d only after a request has been granted. The same holds for SEQ_0.

Specification SEQ guarantees that, once a request has been received, it will be granted, unless the arbiter has to wait for a "done" signal. That is, unless SEQ waits for a "done" signal, progress is guaranteed. Because the token ring in SEQ_0 cannot stop until a pending request is granted, and because a token is consumed when a request is granted, we conclude that SEQ_0 also guarantees progress unless it waits for a "done" signal. Consequently, we have

$$\mathrm{SEQ} \sqsubseteq |[\, t, t_0, t_1, t_2 :: \mathrm{SEQ}_0 \,]|$$

Snippet D from SEQ_0 can be refined as follows:

$$\begin{array}{l} *\,[t!;(g_0!;d?|t_0!;(g_1!;d?|t_1!;(g_2!;d?|t_2!)))] \\ \qquad \sqsubseteq_o \quad *[t!;(d?|t_2!)] \;\;\&\;\; *\,[\,t!;(g_0! \mid t_0!)\,] \;\;\&\;\; *\,[\,t_0!;(g_1! \mid t_1!)\,] \;\;\&\;\; *\,[\,t_1!;(g_2! \mid t_2!)\,] \end{array}$$

We apply the monotonicity of output-persistent refinement, and we get

$$\mathrm{SEQ}_0 \quad \sqsubseteq_o \quad \mathrm{SEQ}_1$$

where

$$\begin{array}{lcll} \mathrm{SEQ}_1 & = & *[\,r_0?;g_0!\,] & \text{(snippet } A\text{)} \\ & \& & *[\,r_1?;g_1!\,] & \text{(snippet } B\text{)} \\ & \& & *[\,r_2?;g_2!\,] & \text{(snippet } C\text{)} \\ & \& & *[t!;(t_2!|d?)] \;\;\&\;\; *\,[\,t!;(g_0! \mid t_0!)\,] \;\;\& & \\ & & *[\,t_0!;(g_1! \mid t_1!)\,] \;\;\&\;\; *\,[\,t_1!;(g_2! \mid t_2!)\,] & \text{(snippet } D'\text{)} \end{array}$$

Following our design method we find implementations for each of the snippets that form SEQ_1:

$$
\begin{array}{lll}
A & \sqsubseteq_o & *[\, r_0?; g_0! \,] \\
B & \sqsubseteq_o & *[\, r_1?; g_1! \,] \\
C & \sqsubseteq_o & *[\, r_2?; g_2! \,] \\
D' & \sqsubseteq_o & *[t!; (t_2?|d?)] \;\|\; *[\, t?; (g_0! \mid t_0!) \,] \;\|\; *[\, t_0?; (g_1! \mid t_1!) \,] \;\|\; *[\, t_1?; (g_2! \mid t_2!) \,]
\end{array}
$$

All refinements above are trivial. Snippets A, B, and C are refined by themselves, and in the refinement of snippet D' we assigned the input and output alphabets such that the alphabet conditions for the network composition are satisfied.

After grouping snippets with common outputs we get

$$
\begin{array}{llll}
\mathrm{SEQ}_1 \;\sqsubseteq & & *[\, r_0?; g_0! \,] \;\&\; *\,[\, t?; (g_0! \mid t_0!) \,] & \text{(ATS)} \\
& \| & *[\, r_1?; g_1! \,] \;\&\; *\,[\, t_0?; (g_1! \mid t_1!) \,] & \text{(ATS)} \\
& \| & *[\, r_2?; g_2! \,] \;\&\; *\,[\, t_1?; (g_2! \mid t_2!) \,] & \text{(ATS)} \\
& \| & *[t!; (t_2?|d?)] & \text{(initialized MERGE)}
\end{array}
$$

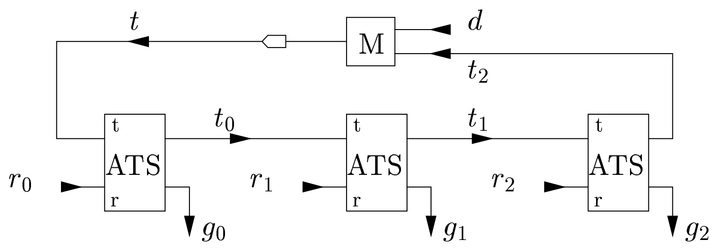


Figure 6.13: A token-ring implementation of a 3-input SEQUENCER

Thus, the 3-input SEQUENCER can be implemented by a network consisting of an initialized MERGE and of three ATS components connected in a ring as shown in Figure 6.13. An initialized MERGE behaves exactly like a regular MERGE, only that it starts the operation by producing an output.

6.8 Summary

In this chapter we demonstrated how ECF models can be applied to finding implementations for a variety of specifications. In particular, we presented a part-wise design method that may help us avoid the state explosion that often occurs when we verify an implementation of a specification of a large component. For example, our derivations of implementations of a FIFO, of a scheduler for dining philosophers, and of a 3-input SEQUENCER can all be generalized to inductive derivations of implementations for these components, where the size of a component is arbitrary. The examples in this section also demonstrate that we can arrive at an implementation that may involve a large degree of concurrency by verifying a number refinements of simple snippets representing simple sequential behaviors. Then we can apply a number of properties of the ECF model and perform a sequence of simple algebraic manipulations to transform implementations of individual snippets into an implementation of the original specification.

Chapter 7

Conclusions

ECF processes are a simple framework for describing concurrent systems. The ECF model is built on top of enhanced characteristic functions which assign labels to traces that represent sequences of communication actions. We use labels in order to express properties of a state of a process after the process has executed a trace. The general ECF model does not depend on a particular set of labels. We only have to assume that the labels enjoy a number of simple properties. Based on the properties of labels, we define operations on processes.

The close connections between labels and processes allows, in many instances, for simple proofs of properties that hold for operations on processes. We agree with Tom Verhoeff, who introduced enhanced characteristic functions [Verhoeff, 1994], that "it is intriguing that so many properties can be lifted from a small algebra on trace labels to ECFs". Tom Verhoeff found "the tediousness and sheer number of details concerning ECFs disappointing". In contrast to Verhoeff's XDI model, which captures the delay-insensitive paradigm, our ECF model is based on the assumption of speed-independence. We find the ECF model quite simple, and we believe that delay-insensitivity was the source of the tediousness that disappointed Tom Verhoeff.

ECF processes have a "plug-and-play" nature. Namely, we only have to choose a set of labels and use it to instantiate a concrete ECF model, where all the properties of the general abstract model still hold. We study two such instantiations, the safety model and the progress model. The safety model addresses safety of processes. The progress model, on the other hand, addresses progress properties on top of safety properties.

In both the safety model and the progress model we were able to provide three equivalent characterizations of refinement. One characterization is based on comparison of trace labels. Another characterization is based on Ebergen's approach [Ebergen, 1991] that requires the correctness of an implementation operating in the environment determined by the reflection of the specification. The third characterization of refinement captures the testing paradigm, which was the basis for Verhoeff's refinement in the XDI model [Verhoeff, 1994]. We see the existence of three equivalent characterizations of refinement as a justification for our definition of refinement and as a justification for the definitions of our correctness conditions.

Choosing whether the composition of processes will hide the connections between processes is an important step in developing a formalism. In a contrast to [Verhoeff, 1994], we chose that the connections between the processes remain visible. We believe that this choice resulted in a simpler definition of process product. In order to gain the ability to conceal connections between components, we defined a separate operation called hiding. In both, the safety model and the progress model, we gave an alternative definition for hiding, which establishes a connection between product and hiding, and between Verhoeff's composition. We see this connection as a justification for our definition of hiding.

In the safety model, the process product has a dual role. First, the product can be used as a network composition, modeling joint operation of *independent* processes. Second, the product can be used for combining behavioral constraints into a specification of *one* component. This dual role of the product mirrors the use of process product (or composition) in previous work such as [Jong and Lin, 1994], [Ebergen et al., 1995], [Pena and Cortadella, 1996] and [Negulescu and Peeters, 1998].

In the progress model, however, we learned that the process product was not an appropriate operation for combining behavioral constraints. In order to be able to define the specification composition, we defined a restricted set of processes, called snippets. Furthermore, in order to gain monotonicity of specification composition, we strengthened the refinement relation. We demonstrated through a number of examples how snippets can express behavioral constraints and how the specification composition can be used for combining snippets into specifications. Introducing restrictions is, of course, always a disappointment. We showed, however, that the restrictions were necessary in order to prove a number of formal properties of specification composition. On the basis of the examples we worked out we believe that the restrictions were not overly cumbersome.

In order to avoid restricting the process domain for the specification composition, we attempted to develop another model, where we attached labels to transitions instead of attaching labels to states. Unfortunately, we have not been successful in defining an appropriate hiding operation. We believe that attaching labels to transitions might be a viable approach, thus we include development of such a model in our plans for future work.

We highlighted three different design techniques that can be used with ECF processes. The Substitution Theorem allows for the standard hierarchical design by means of step-wise refinement. The Factorization Theorem opens the door for a design approach advocated in [Verhoeff, 1994] and further refined in [Mallon et al., 1999]. In this design technique we guess a part of an implementation and the Factorization Theorem gives us a bound for the remainder of the implementation. The third design method, called part-wise refinement, is similar to Ebergen's Separation Theorem [Ebergen, 1991], but now we can also take into account progress concerns. The part-wise design method applies to specifications constructed as a composition of snippets. We can find in isolation implementations for each of the snippets. Then, we apply some theorems and obtain an implementation of the original specification. We find the part-wise design method attractive, because it can prevent state explosion. Namely, implementations of individual snippets tend to be simple enough that there is no danger of state explosion. The rest of the work is then done by applying theorems.

We illustrated the part-wise design method on a number of examples, some of them non-trivial. We felt that larger examples were beyond the scope of this thesis. On the basis of [Ebergen and Berks, 1995] we have strong evidence that the part-wise design method can be applied to a derivation of an implementation of the counterflow-pipeline control circuit [Sproull et al., 1994]. We include that derivation and the search for more design examples in our plans for future work. We also believe that future efforts could be applied towards identifying heuristics that could help us in applying the part-wise design method. We mentioned several such heuristics in our examples.

At the moment, we do not have a tool that would directly support verification, design, or simulation using ECF processes. We were, however, able to use VERDECT [Ebergen and Berks, 1995] to help us with the verification of implementations from Chapter 6. Our use of VERDECT required a rather intricate understanding of the tool and its underlying formalism, because VERDECT does not apply to ECF processes in general. Our list of projects for the future includes development of tools that would help with the design process based on ECF processes.

Finally, we remark that our application of the theory of ECF processes has been, so far, limited to asynchronous circuits. Are there applications in the design and analysis of concurrent programs? Applications of trace theory in Tangram [Berkel et al., 1991] lead us to believe that concurrent programs can be analysed using ECF processes. We leave this application of ECF processes for the future work.

Appendix A

Glossary of symbols

t, s, u, v	traces
ε	the empty trace
A, B	sets
$t \downarrow A$	the projection of trace t on set A
$\sqcup A$	least upper bound of set A
$\sqcap A$	greatest lower bound of set A
A^*	the Kleene closure of set A
Λ	a set of labels
λ, γ, δ	labels
$\sim\lambda$	the reflection of a label
$\lambda \times \gamma$	the product of labels
$\lambda \sqsubseteq \gamma$	label λ is refined by label γ
$\mathcal{PROC}(I, O)$	the set of processes with input alphabet I and output alphabet O
P, Q, R	processes
$\mathsf{ABORT}(I, O)$	the abort process with input alphabet I and output alphabet O

$\mathsf{ABORT}(I, O)$	the miracle process with input alphabet I and output alphabet O
$\mathbf{i}.P$	the input alphabet of process P
$\mathbf{o}.P$	the output alphabet of process P
$\mathbf{a}.P$	the alphabet of process P
$\mathbf{f}.P$	the enhanced characteristic function for process P
$\mathbf{l}.P$	the set of legal traces of process P
$\mathbf{u}_\nabla.P$	the set of ∇-unhealthy traces of process P
$\mathbf{u}_\Delta.P$	the set of Δ-unhealthy traces of process P
$\mathbf{u}.P$	the set of unhealthy traces of process P
$\lfloor P \rfloor$	process P rounded down
$\lceil P \rceil$	process P rounded up
$[P]$	process P rounded
$\sim P$	the reflection of process P
$P \times Q$	the product of processes P and Q
$P \otimes Q$	the rounded product of processes P and Q
$P \parallel Q$	the network composition of processes P and Q
$P \,\&\, Q$	the specification composition of snippets P and Q
$\|[A :: P]\|$	process P where outputs in A were hidden
$P \sqsubseteq Q$	process P is refined by process Q
$P \preceq_o Q$	process Q is output-persistent with respect to process P
$P \sqsubseteq_o Q$	snippet P is output-persistence refined by snippet Q
$\mathbf{safe}.P$	process P satisfies the safety condition
$\mathbf{prog}.P$	process P satisfies the progress condition
$\mathbf{correct}.P$	process P satisfies the correctness condition

Appendix B

Ordered sets and lattices

In this appendix we briefly summarize some definitions and theorems from the theory of ordered sets and lattices. We refer the reader to [Davey and Priestley, 1990] for more details.

B.1 Relations

Let $\sqsubseteq$ be a binary relation on set A; that is, $\sqsubseteq\ \subseteq\ A \times A$. We often write $u \sqsubseteq v$ for $(u, v) \in \sqsubseteq$. The following table summarizes common terminology for relations.

R is	when
reflexive	$u \sqsubseteq u$ for all $u \in A$
anti-reflexive	$\neg(u \sqsubseteq u)$ for all $u \in A$
symmetric	$u \sqsubseteq v \Leftrightarrow v \sqsubseteq u$ for all $u, v \in A$
antisymmetric	$u \sqsubseteq v \ \wedge\ v \sqsubseteq u \ \Rightarrow\ u = v$ for all $u, v \in A$
transitive	$u \sqsubseteq v \ \wedge\ v \sqsubseteq w \ \Rightarrow u \sqsubseteq w$ for all $u, v, w \in A$

A relation is called *pre-order* when it is reflexive and transitive. An antisymmetric pre-order is called *partial order*. A partial order is a *total order*, when $u \sqsubseteq v \vee v \sqsubseteq u$ for any $u, v \in A$.

B.2 Ordered sets and lattices

For partial order $\sqsubseteq$ on set A, we call pair $(A, \sqsubseteq)$ a partially ordered set or a *poset*.

Let $(A, \sqsubseteq)$ be a poset and let $B \subseteq A$. Then, $(B, \sqsubseteq')$ is also a poset for $\sqsubseteq'$ being the restriction of $\sqsubseteq$ to B. It is customary to denote the restricted order $\sqsubseteq'$ by $\sqsubseteq$.

$v \in A$ is a *lower bound* of set B, denoted by $v \sqsubseteq B$, and defined as

$$(\forall u :\ u \in B :\ v \sqsubseteq u\,)$$

Dually, $v \in A$ is an *upper bound* of set B, denoted by $B \sqsubseteq v$, and defined as

$$(\forall u :\ u \in B :\ u \sqsubseteq v\,)$$

Set B can have several upper bounds and several lower bounds. We call v the *least* in B, or *minimum* in B, when $u \in B$ and $u \sqsubseteq B$. Dually, v is called *greatest* in B, or the *maximum* of B, when $v \in B$ and $B \sqsubseteq v$. When the minimum and the maximum of BC exist, they are denoted by $\min B$ and $\max B$, respectively.

We call v a *greatest lower bound*, or *infimum*, of B, when v is the greatest in the set of all lower bounds of B. If a greatest lower bound exists, it is unique, and we denote it by $\sqcap B$. We write $u \sqcap v$ for $\sqcap\{u, v\}$. Dually, v is called a *least upper bound*, or *supremum*, of B if v is the least in the set of all upper bounds of B. If it exists, the least upper bound of B is unique and is denoted by $\sqcup B$. We write $u \sqcup v$ for $\sqcup\{u, v\}$.

Poset $(A, \sqsubseteq)$ is a *lattice* when $u \sqcap v$ and $u \sqcup v$ exist for all $u, v \in A$. If $(A, \sqsubseteq)$ is a lattice, then $\sqcap$ and $\sqcup$ can be seen as binary operators on A. $\sqcap$ and $\sqcup$ enjoy the following properties for any $u, v, w \in A$:

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
u \sqsubseteq (v \sqcap w) & \Leftrightarrow & u \sqsubseteq v \ \wedge \ u \sqsubseteq w & u \sqsubseteq v & \Leftrightarrow & u \sqcap v = u \\
(u \sqcup v) \sqsubseteq w & \Leftrightarrow & u \sqsubseteq w \ \wedge \ v \sqsubseteq w & u \sqsubseteq v & \Leftrightarrow & u \sqcup v = v
\end{array}
$$

Poset $(A, \sqsubseteq)$ is a *complete lattice* when $\sqcap B$ and $\sqcup B$ exist for any $B \subseteq A$. Equivalently, $(A, \sqsubseteq)$ is a complete lattice when $\max A$ exists, and $\sqcap B$ exists for any $B \subseteq A$. Any finite lattice is a complete lattice.

Appendix C

Proofs

Lemma 3.6.2

The following holds for labels λ and γ:

$$(\sim\!\lambda \times \gamma \sqsupseteq \bigcirc) \;\Leftrightarrow\; (\lambda \sqsubseteq \gamma)$$

Proof: The table below lists all pairs of labels λ and γ. Notice that, for all pairs of labels, $\lambda \sqsubseteq \gamma$ coincides exactly with $\sim\!\lambda \times \gamma \neq \bot$.

λ	γ	$\sim\!\lambda$	$\sim\!\lambda \times \gamma$	$\lambda \sqsubseteq \gamma$
$\bot$	$\bot$	$\top$	$\top$	✓
$\bot$	$\bigcirc$	$\top$	$\top$	✓
$\bot$	$\top$	$\top$	$\top$	✓
$\bigcirc$	$\bot$	$\bigcirc$	$\bot$	✗
$\bigcirc$	$\bigcirc$	$\bigcirc$	$\bigcirc$	✓
$\bigcirc$	$\top$	$\bigcirc$	$\top$	✓

λ	γ	$\sim\!\lambda$	$\sim\!\lambda \times \gamma$	$\lambda \sqsubseteq \gamma$
$\top$	$\bot$	$\bot$	$\bot$	✗
$\top$	$\bigcirc$	$\bot$	$\bot$	✗
$\top$	$\top$	$\bot$	$\top$	✓

$\square$

Lemma 3.6.3

For process P, where $\mathbf{a}.P = \mathbf{o}.P$, we have:

$$\boldsymbol{safe}.P \;\Leftrightarrow\; (\forall t:\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \sqsupseteq \bigcirc\,)$$

Proof:

$$
\begin{array}{ll}
& \mathbf{safe}.P \\
\Leftrightarrow & \{\text{ Definition 3.5.1 }\} \\
& \big(\forall t:\ t \in (\mathbf{l}.P)(\mathbf{o}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.(t) \neq \bot\big) \\
\Leftrightarrow & \{\ \mathbf{i}.P = \emptyset, \text{ thus } \mathbf{o}.P = \mathbf{a}.P\ \} \\
& \big(\forall t:\ t \in (\mathbf{l}.P)(\mathbf{a}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.(t) \neq \bot\big) \\
\Leftrightarrow & \{\text{ Equation 2.4 — } \bot\text{-persistence }\} \\
& \big(\forall t:\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.(t) \neq \bot\big) \\
\Leftrightarrow & \{\text{ Equation 3.2 — partial order; Calculus }\} \\
& \big(\forall t:\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.(t) \sqsupseteq \bigcirc\big)
\end{array}
$$

□

Lemma 3.6.4

Let P and Q be processes, such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P \subseteq \mathbf{o}.Q$. Then,

$$(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \bigcirc) \;\Leftrightarrow\; (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \bigcirc)$$

Proof: First we assume $(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \bigcirc)$. Because $\mathbf{l}.({\sim}P \times$

$Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\} \subseteq (\mathbf{a}.Q)^*$, we have, by definition of product

$$
\begin{array}{ll}
& (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.(\sim P \times Q).t \sqsupseteq \bigcirc) \\
\Rightarrow & (\forall t : t \in \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\} : \mathbf{f}.\sim P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \bigcirc)
\end{array}
$$

Now take trace $t \in (\mathbf{a}.Q)^*$ such that $t \notin \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\}$. Then, $t = t'at''$, where $t' \in \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q)$ and $t'a \notin \mathbf{l}.(\sim P \times Q)$. By assumption, $\mathbf{f}.(\sim P \times Q).t'a \sqsupseteq \bigcirc$. Because $t'a$ is not a legal trace in $\sim P \times Q$, it must be the case that $\mathbf{f}.(\sim P \times Q).t'a = \top$. By Equation 3.5 and by Table 3.1 we know that either $\mathbf{f}.(\sim P).(t'a \downarrow \mathbf{a}.P) = \top$ or $\mathbf{f}.Q.t'a = \top$. By $\top$-persistence, either $\mathbf{f}.(\sim P).(t \downarrow \mathbf{a}.P) = \top$ or $\mathbf{f}.Q.t = \top$. Consequently, $\mathbf{f}.\sim P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t = \top \sqsupseteq \bigcirc$.

Now we prove the implication from right to left:

$$
\begin{array}{ll}
& (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.\sim P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \bigcirc) \\
\Rightarrow & \{\ \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\} \subseteq (\mathbf{a}.Q)^*\ \} \\
& (\forall t : t \in \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\} : \mathbf{f}.\sim P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \bigcirc) \\
\Rightarrow & \{\ \text{Definition 2.4.1 – product, Definition 2.3.5 – reflection}\ \} \\
& (\forall t : t \in \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\} : \mathbf{f}.(\sim P \times Q).t \sqsupseteq \bigcirc) \\
\Rightarrow & \{\ \bot\text{-persistence}\ \} \\
& (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.(\sim P \times Q).t \sqsupseteq \bigcirc)
\end{array}
$$

□

Lemma 3.8.2

Let P and R be processes such that $\mathbf{i}.R = \mathbf{i}.P$ and $\mathbf{o}.R = \mathbf{o}.P - A$ for some set A. Then

$$
\textbf{\textit{safe}}.(\sim R \times P) \ \Leftrightarrow\ R \sqsubseteq |[\, A :: P\,]|
$$

Proof:

$$\begin{array}{ll}
& \mathbf{safe}.(\sim R \times P) \\
\Leftrightarrow & \{\ \mathbf{a}.(\sim R \times P) = \mathbf{o}.(\sim R \times P) = \mathbf{a}.P, \text{ Lemma 3.6.3 }\} \\
& (\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.(\sim R \times P) \sqsupseteq \bigcirc) \\
\Leftrightarrow & \{\ \text{Lemma 3.6.4 }\} \\
& (\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.\sim R.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.P.t \sqsupseteq \bigcirc) \\
\Leftrightarrow & \{\ \text{Definition 2.3.5, Lemma 3.6.2 }\} \\
& (\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.R.(t \downarrow \mathbf{a}.P) \sqsubseteq \mathbf{f}.P.t) \\
\Leftrightarrow & \{\ \text{Calculus }\} \\
& \big(\forall s : s \in (\mathbf{a}.R)^* : (\forall t : t \in (\mathbf{a}.P)^* \wedge t \downarrow \mathbf{a}.R = s : \mathbf{f}.R.s \sqsubseteq \mathbf{f}.P.t)\big) \\
\Leftrightarrow & \{\ \text{Calculus }\} \\
& \big(\forall s : s \in (\mathbf{a}.R)^* : \mathbf{f}.R.s \sqsubseteq (\sqcap t : t \in (\mathbf{a}.P)^* \wedge t \downarrow \mathbf{a}.R = s : \mathbf{f}.P.t)\big) \\
\Leftrightarrow & \{\ \mathbf{i}.R = \mathbf{i}.P,\ \mathbf{o}.R = \mathbf{o}.R - A, \text{ Definition 2.6.1 -- hiding }\} \\
& (\forall s : s \in (\mathbf{a}.R)^* : \mathbf{f}.R.s \sqsubseteq \mathbf{f}.|[\, A :: P \,]|.s) \\
\Leftrightarrow & \{\ \text{Definition 2.3.1 -- refinement }\} \\
& R \sqsubseteq |[\, A :: P \,]|
\end{array}$$

□

Lemma 3.8.3

Let $\mathcal{A}$ be a set of processes. Then,

$$\sqcup\{P :\ P \in \mathcal{A} :\ P\} \ = \ \sim\sqcap\{P :\ P \in \mathcal{A} :\ \sim P\}$$

Proof: Let $R = \sqcup\{P :\ P \in \mathcal{A} :\ P\}$. By definition of $\sqcup$, R is an upper bound of $\mathcal{A}$

$$(\forall P :\ P \in \mathcal{A} :\ P \sqsubseteq R)$$

and R is smaller than any other upper bound of R

$$(\forall Q:\ Q \text{ an upper bound of } \mathcal{A}:\ R \sqsubseteq Q\,)$$

$$
\begin{array}{ll}
& (\forall P:\ P \in \mathcal{A}:\ P \sqsubseteq R\,)\ \wedge\ (\forall Q:\ Q \text{ an upper bound of } \mathcal{A}:\ R \sqsubseteq Q\,) \\
\Leftrightarrow & \{\ \text{Property 2.3.6: } P \sqsubseteq R\ \Rightarrow\ {\sim}R \sqsubseteq {\sim}P\ \} \\
& (\forall P:\ P \in \mathcal{A}:\ {\sim}R \sqsubseteq {\sim}P\,)\ \wedge\ (\forall Q:\ Q \text{ an upper bound of } \mathcal{A}:\ {\sim}Q \sqsubseteq {\sim}R\,) \\
\Leftrightarrow & \{\ \text{Write } S = {\sim}R.\ Q \text{ is an upper bound of } \mathcal{A} \text{ implies, by Property 2.3.6, that} \\
& \quad {\sim}Q \text{ is a lower bound of } \mathcal{A}.\ \} \\
& (\forall P:\ P \in \mathcal{A}:\ S \sqsubseteq {\sim}P\,)\ \wedge\ (\forall Q:\ Q \text{ a lower bound of } \mathcal{A}:\ Q \sqsubseteq S\,)
\end{array}
$$

By definition of $\sqcap$, we get

$$S = \sqcap\{P:\ P \in \mathcal{A}:\ {\sim}P\}$$

Because $R = {\sim}S$, we have

$$R = \sqcup\{P:\ P \in \mathcal{A}:\ P\} = {\sim} \sqcap \{P:\ P \in \mathcal{A}:\ {\sim}P\}$$

$\square$

Lemma 4.6.4

For process P, where $\mathbf{a}.P = \mathbf{o}.P$, we have:

$$\boldsymbol{safe}.P\ \wedge\ \boldsymbol{prog}.P\ \ \Leftrightarrow\ \ (\forall t:\ t \in (\mathbf{a}.P)^*:\ \mathbf{f}.P.t \sqsupseteq \square\,)$$

Proof:

$$\mathbf{safe}.P \ \wedge \ \mathbf{prog}.P$$

$\Leftrightarrow$ { Definitions 4.6.1 and 4.6.2 }

$$\big(\forall t :\ t \ \in \ (\mathbf{l}.P)(\mathbf{o}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.(t) \neq \bot\big) \ \wedge \ \big(\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \neq \Delta\big)$$

$\Leftrightarrow$ { $\mathbf{i}.P = \emptyset$, thus $\mathbf{o}.P = \mathbf{a}.P$ }

$$\big(\forall t :\ t \ \in \ (\mathbf{l}.P)(\mathbf{a}.P) \cup \{\varepsilon\} :\ \mathbf{f}.P.(t) \neq \bot\big) \ \wedge \ \big(\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \neq \Delta\big)$$

$\Leftrightarrow$ { Equation 2.4 — $\bot$-persistence }

$$\big(\forall t :\ t \ \in \ (\mathbf{a}.P)^* :\ \mathbf{f}.P.(t) \neq \bot\big) \ \wedge \ \big(\forall t :\ t \in (\mathbf{a}.P)^* :\ \mathbf{f}.P.t \neq \Delta\big)$$

$\Leftrightarrow$ { Equation 4.1 — partial order; Calculus }

$$\big(\forall t :\ t \ \in \ (\mathbf{a}.P)^* :\ \mathbf{f}.P.(t) \sqsupseteq \Box\big)$$

$\Box$

Lemma 4.6.5

The following holds for labels λ and γ:

$$\lambda \sqsubseteq \gamma \ \ \Leftrightarrow \ \ ({\sim}\lambda \times \gamma) \sqsupseteq \Box$$

Proof: We first inspect all pairs of labels λ and γ for which $\lambda \sqsubseteq \gamma$ holds. The following list shows that the left-hand side of the equivalence above implies the right-hand side:

$$\begin{array}{lcccccc}
\bot \sqsubseteq \bot & : & {\sim}\bot \times \bot & = & \top \times \bot & = & \top \\
\bot \sqsubseteq \Delta & : & {\sim}\bot \times \Delta & = & \top \times \Delta & = & \top \\
\bot \sqsubseteq \Box & : & {\sim}\bot \times \Box & = & \top \times \Box & = & \top \\
\bot \sqsubseteq \nabla & : & {\sim}\bot \times \nabla & = & \top \times \nabla & = & \top \\
\bot \sqsubseteq \top & : & {\sim}\bot \times \top & = & \top \times \top & = & \top \\
\Delta \sqsubseteq \Delta & : & {\sim}\Delta \times \Delta & = & \nabla \times \Delta & = & \nabla \\
\Delta \sqsubseteq \Box & : & {\sim}\Delta \times \Box & = & \nabla \times \Box & = & \nabla \\
\Delta \sqsubseteq \nabla & : & {\sim}\Delta \times \nabla & = & \nabla \times \nabla & = & \nabla \\
\Delta \sqsubseteq \top & : & {\sim}\Delta \times \top & = & \nabla \times \top & = & \top \\
\Box \sqsubseteq \Box & : & {\sim}\Box \times \Box & = & \Box \times \Box & = & \Box \\
\Box \sqsubseteq \nabla & : & {\sim}\Box \times \nabla & = & \Box \times \nabla & = & \nabla \\
\Box \sqsubseteq \top & : & {\sim}\Box \times \top & = & \Box \times \top & = & \top \\
\nabla \sqsubseteq \nabla & : & {\sim}\nabla \times \nabla & = & \Delta \times \nabla & = & \nabla \\
\nabla \sqsubseteq \top & : & {\sim}\nabla \times \top & = & \Delta \times \top & = & \top \\
\top \sqsubseteq \top & : & {\sim}\top \times \top & = & \bot \times \top & = & \top
\end{array}$$

Now we check the implication in the opposite direction. The encircled entries in the table below are the entries for which the right-hand side of the equivalence above holds:

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

The encircled entries exactly correspond to the list above, where we assumed that the left-hand side of the equivalence holds. This means that the implication holds in both ways. $\Box$

Lemma 4.6.6

Let P and Q be processes, such that $\mathbf{i}.P = \mathbf{i}.Q$ and $\mathbf{o}.P \subseteq \mathbf{o}.Q$. Then,

$$(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \Box) \;\Leftrightarrow\; (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \Box)$$

Proof: First we assume $(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \Box)$. Because $\mathbf{l}.({\sim}P \times Q)\mathbf{a}.({\sim}P \times Q) \cup \{\varepsilon\} \subseteq (\mathbf{a}.Q)^*$, we have, by definition of product

$$\begin{array}{ll} & (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.({\sim}P \times Q).t \sqsupseteq \Box) \\ \Rightarrow & (\forall t : t \in \mathbf{l}.({\sim}P \times Q)\mathbf{a}.({\sim}P \times Q) \cup \{\varepsilon\} : \mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \Box) \end{array}$$

Now take trace $t \in (\mathbf{a}.Q)^*$ such that $t \notin \mathbf{l}.({\sim}P \times Q)\mathbf{a}.({\sim}P \times Q) \cup \{\varepsilon\}$. Then, $t = t'at''$, where $t' \in \mathbf{l}.({\sim}P \times Q)\mathbf{a}.({\sim}P \times Q)$ and $t'a \notin \mathbf{l}.({\sim}P \times Q)$. By assumption, $\mathbf{f}.({\sim}P \times Q).t'a \sqsupseteq \Box$. Because $t'a$ is not a legal trace in ${\sim}P \times Q$, it must be the case that $\mathbf{f}.({\sim}P \times Q).t'a = \top$. By Equation 3.5 and by Table 3.1 we know that either $\mathbf{f}.{\sim}P.(t'a \downarrow \mathbf{a}.P) = \top$ or $\mathbf{f}.Q.t'a = \top$. By $\top$-persistence, either $\mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) = \top$ or $\mathbf{f}.Q.t = \top$. Consequently, $\mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t = \top \sqsupseteq \Box$.

Now we prove the implication from right to left:

$$\begin{array}{ll} & (\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \Box) \\ \Rightarrow & \{\ \mathbf{l}.({\sim}P \times Q)\mathbf{a}.({\sim}P \times Q) \cup \{\varepsilon\} \subseteq (\mathbf{a}.Q)^*\ \} \\ & (\forall t : t \in \mathbf{l}.({\sim}P \times Q)\mathbf{a}.({\sim}P \times Q) \cup \{\varepsilon\} : \mathbf{f}.{\sim}P.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.Q.t \sqsupseteq \Box) \\ = & \{\ \text{Definition 2.4.1 — product}\ \} \end{array}$$

$(\forall t : t \in \mathbf{l}.(\sim P \times Q)\mathbf{a}.(\sim P \times Q) \cup \{\varepsilon\} : \mathbf{f}.(\sim P \times Q).t \sqsupseteq \Box)$

$\Rightarrow \quad \{\ \bot\text{-persistence}\ \}$

$(\forall t : t \in (\mathbf{a}.Q)^* : \mathbf{f}.(\sim P \times Q).t \sqsupseteq \Box)$

$\Box$

Lemma 4.10.2

Let P and R be processes such that $\mathbf{i}.R = \mathbf{i}.P$ and $\mathbf{o}.R = \mathbf{o}.P - A$ for some set A. Then

$$\boldsymbol{correct}.(\sim R \times P) \quad \Leftrightarrow \quad R \sqsubseteq |[\, A :: P \,]|$$

Proof:

$\mathbf{correct}.(\sim R \times P)$

$\Leftrightarrow \quad \{\ \mathbf{a}.(\sim R \times P) = \mathbf{o}.(\sim R \times P) = \mathbf{a}.P$, Lemma 4.6.4 $\}$

$(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.(\sim R \times P) \sqsupseteq \Box)$

$\Leftrightarrow \quad \{$ Lemma 4.6.6 $\}$

$(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.\sim R.(t \downarrow \mathbf{a}.P) \times \mathbf{f}.P.t \sqsupseteq \Box)$

$\Leftrightarrow \quad \{$ Lemma 4.6.5 $\}$

$(\forall t : t \in (\mathbf{a}.P)^* : \mathbf{f}.R.(t \downarrow \mathbf{a}.P) \sqsubseteq \mathbf{f}.P.t)$

$\Leftrightarrow \quad \{$ Calculus $\}$

$\big(\forall s : s \in (\mathbf{a}.R)^* : (\forall t : t \in (\mathbf{a}.P)^* \wedge t \downarrow \mathbf{a}.R = s : \mathbf{f}.R.s \sqsubseteq \mathbf{f}.P.t)\big)$

$\Leftrightarrow \quad \{$ Calculus $\}$

$\big(\forall s : s \in (\mathbf{a}.R)^* : \mathbf{f}.R.s \sqsubseteq (\sqcap t : t \in (\mathbf{a}.P)^* \wedge t \downarrow \mathbf{a}.R = s : \mathbf{f}.P.t)\big)$

$\Leftrightarrow \quad \{\ \mathbf{i}.R = \mathbf{i}.P,\ \mathbf{o}.R = \mathbf{o}.R - A$, Definition 2.6.1 – hiding $\}$

$(\forall s : s \in (\mathbf{a}.R)^* : \mathbf{f}.R.s \sqsubseteq \mathbf{f}.|[\, A :: P \,]|.s)$

$\Leftrightarrow \quad \{$ Definition 2.3.1 – refinement $\}$

$R \sqsubseteq |[\, A :: P \,]|$

$\square$

Lemma 5.5.4

For snippets P, Q, and R, we have

$$\lfloor \lfloor P \times Q \rfloor \times R \rfloor = \lfloor P \times Q \times R \rfloor$$

Proof: For trace t, we prove

$$\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \mathbf{f}.\lfloor P \times Q \times R \rfloor.t$$

First, assume that $\mathbf{f}.\lfloor P \times Q \rfloor.(t \downarrow \mathbf{a}.(P \times Q)) = \mathbf{f}.(P \times Q).(t \downarrow \mathbf{a}.(P \times Q))$. That is, no rounding down takes place. In this case, $\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \mathbf{f}.\lfloor P \times Q \times R \rfloor.t$.

Now assume $\mathbf{f}.\lfloor P \times Q \rfloor.t = \square$ and $\mathbf{f}.(P \times Q).t = \nabla$. That is, rounding down takes place, because t is ∇-unhealthy in $P \times Q$:

$$(\forall a :\ a \in \mathbf{o}.(P \times Q) : \mathbf{f}.(P \times Q).(ta \downarrow \mathbf{a}.(P \times Q)) = \top) \tag{C.1}$$

We have four possibilities for $\mathbf{f}.R.(t \downarrow \mathbf{a}.R)$:

- $\mathbf{f}.R.(t \downarrow \mathbf{a}.R) = \bot$: Then, by the definitions of product and of rounding, $\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \mathbf{f}.\lfloor P \times Q \times R \rfloor.t = \bot$.

- $\mathbf{f}.R.(t \downarrow \mathbf{a}.R) = \top$: Then, by the definitions of product and of rounding, $\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \mathbf{f}.\lfloor P \times Q \times R \rfloor.t = \top$.

- $\mathbf{f}.R.(t \downarrow \mathbf{a}.R) = \square$: Then, by the definitions of product and of rounding, $\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \square$. Furthermore, $\mathbf{f}.\lfloor P \times Q \times R \rfloor.t = \nabla$. However, by Equation C.1, we have

$$(\forall a:\ a \in \mathbf{o}.(P \times Q) : \mathbf{f}.(P \times Q \times R).ta = \top)$$

Furthermore, because $\mathbf{f}.R.(t \downarrow \mathbf{a}.R) = \Box$ and because R is a snippet, it follows that

$$(\forall a:\ a \in \mathbf{o}.R : \mathbf{f}.(P \times Q \times R).ta = \top)$$

Therefore, t is ∇-unhealthy in $P \times Q \times R$ and rounding down takes place, so $\mathbf{f}.\lfloor P \times Q \times R \rfloor.t = \Box$.

- $\mathbf{f}.R.(t \downarrow \mathbf{a}.R) = \nabla$: Here we consider two cases. In the first case,

$$(\forall a:\ a \in \mathbf{o}.R\ \wedge\ a \notin \mathbf{o}.(P \times Q) : \mathbf{f}.R.ta = \top)$$

In this case, t is ∇-unhealthy in $P \times Q \times R$ and rounding down takes place, so $\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \mathbf{f}.\lfloor P \times Q \times R \rfloor.t = \Box$. In the second case,

$$(\exists a:\ a \in \mathbf{o}.R\ \wedge\ a \notin \mathbf{o}.(P \times Q) : \mathbf{f}.R.ta \neq \top)$$

Here, no rounding down takes place, thus $\mathbf{f}.\lfloor \lfloor P \times Q \rfloor \times R \rfloor.t = \mathbf{f}.\lfloor P \times Q \times R \rfloor.t = \nabla$

$\Box$

Lemma 5.5.7

For snippets P, Q, and R, we have

$$P \sqsubseteq_o Q\ \Rightarrow\ P \times R \sqsubseteq_o Q \times R$$

Proof: Recall

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

Our proof is by contradiction. We assume $P \sqsubseteq_o Q$ and we show that $P \times R \not\sqsubseteq_o Q \times R$ leads to a contradiction to the assumption.

By monotonicity of product, we know that $P \times R \sqsubseteq Q \times R$. Consequently, if $P \times R \not\sqsubseteq_o Q \times R$, then $Q \times R$ is not output-persistent with respect to $P \times R$. That is, there must exist a trace $t \in \mathbf{l}.(P \times R)$ and symbol $a \in \mathbf{o}.(P \times R)$, such that $\mathbf{f}.(P \times R).ta \neq \top$ and $\mathbf{f}.(Q \times R).ta = \top$. From Equation 4.3 and the product table for labels we conclude that $\mathbf{f}.P.(ta \downarrow \mathbf{a}.P) \neq \top$ and $\mathbf{f}.R.(ta \downarrow \mathbf{a}.R) \neq \top$. On the other hand, $\mathbf{f}.(Q \times R).ta = \top$. Because $\mathbf{f}.R.(ta \downarrow \mathbf{a}.R) \neq \top$, it must be the case that $\mathbf{f}.Q.(ta \downarrow \mathbf{a}.Q) = \top$.

If $a \in \mathbf{o}.P$, then we have a contradiction to our assumption that $P \sqsubseteq_o Q$: By output-persistence, $\mathbf{f}.P.(ta \downarrow \mathbf{a}.P) \neq \top$ implies $\mathbf{f}.Q.(ta \downarrow \mathbf{a}.Q) \neq \top$.

If $a \notin \mathbf{o}.P$ and $\mathbf{f}.Q.(ta \downarrow \mathbf{a}.Q) = \top$, then we have two cases to address:

- If $\mathbf{f}.Q.\varepsilon = \top$, then $Q = \mathsf{MIRACLE}$. This contradicts the assumption that process Q is a snippet.

- $\mathbf{f}.Q.\varepsilon \neq \top$. By $\top$-persistence we know that there exists the shortest non-empty prefix s of trace $ta \downarrow \mathbf{a}.Q$, such that $\mathbf{f}.Q.s = \top$. Trace s can be written as $s = t'x$, where x is a symbol and t' is a trace. Because s is the shortest prefix of $ta \downarrow \mathbf{a}.Q$ such that $\mathbf{f}.Q.s = \top$, we know that $\mathbf{f}.Q.t' \neq \top$. Because Q is a snippet, we refer to condition 4 of Definition 5.4.1 and conclude that $x \in \mathbf{o}.Q$. $P \sqsubseteq_o Q$ implies that $\mathbf{o}.P = \mathbf{o}.Q$, thus $x \in \mathbf{o}.P$. Furthermore, because $\mathbf{f}.P.(ta \downarrow \mathbf{a}.P) \neq \top$, we know that $\mathbf{f}.P.t'x \neq \top$. Consequently, we have a contradiction to $P \sqsubseteq_o Q$: We showed above that $\mathbf{f}.Q.t'x = \top$, but, by output-persistence, $\mathbf{f}.P.t'x \neq \top$ implies that $\mathbf{f}.Q.t'x \neq \top$.

$\square$

Lemma 5.5.8

For processes P, Q, and R,

$$P \sqsubseteq_o Q \;\Rightarrow\; \lfloor P \rfloor \sqsubseteq_o \lfloor Q \rfloor \tag{C.2}$$

Proof: We prove that, after trace t has been rounded down in process Q, its label is still at least as large as the label of the same trace in process P:

$$\mathbf{u}_{\nabla}.Q \subseteq \mathbf{u}_{\nabla}.P \cup \left\{t : \mathbf{f}.P.t \in \{\bot, \Box\} : t\right\} \tag{C.3}$$

Take trace $t \in \mathbf{u}_{\nabla}.Q$. Then, $\mathbf{f}.Q.t = \nabla$ and $(\forall a : a \in \mathbf{o}.Q : \mathbf{f}.Q.ta = \top)$.

If $\mathbf{f}.P.t = \bot$ or $\mathbf{f}.P.t = \Box$, then Equation C.3 holds.

If $f.P.t = \nabla$, then either $t \in \mathbf{u}_{\nabla}.P$ or $t \notin \mathbf{u}_{\nabla}.P$. If $t \in \mathbf{u}_{\nabla}.P$, then Equation C.3 holds. On the other hand, if $t \notin \mathbf{u}_{\nabla}.P$, then $(\exists a : a \in \mathbf{o}.P : \mathbf{f}.P.ta \neq \top)$. Because $P \sqsubseteq_o Q$ and $\mathbf{o}.P = \mathbf{o}.Q$, we know that $\mathbf{f}.Q.ta \neq \top$. That is, trace t cannot be a ∇-unhealthy trace in Q, which contradicts our assumption that $t \in \mathbf{u}_{\nabla}.Q$.

Finally, $\mathbf{f}.P.t$ cannot be equal to $\top$, because $\mathbf{f}.P.t \sqsubseteq \mathbf{f}.Q.t$ and $\mathbf{f}.Q.t = \nabla$. $\square$

Bibliography

[Badeau et al., 1992] Badeau, R. W. et al. (1992). A 100-MHz Macropipelined VAX Microprocessor. *IEEE Journal on Solid-State Circuits*, 27(11):1585–1598.

[Benko, 1993] Benko, I. (1993). The Committee Problem and Delay-Insensitive Circuits. Master's thesis, University of Waterloo, Department of Computer Science.

[Benko and Ebergen, 1994] Benko, I. and Ebergen, J. (1994). 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.

[Berkel et al., 1991] Berkel, K. v., Kessels, J., Roncken, M., Saeijs, R., and Schalij, F. (1991). The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389.

[Black, 1986] Black, D. L. (1986). On the existence of delay-insensitive arbiters: Trace theory and its limitations. *Distributed Computing*, 1:205–225.

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

[Brunvand, 1995] Brunvand, E. (1995). Low latency self-timed flow-through FIFOs. In Dally, W. J., Poulton, J. W., and Ishii, A. T., editors, *Advanced Research in VLSI*, pages 76–90. IEEE Computer Society Press.

[Brzozowski and Ebergen, 1992] Brzozowski, J. A. and Ebergen, J. C. (1992). On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360.

[Brzozowski and Seger, 1995] Brzozowski, J. A. and Seger, C.-J. H. (1995). *Asynchronous Circuits*. Springer-Verlag.

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

[Chaney and Molnar, 1973] Chaney, T. J. and Molnar, C. E. (1973). Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22:421–422.

[Clark and Molnar, 1974] Clark, W. A. and Molnar, C. E. (1974). Macromodular computer systems. In Stacey, R. W. and Waxman, B. D., editors, *Computers in Biomedical Research*, pages 45–85. Academic Press.

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

[Davis and Nowick, 1995] Davis, A. and Nowick, S. M. (1995). Asynchronous circuit design: Motivation, background, and methods. In Birtwistle, G. and Davis, A., editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag.

[Day and Woods, 1995] Day, P. and Woods, J. V. (1995). Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272.

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

[Dill, 1989] Dill, D. L. (1989). *Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press.

[Ebergen, 1989] Ebergen, J. C. (1989). *Translating Programs into Delay-Insensitive Circuits*. CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam.

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

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

[Ebergen and Berks, 1995] Ebergen, J. C. and Berks, R. (1995). VERDECT: A verifier for asynchronous circuits. *TCCA Newsletter*.

[Ebergen et al., 1993] Ebergen, J. C., Bertrand, P. F., and Gingras, S. (1993). Solving a mutual exclusion problem with the RGD arbiter. In Furber, S. and Edwards, M., editors, *IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies*. Elsevier, Amsterdam.

[Ebergen et al., 1995] Ebergen, J. C., Segers, J., and Benko, I. (1995). Parallel program and asynchronous circuit design. In Birtwistle, G. and Davis, A., editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag.

[Furber and Day, 1996] Furber, S. B. and Day, P. (1996). Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253.

[Furber et al., 1996] Furber, S. B., Day, P., Garside, J. D., Paver, N. C., and Temple, S. (1996). AMULET2e. In Muller-Schloer, C., Geerinckx, F., Stanford-Smith, B., and van Riet, R., editors, *Embedded Microprocessor Systems.* Proceedings of EMSYS'96 - OMI Sixth Annual Conference.

[Furber et al., 1993] Furber, S. B., Day, P., Garside, J. D., Paver, N. C., and Woods, J. V. (1993). A micropipelined ARM. In Yanagawa, T. and Ivey, P. A., editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10.

[Furber et al., 1998] Furber, S. B., Garside, J. D., and Gilbert, D. A. (1998). AMULET3: A high-performance self-timed ARM microprocessor. In *Proc. International Conf. Computer Design (ICCD).*

[Gageldonk et al., 1998] Gageldonk, H. v., Baumann, D., van Berkel, K., Gloor, D., Peeters, A., and Stegmann, G. (1998). An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall.

[Hopcroft and Ullman, 1979] Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

[Huffman, 1964] Huffman, D. A. (1964). The synthesis of sequential switching circuits. In Moore, E. F., editor, *Sequential Machines: Selected Papers.* Addison-Wesley.

[Jong and Lin, 1994] Jong, G. d. and Lin, B. (1994). A communicating petri net model for the design of concurrent asynchronous modules. In *Proc. ACM/IEEE Design Automation Conference*, pages 49–55.

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

[Keller, 1974] Keller, R. M. (1974). Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33.

[Kessels and Marston, 1997] Kessels, J. and Marston, P. (1997). Designing asynchronous standby circuits for a low-power pager. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 268–278. IEEE Computer Society Press.

[Liu et al., 1993] Liu, Y., , Aldwinckle, J., Birtwistle, G., and Stevens, K. (1993). Ttesting the Consequences of Specifications in the Modal μ-Calculus. In *Canadian Conference On Electrical and Computer Engineering*.

[Mallon, 1997] Mallon, W. (1997). Personal communication.

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

[Martin, 1985] Martin, A. J. (1985). The design of a self-timed circuit for distributed mutual exclusion. In Fuchs, H., editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press.

[Martin, 1990] Martin, A. J. (1990). Programming in VLSI: From communicating processes to delay-insensitive circuits. In Hoare, C. A. R., editor, *Developments in Con-*

currency and Communication, UT Year of Programming Series, pages 1–64. Addison-Wesley.

[Martin, 1993] Martin, A. J. (1993). Synthesis of asynchronous VLSI circuits. In Birtwistle, G., editor, *Proceedings of the VII Banff Workshop: Asynchronous Hardware Desing.*

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency.* Prentice-Hall.

[Molnar, 1993] Molnar, C. (1993). Personal communication.

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

[Molnar et al., 1992b] Molnar, C., Sutherland, I., and Jones, I. (1992b). A Petri-net weave. Sun Microsystems Laboratories Memo 92:0357.

[Muller and Bartky, 1959] Muller, D. E. and Bartky, W. S. (1959). A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press.

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

[Negulescu and Brzozowski, 1995] Negulescu, R. and Brzozowski, J. A. (1995). Relative liveness: From intuition to automated verification. In *Asynchronous Design Methodologies*, pages 108–117. IEEE Computer Society Press.

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

[Nicola and Hennessy, 1983] Nicola, R. D. and Hennessy, M. (1983). Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133.

[Peeters, 1990] Peeters, A. (1990). Decomposition of delay-insensitive circuits. Computing Science Notes 90/04, Dept. of Math. and C.S., Eindhoven Univ. of Technology.

[Pena and Cortadella, 1996] Pena, M. and Cortadella, J. (1996). Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press.

[Rem et al., 1983] Rem, M., van de Snepscheut, J. L. A., and Udding, J. T. (1983). Trace theory and the definition of hierarchical components. In Bryant, R., editor, *Proceedings of Third Caltech Conference on VLSI*, pages 225–239. Computer Science Press.

[Seitz, 1980] Seitz, C. L. (1980). Ideas about arbiters. *Lambda*, First quarter(1):10–14.

[Sproull et al., 1994] Sproull, R. F., Sutherland, I. E., and Molnar, C. E. (1994). The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59.

[Stevens et al., 1993] Stevens, K., Aldwinckle, J., Birtwistle, G., and Liu, Y. (1993). Designing Parallel Specifications in CCS. In *Canadian Conference On Electrical and Computer Engineering*.

[Stevens, 1994] Stevens, K. S. (1994). *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, Dept. of Computer Science, University of Calgary, Canada.

[Sutherland, 1989] Sutherland, I. E. (1989). Micropipelines. *Communications of the ACM*, 32(6):720–738.

[Turing, 1947] Turing, A. (1947). Lecture to the London Mathematical Society on 20 February 1947. In *Charles Babbage Institute Reprint Series for the History of Computing, Vol. 10, 1986*. MIT Press.

[Udding, 1984] Udding, J. T. (1984). *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology.

[van Berkel and Saeijs, 1988] van Berkel, C. H. K. and Saeijs, R. W. J. J. (1988). Compilation of Commnicating Processes into Delay-Insensitive Circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 157–162, Rye Brook, New York. Computer Science Press.

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

[van Berkel et al., 1994] van Berkel, K., Burgess, R., Kessels, J., Peeters, A., Roncken, M., and Schalij, F. (1994). A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439.

[van Berkel and Rem, 1995] van Berkel, K. and Rem, M. (1995). VLSI programming of asynchronous circuits for low power. In Birtwistle, G. and Davis, A., editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag.

[van de Snepscheut, 1985] van de Snepscheut, J. L. A. (1985). *Trace Theory and VLSI Design*. Springer-Verlag, Heidelberg.

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

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

[Verhoeff, 1998b] Verhoeff, T. (1998b). Encyclopedia of Delay-Insensitive Systems (EDIS). `http://www.win.tue.nl/cs/pa/edis/edis.html`.

[Weste and Eshraghian, 1993] Weste, N. and Eshraghian, K. (1993). *Principles of CMOS VLSI Design*. Addison-Wesley, second edition edition.

[Williams, 1994] Williams, T. E. (1994). Performance of iterative computation in self-timed rings. *Journal of VLSI Signal Processing*, 7(1/2):17–31.