Process Spaces and Formal Verification of Asynchronous Circuits

by

Radu Negulescu

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, 1998

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

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

We are not seeking simplicity in art,

but we usually arrive at it as we approach the true nature of things.

Constantin Brancusi, 1876–1957

Abstract

This thesis proposes process spaces, a simple and unified treatment for concurrency issues such as parallel composition, refinement, deadlock, livelock, and starvation. Processes are modeled as contracts over which executions may occur. The main innovation is that executions are abstract; this leads to a very general model. For trace-based executions, process spaces relate closely to trace theory and CSP, except that we do not attach alphabets or connectivity restrictions to processes.

We revise several algebraic properties of process compositions and comparisons that are commonly known from concurrency theory. A novel transform reveals symmetries among usual process operations.

For finite-trace processes, we have a tool that uses a public-domain BDD library. This tool fully supports modular and hierarchical verification, and draws on the flexibility of the underlying formalism to address several niche applications. One application is to detect switch-level faults in asynchronous MOS circuits. Another is to verify sufficiency of relative delay constraints by handling them as metric-free processes, without requiring post-layout numerical information on delays.

By comparing processes for liveness and progress, we obtain a classification of concurrency faults. We also determine links among implicit liveness, progress, and safety properties, so one can specify just the simpler safety properties, and then assume typical liveness and progress properties. We briefly discuss some promising applications that involve analog trajectories and true concurrency.

To handle relabelings, hidings, and derivatives, we construct them as process maps arising from relations on execution sets. We obtain several algebraic properties, including criteria for performing verifications on images of processes by over-approximation, under-approximation, and independence with respect to such maps.

Acknowledgements

My thesis is a product of interaction with the Maveric group at the University of Waterloo, and I wish to thank everyone involved: Janusz A. Brzozowski, my supervisor, has invested valuable ideas and a considerable amount of effort in directing and reviewing my work; Jo C. Ebergen has introduced me to a modern view of asynchronous circuits through a very interesting graduate course and has made key suggestions as well; Robert T. Berks, Robert D. Black, Chuan-Jin (Richard) Shi, Piotr R. Sidorowicz, Signe J. Silver, and the other student members of the Maveric group have contributed constructive feedback for my presentations and writings, and a friendly and supportive environment; and, inspiration has also come from discussions with Charles E. Molnar, Tom Verhoeff, and other visitors to our group.

My external examiner, David L. Dill, and the other members of my thesis committee, Joanne M. Atlee, Janusz A. Brzozowski, Jo C. Ebergen, and Bruno R. Preiss, have provided insightful suggestions and relevant bibliography; I thank them for their interest and careful reading of the manuscript.

I wish to express my gratitude to the late Charles E. Molnar for offering me a summer internship at Sun Microsystems Laboratories, which allowed me to synchronize my theory and tool with the asynchronous practice. This internship has been one of the bright spots of my PhD program, and I would like to thank the entire asynchronous team at Sun Microsystems Laboratories for their pleasant cooperation. I also thank Ad M.G. Peeters from Philips Research Laboratories and André Bouwer, John F. Chappel, and Craig Farnsworth from Cogency Technology for their help in applying my tool to their circuits. Ad Peeters has made a direct contribution to the applications reported in Chapter 4, by co-authoring a paper. These collaborations have been very beneficial for my work.

The Model Checking Group at Carnegie Mellon University is gratefully acknowledged for making available a BDD library that I have used in the tool described in Chapter 3. Also, I thank Dan Calistrate from the University of Calgary for quick briefings regarding mathematical background.

Special thanks go to Maria, my wife, who has done so much to help me, comfort me, and improve my morale throughout the ups and downs of graduate studies. My parents and my sister have always been cheering from afar, never failing in their support, and they have encouraged me greatly. Also, I am obliged to all my friends for many a social occasion and other gratifying activities.

Finally, I would like to acknowledge the financial support for my PhD studies, provided by grant OGP0000871 from the Natural Sciences and Engineering Research Council of Canada (NSERC), by an Ontario Graduate Scholarship, by a grant and a scholarship from the Information Technology Research Centre of Ontario, by teaching assistantships from the Department of Computer Science at the University of Waterloo, and by Sun Microsystems Laboratories directly through an internship salary and indirectly through a grant to our group that has prompted matching funds from NSERC.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Discrete-state systems are practically everywhere; some examples are digital circuits (synchronous and asynchronous), programs and data structures, communication protocols, and work flows in a factory. Discrete-state systems have finite or countable state spaces. Because such systems may consist of several interacting parts, they are sometimes referred to as concurrent systems; however, sequential programs are also important members of this class. Discrete-state systems are exposed to diverse modes of failure, including illegal inputs and outputs, deadlock, livelock, unfairness, and performance faults. Many models have been proposed to capture these diverse phenomena in research areas such as software engineering, digital circuits, and distributed systems.

Despite their diversity, discrete-state systems have several common aspects. The fact that their state spaces are finite or countable is essential for their analysis, but it is not the main unifying aspect. More importantly, there are common mechanisms by which such systems can be composed and compared, and these compositions and comparisons satisfy several common properties that are basic and useful.

For an example of common properties of discrete-state systems, consider the phrase 'implementation r satisfies specification q'; that is, r can replace q in a system, without introducing new dangers of incorrect operation in that system. Furthermore, the specification q may be an implementation for a higher-level specification p. (This viewpoint tacitly assumes that the specification and the implementation are represented by formal objects of the same nature, although implementation models typically contain more details than the higher-level specifications.) Then, a common-sense property is that, if q satisfies p and r satisfies q, then r satisfies p, and this can be applied for several levels of specifications. This property is known as successive refinement or hierarchical verification.

The discussion of hierarchical verification above is not tied to particular models or their levels of detail, and not even to the criteria for comparison. On the basis of this observation, we believe it is beneficial to keep such discussions simple and general, by postponing, as much as possible, references to modeling details or specific correctness concerns. We propose to study discrete-state systems by means of abstract executions, that is, without referring to the form or structure of the executions. Executions can be finite sequences, infinite sequences, time-stamped sequences, etc., but we consider them to be just elements of an arbitrary set.

Just as concepts from Newtonian mechanics are modeled by numbers, properties of discrete-state systems can be modeled by sets of abstract executions. Then, common-sense laws like hierarchical verification can be studied by elementary set calculus. It may be non-trivial to relate abstract executions to real systems, just as it is non-trivial to explain to a novice what force is; however, the modeling disciplines should become easy with practice. The eventual benefit of such an approach is that one could apply generic laws, like $F = ma$, for very diverse systems and at all levels of analysis where the general paradigm applies.

1.1 Objectives of the Thesis

The purpose of this thesis is to take steps towards a general theory of discrete-state systems along the lines suggested above, while making sure that sufficiently many practical applications exist to serve as proofs of concept. This often involves reworking some operations and properties from previous theories of concurrency (which will be discussed below), but some novel operations and properties are introduced as well. More importantly, we adopt an original viewpoint that regards executions as an abstract notion, and makes no references to states, variables, events, ports, or actions; this results in some distinguishing aspects of the proposed approach.

In terms of theory, two main steps are taken. One is a general formalism that models processes by means of abstract executions, that is, without references to structural details of the executions. We call this formalism *process spaces.* The other step is a study of maps between processes by means of relations between executions; we call the mappings in this study *process abstractions.*

We give examples from several fields, ranging from concurrent programs to electrical networks. For in-depth applications, we focus on formal verification techniques for asynchronous circuits. Asynchronous circuits are designed without the assumption of a global clock, and can be viewed as networks of digital components that operate in parallel and synchronize on voltage transitions of the interconnecting wires. For this reason, asynchronous circuits make good test-benches for any theories that deal with concurrency issues in general discrete-state systems. Formal verification aims to search exhaustively for flaws of certain types in a given system, which might be composed of several parts and compared to a specification. Thus, formal verification of asynchronous circuits offers a practical setting for experimenting with correctness concerns and operations on discrete-state systems.

After establishing the basis for verifying asynchronous circuits in process spaces, we develop two niche applications that are of practical interest, and we illustrate them on published designs. We needed a high-performance tool to allow us to study non-trivial examples. We have implemented the process space operations and conditions for execution sets that are regular languages of finite words. BDDs (binary decision diagrams) are data structures and algorithms for manipulating Boolean functions. Many verification problems are computationally intensive, and this applies for BDD-based algorithms too in the worst cases; however, in typical cases BDDs are known to increase performance by many orders of magnitude compared to traditional algorithms for these problems. Our tool represents large state spaces by characteristic functions of sets, and uses a public-domain BDD library for the routine operations on large Boolean functions.

Many recent asynchronous designs tend to rely on timing to achieve speed (see [MJCL97]) or energy and area savings (see [BHP95] and [Pee96]). In the VLSI design of asynchronous circuits, the numerical bounds on component delays are not known with sufficient precision until after setting the layout, transistor sizes, and wire lengths. On the other hand, the effort for optimizing the layout is substantial, and thus it is important to verify a circuit, as much as possible, before layout. We model relative delay constraints of a certain form as metric-free processes that simply ensure orderings of events; this way, the verification does not refer to numerical bounds on delays, can be performed before layout, and all the generic laws deduced for processes fully apply.

Another problem we address is the detection of switch-level faults in MOS transistor networks. At the switch level, the behavior of MOS transistor networks is approximated by on/off switches. Depending on the design style and local optimizations, switch-level faults include floating states, collisions, and shorts. In floating states, certain circuit nodes are disconnected from both ground and power supply; in collisions and shorts, certain circuit nodes are connected to both ground and power supply, with the additional problem that the connections are of roughly equal strengths in the case of collisions. We propose a verification technique that detects such switch-level faults in asynchronous circuits.

Given the generality of process spaces, one may wonder what non-trivial properties of discrete-state systems can be captured. It turns out that many important parts of the theory of concurrency can be adapted to abstract executions. Also, we have found some new properties. Certain things are lost or gained, however, in this generalization. Properties like hierarchical verification were usually given in previous formalisms under various restrictions pertaining to connectivity: no two outputs can be connected, only processes with the same inputs and outputs can be compared, etc. Since we do not refer to structural detail of executions, we do not impose similar restrictions. This generalization is done not by extending existing models with new features, but rather by removing existing features. Using case studies, we confirm that the unrestricted properties do apply and are sufficient at least for our formal verification applications. In addition, we have indications that the generalized properties apply to other systems and types of analysis as well.

Our investigation was started as a study of correctness concerns for asynchronous circuits. Finitary descriptions are regarded as insufficiently powerful for various correctness concerns, like liveness, which are treated by means of infinite words (see for instance [AS85] and [Dil89] for two prominent treatments of liveness); [Bla86] even gives an argument that general liveness properties cannot be captured by finite traces. On the other hand, many of the common descriptions of circuits available in practice are equivalent in power to such finitary descriptions.

Accordingly, we had undertaken to define and study whatever correctness concerns we can in terms of finitary descriptions only. Although we do not challenge the argument of [Bla86], we have noticed that default liveness properties can be uniquely associated to such descriptions in a way that does correspond to our expectations for liveness properties in many asynchronous circuits. The same holds for a slightly different class of properties, known as progress properties.

We also study the algebraic properties of liveness and progress representations, and we propose a classification of liveness and progress faults that matches our intuitive notions of deadlock, livelock, and starvation; we call them collectively *lock faults*. For these purposes, we found it more natural to refer to infinite traces rather than to finitary descriptions. Nevertheless, users can still apply finitary descriptions for ease of specification, extract the default liveness and progress properties mechanically from such descriptions, and then apply the generic properties of processes or the classification of lock faults on the resulting processes, also mechanically. This way, users can have the best of both worlds. Finally, there are strong indications that our treatment of liveness and progress applies to other types of discrete-state systems as well; we point that out with small examples.

We continue the general theory of process spaces by a study of maps that preserve binary operations; these maps (called process abstractions) generalize other process operations, and allow for reduced verifications of processes. Some important operations in concurrency theory take as arguments not just processes, but also sets or sequences of actions; some examples are projections and hidings, where certain actions are deleted from executions, and derivatives, where an initial part of an execution is deleted. Is it possible to study all these operations without referring to the structure of executions? It turns out that projections, hidings, derivatives, and several other operations are just particular cases of process abstractions; as such, we could indeed assimilate these operations within the abstract execution framework. At the same time, general process abstractions capture many properties of practical interest. We give particular emphasis to criteria that authorize us to perform verifications on images of processes through a process abstraction rather than directly on the original processes. These criteria and other generic algebraic properties we study for process abstractions are inherited by projections, relabelings, derivatives, and other operations, as particular cases of process abstractions.

1.2 Previous Work

The literature on process modeling is vast, and unfortunately we cannot do justice to all of it within the scope of this thesis. We therefore restrict our survey to some of the better-known approaches from the asynchronous community and elsewhere.

1.2.1 Models of Concurrency

Process spaces are closely related to several previous treatments of concurrency that model discrete-state systems essentially by their finite or infinite sequences of events; let us refer to models of this type as *language-oriented* models. Some of these related models are: the prefix-closed and complete trace theories in [Dil89], the trace theory used in [Ebe91a], the trace, failure, and divergence semantics in [Hoa85], the receptive process theory in [Jos92], the CFFD (chaos-free failures divergences) model in [VK98, VT95], and the theory of delay-insensitive systems in [Ver94b].

Our model for a process consists of two sets of abstract executions, representing an assumption/guarantee specification: one set represents guarantees about the behavior of the device, and the other set represents assumptions about the behavior of the environment. Assumption/guarantee specifications are a natural idea, which can be traced as far back as [Pnu77] and even Hoare's triples [Hoa69]. In CSP (communicating sequential processes) [Hoa85], non-deterministic processes are formalized essentially by two sets of behaviors (failures and divergences). In [Dil89], the two sets contain behaviors of the same type (sequences of symbols), and the idea is applied at two different levels of detail, using finite and infinite sequences. The processes of the extended DI (extended delay-insensitive) model in [Ver94b] also introduce assumptions and guarantees regarding two correctness issues (safety and deadlock), resulting in specifications with several sets of finite sequences of symbols. A separate development regarding assumption/guarantee specifications has been proposed in [AL95], using temporal logic [Lam94].

Although several theories of concurrency cited above use more than one type of sequence of events to describe the behaviors of their systems, they have made no proposals to use abstract executions. This is an important difference between process spaces and the theories cited above. Even for those process spaces that use sequences of events as executions, there are further differences regarding unusual connectivity cases. We discuss the impact of these differences on our niche applications, where we exploit precisely these connectivity cases that are ruled out by other models. We do not aim to prove that any of these models, including ours, is right or wrong; we only argue that our model is useful, because it fits naturally at least some applications that might contort other models, and because our model captures credible algebraic properties for these applications.

Although there are some differences on connectivity restrictions and other technicalities, we believe there is a basic agreement between particular instances of process spaces and the theories cited above, and among the theories above. In particular, several of the generic properties of processes, such as hierarchical verification, have been known in many of these previous theories; our contribution, nevertheless, is to generalize them for arbitrary types of executions, without major modifications. In addition, we have found several new properties, mostly referring to maps between processes and to symmetries of process spaces.

Several of the theories above have a reflection or mirroring operation that induces a duality between devices and their environments; [Ver94b] even employs a symmetric notation. However, we are the first to complete this duality and to state it formally. Also, we propose a rotation operation that induces a ternary symmetry between common process operations. J. A. Brzozowski was the first to notice that process spaces are particular cases of ternary algebras [BLN96]. However, the rotation operation, the principle of ternary symmetry, and the resulting links between common process operations are our contribution and do not exist in general ternary algebras as presented for instance in [BS95] and [BLN96].

Since the early days of concurrency theory, there have been extensive efforts to establish an abstract view of discrete-state systems by axiomatic approaches. The resulting models, to which we refer as *process algebras*, define processes by the laws they satisfy. CSP, for instance, combines a process algebra viewpoint with the language-oriented viewpoint. Other algebraic approaches include, but are not limited to, the algebraic theory of processes of [Hen88], ACP (algebra of communicating processes) of [BK85], the laws of CCS (calculus of communicating systems) [Mil89], Circal [Mil94], and, for asynchronous circuits, the algebraic model of [BC88] and the algebra for delay-insensitive circuits of [JU90]. Term algebras have been used to describe general concurrent systems in [Hen88] and VLSI circuits in [Car82], subsuming both the behavior and the connectivity of such systems.

Process algebras have captured many of the generic properties of processes, like hierarchical verification, but those process algebras known to us also include laws that refer to structural details of executions. Some examples are laws for occurrence of individual events, laws for sequential composition (which assume that executions can be concatenated), and references to alphabets of actions. In our opinion, such references to structure have prevented the development of a general theory that would be truly independent of the level of analysis of a discrete-state system.

Other main approaches to the verifications of discrete-state systems are based on Petri nets, temporal and modal logics, and various types of automata ("operational semantics"). These formalisms provide data structures and algorithms for handling languages of finite or infinite words, but they are not concerned with reasoning in terms of abstract (structure-less) executions, relying instead on occurrences of events. Still, a discussion of modeling paradigms is in order.

It has been noted that the CCS notion of correctness relative to a specification, called "observation equivalence", makes finer distinctions than some language-oriented models; see for instance [Mil90]. On the other hand, a notion of relative correctness based on equivalence between specification and implementation is not very meaningful for our verification problems, because typically an implementation corresponds to just one of many behavior cases permitted by a specification; what we need is a directed (one-way) notion of relative correctness which allows an implementation to fill in some of the "don't care" parts of the specification. Consider for instance a circuit with two data outputs a and b specified to switch from low to high concurrently. Let $a\uparrow$ and $b\uparrow$ be rising transitions on the voltage signals a and b, and consider that the specification permits these transitions to occur in either order; that is, both $a\uparrow b\uparrow$ and $b\uparrow a\uparrow$ are legal behaviors. However, due to relative timing, the implementation might always issue the $a\uparrow$ transition first, that is, $b\uparrow a\uparrow$ might never happen. This is a common situation in many real-life systems, but the "observation equivalence" viewpoint would consider it incorrect, because the behavior $b\uparrow a\uparrow$ cannot be observed in the implementation. By contrast, unless the specification imposes some fairness constraints, we would consider that this circuit meets this specification. (Fairness constraints can also be specified in our model and in some of the language-oriented models above, simply by ruling out certain infinite executions; see Chapter 6.) This is not to say that CCS cannot be used in other manners to fit our problems or other problems, but just to explain our basic reason for adopting a language-oriented modeling paradigm in spite of the differences from CCS. We demonstrate in applications how our paradigm works. A critique of CCS is beyond the scope of this thesis.

The language-oriented models cited above represent situations where two events occur at the same time by considering that the two events may occur in both orders, although not precisely at the same time; this viewpoint is known as "interleaving semantics". A different paradigm, known as "true concurrency", is offered by models such as [Maz86] that distinguish between two events occurring at the same time and two events that may occur in either order. Although the usual interleaving semantics lose some information regarding causal dependences between events, they are often more tractable than the true concurrency models. For convenience, we adopt interleaving semantics in most of our treatment, but we briefly show in Chapter 7 how a true concurrency viewpoint can also be accommodated in process spaces.

1.2.2 Methods for Asynchronous Circuit Verification

Our technique for including relative delay constraints in a metric-free analysis of asynchronous circuits appears to be novel. Previous timing analysis methods for asynchronous circuits from [ACD+92], [AD94], [CDYC97], [LMBSV92], [MD92], [Mye95], or [RM94], allow one to perform a more detailed analysis than we do here, but they require numerical bounds on individual delays, whereas in our case studies such bounds were not even available. The timing-reliability problems in [KN94] are related to the problem we are studying, but [KN94] focuses on quantitative analysis for known delay constraints, rather than on finding and verifying the constraints. In [BY75] and [BY76], almost-equal-delay models were proposed to analyze a circuit in a metric-free way under the assumption that the gate delays are roughly equal; however, [BY75] and [BY76] do not address our problems because their delay constraints are fixed (always a triangle inequality).

Switch-level analysis methods have been proposed in [BCDM86], [Bry87], [BS95], [KS95], [KSL95], and [ZH92], among others. However, except for [BS95], these methods have focused only on steady-state or synchronous behavior. The technique we propose here aims to complement the methods above and to provide additional flexibility. Our analysis is event-driven, in the sense that it focuses on signal transitions rather than signal levels and does not assume (nor forbid) synchrony or stabilization of voltage signals.

1.2.3 Liveness and Progress

The complete trace structures of [Dil89] can be used to capture both liveness and progress properties, and the differences between our general liveness and progress processes and [Dil89] have to do mainly with connectivity restrictions and other technicalities. The models of [Ver94b] also capture some progress and liveness faults, in the form of global deadlock, where everything may stop in a system while some events are demanded. We use related finalization processes that differ from [Ver94b] mainly on connectivity and other technicalities.

On the other hand, our idea of liveness properties implicitly attached to finitary specifications appears to be new. Without challenging the argument of [Bla86] that finitary specifications are insufficiently powerful to capture arbitrary liveness properties, we point out that in many discrete-state systems the liveness properties are not arbitrary, but they are correlated to safety properties, and we demonstrate this on the main example from [Bla86]. In [AS85], an exhaustive characterization of liveness properties has been proposed. However, nothing is said about which of the properties in the class defined by [AS85] should be used for a liveness condition. The models of [LT87] provide more insight in this respect, but their condition for liveness fails to detect certain cases of unfairness [Neg95, p. 30]. The condition for "completeness with respect to specification" in [Ebe91a] prescribes essentially that every specified behavior must have a counterpart in the implementation. This condition does not seem very meaningful for our problems, for reasons similar to those mentioned in regards to process equivalences; besides, many of the liveness and progress faults we study are outside the scope of [Ebe91a].

Our models of default progress processes that capture progress properties implicitly attached to finitary specifications appear to be related to a "finitary fairness" type of assumption recommended in [AH94] and [VK98]. However, [AH94] does not investigate the algebraic properties of finitary fairness specifications, focusing instead on modeling issues and case studies. A notion of 'an implementation satisfies a specification' is given in [VT95], and it is shown in [VK98] that it can capture finitary fairness, but there connectivity restrictions play an important role.

Our general liveness and progress processes induce a formal classification of liveness and progress faults into deadlock, livelock, and starvation. This classification appears to capture common intuition as expressed for instance in certain examples from [Dil89] and [Ver94b]. Although diverse formalizations have been proposed for deadlock or other lock faults, (see for instance [BS95], [Hoa85], [Sme90], and [Ver94b]), we are not aware of a formal classification related to ours.

1.2.4 Process Abstractions

Most of the theories of concurrency mentioned above incorporate some notion of hiding internal events of a system from an externally visible interface. Also, most of these previous theories have some notion of derivative of a process, representing the behavior of a system after certain events have occurred. However, structural details, such as occurrences of actions, have been given key roles in studying such maps between processes. By contrast, we unify these concepts and investigate them on the basis of abstract executions. Besides, in process spaces we impose no restrictions on actions, and we do not even distinguish between input, output, internal, and external actions; instead, we rely on different effects of such actions on the legality of executions.

Reduced representation of systems is also a natural idea, which has received a lot of attention in previous treatments of concurrency. Using reduced representations, one can verify circuits at lower levels of modeling [KM91], or of larger sizes [CGL92]. The criterion for exact approximation in [CGL92] is related to a particular case of a criterion for independence that we propose; these criteria determine cases where verifications on images of processes through a map are equivalent to direct verifications on the original processes. Also, abstractions have been studied by means of Galois connections in [LGS+95] and [Sif83], and we also give Galois connection properties that correlate orders in different process domains. However, none of the methods above has been aimed towards abstract executions; in fact, [CGL92], [LGS+95], and [Sif83] take an operational viewpoint based on transition systems. Moreover, [KM91] does not allow for uncertainty of representations because their maps attach exactly one image to each execution or trajectory of a system. Methods that approximate state sets have been used, for instance, in [CDYC97], [DWT95], and [SYP+97], but they are also based on an operational approach.

1.3 Notes on Terminology

From our introduction of process spaces in [Neg95], we have changed some inappropriate terms. The 'goal' executions used to be called 'contract' executions, but that label could be confused with the idea of a contract between environment and device, to which we refer frequently. More importantly, 'escape' executions were called 'error' executions. We have changed this term because of possible confusion with the idea of an error state of a computer program or a pocket calculator, that may be reached following illegal inputs; such error states should normally be related to our 'reject' executions rather than to 'escape' executions.

We use double quotes “” for citations, and single quotes ‘’ for some informal or undefined terms.

Also, we do not include within quotes punctuation from the surrounding text. We follow the advice of P. R. Halmos in [SHSJ73] that “the comma or period should come where the logic of the situation forces it to come”, although, as P. R. Halmos also mentions, “There appears to be an international typographical decree that a period or a comma immediately to the right of a quotation is “ugly”.”.

Chapter 2

Process Spaces

We introduce process spaces, a formalism that deals with the idea of a contract between a device and its environment in a general manner. Such contracts specify the device-environment interface in terms of allowed executions, but we build our formalism without reference to a particular type or interpretation for executions. Executions can be sequences of events, functions of time, etc., but we consider them to be just elements of an arbitrary set $\mathcal{E}$. In fact, we postpone the decision of what the executions are until after determining the algebraic structure of our theory. In this sense, our approach is based on abstract executions. This approach permits the users to choose the levels of complexity and precision of a particular application by varying the amount of detail contained in the executions.

We generalize several operations on processes and notions of correctness that form the core of concurrency theory, such as relative and absolute correctness, parallel composition, and non-deterministic choice. (Process abstractions are discussed in a later chapter, also at a general level, but as a separate development.) We also discuss several algebraic properties of these conditions and operations, as well as their use for handling processes in a structured manner.

The correctness conditions of process spaces are abstract patterns for particular correctness concerns, such as safety, liveness, progress, and deadlock-freedom. By this, we obtain a unified theory for all particular concerns that can be cast into the general patterns.

Because we make no assumptions about the structure of the elements of $\mathcal{E}$, the algebraic structure of process spaces boils down to elementary set calculus on subsets of $\mathcal{E}$; this facilitates the proofs of algebraic properties for our processes. In other words, not only do we capture several correctness concerns for the price of one, but also the price we pay is not very high.

The process operations naturally extend to infinite families of processes. Certain process properties can also be taken to the limit, in the sense that they are generalized for infinite families of processes.

The operations and conditions of process spaces are linked by six-way symmetries. We formalize an environment-device symmetry as a duality principle, and also we formulate a ternary symmetry principle based on a 'rotation' transform.

2.1 The Model

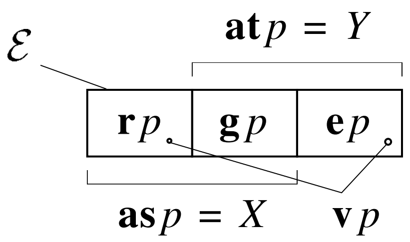
Let $\mathcal{E}$ be an arbitrary set, whose elements are called *executions.*

Definition 2.1 *A* process *over set $\mathcal{E}$ is a pair (X, Y) of subsets of $\mathcal{E}$ such that*

$$X \cup Y = \mathcal{E} \ . \tag{2.1}$$

The process space *of $\mathcal{E}$, denoted by $\mathcal{S}_{\mathcal{E}}$, is the set of all processes over $\mathcal{E}$.*

In the remainder of this chapter, we consider all processes to be defined over the same execution set $\mathcal{E}$; we simply say "a process" instead of "a process over $\mathcal{E}$". The set $\mathcal{E}$, however, may vary among applications.



Notation	Name	Set
$\mathbf{as}\,p$	accessible	X
$\mathbf{at}\,p$	acceptable	Y
$\mathbf{r}\,p$	reject	$\overline{Y}$
$\mathbf{g}\,p$	goal	$X \cap Y$
$\mathbf{e}\,p$	escape	$\overline{X}$
$\mathbf{v}\,p$	violation	$\overline{X} \cup \overline{Y}$

Figure 2.1: Execution sets of a process $p = (X, Y)$.

A process (X, Y) represents a contract between a device and its environment: the device guarantees that only executions from X may occur, whereas the environment guarantees that only executions from Y may occur. Executions from X are called *accessible* and those from Y are called *acceptable*—the device can 'access' and 'accept' them, respectively. Thus, a process can be regarded as an assumption/guarantee specification: assuming that only executions from Y may occur, it guarantees that only executions from X may occur.

Our model of processes is closely related to several previous theories of concurrency, mentioned in Section 1.2. The main distinguishing aspect is that our executions are abstract: there are no notions of inputs, outputs, actions, states, or variables involved in a process or its executions. This holds throughout the development of the process space theory in this chapter: many of the operations and properties are similar to usual notions from concurrency theory, and we generalize them for abstract executions. In particular, this generalization requires elimination of connectivity restrictions from process operations, correctness conditions, and their algebraic properties, by making no references to inputs, outputs, etc. Nevertheless, we show in examples how to use our processes to handle discrete-state systems with states and actions; other interpretations for processes are discussed in later chapters.

A process (X, Y) partitions $\mathcal{E}$ into three disjoint subsets, as in Figure 2.1. Executions from $\overline{Y}$ are called *rejects* and must be avoided by the environment; executions from $\overline{X}$ are called *escapes* and must be avoided by the device; executions from $X \cap Y$ are called *goals* and are legal for both device and environment. Executions from $\overline{X} \cup \overline{Y}$, called *violations*, are illegal either for the device or the environment. The accessible, acceptable, escape, reject, goal, and violation sets of process p are denoted by $\mathbf{as}\,p$, $\mathbf{at}\,p$, $\mathbf{e}\,p$, $\mathbf{r}\,p$, $\mathbf{g}\,p$, $\mathbf{v}\,p$, respectively.

Equation (2.1), written equivalently $\overline{X} \cap \overline{Y} = \emptyset$, formalizes a separation of responsibilities between device and environment: none of the executions needs to be avoided by both device and environment. For this reason, the fourth intersection $(\overline{X} \cap \overline{Y})$ is absent from the Venn diagram in Figure 2.1.

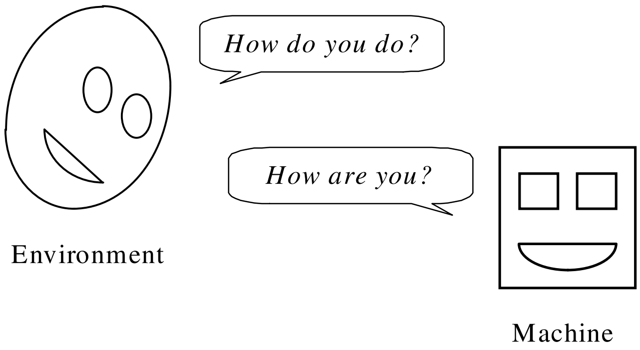


Figure 2.2: Etiquette machine.

Example 2.1 Imagine a simple etiquette machine. Proper etiquette is defined as a conversation containing any number of the phrases "How do you do" and "How are you". The conversation can be arbitrarily long, but finite. However, the available vocabulary includes another phrase, "XXX", that nobody wants to hear. If our machine hears "How do you do" or "How are you", it responds by "How do you do" or "How are you". If it hears "XXX", the machine might become out of order and may produce an arbitrary response or no response at all.

To model this machine in process spaces, we represent conversations as executions, consisting of strings of greetings between the machine and the environment. For instance, $\langle$Environment: How are you$\rangle$ $\langle$Machine: XXX$\rangle$ $\langle$Machine: How are you$\rangle$ is in $\mathcal{E}$, but $\langle$Environment: How do you do XXX$\rangle$ is not in $\mathcal{E}$ because "How do you do XXX" is not one of the available greetings. In this example we indicate in each greeting who delivered it, because it does matter who says "XXX", for instance.

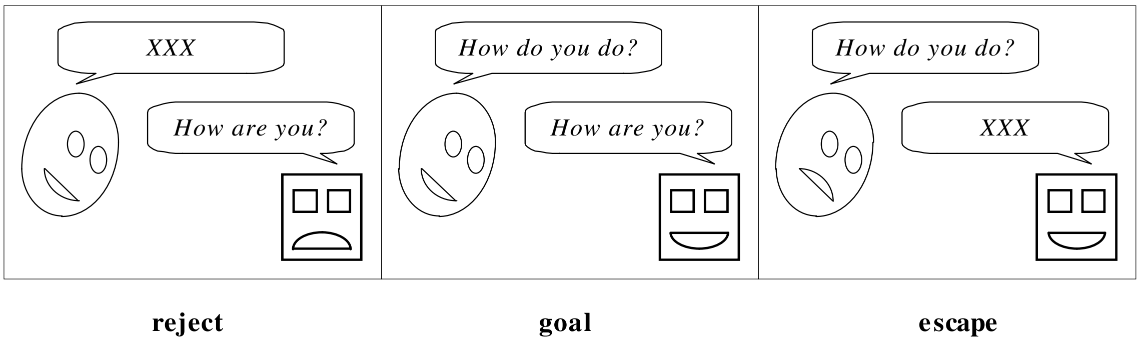
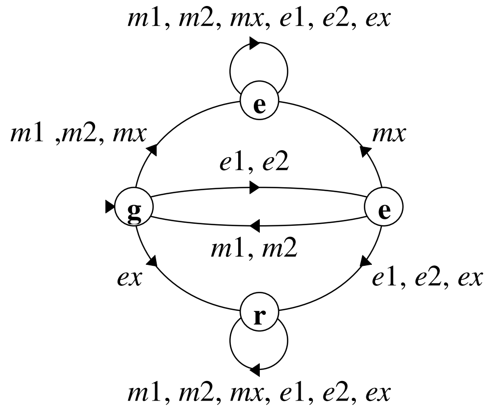


Figure 2.3: Watch your mouth!

Figure 2.3 gives examples of goals, rejects, and escapes for the etiquette machine. In a goal execution, everybody behaves and everybody is content. In a reject execution, the environment behaves improperly and violates the requirements of the machine, whereas in an escape execution, the machine behaves improperly and violates the requirements of the environment.

Greeting	Shorthand
Machine: How do you do	$m1$
Machine: How are you	$m2$
Machine: XXX	mx
Environment: How do you do	$e1$
Environment: How are you	$e2$
Environment: XXX	ex

(a)



(b)

Figure 2.4: Process of the etiquette machine: (a) actions; (b) process.

Figure 2.4 shows in full detail a process for the etiquette machine, as a state machine. The edges are labeled with greetings. To reduce clutter, we use shorthands for the greetings, and we sometimes put several greetings on a single edge. The shorthands are listed in Figure 2.4 (a). The state markings **r**, **g**, and **e** in Figure 2.4 (b) stand for reject, goal, and escape, respectively. The initial state is marked with an incoming arrowhead. A string of greetings is a reject, goal, or escape for this process if that string is spelled by a path starting at the initial state and ending at a state marked **r**, **g**, or **e**, respectively. Since every greeting is possible in every state, and thus every string in $\mathcal{E}$ is on a path starting at the initial state, the rejects, goals, and escapes cover all of $\mathcal{E}$. Any conversations are considered possible, because we don't know what the environment or the machine would do. However, we forbid the environment or the machine from doing certain things by qualifying those things as rejects or escapes.

The formal representation requires us to classify every string of greetings, including some which were more or less hand-waved in the informal descriptions above. This way we obtain a formal description of the interface between the etiquette machine and its environment. For instance, ⟨Environment: XXX⟩ ⟨Machine: XXX⟩ is a reject, because the environment violated the contract first; ⟨Machine: How do you do⟩ ⟨Environment: XXX⟩ ⟨Machine: How are you⟩ is an escape, because the machine is not allowed to speak out of turn; ⟨Environment: How do you do⟩ ⟨Environment: How do you do⟩ ⟨Machine: How are you⟩ is a reject, because the environment is not allowed to speak out of turn, either; the empty string is a goal, because neither the environment nor the machine have done anything wrong; and, the execution consisting of just ⟨Environment: How are you⟩ is an escape, because it would be impolite for the machine not to respond to the greeting. Notice that the machine must leave the escape state on the right by a polite greeting in order to reach a goal state.

Note that, in this example, not only do we forbid undesired greetings to occur, but also we force desired greetings to occur by forbidding stopping at certain states. Although these two types of interdictions are of different nature, they both amount to imposing that certain executions be avoided. More complex properties can be specified by the same mechanism if one uses infinite-word executions or other more detailed types of executions. □

2.2 Correctness Conditions

We use an *absolute* correctness condition, that a process has all by itself, and a *relative* correctness condition, by which an 'implementation' process is compared to a 'specification' process.

The property of absolute correctness in process spaces formalizes that a device operates correctly all by itself, that is, the device imposes no requirements upon the environment; in this sense the device is said to be robust. In terms of avoided executions, this property amounts to an empty reject set of the corresponding process, as in the definition below. A symmetric property (albeit not a desirable one) is that a device offers no guarantees to the environment as to which executions are avoided; in this sense the device is said to be chaotic. This amounts to an empty escape set.

Definition 2.2 *Process p is* robust *if*

$$\mathbf{r}\,p \;=\; \emptyset\ .$$

Process p is chaotic *if*

$$\mathbf{e}\,p \;=\; \emptyset\ .$$

The sets of robust and chaotic processes are denoted by $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{C}_{\mathcal{E}}$, respectively.

Example 2.2 The process in Figure 2.4 is not robust, because the executions beginning with ⟨Environment: XXX⟩ are rejects for that process. □

The only process that is both robust and chaotic has the accessible and acceptable sets equal to $\mathcal{E}$.

Definition 2.3 *The process* void*, denoted by Φ, is $(\mathcal{E}, \mathcal{E})$.*

Void has no escapes and no rejects; thus it neither offers guarantees nor imposes requirements upon the environment. Its behavior reminds one of vacuum.

The property of relative correctness in process spaces formalizes that a process q is a satisfactory substitute for a process p. For this, q should impose fewer requirements and offer more guarantees than p does. In terms of avoided executions, this means that q has a larger acceptable set and a smaller accessible set.

Definition 2.4 *Process p is* refined by *process q, written $p \sqsubseteq q$, if*

$$(\mathbf{at}\, p \subseteq \mathbf{at}\, q) \wedge (\mathbf{as}\, p \supseteq \mathbf{as}\, q) \ .$$

Why is it undesirable to have many accessible executions? If the set of accessible executions is smaller, the behavior of the device is more specified, in the sense that the device has less freedom and obeys tighter constraints.[1]

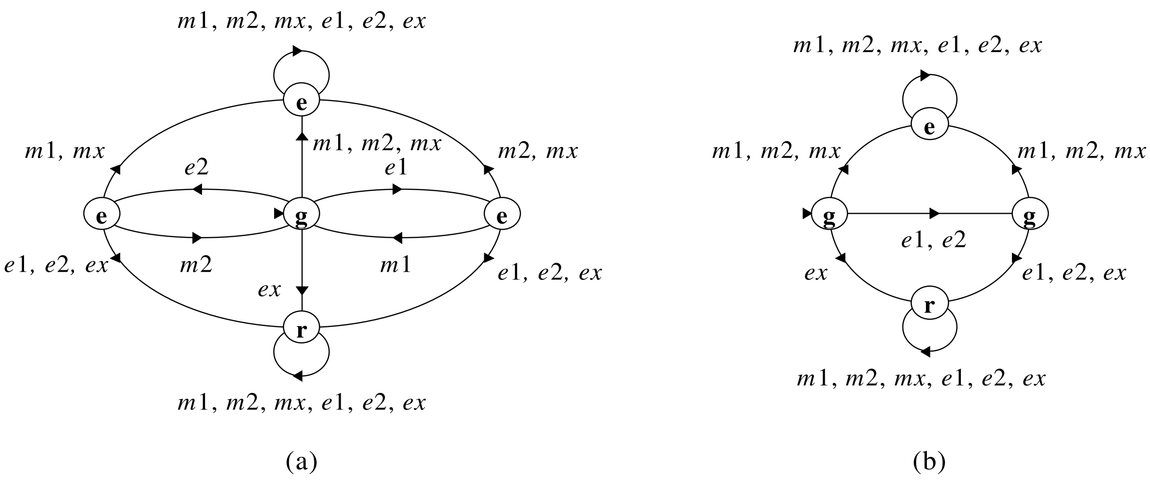


Figure 2.5: Examples for refinement: (a) repeat machine; (b) quiet machine.

Example 2.3 A 'repeat machine' behaves similarly to the etiquette machine, except that, after a polite greeting by the environment, the repeat machine responds

[1]This notion should not be confused with that of 'determinism': we can, for instance, constrain an arbiter device to behave fairly, but for that we use infinite words as executions, in later chapters.

with the same greeting. The process in Figure 2.5 (a) for the repeat machine was obtained from the process in Figure 2.4 (b) by splitting the rightmost state, to remember the particular greeting used by the environment. Now, observe that the repeat machine is not refined by the etiquette machine, since execution $e1\ m2$ is accessible for the etiquette machine but not for the repeat machine. In other words, the etiquette machine is not a good substitute for the repeat machine, since the etiquette machine might commit a gaffe if the environment is stiff enough to require identical responses. □

Example 2.4 A 'quiet machine' imposes upon the environment the same requirements as the original etiquette machine, but does not respond to greetings. The process for such a machine is shown in Figure 2.5 (b). Execution $e1$ is accessible because it is legal for the quiet machine to stop after $e1$. However, execution $e1$ is not accessible for the etiquette machine. The quiet machine is not a good substitute for the etiquette machine, because of stopping after $e1$; the fault is detected as an extraneous accessible execution.

The repeat machine does refine the etiquette machine, as is easily verified. However, it might be considered unfair to respond always by the same greeting. Such unfairness faults are ignored by the process space over finite strings that we use in this example; still, such faults can be detected by a process space that uses infinite words for executions, as will be discussed in later chapters.

One can also verify that neither the repeat machine nor the etiquette machine refines the quiet machine, and the quiet machine does not refine the repeat machine. We perform such verifications by a tool. □

The following property shows that refinement is a partial order.

Proposition 2.1 *For processes p, q and r,*

(a) $p \sqsubseteq p$, *(reflexivity)*

(b) $p \sqsubseteq q \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$, *(transitivity)*

(c) $p \sqsubseteq q \wedge q \sqsubseteq p \Rightarrow p = q$. *(antisymmetry)*

Proof This follows from Definition 2.4 by reflexivity, transitivity, and antisymmetry of the subset relationship. □

Reflexivity and transitivity are common-sense properties that one would expect from a 'worse-than-or-as-good-as' relationship. The antisymmetry property above (Proposition 2.1) concerns the level of abstraction of the model. A model is *fully abstract* if it makes no irrelevant distinctions among its objects (see for instance [Hen88]), where irrelevance is defined by some equivalence relationship, and distinctions are defined by the formal features of the model. In our case, the objects are processes as constructed in Definition 2.1, and they are distinguished or identified by their execution sets. Irrelevance is with respect to an equivalence relationship derived from the relative correctness condition: two processes are equivalent if they refine each other. Proposition 2.1 (c) determines that process spaces are fully abstract, referring to this equivalence.

The following statement determines that robustness as a predicate on processes is monotonic with respect to refinement.

Proposition 2.2 *For processes p and q,*

$$p \in \mathcal{R}_{\mathcal{E}} \ \wedge \ p \sqsubseteq q \ \Rightarrow \ q \in \mathcal{R}_{\mathcal{E}} \ .$$

Proof From the hypothesis, we have:

$$\mathcal{E} = \mathbf{at}\, p \text{ and } \mathbf{at}\, p \subseteq \mathbf{at}\, q \ ;$$

thus, $\mathbf{at}\, q = \mathcal{E}$, meaning that q is robust. $\square$

If p is 'correct by itself' and q is 'at least as good as' p, then one would expect q to be 'correct by itself' as well.

The void process gives another connection between robustness and refinement.

Proposition 2.3 *For process p,*

$$p \in \mathcal{R}_{\mathcal{E}} \ \Leftrightarrow \ \Phi \sqsubseteq p \ .$$

Proof By the definitions of the refinement relationship and the void process,

$$\Phi \sqsubseteq p \quad \Leftrightarrow \quad \mathbf{as}\, p \subseteq \mathcal{E} \ \wedge \ \mathbf{at}\, p \supseteq \mathcal{E} \ .$$

Since $\mathbf{as}\, p \subseteq \mathcal{E}$ is trivially satisfied, we have

$$\Phi \sqsubseteq p \ \Leftrightarrow \ \mathbf{at}\, p \supseteq \mathcal{E} \ \Leftrightarrow \ p \in \mathcal{R}_{\mathcal{E}} \ .$$ $\square$

In words, p is 'correct by itself' iff p is 'at least as good as' a process that is so indifferent to its environment that can be regarded as 'vacuum'.

Refinement admits the following extremal elements.

Definition 2.5 *The process $(\emptyset, \mathcal{E})$ is called* top *and is denoted by $\top$. The process $(\mathcal{E}, \emptyset)$ is called* bottom *and is denoted by $\perp$.*

Proposition 2.4 *For every process p,*

$$\perp \sqsubseteq p \sqsubseteq \top \ .$$

Proof We have

$$\mathbf{as} \perp \ = \mathcal{E} \supseteq \mathbf{as}\, p \supseteq \emptyset \ = \ \mathbf{as} \top \ ,$$

$$\mathbf{at} \perp \ = \emptyset \subseteq \mathbf{at}\, p \subseteq \mathcal{E} \ = \ \mathbf{at} \top \ .$$ $\square$

Notice that top is robust and bottom is chaotic.

2.3 Operations

For dealing with concurrency, it is important to handle a system of interacting processes as a single process that represents their joint behavior. As we shall see, however, there is a difference in whether we consider the devices or the environments of these processes to be interacting. We define two operations for computing joint behaviors: product can be regarded as the law of composition for devices, while exclusive sum can be regarded as the law of composition for environments.

Definition 2.6 *The* product *of processes p and q is a process $p \times q$ such that*

$$\begin{cases} \mathbf{as}\,(p \times q) & = & \mathbf{as}\,p \cap \mathbf{as}\,q \\ \mathbf{at}\,(p \times q) & = & (\mathbf{at}\,p \cap \mathbf{at}\,q) \cup \overline{\mathbf{as}\,p \cap \mathbf{as}\,q} \end{cases}$$

and the exclusive sum *of p and q is a process $p \oplus q$ such that*

$$\begin{cases} \mathbf{as}\,(p \oplus q) & = & (\mathbf{as}\,p \cap \mathbf{as}\,q) \cup \overline{\mathbf{at}\,p \cap \mathbf{at}\,q} \\ \mathbf{at}\,(p \oplus q) & = & \mathbf{at}\,p \cap \mathbf{at}\,q \end{cases}$$

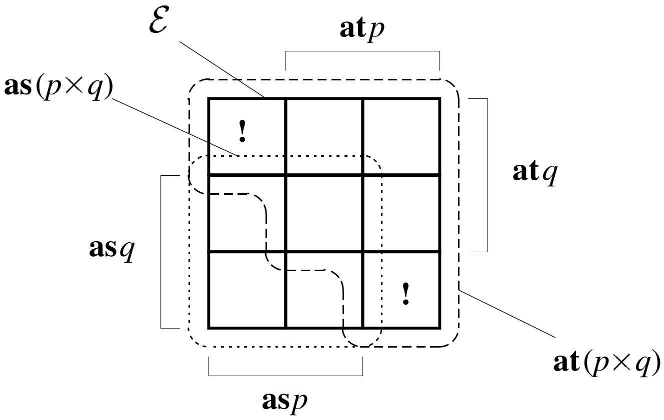


Figure 2.6: The product operation.

To interpret the formal definitions of product and exclusive sum, notice that executions that are accessible to both participating devices are accessible for the system, and executions that are acceptable to both devices are acceptable for the system. However, these two intersections do not cover the entire execution set $\mathcal{E}$: the executions marked '!' in Figure 2.6 have not been accounted for so far.

The '!' executions are escapes for one of the operand processes and rejects for the other. They are violations for the resulting system too, but the question is whether they should be escapes or rejects. If we consider devices to be interacting, we take the position that the system device avoids all executions that are avoided by at least one of the component devices. Accordingly, we define the acceptable set of the product by augmenting $\mathbf{at}\, p \cap \mathbf{at}\, q$ by the executions marked '!'. (Notice that $\mathbf{at}\,(p \times q) = (\mathbf{at}\, p \cap \mathbf{at}\, q) \cup (\mathbf{r}\, p \cap \mathbf{e}\, q) \cup (\mathbf{e}\, p \cap \mathbf{r}\, q)$.)

The exclusive sum operation is similar to the product, except that the '!' executions are rejects, rather than escapes, for the exclusive sum process. This way, if an execution is avoided by the environment of either one of the operand processes, that execution is avoided by the environment of the exclusive sum process too.

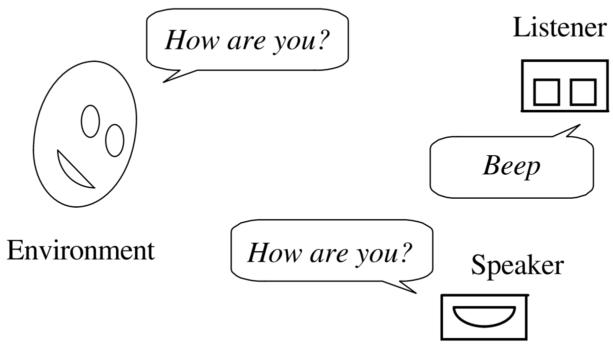
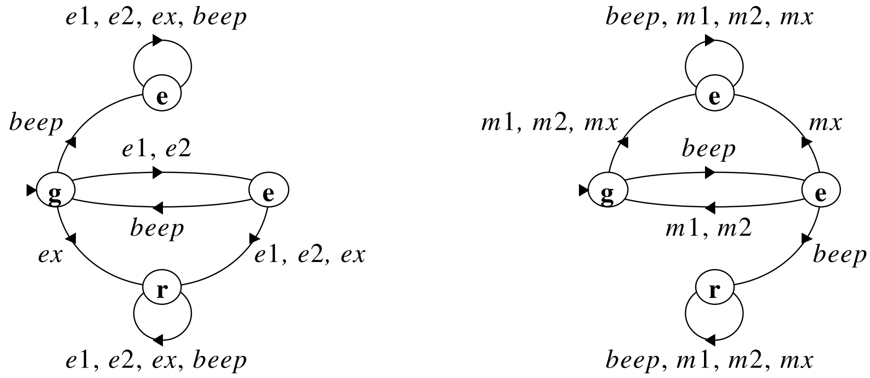


Figure 2.7: Listener and Speaker.

Example 2.5 Consider an etiquette machine which is realized as a system of two parts, call them Listener and Speaker. Listener emits a "beep" each time it hears a greeting from the environment, provided that the greeting is not out of turn and not an "XXX". Speaker delivers a polite greeting each time it hears a "beep".



(self-loops on $m1$, $m2$, mx) (self-loops on $e1$, $e2$, ex)

(a) (b)

Figure 2.8: Example for product: (a) Listener; (b) Speaker.

The processes for Listener and Speaker are shown in Figure 2.8. These processes are over the execution set $\mathcal{E} = \{e1, e2, ex, m1, m2, mx, beep\}^*$, using the shorthands from Figure 2.4 (a) and a new *beep* action. These processes are similar to the process in Figure 2.4, except that certain actions have been omitted from the state machines. We normally omit those actions that are ignored by a device, in the sense that they do not change the state of the device. More precisely, at every state there should be self-loops labeled with those actions. For instance, $m1$, $m2$, mx are ignored by Listener and are omitted from Figure 2.8 (a); these actions would only produce self-loops at every state of Listener.

Now, let us discuss some of the executions of the product of Listener and Speaker. Execution $e2$ *beep* $m1$ is accessible and acceptable for both Listener and

Speaker, since it leads each of them to a goal state. By the definition of product and Figure 2.6, this execution is acceptable and accessible for the product process as well. Execution $e1\ e2$ is accessible and not acceptable for Listener and it is accessible and acceptable for Speaker, since the environment has spoken too fast for Listener, but Speaker has stayed in its initial state, which is a goal state; this execution is accessible and not acceptable for the product process.

An execution corresponding to a square marked '!' in Figure 2.6 is $e1$ *beep beep*; this execution is acceptable and not accessible for Listener and it is accessible and not acceptable for Speaker; thus this execution is acceptable and not accessible for the product process. If a participating device guarantees to avoid a certain execution, the system of devices also guarantees to avoid that execution. □

Example 2.6 The omission of actions that produce self-loops at every state permits us to specify the same machine over different execution sets. The process in Figure 2.4 (b) looks the same if the executions can have action *beep*. One can verify that the process in Figure 2.4 (b) (considered over $\{e1, e2, ex, m1, m2, mx,$ *beep*$\}$) is refined by the product of Listener and Speaker. As will be discussed in Chapter 3, we perform such verifications by a tool. □

Some basic properties of product and exclusive sum are established in the following statement. These operations are idempotent, associative, commutative, and admit the void process as the identity element.

Proposition 2.5 *For processes p, q and r, we have:*

(a) $p \times p = p$, **(a')** $p \oplus p = p$, *(idempotency)*

(b) $(p \times q) \times r = p \times (q \times r)$, **(b')** $(p \oplus q) \oplus r = p \oplus (q \oplus r)$, *(associativity)*

(c) $p \times q = q \times p$, **(c')** $p \oplus q = q \oplus p$, *(commutativity)*

(d) $p \times \Phi = p$, **(d')** $p \oplus \Phi = p$. *(identity element)*

Proof sketch We only consider properties for $\times$; those for $\oplus$ follow by symmetry, as will be discussed later in this chapter.

(a) By idempotency of $\cap$.

(b) Equality of the accessible sets follows immediately from associativity of $\cap$. To show equality of the acceptable sets, we simplify $\mathbf{at}\,((p \times q) \times r)$ by routine calculations, obtaining:

$$\begin{aligned} & \mathbf{at}\,((p \times q) \times r) \\ = \quad & (\mathbf{at}\,p \cap \mathbf{at}\,q \cap \mathbf{at}\,r) \cup \overline{\mathbf{as}\,p \cap \mathbf{as}\,q \cap \mathbf{as}\,r}\ . \end{aligned}$$

Noting that the final form is symmetric in p, q, and r, we also have:

$$\begin{aligned} & (\mathbf{at}\,p \cap \mathbf{at}\,q \cap \mathbf{at}\,r) \cup \overline{\mathbf{as}\,p \cap \mathbf{as}\,q \cap \mathbf{as}\,r} \\ = \quad & \mathbf{at}\,((q \times r) \times p) \\ = \quad & \mathbf{at}\,(p \times (q \times r))\ . \qquad \text{(by commutativity of } \times \text{ (Part (c) below))} \end{aligned}$$

(c) By commutativity of $\cap$.

(d) Routine. □

Informally speaking, the identity element properties above ensure that introducing a void process in a system does not alter the behavior of the system.

A process represents a contract between a device and its environment from the device's point of view; we sometimes need to 'turn the table' by referring to a process that represents the environment's point of view in the same contract.

Definition 2.7 *The* reflection *of process p is a process $-p$ such that*

$$-p \;=\; (\mathbf{at}\,p, \mathbf{as}\,p)\ .$$

In defining reflection, we simply swap the acceptable and accessible sets of the original process.

Reflection is its own inverse, it inverts refinement, and it swaps $\mathcal{R}_\mathcal{E}$ and $\mathcal{C}_\mathcal{E}$.

Proposition 2.6 *For processes p and q,*

(a) $p \;=\; --p\ ,$

(b) $p \sqsubseteq q \;\Leftrightarrow\; -q \sqsubseteq -p\ ,$

(c) $p \in \mathcal{R}_\mathcal{E} \Leftrightarrow -p \in \mathcal{C}_\mathcal{E}\ \ .$

Proof This follows directly from the definitions of $-$, $\sqsubseteq$, $\mathcal{R}_\mathcal{E}$, and $\mathcal{C}_\mathcal{E}$. □

Situations where we do not have complete information about the behavior of a device or environment can be regarded as choices between behavior alternatives. Such situations are modeled by two operations on processes: one operation for choice between devices, and the other for choice between environments.

Definition 2.8 *The* meet *of processes p and q is a process $p \sqcap q$ such that*

$$p \sqcap q \;=\; (\mathbf{as}\,p \cup \mathbf{as}\,q,\ \ \mathbf{at}\,p \cap \mathbf{at}\,q)\ ,$$

and the join *of p and q is a process $p \sqcup q$ such that*

$$p \sqcup q \;=\; (\mathbf{as}\,p \cap \mathbf{as}\,q,\ \ \mathbf{at}\,p \cup \mathbf{at}\,q)\ .$$

Given two alternative processes, the meet and join processes can behave like either of the alternatives, and this behavior choice is made for each execution.

Meet models a choice between devices. If an execution is accessible for either of the alternative devices, the meet device will *not* guarantee to avoid that execution, since we don't know which device was chosen. If an execution is a reject for one of the alternative devices, the meet device will also require its environment to avoid that execution, to forestall the possibility of choosing a device that rejects that execution. Meet can be viewed as an adversary situation,[2] in which we choose the execution and then an adversary chooses a device so as to maximize the possibility of occurrence of faults in a system consisting of the device and its environment.

Dually, join models a non-deterministic choice between environments. The device of $p \sqcup q$ has an acceptable set just large enough and an accessible set just small enough to accommodate an environment that can choose to behave either like $-p$ or like $-q$.

Top and bottom have several notable properties.

Proposition 2.7 *For process p,*

(a) $p \sqcap \top = p$,	**(a')** $p \sqcup \bot = p$,	*(identity elements for* $\sqcap$ *and* $\sqcup$*)*
(b) $p \sqcup \top = \top$,	**(b')** $p \sqcap \bot = \bot$,	*(dominant elements for* $\sqcup$ *and* $\sqcap$*)*
(c) $p \times \top = \top$,	**(c')** $p \oplus \bot = \bot$,	*(dominant elements for* $\times$ *and* $\oplus$*)*
(d) $-\top = \bot$.		

Proof This follows directly from the definitions. □

[2]We thank D. L. Dill for suggesting this interpretation.

Top, a fictitious process, has the miraculous property that all possible flaws in a system of processes are eliminated if top is inserted in that system. Proposition 2.7 (c) indicates that inserting a top process in an arbitrary system will make that system a top too, thus making the system robust.

Figure 2.9 gives a pictorial representation for process spaces. Figure 2.9 (a) indicates the conventions for the diagrams in this figure, Figure 2.9 (b) illustrates a general process space with distinguished processes and the refinement relation, and Figure 2.9 (c) illustrates a process space over a three-element execution set.

In these diagrams, the subsets of $\mathcal{E}$ are represented by disjoint segments on the coordinate axes. Every subset has two segments, one on each axis, equidistant from the origin; notice the positions of Z on the two axes in Figure 2.9 (a). Complementary subsets are represented by segments equidistant from the middle of an axis; notice the segments for $\overline{Z}$ and Z in Figure 2.9 (a). Each pair of subsets of $\mathcal{E}$ is represented by a square whose projections on the axes are the segments of the subsets in the pair; notice the positions of p, $\mathbf{as}\, p$, and $\mathbf{at}\, p$ in Figure 2.9 (a).

With these conventions, a process space $\mathcal{S}_{\mathcal{E}}$ is represented in Figure 2.9 (b): $\mathcal{S}_{\mathcal{E}}$ contains none of the pairs in the heavily shaded area, some of the pairs in the lightly shaded area, and all the pairs in the delimited blank area of Figure 2.9 (b). The upper and right borders in Figure 2.9 (b) are the sets of robust and chaotic processes, denoted by $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{C}_{\mathcal{E}}$, respectively, since the processes on the upper border have the set of acceptable executions equal to $\mathcal{E}$, and the processes on the right border have the set of accessible processes equal to $\mathcal{E}$. The set $\mathcal{D}_{\mathcal{E}}$ contains the pairs of the form $(Z, \overline{Z})$, called *diagonal processes* (just because of their position). The $\sqsubseteq$ signs indicate the direction of the refinement order on sets $\mathcal{R}_{\mathcal{E}}$, $\mathcal{C}_{\mathcal{E}}$ and $\mathcal{D}_{\mathcal{E}}$.

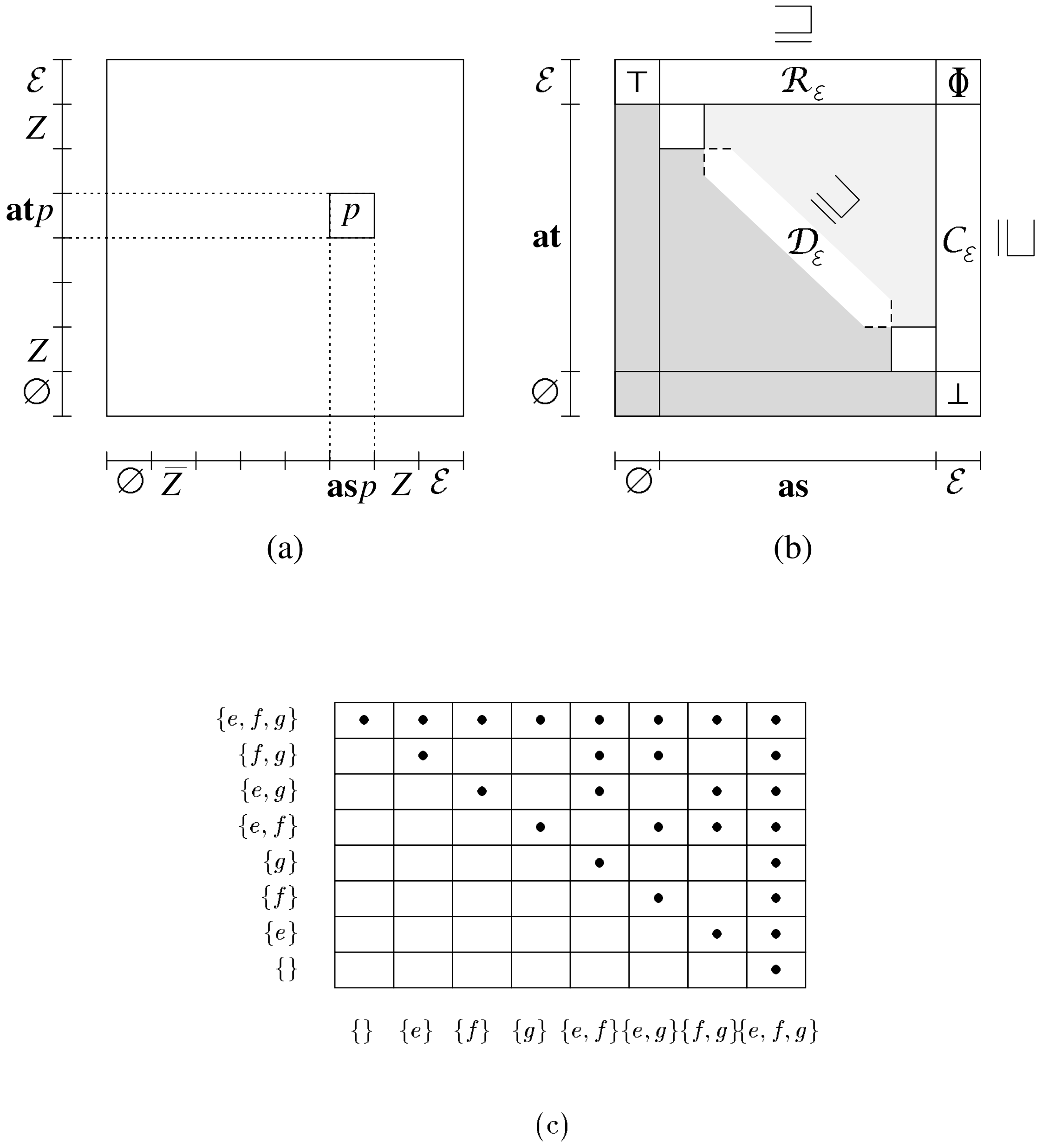


Figure 2.9: Charting a process space: (a) drawing convention; (b) chart; (c) process space over $\{e, f, g\}$.

Figure 2.9 (c) represents $\mathcal{S}_{\{e,f,g\}}$ by the conventions above. Notice that there are 27 processes in this process space; in general, the cardinality of a finite process space is $3^{|\mathcal{E}|}$, where $\mathcal{E}$ is the execution set, also finite.[3] This is because each process assigns one of three designations (reject, goal, escape) to each execution in $\mathcal{E}$.

Throughout our presentation of process spaces, we have mentioned symmetries between environment and device. We formalize these symmetries as follows.

Proposition 2.8 (de Morgan's laws) *For processes p and q,*

(a) $-(p \times q) = -p \oplus -q$, **(a')** $-(p \oplus q) = -p \times -q$,

(b) $-(p \sqcap q) = -p \sqcup -q$, **(b')** $-(p \sqcup q) = -p \sqcap -q$.

Proof This follows immediately from the definitions of the process space operators involved. $\square$

Together, Propositions 2.6 and 2.8 essentially say that reflection is an isomorphism of process spaces. On the basis of these statements, we formulate a duality principle for process spaces.

Remark (duality) Let S be a statement about process spaces. The *dual* of S is a statement S^∂ obtained by replacing in S every occurrence of $\sqsubseteq$ by $\sqsupseteq$, of $\sqcup$ by $\sqcap$, of $\times$ by $\oplus$, of $\mathcal{R}_\mathcal{E}$ by $\mathcal{C}_\mathcal{E}$, of **as** by **at** , of **r** by **e** , and vice versa. (Φ, $-$, **g** , **v** , $\mathcal{E}$, $\mathcal{S}_\mathcal{E}$, and $\mathcal{D}_\mathcal{E}$ are their own duals.) Notice that $S^{\partial\partial} = S$. If statement S holds, then S^∂ holds, too.

In fact, the meet and join operations endow process spaces with a lattice structure, as will be discussed later.

[3] We thank J. A. Brzozowski for these observations.

2.4 Manipulation of Processes

The complexity of manipulating processes grows very rapidly with the sizes of their representations. Therefore, a key idea is to divide a problem into parts, treat the parts separately, and combine the results. We discuss several algebraic properties that support such approaches.

Theorem 2.9 (monotonicity) *For processes p, q, and r,*

$$p \sqsubseteq q \;\Rightarrow\; p \times r \sqsubseteq q \times r \ .$$

Proof First, we show inclusion of the accessible sets.

$$\begin{array}{ll}
& \mathbf{as}\,(p \times r) \supseteq \mathbf{as}\,(q \times r) \\
\Leftrightarrow & \mathbf{as}\,p \cap \mathbf{as}\,r \supseteq \mathbf{as}\,q \cap \mathbf{as}\,r \\
\Leftrightarrow & \overline{(\mathbf{as}\,p \cap \mathbf{as}\,r)} \cap (\mathbf{as}\,q \cap \mathbf{as}\,r) = \emptyset \\
\Leftrightarrow & (\overline{\mathbf{as}\,p} \cup \overline{\mathbf{as}\,r}) \cap (\mathbf{as}\,q \cap \mathbf{as}\,r) = \emptyset \\
\Leftrightarrow & (\overline{\mathbf{as}\,p} \cap \mathbf{as}\,q \cap \mathbf{as}\,r) \cup (\overline{\mathbf{as}\,r} \cap \mathbf{as}\,q \cap \mathbf{as}\,r) = \emptyset \qquad \text{(distributivity)} \\
\Leftrightarrow & \overline{\mathbf{as}\,p} \cap \mathbf{as}\,q \cap \mathbf{as}\,r = \emptyset \qquad (\text{since } \overline{\mathbf{as}\,r} \cap \mathbf{as}\,r = \emptyset) \\
\Leftarrow & \overline{\mathbf{as}\,p} \cap \mathbf{as}\,q = \emptyset \\
\Leftrightarrow & \mathbf{as}\,p \supseteq \mathbf{as}\,q \ .
\end{array}$$

Similarly, we derive

$$\mathbf{at}\,p \cap \mathbf{at}\,r \subseteq \mathbf{at}\,q \cap \mathbf{at}\,r \ .$$

Since $\mathbf{as}\,p \cap \mathbf{as}\,r \supseteq \mathbf{as}\,q \cap \mathbf{as}\,r$, we obtain

$$\begin{array}{ll}
& (\mathbf{at}\,p \cap \mathbf{at}\,r) \cup \overline{(\mathbf{as}\,p \cap \mathbf{as}\,r)} \subseteq (\mathbf{at}\,q \cap \mathbf{at}\,r) \cup \overline{(\mathbf{as}\,q \cap \mathbf{as}\,r)} \\
\Leftrightarrow & \mathbf{at}\,(p \times r) \subseteq \mathbf{at}\,(q \times r) \ .
\end{array}$$

□

If q is 'at least as good as' p, one would expect that coupling the same r to both p and q wouldn't change the situation. However, the fact that r is arbitrary, not related in any ways to p or q, is handy and rather surprising. For instance, nothing prevents us from taking $r = p$, which leads to the following corollary, which tells us that inserting a 'better' part (represented by process q) in a system (represented by process p) can only make the system 'better'.

Corollary 2.10 *For processes p and q,*

$$p \sqsubseteq q \;\Rightarrow\; p \sqsubseteq p \times q \;. \tag{2.2}$$

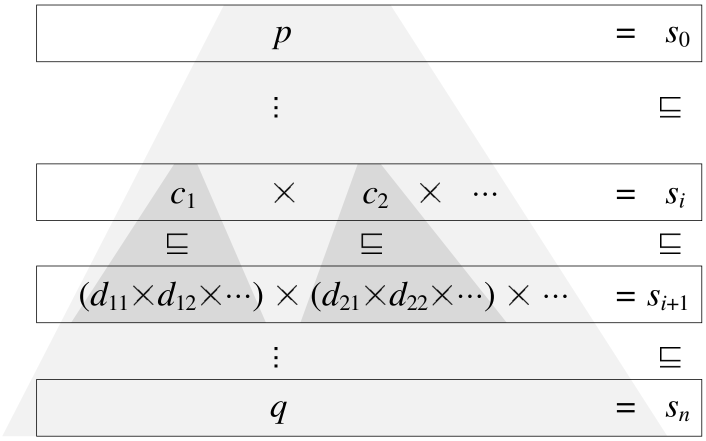


Figure 2.10: Modular and hierarchical verification.

In conjunction with transitivity of refinement (Proposition 2.1 (b)) and commutativity of product (Proposition 2.5 (c)), the monotonicity property of Theorem 2.9 allows for modular and hierarchical verification, as follows.

The problem is to determine whether $p \sqsubseteq q$, where p represents a specification and q an implementation. Typically, one devises a chain of intermediate specifications $s_0, s_1, \ldots, s_n$ such that $s_0 = p$ and $s_n = q$, as in Figure 2.10. Consecutive specifications (including p and q) may be broken into components: $s_i = c_1 \times c_2 \times \cdots$ and $s_{i+1} = (d_{11} \times d_{12} \times \cdots) \times (d_{21} \times d_{22} \times \cdots) \times \cdots$. One verifies, for each j, that $c_j \sqsubseteq d_{j1} \times d_{j2} \times \cdots$. By monotonicity of $\times$ with respect to $\sqsubseteq$, one obtains $s_i \sqsubseteq s_{i+1}$. By the same procedure, one obtains $s_k \sqsubseteq s_{k+1}$ for each k in $\{0, ..., n\}$. By transitivity, $p \sqsubseteq q$ is established.

By combining the modular and hierarchical verification approach above with the idempotency property (Proposition 2.5 (a)), we also allow the components of consecutive specifications to overlap. For instance, one may verify refinement between p and q by breaking p into several parts $p = p_1 \times p_2 \times \cdots$ and by comparing each part to q. The parts might stand for independent properties to be verified individually. If, for each i, $p_i \sqsubseteq q$, then $p \sqsubseteq q$ too.

For finitely many specification levels and finitely many components at each level, the modular and hierarchical verification techniques described above are justified by the following property. (Cases of systems that have infinitely many components will be discussed in Section 2.5.)

Corollary 2.11 *For processes p_1, p_2, q_1, q_2, and q,*

(a) $\quad p_1 \sqsubseteq q_1 \;\wedge\; p_2 \sqsubseteq q_2 \;\Rightarrow\; p_1 \times p_2 \sqsubseteq q_1 \times q_2$,

(b) $\quad p_1 \sqsubseteq q \;\wedge\; p_2 \sqsubseteq q \;\Rightarrow\; p_1 \times p_2 \sqsubseteq q$.

Proof

(a) By Theorem 2.9, we deduce

$$
\begin{aligned}
p_1 \sqsubseteq q_1 &\Rightarrow p_1 \times p_2 \sqsubseteq q_1 \times p_2 \\
p_2 \sqsubseteq q_2 &\Rightarrow p_2 \times q_1 \sqsubseteq q_2 \times q_1 \\
&\Rightarrow q_1 \times p_2 \sqsubseteq q_1 \times q_2 && \text{(by commutativity of } \times)
\end{aligned}
$$

The conclusion follows by transitivity of refinement:

$$p_1 \times p_2 \sqsubseteq q_1 \times p_2 \sqsubseteq q_1 \times q_2 \ .$$

(b) This follows by idempotency of $\times$, by substituting q for q_1 and for q_2 in Part (a) above. □

An alternative definition of refinement is to say that an 'implementation' q is correct with respect to a 'specification' p if q operates correctly in the environment of p. The question arises whether this alternative definition is equivalent to that in Definition 2.4. The following property answers this question, and, by that, connects our relative and absolute notions of correctness.

Theorem 2.12 (verification) *For processes p and q,*

$$p \sqsubseteq q \ \Leftrightarrow\ -p \times q \in \mathcal{R}_{\mathcal{E}} \ .$$

Proof Let $p = (X_1, Y_1)$ and $q = (X_2, Y_2)$. We have:

$$\begin{array}{ll}
 & -p \times q \in \mathcal{R}_{\mathcal{E}} \\
\Leftrightarrow & \mathbf{at}\,(-p \times q) = \mathcal{E} \\
\Leftrightarrow & X_1 \cap Y_2 \cup \overline{Y_1 \cap X_2} = \mathcal{E} \\
\Leftrightarrow & (X_1 \cap Y_2) \cup \overline{Y}_1 \cup \overline{X}_2 = \mathcal{E} \\
\Leftrightarrow & (X_1 \cup \overline{Y}_1 \cup \overline{X}_2) \cap (Y_2 \cup \overline{Y}_1 \cup \overline{X}_2) = \mathcal{E} \quad \text{(by distributivity of } \cup \text{ through } \cap) \\
\Leftrightarrow & (X_1 \cup \overline{X}_2) \cap (Y_2 \cup \overline{Y}_1) = \mathcal{E} \qquad \text{(since } \overline{Y}_1 \subseteq X_1 \text{ and } \overline{X}_2 \subseteq Y_2) \\
\Leftrightarrow & X_1 \cup \overline{X}_2 = \mathcal{E} \ \wedge\ Y_2 \cup \overline{Y}_1 = \mathcal{E} \\
\Leftrightarrow & X_1 \supseteq X_2 \wedge Y_1 \subseteq Y_2 \\
\Leftrightarrow & p \sqsubseteq q \ .
\end{array}$$

□

Theorem 2.12 permits us to verify whether an implementation satisfies a specification by placing the implementation in the environment of the specification, and then checking an absolute correctness condition on their product. Such approaches were taken in [Ebe89, Ebe91a] and further developed in [Dil89] and [Ver94b].

Another alternative definition of refinement uses a testing point of view: q is 'better than or as good as' p if q passes all tests that p passes. Passing a test r can be viewed as the absence of rejects when the device is coupled with r. Theorem 2.13 shows that this testing definition of refinement is equivalent to Definition 2.4, and thereby provides another link between the absolute and relative notions of correctness in process spaces.

Theorem 2.13 (testing) *For processes p and q,*

$$p \sqsubseteq q \;\Leftrightarrow\; \forall r \in \mathcal{S}_{\mathcal{E}} : (r \times p \in \mathcal{R}_{\mathcal{E}} \Rightarrow r \times q \in \mathcal{R}_{\mathcal{E}}) \; .$$

Proof By Theorem 2.12 and Proposition 2.6 (a), we have $r \times p \in \mathcal{R}_{\mathcal{E}} \Leftrightarrow -r \sqsubseteq p$ and $r \times q \in \mathcal{R}_{\mathcal{E}} \Leftrightarrow -r \sqsubseteq q$. Thus, it is sufficient to prove

$$p \sqsubseteq q \;\Leftrightarrow\; \forall r \in \mathcal{S}_{\mathcal{E}} : \; (-r \sqsubseteq p \Rightarrow -r \sqsubseteq q) \; .$$

($\Rightarrow$) By transitivity of refinement (Proposition 2.1 (b)),

$$p \sqsubseteq q \wedge -r \sqsubseteq p \Rightarrow -r \sqsubseteq q \; .$$

($\Leftarrow$) Let $r = -p$. By Proposition 2.6 (a), $-r = p$. By reflexivity of refinement (Proposition 2.1 (a)), $-r \sqsubseteq p$. By hypothesis, $-r \sqsubseteq q$. Finally, since $-r = p$, we have $p \sqsubseteq q$. □

Theorem 2.13 refers to the "testing paradigm", which is commonplace in concurrency theory (see e.g. [DNH83]).

The *design inequality* is

$$p \sqsubseteq q \times r \ ,$$

where process p represents a known specification, process q represents a known part of the implementation, and process r represents the unknown remaining part of the implementation. Theorem 2.14 solves the design inequality by characterizing its solutions as those processes that refine a minimal solution.

Theorem 2.14 (design inequality) *For processes p, q, and r,*

$$p \sqsubseteq q \times r \ \Leftrightarrow \ p \oplus -q \sqsubseteq r \ .$$

Proof

$$\begin{array}{lll}
 & p \sqsubseteq q \times r & \\
\Leftrightarrow & -p \times q \times r \in \mathcal{R}_{\mathcal{E}} & \text{(by Theorem 2.12)} \\
\Leftrightarrow & -(p \oplus -q) \times r \in \mathcal{R}_{\mathcal{E}} & \text{(by Proposition 2.6 (a) and (c'))} \\
\Leftrightarrow & p \oplus -q \sqsubseteq r \ . & \text{(by Theorem 2.12)} \quad \Box
\end{array}$$

Solutions to the design inequality have been proposed previously in [Pra91] and [Ver94b]. The design inequality also relates to the "supervisory control problem" in [RW87], [WR87], and [Sme89].

Example 2.7 In Example 2.5, let p be the process for the etiquette machine, q the process for Speaker, and r the process for Listener. One can verify that $p \oplus -q \sqsubseteq r$, thus $p \sqsubseteq q \times r$. □

Example 2.8 One possible application of the design inequality may be the design of software for embedded systems. In that case, p can be the (known) specification of the embedded system, q the (known) description of the underlying machine, and r the (unknown) specification for the software. Another possible application is the design of interface circuitry for communication protocols: p and q are two interfaces, and r is the specification for a 'glue' circuit. $\square$

Although the design inequality holds promise for applications to synthesis problems (to derive an implementation from a higher-level specification), an inconvenience is that its minimal solution provided by Theorem 2.14 is not necessarily the simplest solution in terms of the number of states. The minimal solution can be simplified by using the process abstractions discussed in Chapter 8. However, we do not pursue this issue further in this thesis, concentrating instead on verification applications.

Off-the-shelf parts and subsystems are often intended to be fool-proof and to have a defined behavior in any environment. For instance, voltage regulated sources would often have overload protection. Such features are modeled in process spaces by the robustness property: the device specified by a process accepts every execution, regardless of how the environment behaves in that execution. At the same time, the environments (e.g. users) should ideally be assumed to be completely unpredictable, in the sense that they offer no guarantees in terms of avoided executions; in process spaces, such unpredictable behaviors are modeled by chaotic processes. This provides motivation for mentioning the following property, which shows how to split a process into a robust and a chaotic part.

Theorem 2.15 (RC decomposition)

For process p,

(a) $p \sqcup \Phi$ *is robust,*

(b) $p \sqcap \Phi$ *is chaotic,*

(c) $p = (p \sqcup \Phi) \times (p \sqcap \Phi)$.

Proof

(a) $p \sqcup \Phi \;=\; (\mathbf{as}\,p \cap \mathcal{E}, \mathbf{at}\,p \cup \mathcal{E}) \;=\; (\mathbf{as}\,p, \mathcal{E}) \;\in\; \mathcal{R}_{\mathcal{E}}$.

(b) $p \sqcap \Phi \;=\; (\mathbf{as}\,p \cup \mathcal{E}, \mathbf{at}\,p \cap \mathcal{E}) \;=\; (\mathcal{E}, \mathbf{at}\,p) \;\in\; \mathcal{C}_{\mathcal{E}}$.

(c) $(p \sqcup \Phi) \times (p \sqcap \Phi)$

$$
\begin{aligned}
&= (\mathbf{as}\,p, \mathcal{E}) \times (\mathcal{E}, \mathbf{at}\,p) \\
&= (\mathbf{as}\,p, \mathbf{at}\,p \cup \overline{\mathbf{as}\,p}) \\
&= p\ . \qquad\qquad (\text{since } \overline{\mathbf{as}\,p} = \mathbf{e}\,p \subseteq \mathbf{at}\,p)
\end{aligned}
$$

□

Robust processes can be regarded as pure guarantees, and chaotic processes as pure requirements. Theorem 2.15 shows that every process is the product of a pure guarantee and a pure requirement, and provides a way to compute the factors.

If all components of a system are robust, is the system itself robust? Proposition 2.16 implies that the product of two robust devices or environments is also robust, and also indicates some other closure properties of the set of robust processes. (The infinite case is also treated, in Section 2.5.)

Proposition 2.16 *$\mathcal{R}_{\mathcal{E}}$ is closed under $\times$, $\oplus$, $\sqcup$, and $\sqcap$.*

Proof Routine. If we let p and q be robust processes, the property follows when we compute the accessible sets of $p \times q$, $p \oplus q$, etc. □

2.5 Sets of Processes

The product and exclusive sum operators extend naturally to sets of processes of arbitrary cardinality. These extended operators obey certain algebraic properties that permit simplified checks of refinement and robustness for possibly infinite systems of processes.

Definition 2.9 *For subset* $\mathcal{B}$ *of* $\mathcal{S}_{\mathcal{E}}$, *let the* product *of* $\mathcal{B}$ *and the* exclusive sum *of* $\mathcal{B}$ *be, respectively,*

$$\begin{aligned}\times\mathcal{B} &= (\textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p,\ \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \cup \overline{\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p})\ ,\\ \oplus\mathcal{B} &= (\textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p \cup \overline{\bigcap_{p\in\mathcal{B}} \mathbf{at}\, p},\ \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p)\ .\end{aligned}$$

The extended and the binary forms of product and exclusive sum are equivalent.

Proposition 2.17 *For processes* p *and* q,

$$\begin{aligned}p \times q &= \times\{p, q\}\ ,\\ p \oplus q &= \oplus\{p, q\}\ .\end{aligned}$$

Proof This follows immediately from the definitions. □

The following property provides a criterion for relative correctness (refinement) between products of sets of processes.

Lemma 2.18 (system refinement) *For process sets* $\mathcal{B}$ *and* $\mathcal{C}$,

$$(\textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p \supseteq \bigcap_{q\in\mathcal{C}} \mathbf{as}\, q)\ \wedge\ (\bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \subseteq \bigcap_{q\in\mathcal{C}} \mathbf{at}\, q)\ \Rightarrow\ \times\mathcal{B} \sqsubseteq \times\mathcal{C}\ .$$

Proof We have:

$$\textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p \supseteq \bigcap_{q\in\mathcal{C}} \mathbf{as}\, q\ \Leftrightarrow\ \left.\begin{array}{r}\overline{\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p} \subseteq \overline{\bigcap_{q\in\mathcal{C}} \mathbf{as}\, q}\\ \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \subseteq \bigcap_{q\in\mathcal{C}} \mathbf{at}\, q\end{array}\right\}\ \Rightarrow\ \mathbf{at}\,(\times\mathcal{B}) \subseteq \mathbf{at}\,(\times\mathcal{C})$$

and

$$\textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p \supseteq \bigcap_{q\in\mathcal{C}} \mathbf{as}\, q\ \Leftrightarrow\ \mathbf{as}\,(\times\mathcal{B}) \supseteq \mathbf{as}\,(\times\mathcal{C})\ .$$

Thus, $\times\mathcal{B} \sqsubseteq \times\mathcal{C}$. $\square$

Remark A "nasty technical problem" regarding systems of infinitely many processes was mentioned in [Ver94b, p. 111]. If we have $p_i \sqsubseteq q_i$ for every $i \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers, do we also have $\times\{p_i|i \in \mathbb{N}\} \sqsubseteq \times\{q_i|i \in \mathbb{N}\}$? In our model the answer is yes, even for uncountable process sets; for such sets, we replace $\mathbb{N}$ by an arbitrary index set I.

For all $i \in I$, $p_i \sqsubseteq q_i$ implies $\mathbf{as}\, p_i \supseteq \mathbf{as}\, q_i$ and $\mathbf{at}\, p_i \subseteq \mathbf{at}\, q_i$. Therefore, $\bigcap_{i\in I} \mathbf{as}\, p_i \supseteq \bigcap_{i\in I} \mathbf{as}\, q_i$ and $\bigcap_{i\in I} \mathbf{at}\, p_i \subseteq \bigcap_{i\in I} \mathbf{at}\, q_i$. By Lemma 2.18, we have $\times\{p_i|i \in I\} \sqsubseteq \times\{q_i|i \in I\}$. $\square$

The converse of Lemma 2.18 does not hold. Take, for instance, $\mathcal{B} = \{\top\}$ and $\mathcal{C} = \{\top, \bot\}$. Although $\times\mathcal{B} = \times\mathcal{C} = \top$, we have $\bigcap_{p\in\mathcal{B}} \mathbf{at}\, p = \mathcal{E} \not\subseteq \emptyset = \bigcap_{p\in\mathcal{C}} \mathbf{at}\, p$.

On the other hand, the following property provides a necessary and sufficient criterion for absolute correctness (robustness) of products of sets of processes.

Lemma 2.19 (system robustness) *For set $\mathcal{B}$ of processes,*

$$\times\mathcal{B} \in \mathcal{R}_\mathcal{E} \iff \bigcap_{p\in\mathcal{B}} \mathbf{as}\, p \subseteq \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \ .$$

Proof

$$\times\mathcal{B} \in \mathcal{R}_\mathcal{E} \iff \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \cup \overline{\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p} = \mathcal{E} \iff \bigcap_{p\in\mathcal{B}} \mathbf{as}\, p \subseteq \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \ . \qquad \square$$

2.6 Lattice Properties

The terminology and notation for meet and join suggest the following connections to the refinement relationship.

Proposition 2.20 (connecting lemma) *For processes p and q,*

$$p \sqsubseteq q \;\Leftrightarrow\; p \sqcup q = q \;\Leftrightarrow\; p \sqcap q = p \;.$$

Proof This follows immediately from definitions, by using the similar properties of $\subseteq$, $\cup$, and $\cap$ for the accessible and acceptable sets of p and q. □

The meet and join operators can be extended to sets of arbitrary cardinality, endowing process spaces with a lattice structure. The following property shows that a process space $\mathcal{S}_\mathcal{E}$ with the refinement partial order is a complete lattice.

Proposition 2.21

For every subset $\mathcal{B}$ of $\mathcal{S}_\mathcal{E}$, there exist unique processes $\sqcup\mathcal{B}$ (called the join *of $\mathcal{B}$) and $\sqcap\mathcal{B}$ (called the* meet *of $\mathcal{B}$) such that, for every process p,*

$$(\forall q \in \mathcal{B} : q \sqsubseteq p) \Leftrightarrow \sqcup\mathcal{B} \sqsubseteq p \;, \qquad \textit{(the least upper bound of } \mathcal{B}\textit{)}$$
$$(\forall q \in \mathcal{B} : p \sqsubseteq q) \Leftrightarrow p \sqsubseteq \sqcap\mathcal{B} \;. \qquad \textit{(the greatest lower bound of } \mathcal{B}\textit{)}$$

Proof idea The operators above have the following forms:

$$\sqcup\mathcal{B} = (\textstyle\bigcap_{q\in\mathcal{B}} \mathbf{as}\, q, \bigcup_{q\in\mathcal{B}} \mathbf{at}\, q) \;,$$
$$\sqcap\mathcal{B} = (\textstyle\bigcup_{q\in\mathcal{B}} \mathbf{as}\, q, \bigcap_{q\in\mathcal{B}} \mathbf{at}\, q) \;.$$ □

In particular, the construction in the proof of Proposition 2.21 establishes the correspondence between the extended and the binary forms of meet and join:

$$p \sqcap q = \sqcap\{p, q\} \;,$$
$$p \sqcup q = \sqcup\{p, q\} \;.$$

The following statement shows that a process space with the refinement partial order is also a distributive lattice.

Proposition 2.22 *For processes p, q, and r,*

(a) $p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$, *(distributivity of $\sqcap$ through $\sqcup$)*

(a') $p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$. *(distributivity of $\sqcup$ through $\sqcap$)*

Proof Use the distributivity properties of union and intersection. $\square$

Refinement and reflection do not induce a Boolean algebra structure upon process spaces, since $\Phi \sqcup -\Phi = \Phi$ but $\Phi \neq \top$ whenever $\mathcal{E} \neq \emptyset$, thus in general we do not have $\Phi \sqcup -\Phi = \top$. However, refinement and reflection do induce a ternary algebra structure [BLN96].

2.7 Symmetries of Process Spaces

In Section 2.3, we stated a duality principle for process spaces, to help organize their algebraic structure. For instance, a property like Theorem 2.12 needs only be stated for $\times$, $\mathcal{R}_{\mathcal{E}}$, and $\sqsubseteq$, and the duality principle entails a similar property for $\oplus$, $\mathcal{C}_{\mathcal{E}}$, and $\sqsupseteq$. Process spaces also have a ternary symmetry and two other dualities, obtained by applying the rotation operators below.

Definition 2.10 *The* forward rotation *of process p is a process*

$$/p \;=\; (\overline{\mathbf{as}\,p} \cup \overline{\mathbf{at}\,p}, \mathbf{as}\,p) \;,$$

and the backward rotation *of p is a process*

$$\backslash p \;=\; (\mathbf{at}\,p, \overline{\mathbf{as}\,p} \cup \overline{\mathbf{at}\,p}) \;.$$

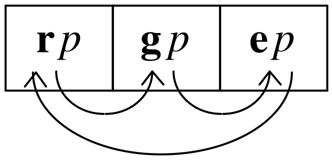


Figure 2.11: Forward rotation.

One can visualize forward rotation as moving the contents of the reject, goal, and escape sets of a process as shown in Figure 2.11. Other substitutions performed by rotation are listed in Table 2.1.

Forward and backward rotations are duals, and also they are related as follows.

Proposition 2.23 *For process p,*

(a) $///p \;=\; p\;,$

(b) $\backslash p \;=\; //p\;.$

Proof

(a) $///p \;=\; //(\overline{\mathbf{as}\,p} \cup \overline{\mathbf{at}\,p}, \mathbf{as}\,p) \;=\; //(\mathbf{v}\,p, \mathbf{as}\,p) \;=\; /(\overline{\mathbf{v}\,p} \cup \overline{\mathbf{as}\,p}, \mathbf{v}\,p)$

$= \;/(\mathbf{at}\,p, \mathbf{v}\,p) \;=\; (\overline{\mathbf{at}\,p} \cup \overline{\mathbf{v}\,p}, \mathbf{at}\,p) \;=\; (\mathbf{as}\,p, \mathbf{at}\,p) \;=\; p$

(b) As in Part (a),

$//p \;=\; (\mathbf{at}\,p, \mathbf{v}\,p) \;=\; \backslash p\;.$ □

It follows that forward and backward rotations are inverses of each other; thus they are bijective functions.

Rotations link the top, bottom, and void in a process space, and the robust, chaotic, and diagonal processes.

Proposition 2.24

(a) *For process p,*

$$p \in \mathcal{R}_{\mathcal{E}} \;\Leftrightarrow\; /\, p \in \mathcal{D}_{\mathcal{E}} \;\Leftrightarrow\; //\, p \in \mathcal{C}_{\mathcal{E}} \ .$$

(b)

$$\Phi \;=\; /\bot \;=\; //\top \ .$$

Proof This follows immediately from definitions. □

The binary operators of process spaces are also linked by rotations. In particular, rotations can be viewed as transforming product into a non-deterministic choice: rotate the operand processes, apply meet, and then rotate the result back, to obtain the product of the original operands. Similarly, meet can be transformed into product. By duality, similar relationships can be established between join and exclusive sum.

Theorem 2.25 (rotation) *For processes p and q,*

$$p \times q = \backslash(/p \sqcap /q) \ .$$

Proof

$$\mathbf{as}\,\backslash(/p \sqcap /q) \;=\; \mathbf{at}\,/p \cap \mathbf{at}\,/q \;=\; \mathbf{as}\,p \cap \mathbf{as}\,q \;=\; \mathbf{as}\,(p \times q) \ ;$$

$$\begin{aligned}
& \mathbf{at}\,\backslash(/p \sqcap /q) \\
= \quad & \mathbf{v}\,(/p \sqcap /q) && \text{(by definition of /)} \\
= \quad & \mathbf{r}\,/p \cup \mathbf{r}\,/q \cup (\mathbf{e}\,/p \cap \mathbf{e}\,/q) && (\text{since } \mathbf{v}\,(/p \sqcap /q) = \mathbf{r}\,(/p \sqcap /q) \cup \mathbf{e}\,(/p \sqcap /q)) \\
= \quad & \mathbf{e}\,p \cup \mathbf{e}\,q \cup (\mathbf{g}\,p \cap \mathbf{g}\,q) \\
= \quad & \mathbf{at}\,(p \times q) \ . && \text{(see the Venn diagram in Figure 2.6)}
\end{aligned}$$

□

To complete the ternary symmetry, we need to introduce two binary operators on processes, in addition to the previously defined ones. It is remarkable that only two new binary operators suffice, given that four binary operators were defined previously; some links already exist among the previous operators.

Definition 2.11 *The* exclusive product *and the* sum *of processes p and q are the following processes, respectively:*

$$p \otimes q = \backslash(/p \times /q) \ ,$$
$$p + q = /(\backslash p \oplus \backslash q) \ .$$

Sum and exclusive product extend to sets of processes in a manner similar to Definition 2.9 and Proposition 2.21.

Sum and exclusive product can be connected by some other transformation laws to the previously defined binary operators of process spaces. Sum and exclusive product are duals, so we only give the law for exclusive product.

Proposition 2.26 *For processes p and q,*

$$p \otimes q \ = \ /(\backslash p \sqcap \backslash q) \ .$$

Proof

$$\begin{array}{lll} & (/p \times /q) & \\ = & \backslash\backslash(//p \sqcap //q) & \text{(cf. Theorem 2.25)} \\ = & /(\backslash p \sqcap \backslash q) & \text{(by Proposition 2.23 (a) and (b))} \quad \square \end{array}$$

In a similar manner, we can transform the reflection operator to obtain two new unary operators on processes. These operators have their own de Morgan's laws and duality principles.

Definition 2.12 *For process p, let*

$$\div p = \backslash - /p \ ,$$
$$\ominus p = / - \backslash p \ .$$

We can also transform the refinement relationship to obtain two new partial orders. One of these new orders, suggested by J. A. Brzozowski, has an interesting meaning.

Definition 2.13 *Process q is* transparent *for process p, written $p \leq q$, if*

$$p \;=\; p \times q \;.$$

Thus, q does not affect the operation of p in a joint behavior. Void is the most transparent process: in a system, it does not affect the operation of other processes. Top is the least transparent process: in a system, it masks everything else by making the entire system a top.

Proposition 2.27 *For process p,*

$$\top \;\leq\; p \leq\; \Phi \;.$$

Proof For every process p,

$$p \times \Phi = \; p \;,$$
$$\top \times p = \; \top \;. \qquad \square$$

The transparency order can be obtained by 'rotating' the refinement order.

Proposition 2.28 *For processes p and q,*

$$p \leq q \;\Leftrightarrow\; /p \sqsubseteq /q \;.$$

Proof By Theorem 2.25 and Proposition 2.23, we obtain

$$p \leq q \;\Leftrightarrow\; p = p \times q \;\Leftrightarrow\; p = \backslash(/p \sqcap /q) \;\Leftrightarrow\; /p = /p \sqcap /q \;.$$

By Proposition 2.20,

$$/p = /p \sqcap /q \;\Leftrightarrow\; /p \sqsubseteq /q \;. \qquad \square$$

Another order can be obtained by further rotating transparency.

Definition 2.14 *Process q is* more sensitive *than process p, written $p \trianglelefteq q$, if*

$$/p \le /q \;.$$

The order $\trianglelefteq$ is the dual of $\ge$.

as	$\to$	**at**	$\to$	**v**	$\to$	**as**
e	$\to$	**r**	$\to$	**g**	$\to$	**e**
$\mathcal{C}_\mathcal{E}$	$\to$	$\mathcal{R}_\mathcal{E}$	$\to$	$\mathcal{D}_\mathcal{E}$	$\to$	$\mathcal{C}_\mathcal{E}$
$\top$	$\to$	$\bot$	$\to$	Φ	$\to$	$\top$
$\sqcap$	$\to$	$\otimes$	$\to$	$\times$	$\to$	$\sqcap$
$\sqcup$	$\to$	$\oplus$	$\to$	$+$	$\to$	$\sqcup$
$\sqsubseteq$	$\to$	$\trianglelefteq$	$\to$	$\le$	$\to$	$\sqsubseteq$
$-$	$\to$	$\ominus$	$\to$	$\div$	$\to$	$-$

Table 2.1: Substitutions made by the ternary symmetry principle.

On account of Theorem 2.25 (a), Propositions 2.23, 2.24, 2.26 and 2.28, and Definitions 2.11, 2.12, and 2.14, we formulate a ternary symmetry principle for process spaces.

Remark (ternary symmetry) Let S be a statement about process spaces. The *forward ternary correspondent* of S is a statement S^τ obtained by the substitutions in Figure 2.1. (The rotation operators and the sets $\mathcal{E}$ and $\mathcal{S}_\mathcal{E}$ are their own substitutes.) Notice that $S^{\tau\tau\tau} = S$. If statement S holds, then S^τ holds, too.

Example 2.9 The forward ternary correspondent of Proposition 2.7 (b) is Proposition 2.7 (c'). $\square$

The duality and ternary symmetry of process spaces can be summarized into a six-way symmetry based on the group of permutations of three elements. Let us call $\times$, $\sqcap$, $\otimes$, $\oplus$, $+$, and $\sqcup$ *process compositions*, and $\sqsubseteq$, $\trianglelefteq$, $\leq$, $\sqsupseteq$, $\trianglerighteq$, $\geq$ *process comparisons*. Process compositions and comparisons correspond to the six possible orderings of set $\{\mathbf{r}\,,\mathbf{g}\,,\mathbf{e}\}$, as follows. We associate $\sqsubseteq$ and $\sqcup$ to ordering $(\mathbf{r}\,,\mathbf{g}\,,\mathbf{e})$, and we apply the replacements prescribed by the duality or ternary symmetry principles to the comparison, composition, and ordering above, any number of times but simultaneously to these three objects. Then, we obtain that each process composition and each process comparison will be associated invariantly with an ordering of $\{\mathbf{r}\,,\mathbf{g}\,,\mathbf{e}\}$. These correspondences also extend to the unary operators $-$, $\ominus$, and $\div$; however, each of these unary operators has two attached orderings, derived from associating $(\mathbf{r}\,,\mathbf{g}\,,\mathbf{e})$ and $(\mathbf{e}\,,\mathbf{g}\,,\mathbf{r})$ to $-$.

Although at the moment we do not have direct practical applications for the symmetries above, the duality and ternary symmetry principles are useful because they help us to keep track of the properties of many process space operations, without having to prove each algebraic property separately for every combination of operations.

Chapter 3

Handling Safety and Finalization

In this chapter we use processes over finite words for studying two important correctness concerns: safety and finalization. For many applications, the execution sets are regular languages of finite words and can be represented by a type of finite automaton. Certain aspects of modeling of systems by state machines over finite executions have already been illustrated in the examples of Chapter 2, and will be demonstrated again in the application to circuit verification; here, we briefly discuss the basics of such modeling.

An obstacle to the verification of discrete-state systems is state explosion, that is, the fact that the number of states of such a system can grow exponentially with the number of components. Our basic verification problems are computationally intractable in the worst cases.

On the other hand, there are algorithms that perform well on typical cases of practical interest, even though the worst cases are still intractable. We have implemented operations on finite automata in a tool called FIREMAPS. The label FIREMAPS stands for finitary and regular manipulation of processes and systems.

For efficiency, FIREMAPS uses a package of BDD routines obtained from the Model Checking Group at Carnegie Mellon University [BRB90]. BDDs (binary decision diagrams) are data structures for representing Boolean functions [Bry86]. Extensive experimental evidence indicates that, in many cases, the efficiency of BDD-based algorithms can be better by orders of magnitude than the traditional representations of Boolean functions (see for instance [Bry86] and [BRB90]); this observation is confirmed for typical cases of our applications too.

3.1 Safety and Finalization

Safety can be regarded as avoidance of illegal incidents, and finalization can be regarded as avoidance of global deadlock. In Example 2.1, we don't want "XXX" greetings, and we don't want the machine to have the right to remain silent after a polite, timely greeting from the environment. Following an informal description from [LL90], *safety* imposes that 'something bad does not happen'. Also, let us refer to avoidance of global deadlock as *finalization*, describing it informally as 'something good does happen'. An example of what we call the finalization correctness concern is the termination property for sequential programs, which essentially says that a program does not loop forever and will eventually produce results. However, finalization is more general than termination, because finalization also applies to cases where inputs continue to be fed in after the results are produced, and to cases where several devices operate in parallel. Also, finalization relates to the more general correctness concerns of 'liveness' and 'progress', discussed in Chapter 6.

In our main applications, we take an event-oriented point of view. *Events* are atomic changes of state, possibly occurring simultaneously in several processes, like the greetings in the environment-machine examples in Chapter 2. In this interpretation, the expressions 'something bad' and 'something good' above refer to undesired and desired events, respectively.

To model synchronization of state changes in different processes, events bear labels called *actions*. In the Chapter 2 examples, the actions are $e1$, ..., *beep*. One way to study discrete-state systems is to take executions to be strings of actions, which can be finite or infinite. These strings can be interpreted as partial or total observations of a system's behavior; in the context of such an interpretation, we refer to strings as *traces*. Formally, the terms 'string', 'word', and 'trace' are synonymous; however, 'trace' carries the connotation of being interpreted as a recording of events that actually occur in a system.

In this chapter, we use finite traces only. An observation of the behavior at a device-environment interface can be interpreted as a partial observation representing the behavior since the time the system was started until a certain moment in time, or as a total observation representing the entire operation of a system until it comes to a stop (i.e., until it ceases to issue new events). Correspondingly, we call a finite trace *partial* or *total* in the context of these interpretations.

Total finite traces deal only with situations where a system does come to a (legal or illegal) stop; if no-stop situations are of interest, one can use a different type (infinite words) or interpretation for executions, as discussed in Chapter 6.

The interpretations above as partial and total observations amount to two ways of describing a system s by a process over finite traces. These are two different ways of classifying finite strings of actions as accessible and/or acceptable. For dealing with partial finite traces, we construct the *safety process* of the system, a process σs over $\mathcal{U}^*$; for total finite traces, we construct the *finalization process*, a process φs over $\mathcal{U}^*$.

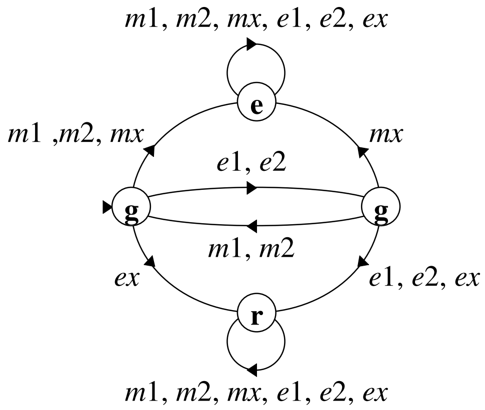


Figure 3.1: Safety process of the etiquette machine.

Example 3.1 For the etiquette machine in Example 2.1, execution ⟨Environment: How are you?⟩ is an escape of the finalization process, since it is the responsibility of the machine to respond to the greeting, thereby ensuring that this execution may not occur as a total finite trace of the system. At the same time, this execution is a goal of the safety process, since it is a partial observation of the legal behavior ⟨Environment: How are you?⟩ ⟨Machine: How are you?⟩. The process in Figure 2.4 is the finalization process of the etiquette machine, whereas Figure 3.1 here represents its safety process. □

Often, there is a tradeoff between the effort for analysis and the number of correctness issues that we care to specify and verify. Although safety and finalization are different correctness concerns and should be verified individually, for many real-life systems there are relationships between safety and finalization similar to those occurring in Example 3.1, in the sense that safety flaws are usually also finalization flaws. This is because, after a safety flaw, some systems might deadlock, producing a finalization flaw. For example, execution $e1mx$ is an escape both for the safety process in Figure 3.1 and for the finalization process in Figure 2.4 (b). For this reason, we usually verify finalization only.

Example 3.2 In Example 2.3, we noted that the etiquette machine does not satisfy the repeat machine specification, because execution $e1\ m2$ is not accessible for the latter. This mismatch is a safety fault since the undesired event $m2$ happens; nevertheless, it is also a finalization flaw, because execution $e1\ m2$ leads the specification to an illegal stopping point. □

The algebraic structure of Chapter 2 is inherited by the safety and finality processes by simply taking the execution set to be $\mathcal{U}^*$.

3.2 Process Automata

If its execution sets are regular languages, a (safety or finalization) process over $\mathcal{U}^*$ can be represented as a finite automaton. To specify the effects of all actions of $\mathcal{U}$, we specify individually the effects of actions from a subset Σ of $\mathcal{U}$, called the *alphabet* of the automaton, and we let actions from outside the alphabet to occur arbitrarily without changing state. However, it is important to keep in mind that our safety and finalization processes are all over the same set of executions ($\mathcal{U}^*$) and do not have individually designated alphabets; finite automata with alphabets are only a means to specify processes for which only finitely many actions can induce changes of state, as opposed to self-loops at every state. Also, for general processes the execution sets might not be regular; finite automata are just a means to specify processes whose execution sets are actually regular.

Definition 3.1 *A* process automaton *over $\mathcal{U}$ is a tuple $\mathcal{P} = \langle \Sigma, Q, I, E, Q_{\mathrm{as}}, Q_{\mathrm{at}} \rangle$, where*

1. Σ *is a finite subset of* $\mathcal{U}$, *called the* alphabet *of* $\mathcal{P}$.

2. Q *is a finite set of* states.

3. $I \subseteq Q$ *is a set of* initial states.

4. Q_{as} *and* Q_{at} *whose elements are called* accessible *and* acceptable, *respectively, are subsets of* Q *such that* $Q = Q_{\mathrm{as}} \cup Q_{\mathrm{at}}$.

5. $E \subseteq Q \times \Sigma \times Q$ *is a set of* edges.

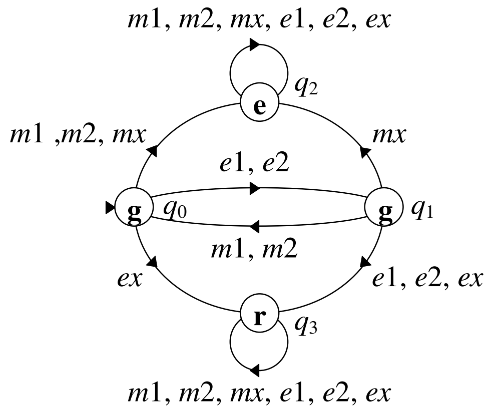


Figure 3.2: Example of a process automaton.

Example 3.3 The automaton in Figure 3.2 is the same as that in Figure 3.1, except that we have given labels to states for easy reference. This automaton has alphabet $\{e1,\ e2,\ ex,\ m1,\ m2,\ mx\}$, state set $\{q_0,\ q_1,\ q_2,\ q_3\}$, set $\{q_0\}$ of initial states, set $\{q_0, q_1, q_3\}$ of accessible states, set $\{q_0, q_1, q_2\}$ of acceptable states, and a set of 24 edges (4 states times 6 labels), including $(q_0, e1, q_1)$ and (q_3, mx, q_3), but not (q_1, ex, q_2). $\square$

To specify a process by a process automaton, we need to classify each sequence of actions from $\mathcal{U}^*$ as accessible or acceptable for the process. For this, we handle actions from Σ in the usual way, and we let actions from $\mathcal{U}\backslash\Sigma$ to occur arbitrarily without changing state.

Definition 3.2 *For process automaton $\mathcal{P}$ with alphabet Σ, a* path *of $\mathcal{P}$ is a finite sequence $c = (q_0, \sigma_1, q_1) \ldots (q_{k-1}, \sigma_k, q_k)$ of consecutive edges; the* length *of this path is k. Word $\gamma_1 \ldots \gamma_n$ of $\mathcal{U}^*$ is* spelled *by the path c if, by deleting from $\gamma_1 \ldots \gamma_n$ all symbols from outside the alphabet of $\mathcal{P}$, one obtains word $\sigma_1 \ldots \sigma_k$. Path c is a* run *if $q_0 \in I$.*

Example 3.4 For the process automaton in Figure 3.2, if $\mathcal{U} = \{e1,\ e2,\ ex,\ m1,\ m2,\ mx,\ a,\ b\}$ then the path $(q_0, e1, q_1)\ (q_1, mx, q_2)$ is a run and spells $e1\ b\ mx$ and $e1\ mx$, but not $e1\ e2\ mx$. □

Definition 3.3 *With each process automaton $\mathcal{P} = \langle \Sigma, Q, I, E, Q_{\mathrm{as}}, Q_{\mathrm{at}} \rangle$ we associate a pair $\mathbf{pr}\,\mathcal{P} = (X, Y)$ of subsets of $\mathcal{U}^*$, as follows. For every word $w \in \mathcal{U}^*$,*

1. *$w \in X$ if and only if there exists a run in $\mathcal{P}$ that spells w and ends in Q_{as}.*
2. *$w \in Y$ if and only if there are no runs in $\mathcal{P}$ that spell w and end in $Q\backslash Q_{\mathrm{at}}$.*

Proposition 3.1 *For every process automaton $\mathcal{P}$ over $\mathcal{U}$, $\mathbf{pr}\,\mathcal{P}$ is a process over $\mathcal{U}^*$.*

Proof Let $\mathcal{P}$, X, and Y be as in Definition 3.3. For arbitrary word $w \in \mathcal{U}^*$, let $\delta^*(I, w)$ be the set of end states of all runs spelling w. We have two cases for $\delta^*(I, w)$.

If $\delta^*(I, w) \cap Q_{\rm as} \neq \emptyset$, then, by Part 1 of Definition 3.3, we have $w \in X$.

If $\delta^*(I, w) \cap Q_{\rm as} = \emptyset$, then $\delta^*(I, w) \subseteq Q \backslash Q_{\rm as} \subseteq Q_{\rm at}$, by Part 4 of Definition 3.1. In this case, we have by Part 2 of Definition 3.3 that $w \in Y$. □

Note that Proposition 3.1 does not assume that the automaton is complete; that is, there might be no initial states, some states might not have successors by certain actions, and, for some w, $\delta^*(I, w)$ might be empty.

We normally work with deterministic process automata, defined below. Nondeterministic process automata arise, for instance, as a result of hiding internal actions, but in those cases we prefer to determinize them before performing any other operations.

Definition 3.4 *A process automaton is* deterministic *if it has exactly one initial state, and for each $q \in Q$ and $\sigma \in \Sigma$ there is exactly one edge of the form (q, σ, q').*

Example 3.5 The automaton in Figure 3.2 is deterministic. □

Note that, in a deterministic process automaton, for each word there exists exactly one run spelling that word. Also notice that Conditions 1 and 2 in Definition 3.3 are symmetric if the process automaton is deterministic: if exactly one run in $\mathcal{P}$ spells word w, the statement "there are no runs in $\mathcal{P}$ that spell w and end in $Q \backslash Q_{\rm at}$" is equivalent to "there exists a run in $\mathcal{P}$ that spells w and ends in $Q_{\rm at}$".

3.3 A BDD-Based Implementation

Our basic verification problems are to decide robustness for systems of finitely many processes, as in Lemma 2.19, or refinement between a single specification process and the product of a system of finitely many processes. By Theorem 2.12, this refinement problem can be reduced to the robustness problem above by taking the reflection of the specification process.

As in most verification problems, for checking robustness of a system of processes we have to deal with state explosion, that is, with the fact that the number of states of the product of n processes may grow exponentially with n in the worst case. We show below that, unless new algorithms are found that can solve the worst cases of PSPACE-hard problems efficiently, the problem of deciding robustness of the product of many process automata is intractable in the worst case.

Proposition 3.2 *Deciding robustness of the product of a variable number n of process automata is PSPACE-hard.*

Proof We use reduction from the "finite state automata intersection" problem, which is to decide emptiness of the intersection of the languages accepted by finitely many deterministic finite automata $M_1, \ldots, M_n$ that have the same alphabet Σ. The finite state automata intersection problem is known to be PSPACE-hard [GJ79, p. 266].[1] Here we have referred to the usual notion of finite automata which classify each word as accepted or rejected, whereas a process classifies each word as accessible or escape, as acceptable or reject, and as violation or goal.

Let L_i be the language accepted by M_i. We have that $\bigcap_{i\in\{1,\ldots,n\}} L_i$ is empty if and only if $\bigcap_{i\in\{2,\ldots,n\}} L_i \subseteq \overline{L_1}$, and thus, by Lemma 2.19, if and only if $(\Sigma^*, \overline{L_1}) \times (\times_{i\in\{2,\ldots,n\}}(L_i, \Sigma^*)) \in \mathcal{R}_{\Sigma^*}$.

By altering M_i, one can construct a process automaton for (L_i, Σ^*) by labeling the accepting states of M_i as accessible and by labeling all states of M_i as acceptable

[1] We thank J. O. Shallit for this reference.

in the resulting process automaton. Similarly, one obtains a process automaton for $(\Sigma^*, \overline{L_1})$ by labeling the non-accepting states of M_1 as acceptable and by labeling all states of M_1 as accessible in the resulting process automaton. □

Since the proof above used deterministic automata only, the problem of deciding robustness of the product of a variable number of deterministic process automata is also PSPACE-hard. A similar problem involving automata on infinite words is mentioned in [Kur94] to be PSPACE-complete.

The state explosion problem is partly alleviated by using a modular and hierarchical verification approach as discussed in Section 2.4. This way, one can break an overall refinement or robustness problem into subproblems, each involving a smaller number of automata, so that the subproblems are more tractable.

Another way to tame state explosion on typical cases is offered by BDDs. We have implemented various operations on process automata in a tool, called FIREMAPS, that uses a public-domain BDD library [BRB90] for the basic routines for manipulating large Boolean functions.

FIREMAPS has over one hundred operators, including all process space operations and conditions mentioned in Chapter 2, and also certain process maps like the projections and derivatives that will be discussed in Chapter 8. The main usage in the current applications is to verify robustness and refinement for systems of processes as described above. If the verified system is not robust, the tool can find a shortest reject trace and then simulate its effects on the processes in the system. These features are important for finding and diagnosing flaws. Also, FIREMAPS has capabilities for reading and writing processes as process automata, several operators for constructing frequently used process models for circuit components and delay constraints, and a front-end for reading and writing AND/IF (Andover interchange format) specifications (see [Ver]).

FIREMAPS has a script language which uses the prefix (a.k.a. Polish) notation for commands and expressions, where the operator comes first and then the operands; in turn, the operands can be other prefix expressions or constants. For instance, $a \wedge (\neg b \vee c)$ can be written $\wedge\ a \vee \neg\ b\ c$ in Polish notation. This way, parsing is made trivial and expressions have less clutter, because there is no need for parentheses. Also, very importantly, the users do not need to memorize operator precedences. The script language of FIREMAPS supports variable assignment and several types of data: process, list of processes, list of actions, integer, list of integers, and bit. (Condition operators, like robustness, return bits.) Using the prefix notation, we found it easy to extend the script language whenever new operators were needed.

The standard BDDs (used in FIREMAPS) express Boolean functions of the form $f : \{0,1\}^V \to \{0,1\}$, where V is a finite set, whose elements are referred to as Boolean variables. This formalization follows loosely the treatment of symbolic analysis in [BS95], except that for us it is important to know which variables are used in a particular Boolean function, not just to know how many arguments that function has; hence we refer to $\{0,1\}^V$ instead of $\{0,1\}^n$ for a number n. The elements of $\{0,1\}^V$ are called *valuations* for V and are functions of the form $x : V \to \{0,1\}$, assigning a Boolean value to each variable. A valuation can also be regarded as a binary vector with $|V|$ bits, where $|\cdot|$ denotes the cardinality of a set, except that at some points we refer to the elements of V, not just to the cardinality of V.

To manipulate automata by Boolean functions, one can encode the actions, states, and edges as binary vectors over some of the Boolean variables available, and represent the finite sets of actions, states, and edges by characteristic functions of sets of binary vectors. Given a finite set $\mathcal{D}$, we encode it by an injective function $\pi_{\mathcal{D}} : \mathcal{D} \to \{0,1\}^{W_{\mathcal{D}}}$, where $W_{\mathcal{D}} \subseteq V$. Under this encoding, the set $\mathcal{D}$ is represented by a function $\chi_{\mathcal{D}} : \{0,1\}^V \to \{0,1\}$ such that $\chi_{\mathcal{D}}(x) = 1$ if and only if the components of x corresponding to $W_{\mathcal{D}}$ are the code-word of an element of $\mathcal{D}$. That is, $\chi_{\mathcal{D}}(x) = 1$ if and only if $\exists\ d \in \mathcal{D} : x|_{W_{\mathcal{D}}} = \pi_{\mathcal{D}}(d)$, where $\cdot|.$ denotes restriction of a function to a sub-domain.

Notice that the function $\chi_{\mathcal{D}}$ depends only on the variables in $W_{\mathcal{D}}$. However, we defined $\chi_{\mathcal{D}}$ over all variables in V rather than just $W_{\mathcal{D}}$ for two reasons: to follow closely the BDD implementation, and because we need conjunctions or disjunctions of functions that may depend on different variable sets.

In FIREMAPS, all actions are encoded over a set $W_{\mathcal{U}}$ of Boolean variables. Alphabets are represented by the corresponding characteristic functions. For simplicity, we take the characteristic function of $\mathcal{U}$ to be constant 1.

The states of a process automaton $\langle \Sigma, Q, I, E, Q_{\mathrm{as}}, Q_{\mathrm{at}} \rangle$ are represented over a set W_Q of variables, where $2^{|W_Q|} \geq |Q|$ and W_Q is disjoint from $W_{\mathcal{U}}$. To each process automaton, we associate functions for Q, I, Q_{as}, and Q_{at}, all these functions being from $\{0,1\}^{W_Q}$ to $\{0,1\}$.

By inserting at every state self-loops on actions from outside the alphabet of a process automaton, the edge set of that process automaton is a subset of $Q \times \mathcal{U} \times Q$. The edges are represented over the variable set $W_Q \cup W_{\mathcal{U}} \cup W'_Q$ where the variables in W'_Q encode the next state, those in W_Q encode the current state, and those in $W_{\mathcal{U}}$ encode the action of an edge. The set W'_Q is disjoint from W_Q and $W_{\mathcal{U}}$, and there is a bijective function $s : W_Q \to W'_Q$ that defines the next state encoding as the function $\pi'_Q : Q \to \{0,1\}^{W'_Q}$ that satisfies $\forall q \in Q,\ \eta \in W'_Q : (\pi'_Q(q))(\eta) = (\pi_Q(q))(s^{-1}(\eta))$.

The product operation is implemented as follows. If $\mathcal{P}^1$ and $\mathcal{P}^2$ are the operands and $\mathcal{P}$ is the result, then:

1. $\chi_\Sigma = \chi_{\Sigma^1} \vee \chi_{\Sigma^2}$
2. $\forall \xi \in W_{Q^1} : s(\xi) = s^1(\xi)$ and $\forall \xi \in W_{Q^2} : s(\xi) = s^2(\xi)$
3. $\chi_Q = \chi_{Q^1} \wedge \chi_{Q^2}$
4. $\chi_I = \chi_{I^1} \wedge \chi_{I^2}$
5. $\chi_E = \chi_{E^1} \wedge \chi_{E^2}$
6. $\chi_{Q_{\mathrm{as}}} = \chi_{Q^1_{\mathrm{as}}} \wedge \chi_{Q^2_{\mathrm{as}}}$
7. $\chi_{Q_{\mathrm{at}}} = \chi_Q \wedge \neg((\chi_{Q^1_{\mathrm{as}}} \wedge \neg\chi_{Q^2_{\mathrm{at}}}) \vee (\neg\chi_{Q^1_{\mathrm{at}}} \wedge \chi_{Q^2_{\mathrm{as}}}))$

This implementation of product follows the definition closely, except that the sets of state variables of $\mathcal{P}^1$ and $\mathcal{P}^2$ are not necessarily disjoint. Sharing of state variables among different process automata can provide substantial savings for the BDD routines, while leaving the reachable part of the resulting state spaces unchanged. Sharing of state variables requires certain compatibility conditions among the process automata involved and among their Boolean function representations. We do not go deeper into this issue, but we note that the product automaton is compatible in this sense with the factors, justifying variable sharing between the product and each of the factors. The same holds for the result and the operand of reflection for deterministic process automata. The idea of sharing state variables is not new; see e.g. [NS95].

Reflection is implemented by simply swapping the functions for Q_{as} and Q_{at}. This operation requires the argument to be a deterministic process automaton.

For general process automata, we apply a determinization operation first. The current FIREMAPS implementation for determinization is much less efficient than the FIREMAPS implementations for product or robustness. For this reason, we try not to use non-deterministic automata in practical applications at all, although the current performance of determinization was sufficient to enable experimentation with the model. (As usual, arbiters and other non-deterministic systems can be modeled by deterministic automata by simply assigning different actions to the options of a non-deterministic choice; examples will be given in later chapters.)

Other binary process space operations are implemented similarly to the product, except for Steps 6 and 7, that have to be adjusted to correspond to the operation.

Robustness is verified by variations of the common reachability analysis algorithm, which determines whether at least one reject state is reachable from the initial states. Savings of several orders of magnitude are obtained if we eliminate from χ_F all transitions to permanent-escape states right at the start, that is, when we compute χ_F to states from which only escape states are reachable. Additional savings are obtained by other optimizations.

Dynamical variable reordering is known to produce more compact BDDs and thus to reduce the memory requirements of the routines, but in our experience this comes at large penalties in response time. For this reason, we are not using dynamical variable reordering.

For process automata $\mathcal{P}^1$ and $\mathcal{P}^2$, we check refinement by checking robustness of $-\mathcal{P}^1 \times \mathcal{P}^2$. Thus refinement is also checked by reachability analysis, and it requires $\mathcal{P}^1$ to be a deterministic process automaton.

With minor modifications, the algorithm above is used to find and return a reject execution for a process, if that process is not robust.

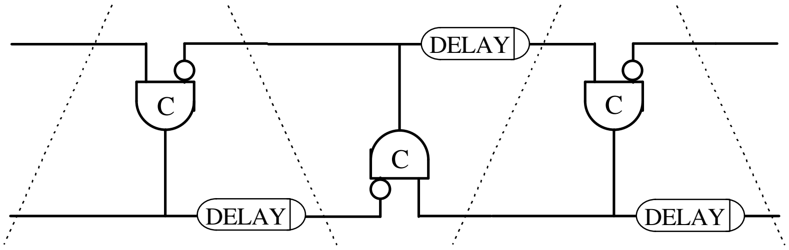


Figure 3.3: Basic micropipeline control circuit.

To evaluate the performance of FIREMAPS, we have used as a benchmark the basic micropipeline control circuit from [Sut89]. For details of the operation of this circuit, we refer the reader to [Sut89]; however, we show the schematics of three stages of this circuit in Figure 3.3 to help in discussing the complexity of the benchmark.

On this benchmark, we compare the results of FIREMAPS with those of VERDECT, a tool based on explicit state exploration. For a description of VERDECT and some non-benchmark demonstration circuits for that tool, see [EB95]. VERDECT was previously used to verify circuits related to this benchmark, in [Sid95] for instance.

Using FIREMAPS, we have checked refinement between the finalization process for an overall specification and the product of finalization processes given as models of the stages. Each stage was represented by a deterministic process automaton having 10 states. Our specifications were deterministic process automata having $4n + 6$ states, where n is the number of stages of the circuit.

The circuit passed our verifications when the numbers of stages of the specification and the implementation were the same. The number of states of the resulting system for a micropipeline control with n stages and the reflection of the specification process was $(4n + 6)10^n$, but only very few of these states were reachable.

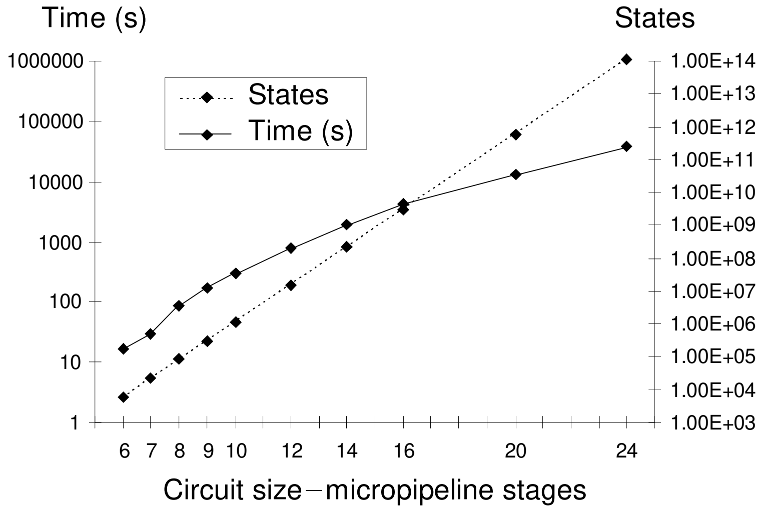


Figure 3.4: Performance plot for the micropipeline benchmark.

Still, the number of states that were actually reached grew exponentially with the number of stages, as shown by the straight line plotted with a dotted line on the logarithmic scale in Figure 3.4. For 9 stages or less, FIREMAPS and VERDECT reached precisely the same numbers of states. For 10 stages or more, we could not perform a comparison because VERDECT exhausted the memory of the workstation; FIREMAPS reached the RAM limit and started swapping only at 24 stages. For 6 and 7 stages, our run-times were larger than VERDECT's, but they were smaller than one minute. Our run-times were smaller than VERDECT's for 8 and 9 stages. The time taken by our tool for this circuit is indicated by a plain line in Figure 3.4, and for this case, it appears to be sub-exponential. These tasks were performed on a Sun Sparc 10/30 workstation with 96 Mbytes of RAM.

It may be objected that we should not compare verification methods for a circuit that is known to be correct and can be verified by hand by using algebraic techniques. However, for the sizes of the circuits that both VERDECT and FIREMAPS could handle, these tools reached the same numbers of states; this tells us that the comparison was done on similar state spaces, rather than taking shortcuts.

Nevertheless, in fairness to methods based on explicit state representations, it must be pointed out that BDDs are known to have exponential growth for some verification problems [Bry86], similar to the exponential growth of explicit state representations. Other techniques, such as those in [McM92], can be used in combination with BDDs to reduce further the computational effort for verification. We do not address the question of which problems may or may not produce exponential growth of BDDs.

Also, these benchmark verifications were brute-force; both VERDECT and FIREMAPS can deal with larger circuits by applying modular and hierarchical verification. In particular, modular and hierarchical verification works well for the micropipeline circuit; see [DNS92].

We have also had the opportunity to compare FIREMAPS against the results of another explicit state exploration tool used in [NS95]. This time we compared to the performance reported by [NS95] for a non-benchmark circuit with a less regular structure. FIREMAPS performed better by orders of magnitude; the details are given in Chapter 5.

As a sanity check for our verifications of the micropipeline control circuit, we also have altered the specification process to correspond to a different number of stages than actually present in the implementation, and in those situations we have obtained counter-example traces, as expected.

Our verifications of the micropipeline control circuit from [Sut89] did not include relative timing for the "DELAY" elements; on the other hand, we have verified other micropipeline control circuits under relative timing assumptions [MJCL97].

Chapter 4

Relative Delay Assumptions

A way to reduce the area or increase the performance of asynchronous circuits is to make timing assumptions. Such assumptions often boil down to imposing that, of two circuit paths that start at the same point, one path is faster than the other. Such assumptions are used for instance in circuits from [BHP95], [MJCL97], [PDF$^+$98], and [Pee96]. We call speed-dependences of this form *chain constraints*.

To verify a circuit that relies on chain constraints, one might (i) size the component delays to meet the delay constraints and then verify the circuit with known bounds on component delays, or (ii) verify the circuit solely on the basis of given delay constraints, then size the component delay bounds to meet the delay constraints. Precise bounds on individual component delays can be calculated only after knowing transistor sizes and wire lengths; for this reason, let us call these approaches *post-layout* and *pre-layout* verification, respectively. Pre-layout verification can speed up the design by avoiding optimizing delays in an incorrect circuit, by allowing migration to another implementation technology for a verified circuit, and by keeping the verification simple because a minimum of information is used.

In this chapter, we discuss the basics of asynchronous circuit modeling by means of finalization processes. Then, we show how to incorporate chain constraints in the analysis by formalizing them as finalization processes as well. Chain constraint processes can be coupled and compared to other processes representing circuit specifications or components, leading to a flexible technique that is applicable before layout.

Previous timing analysis methods for asynchronous circuits such as [ACD+92], [AD94], [CDYC97], [LMBSV92], [MD92], [Mye95], or [RM94] allow for a more detailed analysis than we do here, but they require numerical bounds on individual delays, whereas in our case studies such bounds are not even available. Also, the absence of numeric information in our finalization models can simplify the analysis, making it more tractable and more suitable for BDD algorithms. On the other hand, we do not address performance concerns such as the response time and the throughput of a circuit.

Other approaches for non-numeric analysis of circuits under relative delay constraints have been proposed in [BY75] and [BY76], but their delay constraints are always of the form that the delay of one gate is less than the sum of the delays of two other gates. In our problems, however, we encounter more diverse constraints and we try to avoid imposing unnecessary constraints; this requires a more versatile technique. In addition, we can use algebraic properties from the underlying formalism, such as the properties for modular and hierarchical verification.

We show on case studies how to use our technique to verify single-rail handshake circuits from [Pee96], a class of asynchronous circuits with provable industrial viability. We also report our experience with the following problems, which are encountered in these case studies:

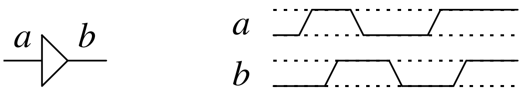
- significance of hazards (are they true faults? can they be ignored?);
- models for the behavior of certain CMOS cells in the presence of hazards;
- comparisons to Spice-like, analog simulations of dynamical system circuit models;
- extraction of chain constraints from timing assumption of the type described in [BHP95], called extended isochronic forks;
- interpretations of delay constraints on analog waveforms;
- importance of uncertainty in the switching thresholds of CMOS cells ('threshold spreads') for the implementation of chain constraints;
- assessment of tolerance of delay fluctuations for an asynchronous circuit that uses chain constraints.

Also see [NP98], [Neg97a], and [Neg97b]; this chapter is based on [NP98].

4.1 Cell Models

To model digital circuits in an asynchronous, event-oriented manner, we represent their waveforms by traces. Events are associated to signal transitions in the circuit. We do not make a formal distinction between rising and falling transitions, which are distinguished anyway by their effects on the state of the circuit.

Example 4.1 The waveform in Figure 4.1 (b) may be produced by a buffer component, shown in Figure 4.1 (a), which repeats at the output the logical level of the input signal. This waveform is represented by trace *ababab*. □



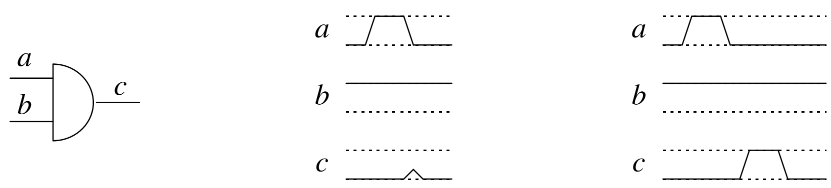
(a) (b)

Figure 4.1: Waveform of a buffer.

In this section we discuss process space models of Boolean gates and some other circuit components, such as C-elements. (The basic C-element has two inputs and one output. If the inputs signal have the same logical value, the output reproduces that value; otherwise, the output is constant. Thus, C-elements are not combinational gates.) Such digital components can be modeled by finite state machines; however, there are some subtleties, because the behavior of a gate is not fully standardized in the presence of hazards.

We are mostly interested in components that have a single output and whose behavior is described by a Boolean function of the inputs and the output; we call this function the *instability function*. The instability function specifies when the output value does not correspond to the input values, and thus the output signal is about to undergo a transition. We refer to all such components as CMOS cells. For example, consider an AND gate having inputs a, b, and c, and output d. The instability function is $(a \wedge b \wedge c) \oplus d$, where $\oplus$ is exclusive-or. (Although we use the same sign for exclusive sum and exclusive-or, it will be clear from the context which of these operators is used.) Noting that exclusive-or is the negation of equivalence, we see that the AND is unstable whenever d is different from the conjunction of a, b, and c. For another example, consider a C-element with inputs a and b and output c. The instability function is $(a \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge \overline{b} \wedge c)$. In words, the C-element is unstable whenever a and b are high and c is low, or a and b are low and c is high.

In general, a CMOS cell that has output b and realizes the Boolean function $F(a_1, a_2, ...)$ has the instability function $b \oplus F(a_1, a_2, ...)$, and a CMOS cell that realizes Martin's production rules [Mar90] $U(a_1, a_2, ...) \to b\uparrow$ and $D(a_1, a_2, ...) \to b\downarrow$ has the instability function $(U(a_1, a_2, ...) \wedge \overline{b}) \vee (D(a_1, a_2, ...) \wedge b)$.



(a) (b) (c)

Figure 4.2: Responses to a short pulse: (a) gate and signals; (b) incomplete transition; (c) delayed pulse.

The behavior of a CMOS cell is to a large extent standardized, but still there are some situations where the behavior depends on the particular implementation. For example, consider the AND gate in Figure 4.2 (a). Suppose input b is high and input a is low. Suppose a rises and then falls again before the output gets a chance to rise. What happens next? Depending on the implementation and timing, the output signal might stay low, as in Figure 4.2 (b), or it might have a delayed pulse, as in Figure 4.2 (c). CMOS cell models where an output transition is canceled upon retraction of the input excitation, as in Figure 4.2 (b), are called *inertial* models. This matches the traditional definition of inertial models (see e.g. [BS95]) except for not specifying numerical bounds on the delay of the inertial cell. Also, models where the gate can 'store' up to k output transitions and deliver them after the input excitation has been retracted, as in Figure 4.2 (c) for $k \geq 2$, are called *k-bounded* [LMBSV92].

Let us refer to situations where a CMOS cell becomes unstable and then stable again before completing an output transition, as in Figure 4.2, as *hazards*. The usual dynamic and static hazards (see e.g. [BS95]) are subcases of this definition.

We use finalization processes to specify the behaviors of CMOS cells of the types described above. Since several behaviors of a cell may be expected in a hazard situation, we need to know whether hazards may occur and, if so, for which cells. For this purpose, one can simply forbid hazards at first by using cell specifications that declare hazards illegal. We call such CMOS cell specifications *hazard-intolerant* models. Then, any hazards will be detected as violations of the cell specifications. Should any hazards occur, one can use more detailed specifications to model the particular behaviors of the cells exposed to hazards.

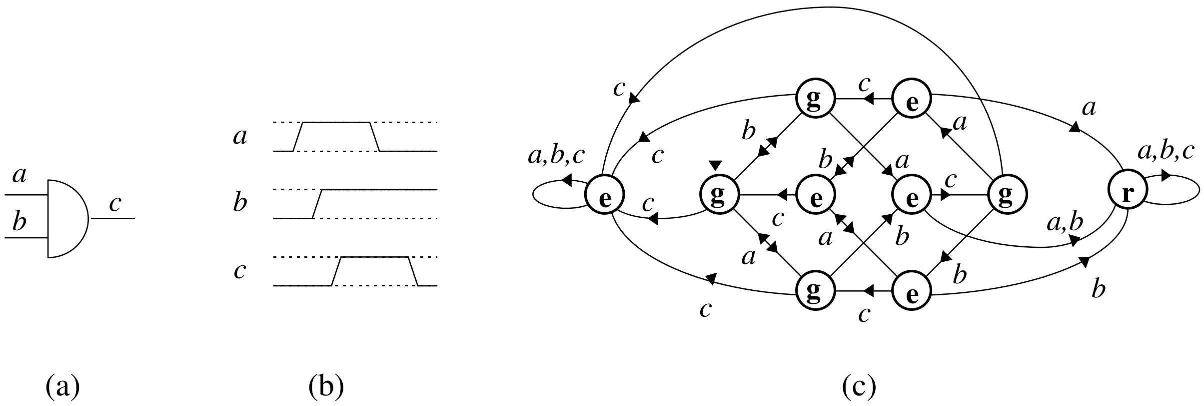


Figure 4.3: Hazard-intolerant model: (a) gate and signals; (b) execution *abcac*; (c) process.

For an example of a finalization process for a hazard-intolerant model, consider the AND gate in Figure 4.3 (a). The corresponding process automaton is shown as a state machine in Figure 4.3 (c). The state machine has ten states: eight states for the eight possible combinations of logic levels of the signals of the gate, one permanent-escape state for executions containing an illegal output, and one permanent-reject state for executions containing an illegal input. Unless otherwise specified, we assume that in the initial state all signals are low. Then, in the neighboring state after an a transition, the signal values become a high, b low, c low; and so on. For instance, execution $abcac$ is a goal because it leads to a state where a and c are low and b is high. Execution $abcb$ is an escape, since the gate should continue by issuing another c event. Infinite executions like $abcbbb\ldots$ are beyond the scope of finalization processes, which classify finite traces only; infinite traces will be discussed in Chapter 6.

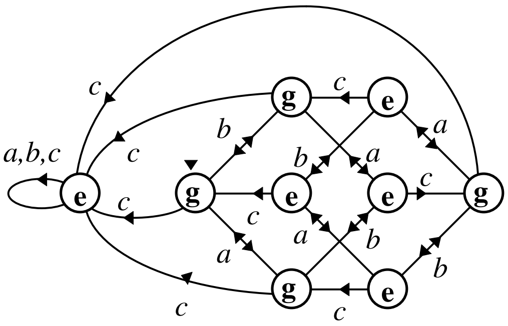


Figure 4.4: Process for the inertial model of the gate in Figure 4.3 (a).

For an example of a finalization process for an inertial model, consider the process in Figure 4.4, representing the inertial model of the AND gate in Figure 4.3 (a). This process is similar to the hazard-intolerant model, except that input transitions from an unstable state to a stable state are permitted. For instance, execution abb represents a hazard as described above, but it leads to a goal state of the inertial model in Figure 4.4. This amounts to assuming that hazards are ignored, and so are runt pulses at the input. One cannot rely on this assumption in general, but this assumption does hold for certain gates, depending on their layout parameters. Inertial models have to be validated by other means, such as analog simulations for the individual gates involved.

The hazard-intolerant models extend to arbitrary CMOS cells the asynchronous models used e.g. in [Dil89] for Boolean gates, while the inertial models correspond to the widely known inertial delay models described e.g. in [BS95].

FIREMAPS provides operators for building CMOS cell models directly from the instability functions of the cells. Currently there are operators only for hazard-intolerant and inertial models, but one can easily build k-bounded models for even k by attaching buffers at the output of a cell. Ideal-delay models, which can store arbitrarily many output transitions, are not finite-state and are not supported.

4.2 Chain Constraints

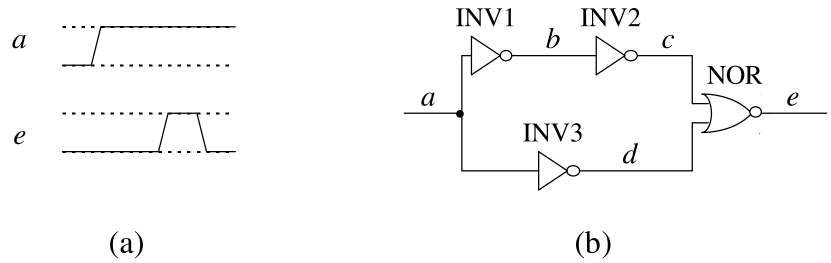


Figure 4.5: Is the waveform (a) realized by the circuit (b)?

Consider the problem of designing a circuit that responds to a rising input transition by issuing an output pulse, as in Figure 4.5 (a). A straightforward implementation would be that in Figure 4.5 (b): from an initial stable state where a is low, the rising transition on a causes falling transitions on b and d, then e rises, after which c rises and e falls.

If INV1 and INV2 are quicker than INV3 and NOR, it is possible that the output pulse will never occur or will be corrupted, since c might rise and pull e down before e completes the upgoing transition. To prevent this problem, one may introduce

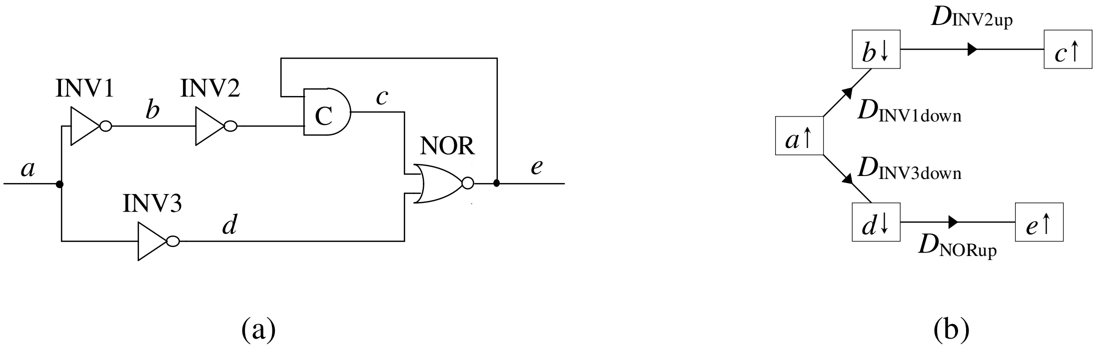


Figure 4.6: Ensuring correct behavior: (a) by extra circuitry; (b) by relative delays.

extra circuitry (a C-element), as in Figure 4.6 (a). However, the extra circuitry increases the area, delay, and power consumption of the circuit. Fortunately, a cheaper and more efficient solution is available. One can size the component delays in such a way that

$$D_{a\uparrow b\downarrow} + D_{b\downarrow c\uparrow} > D_{a\uparrow d\downarrow} + D_{d\downarrow e\uparrow} \tag{4.1}$$

where D's represent gate delays, and $\uparrow$ and $\downarrow$ refer to rising or falling transitions on output signals. The two chains of delays involved are shown in Figure 4.6 (b).

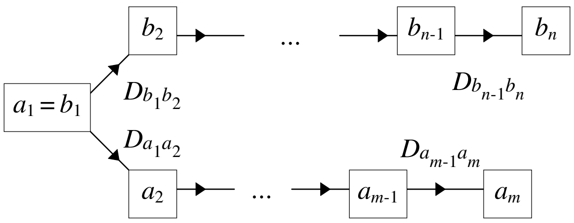


Figure 4.7: General form of competing delay chains.

At this point we can define *chain constraints* more precisely, as relative delay assumptions of the form

$$D_{b_1b_2} + \cdots + D_{b_{n-1}b_n} > D_{a_1a_2} + \cdots + D_{a_{m-1}a_m} \tag{4.2}$$

that ensure that a_m will occur before b_n, where $a_1, \ldots, a_m, b_1, \ldots, b_n$ are events such that a_1 is the same as b_1, and the D's are delays between events; see Figure 4.7. Instead of the expression in equation (4.2), we often use the shorthand

$$D_{b_1 \ldots b_n} > D_{a_1 \ldots a_m} \; . \tag{4.3}$$

Such delay inequalities are conditional upon the actual occurrence of the chains of events involved. In many systems, however, it is possible that the chains are disrupted by events occurring out of order, and that the chain that is supposed to take a shorter time actually stops midway. Thus, an analysis of chain constraints based on linear programming may encounter difficulties regarding deadlock and non-determinism; instead of taking that approach, we model chain constraints by processes like any other components in a discrete-state system. Also, a constraint of the form (4.2) is in general different from the simple delay inequality $D_{b_1 b_n} > D_{a_1 a_m}$; we shall discuss this point on an example at the end of this section. This further motivates the handling of chain constraints by means of processes rather than linear programming.

We propose a technique for verifying whether a given set of chain constraints ensures correctness of a circuit. Additionally, this technique can assist in finding sufficient chain constraints for a given circuit and specification, by indicating potential flaws and executions where the flaws are manifest.

For example, consider again the circuit in Figure 4.5 (b). The pulse specification in Figure 4.5 (a) is represented by the finite-state process in Figure 4.8. From the initial state, a is supposed to switch, then e, and then e again. Illegal inputs lead to a reject state, and illegal outputs to an escape state. To simplify this example, we have chosen to disallow cyclic operation of the circuit and specifying only a one-time use, by making a second a event illegal; in general, our technique does

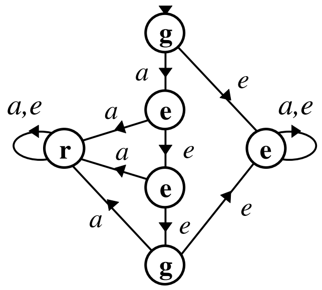


Figure 4.8: Specification process.

apply for cyclic operation as well, as demonstrated by the case studies later in this chapter. Stopping before two e transitions is also illegal, and the corresponding states are escape states. Stopping is legal after two e transitions, and before the first a transition. Note that the circuit does not actually have to be used: the environment is allowed to stay in the initial state and never do anything. Events other than a and e are ignored by this interface specification, meaning that they do not change the state of the process. The problem is to determine whether the specification process is refined by the product of the component processes. In this case, the component processes are hazard-intolerant models of the behaviors of the cells in Figure 4.5 (b), which are mechanically extracted from standard cell descriptions as discussed in Section 4.1.

FIREMAPS reported that refinement does not hold and produced counter-example execution *abcd*, indicating a hazard at the NOR gate. This execution shows that the circuit might end up in a stable state where signals a and c are high and signals b, d, and e are low, while an e transition is expected. Thus, in absence of delay constraints, the circuit might not behave as specified.

To deal with the flaw without introducing new cells, we impose constraint (4.1). To verify the circuit with the constraint, we model constraint (4.1) as a process that forbids certain executions by declaring them to be escapes. It is the responsibility of an imaginary constraint device to avoid executions such as *abcd*, *abdc*, *abd*. FIREMAPS reported that refinement holds between the specification process in Figure 4.8 and the product of the new constraint process and the cell processes.

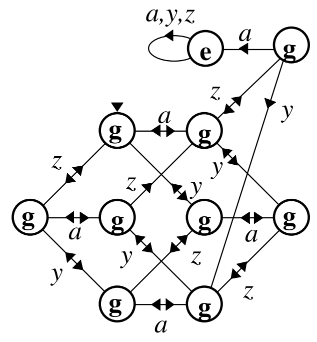


Figure 4.9: Process for chain constraint $D_{z\uparrow a\downarrow} > D_{z\uparrow y\downarrow}$, assuming a and z start low and y starts high.

To illustrate the chain constraint processes, we use a chain constraint which is simpler than that in (4.1) and that actually arose in one of our case studies. Consider the process automaton in Figure 4.9 representing the (rather simplistic) chain constraint $D_{z\uparrow a\downarrow} > D_{z\uparrow y\downarrow}$ under the assumption that, in the initial state, a and z are low and y is high. (In applications, we often change the initial states of process automata, to correspond to the actual initial values of signals in a circuit.) Here the direction of signal transitions is meaningful, indicating that the constraint applies from a state where z is low and a and y are high. This state, called the *base state* of the constraint, needs not be the initial state of the system; in Figure 4.9, the base state is reached by a from the initial state. A cube-like set of states keeps track of the levels of the three signals involved. Execution $a\uparrow z\uparrow a\downarrow z\downarrow$ violates the constraint and is declared to be an escape, thereby making it the responsibility of an imaginary constraint device to avoid such executions. (In execution $a\uparrow z\uparrow a\downarrow z\downarrow$, the event $a\downarrow$ is illegal because a $y\downarrow$ should have occurred before $a\downarrow$.) Executions such as $a\uparrow z\uparrow y\downarrow$, which obey the constraint, or $a\uparrow z\uparrow z\downarrow$, which disrupt the chains (meaning that the ordering of the events in the execution does not match the orderings in the chains), are not restricted by the constraint process, which classifies them as goals.

The constraint process is built automatically by FIREMAPS from the lists of actions involved, a trace leading to the base state where the race starts, and the two competing delay chains. For the chain constraint in (4.1), the actions involved in the constraint are a, b, c, d, and e; the competing chains start at the initial state; the chain with the longer delay is abc; and, the chain with the shorter delay is ade. For the chain constraint in Figure 4.9, the actions involved in the constraint are a, y, and z, the base state is reached by a, and the chains are $z\ a$ and $z\ y$.

As we have mentioned, we do not deal with chain constraints as linear inequalities. The question arises whether one can simply sum up the delays in a chain constraint and use a simpler constraint. For instance, would the chain constraint

$$D_{a\uparrow c\uparrow} > D_{a\uparrow e\uparrow} \tag{4.4}$$

have the same effect on the circuit in Figure 4.5 (b) as the constraint (4.1)? The answer is no, because chain constraint (4.4) turns out to be too strong, as we now show.

We used FIREMAPS to find counter-examples to refinement between the circuit with chain constraint (4.4) and the circuit with chain constraint (4.1). The tool has produced execution $abaadc$, which is allowed by (4.1), but is inappropriately ruled out by (4.4), for the following reasons. Notice that the first a in $abaadc$ matches both the a in (4.1) and the a in (4.4). However, the chain abc in (4.1) is disrupted by the second a of $abaadc$; thus, the constraint in (4.1) does not apply to $abaadc$.

On the other hand, the constraint in (4.4) does rule out *abaadc*, because the third a in *abaadc* also matches the a in (4.4). Thus, (4.4) should not be used instead of (4.1).

In general, one must be careful when deciding to simplify a chain constraint, because a simpler delay inequality such as (4.4) may inappropriately guarantee to avoid some potentially dangerous executions, and thus may mask bugs if it is used instead of the more detailed constraint. Nevertheless, one may use a simplified constraint if one checks that a subcircuit with a simplified constraint is equivalent to that subcircuit with the original constraint, in the sense of double refinement.

4.2.1 Extended Isochronic Fork Assumptions

An extended isochronic fork assumption of depth n [BHP95] involves the delays in two paths that start at the same circuit node and go through n gates each. In a first approximation, let us ignore wire delays and rise and fall times. An extended isochronic fork assumption can be symmetric or asymmetric.

An asymmetric fork assumes that one of the paths always takes longer than the other. This amounts to an inequality of the form:

$$D_1 + \cdots + D_n > D'_1 + \cdots + D'_n \tag{4.5}$$

where $|\cdot|$ denotes the absolute value, and $D_1, \ldots, D_n$ and $D'_1, \ldots, D'_n$ are the gate delays in the two paths. To put (4.5) into a chain constraint form, we need to translate the gate delays into delays between events. This can be done in several ways, depending on whether events represent rising or falling signal transitions, thus producing chain constraints with different base states. Furthermore, one might insert extra delays in the right-hand sides to constrain events that are not consecutive;

such an example was encountered in [Neg97a]. This might seem complicated, but usually it is quite clear which chain constraints need to be selected, by examining the counter-example executions that are produced by the tool when checking the circuit *without* the respective constraints. Moreover, in most cases, just a few of the constituent chain constraints of each fork suffice to guarantee correctness, and we can ignore the rest of them, facilitating both the analysis and the implementation of the circuit.

A symmetric extended isochronic fork assumes that the difference of the delays of the two circuit paths involved is smaller than the delay of any gate that has at least one input connected at the end of either path. A symmetric extended isochronic fork assumption can be split into inequalities of the form:

$$|(D_1 + \cdots + D_n) - (D'_1 + \cdots + D'_n)| < D'' \tag{4.6}$$

where $D_1, \ldots, D_n$ and $D'_1, \ldots, D'_n$ are the gate delays in the two paths, and D'' is the delay of any gate that has at least one input connected at the end of one of the paths. Further, inequality (4.6) splits into:

$$D_1 + \cdots + D_n < D'_1 + \cdots + D'_n + D'' \tag{4.7}$$

$$\text{and} \quad D'_1 + \cdots + D'_n < D_1 + \cdots + D_n + D'' \tag{4.8}$$

One can extract chain constrains from inequalities (4.7) and (4.8), just like from an asymmetric extended isochronic fork.

One way to handle the genuine isochronic forks from [Mar90] is as depth-one extended isochronic forks, modeling the branch delays by inertial buffers.

4.2.2 Thresholds and Constraint Interpretations

Chain constraints were defined in Subsection 4.2 under a discrete-state assumption, but, to implement them in a circuit, one needs to translate them into constraints on the dynamical-system parameters of the circuit. Aiming for a safe implementation, we propose a translation which is always pessimistic.

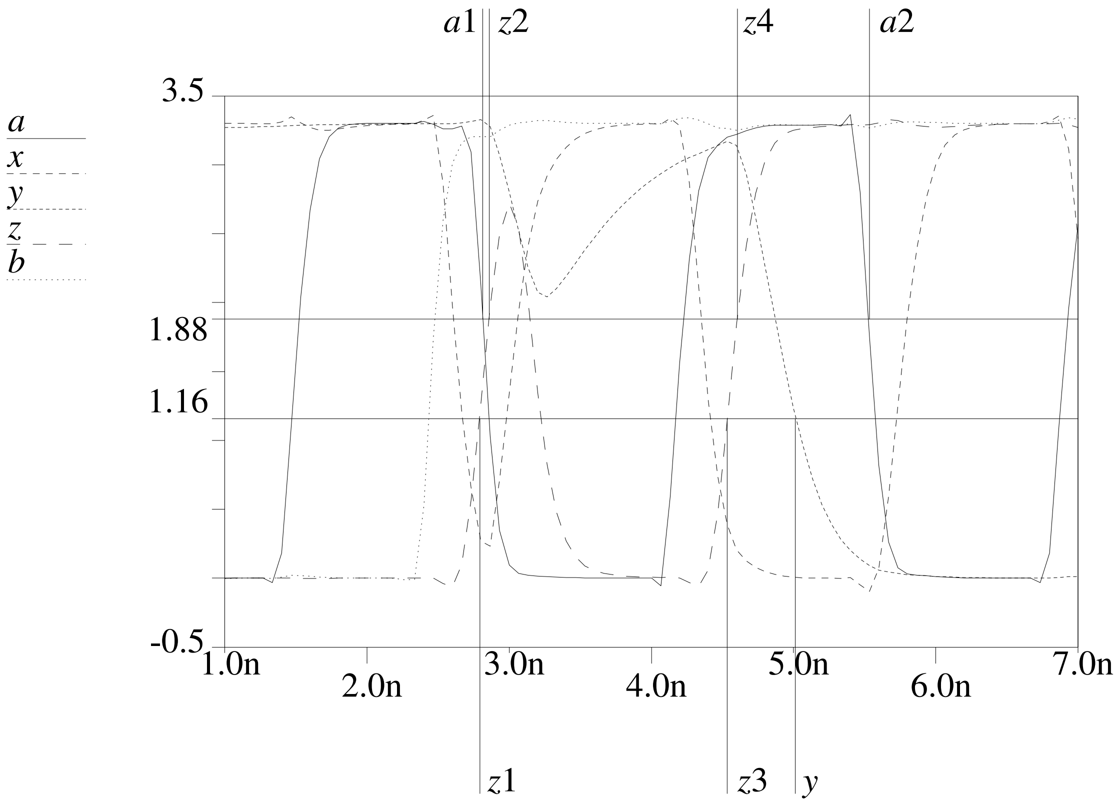


Figure 4.10: Meaning of delay constraint $D_{z\uparrow a\downarrow} > D_{z\uparrow y\downarrow}$, and simulation of a failure mode prevented by this constraint.

Links between the discrete-state and dynamical system viewpoints have been extensively studied, for instance, in [GC94] and [KM91]; however, we need to discuss the issue from a different perspective. The intention of [GC94] is to rely solely on topological features of the phase spaces, whereas, to define delays, we associate events to crossings of voltage thresholds. In [KM91] the phase spaces are partitioned into fixed boxes corresponding to discrete states, whereas we need to allow some uncertainty for the thresholds. We sometimes associate more than one discrete execution to each analog trajectory; for example, the waveform in Figure 4.10 can be represented by either one of the executions

$$a{\uparrow}b{\uparrow}x{\downarrow}z{\uparrow}a{\downarrow}x{\uparrow}z{\downarrow}a{\uparrow}x{\downarrow}z{\uparrow}y{\downarrow}a{\downarrow}x{\uparrow}a{\uparrow} \quad \text{or}$$
$$a{\uparrow}b{\uparrow}x{\downarrow}a{\downarrow}z{\uparrow}x{\uparrow}z{\downarrow}a{\uparrow}x{\downarrow}z{\uparrow}y{\downarrow}a{\downarrow}x{\uparrow}a{\uparrow}$$

since the first $a{\downarrow}$ and $z{\uparrow}$ transitions cross their thresholds concurrently.

cell	a	b	c
INV	1.55 V		
NOR2	1.61 V	1.73 V	
NOR3	1.60 V	1.68 V	1.78 V
NAND2	1.46 V	1.36 V	
NAND3	1.40 V	1.35 V	1.26 V

Table 4.1: Simulated logic thresholds, per input, for cells from a 0.5μ CMOS library, operated at 3.3 V.

We consider two logical thresholds, T_{low} and T_{high}, that enclose the range of gate thresholds that are physically possible, and maybe some extra margins to account for fluctuations. Due to dependences on transistor widths and other effects, the physical thresholds for a given technology differ among gates and even among inputs of the same gate. As in [Ber92], we define physical thresholds by matching saturation currents through the pull-down and pull-up networks of a cell. Some other definitions of physical thresholds can also be accommodated in our method, as long as they allow for the variation in threshold values. (However, a definition based on unit gain points would make the threshold ranges unnecessarily wide.) In [NP98], such threshold values were determined by analog simulations, and we list them in Table 4.1. As in [NP98], we use a logical threshold range 1.16–1.88 V for the circuits we analyze, to cover the simulated threshold values plus and minus a margin of 0.1 V, chosen arbitrarily.

The purpose of a chain constraint of the form (4.2) is to ensure that event (signal transition) a_m occurs before b_n. We take this to mean that the signal of a_m *leaves* the logical threshold range before the signal of b_n *enters* the range.

The meaning of a chain constraint $D_{z\uparrow a\downarrow} > D_{z\uparrow y\downarrow}$ is illustrated in Figure 4.10. This constraint is applied twice, corresponding to two different ways of matching events $z\uparrow$, $a\downarrow$, and $y\downarrow$ with the actual waveform. This constraint is first interpreted as $D_{za_1} > D_{zy}$, where z can be a point on the time axis situated between z_1 and z_2. In this first interpretation, the constraint is clearly violated. A second interpretation is $D_{z'a_2} > D_{z'y}$, where z' is a point between z_3 and z_4; here, the constraint is obeyed.

A consequence of the proposed interpretations for chain constraints is that, if a wider threshold range is used, the chain constraints become stricter and harder to implement. This observation agrees with the concern expressed in [BHP95] and [Ber92] for reducing the possible variation of the thresholds. In our method, however, threshold spreads are not crucial. If the chain constraints themselves are not very tight (if their safety margin is high—see Subsection 4.2.3) then we might afford large threshold spreads. Note however that this observation applies to the implementation of chain constraints and affects their verification only indirectly, by guiding the choice of the constraints we care to verify.

4.2.3 Safety Margins

To estimate how hard it is to implement chain constraints, one can use the safety margin measure from [KN94], which evaluates the timing reliability of a circuit in the presence of delay fluctuations. This is a side issue that does not directly regard verification, but provides a way to use the results of our technique in a practical setting.

Given a constraint of the form (4.2), its safety margin can be defined as the ratio i/j, where i and j are the numbers of inversions on the greater and smaller sides of the inequality sign, respectively. Note that we count CMOS inversions rather than gates. A larger ratio grants more leeway for the delays involved, presumably making the constraints easier to implement. Larger i and j with a constant $i - j$ difference produce higher ratios and thus constraints that may be harder to implement. As in [KN94], the safety margin of a set of chain constraints is the *smallest* of the safety margins of the chain constraints in the set. In general, a circuit may admit multiple sets of chain constraints so that each set of constraints alone suffices to guarantee correctness. The safety margin of a circuit can be defined as the *largest* of the safety margins of these sets.

The number of CMOS inversions is an oversimplification of the delay of a gate, although it does serve as a first-order approximation. More detailed models can be used as well, for instance models that take the capacitive load into account as in [SS91]; such extensions, however, are not pursued further in this thesis.

4.2.4 Case Studies: Handshake Circuits

In [Pee96], the implementation of handshake circuits [Ber93] in a generic standard-cell library (containing no dedicated asynchronous cells like C-elements) is described. The time dependences in these circuit implementations generally take the form of chain constraints. The analysis technique described above is applied to specify these timing assumptions precisely and to verify them.

C-Element with Delay Constraints

Figure 4.11 shows a circuit from [Pee96] that implements a three-input C-element in a speed-dependent manner. The '==' sign denotes an extended isochronic fork of depth one, as defined in [BHP95], and an extra gate with input z is assumed to intervene in the environment delay from z to each of a, b, and c. Following [Pee96, p. 70], this fork assumption is taken to be asymmetric.

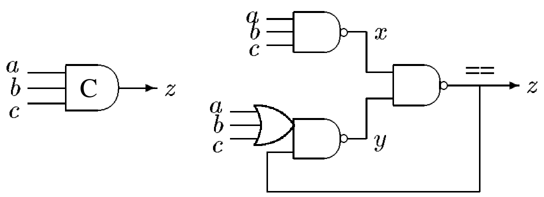


Figure 4.11: Symbol and implementation for a three-input C-element[Pee96, p. 70].

First, we check whether the circuit satisfies its specification in the absence of delay assumptions. The specification is the hazard-intolerant model of a three-input C-element. The circuit is modeled as the product of hazard-intolerant models of the circuit gates (extracted from standard cell descriptions as discussed in Section 4.1) assuming that a, b, c, and z are initially low and x and y are initially high. To check whether the circuit satisfies the specification without delay assumptions, we verify by FIREMAPS the refinement relationship between the specification process and the circuit model. The answer is that refinement does not hold, and the counter-example that is produced is execution $c\uparrow b\uparrow a\uparrow x\downarrow z\uparrow a\downarrow a\uparrow y\downarrow$. This execution is a reject for the NAND with output x, and a goal for all other gate processes and the specification process. The smallest reject prefix is $c\uparrow b\uparrow a\uparrow x\downarrow z\uparrow a\downarrow a\uparrow$, indicating that a hazard may occur at the NAND, although it is not a hazard for the specification. To see whether this hazard may produce a fault that is externally visible, we replace the circuit gate models by inertial models, and we obtain $c\uparrow b\uparrow a\uparrow x\downarrow z\uparrow a\downarrow x\uparrow z\downarrow$, which is an escape for the specification and a goal for the circuit gates. This execution contains an illegal $z\downarrow$ transition.

An execution similar to the counter-example to refinement above was obtained by the analog simulation in Figure 4.10, confirming that the fault detected by the discrete-state analysis without delay constraints is realistic for the actual circuit.

To avoid the danger above, [Pee96] uses the delay assumption of an asymmetric extended isochronic fork (denoted by '==' in Figure 4.11). To avoid the illegal $z\downarrow$ in $c\uparrow b\uparrow a\uparrow x\downarrow z\uparrow a\downarrow x\uparrow z\downarrow$, we only need to ensure that after $z\uparrow$ the internal signal y has enough time to stabilize before the next input transition. Thus we only impose the following chain constraints, which derive from the fork assumption for the delay introduced by the extra environment gate mentioned above:

$$D_{z\uparrow a\downarrow} > D_{z\uparrow y\downarrow} \tag{4.9}$$

$$D_{z\uparrow b\downarrow} > D_{z\uparrow y\downarrow} \tag{4.10}$$

$$D_{z\uparrow c\downarrow} > D_{z\uparrow y\downarrow} \tag{4.11}$$

Chain constraint (4.9) was illustrated in Figure 4.9 and Figure 4.10. Actions other than a, y, and z do not change the state of the chain constraint process and are omitted from Figure 4.9. This chain constraint guarantees avoidance of the execution $cbaxzaxz$ above by making it an escape of its process.

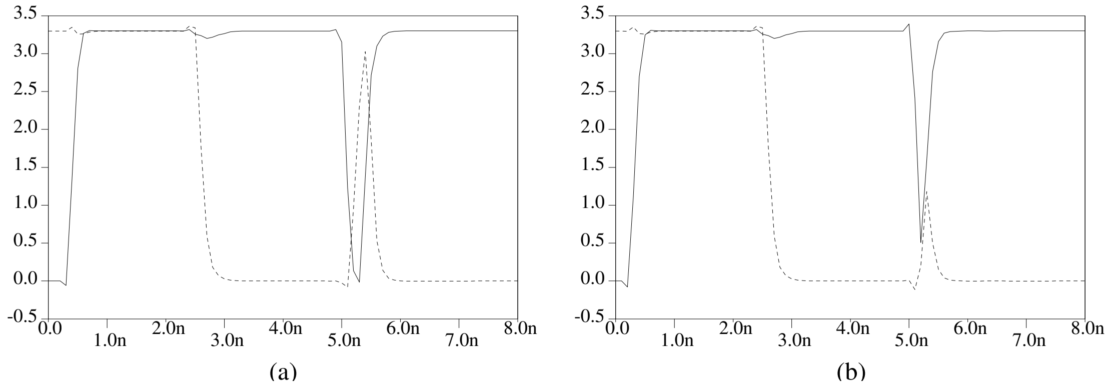


Figure 4.12: Simulated response to an input pulse, for a NAND gate with output x (dotted line) and negative pulses of varying lengths on input a (solid line): (a) an output pulse is produced, but it overlaps the input pulse; (b) the output spike becomes too small.

Still, the product of chain constraint processes and the hazard-intolerant processes of the circuit gates do not refine the specification, a counter-example execution being $c\!\uparrow\! b\!\uparrow\! a\!\uparrow\! x\!\downarrow\! z\!\uparrow\! y\!\downarrow\! a\!\downarrow\! a\!\uparrow$. This execution shows a hazard caused by $a\!\uparrow$ at the NAND gate that drives x, although this behavior is legal for the environment. We need to determine whether this hazard can lead to a 'true' fault (externally visible), or whether it can be ignored.

Under an inertial model for this NAND gate and hazard-intolerant models for all other gates, the circuit with the chain constraints above does satisfy the hazard-intolerant three-input C-element specification. But is an inertial model appropriate for this gate?

Since the NAND gate that drives x is an inverting gate and its load is not abnormally small, one may expect that an inertial model will be appropriate for this gate. The verification procedure can exhaustively compare the discrete-state models of the components and the specification, but other methods have to be employed to determine whether these discrete-state models properly describe the actual, physical behavior of the system. To validate the model for this NAND gate, we have simulated negative pulses of varying durations on input signal a, while keeping b and c high. We have made the input transitions abrupt, to make it easier for the output pulse to be delayed beyond the input pulse and expose non-inertial behavior, if any. The gate parameters and the load had typical values currently used for this circuit. (This load was the capacitance of an internal node.) The response of the gate was indeed inertial: either the output pulse overlapped the input pulse, as in Figure 4.12 (a), or the output spike was too small to be perceived as a pulse by other gates, as in Figure 4.12 (b). That is, no output pulses were started after the input pulse vanished. In other words, if the input pulse is too short, it will be ignored rather than causing a delayed output pulse; these simulations agree with the inertial assumption for this gate.

Multiplexer Control

Figure 4.13 shows a multiplexer component taken from [BHP95, Pee96]. The environment of this component is assumed to guarantee mutual exclusion between handshakes on two handshake channels a and b that transmit data and control signals. This component will forward incoming data from a or b along c and, when acknowledged through c, send an acknowledgement along a or b, depending on the origin of the data. This component thus multiplexes data from a and b onto channel c.

In the single-rail implementation of this component, data and control are treated separately. A control circuit (shown in Figure 4.13) connects to the control wires of each handshake channel (denoted by subscripts r for request and a for acknowledge)

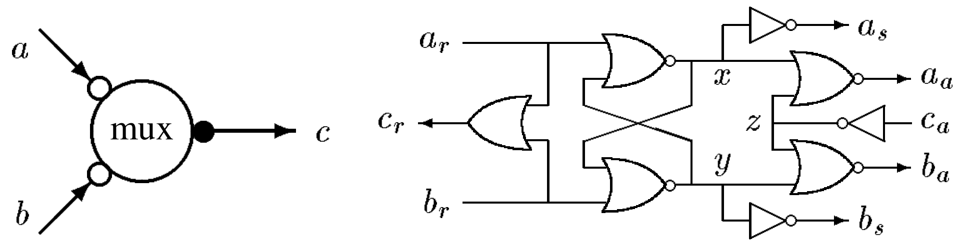


Figure 4.13: Multiplexer component and control circuit [Pee96, p. 96].

and from this generates select wires (denoted by subscript s). The select wires connect to the data part of the circuit, which for each bit i implements function $c_i = (a_i \wedge a_s) \vee (b_i \wedge b_s)$. The data on output c should be valid at least from $c_r\downarrow$ until $c_a\downarrow$, given that the input also guarantees this late data-valid scheme [Pee96].

The specification of the multiplexer control circuit is given by the following command [Pee96]:

$$\begin{array}{ll} *\ (& b_r\uparrow\ ;\ (a_s\downarrow \| \ b_s\uparrow \| \ c_r\uparrow\ ;\ c_a\uparrow\ ;\ b_a\uparrow\ ;\ b_r\downarrow)\ ;\ c_r\downarrow\ ;\ c_a\downarrow\ ;\ b_a\downarrow \\ \mid & a_r\uparrow\ ;\ (a_s\uparrow \| \ b_s\downarrow \| \ c_r\uparrow\ ;\ c_a\uparrow\ ;\ a_a\uparrow\ ;\ a_r\downarrow)\ ; c_r\downarrow\ ;\ c_a\downarrow\ ;\ a_a\downarrow) \end{array}$$

In the command, operator '$*$' is used for repetition, ';' for concatenation, '$\|$' for parallel composition, and '$|$' for choice. They are listed here in decreasing order of precedence.

Transitions on a_s and b_s are conditional, that is, they will be ignored if their corresponding signals already have the targeted levels. For instance, $a_s\uparrow a_s\uparrow$ is legal, but the second $a_s\uparrow$ has no effect. In the verification we used a more complex but equivalent command, with unconditional transitions, in which an initial state for these signals is assumed. We assumed that signals y, z, and a_s start high and all the other signals start low. The analysis would be symmetric for an initial state where only x, z, and b_s are high.

First, we check whether the circuit satisfies the specification under the following delay assumptions indicated in [Pee96], and hazard-intolerant models for all cells:

$$D_{a_r\uparrow c_r\uparrow c_a\uparrow z\downarrow} > D_{a_r\uparrow x\downarrow a_s\uparrow} \tag{4.12}$$
$$D_{b_r\uparrow c_r\uparrow c_a\uparrow z\downarrow} > D_{b_r\uparrow y\downarrow b_s\uparrow} \tag{4.13}$$

and we obtain an execution

$$b_r\uparrow c_r\uparrow c_a\uparrow y\downarrow b_s\uparrow z\downarrow a_a\uparrow a_r\uparrow b_a\uparrow$$

that contravenes the specification because $a_a\uparrow$ inappropriately occurs in response to $b_r\uparrow$. (Notice that $a_r\uparrow$ does not constitute a violation of the mutual exclusion requirement by the environment, since the device faulted first by issuing $a_a\uparrow$.)

To avoid this danger we need to force an $x\uparrow$ event to occur before $z\downarrow$. Thus we introduce by hand two new constraints for $a_s\downarrow$ and $b_s\downarrow$ similar to those for $a_s\uparrow$ and $b_s\uparrow$:

$$D_{b_r\uparrow c_r\uparrow c_a\uparrow z\downarrow} > D_{b_r\uparrow y\downarrow x\uparrow a_s\downarrow} \tag{4.14}$$
$$D_{a_r\uparrow c_r\uparrow c_a\uparrow z\downarrow} > D_{a_r\uparrow x\downarrow y\uparrow b_s\downarrow} \tag{4.15}$$

With chain constraints (4.12), (4.13), (4.14), and (4.15), the circuit satisfies the specification.

Let us compute the safety margin for constraints (4.12)–(4.15). Assuming $D_{c_r\uparrow c_a\uparrow}$ is larger than two inversion delays, we obtain a safety margin of 5/3. We have not verified the datapath, but it may add two extra inversion delays to the delay of the inverters that control a_s and b_s. With these extra delays, the safety margin becomes 5/5. If we also worry that $D_{c_r\uparrow c_a\uparrow}$ might be smaller than two inversion delays, which could happen, for instance, if there are extended isochronic forks in the environment, the safety margin is 3/5, which is unacceptably small.

Fortunately, the constraints above could be split up and relaxed, by binding only x and y to $z{\downarrow}$, and a_s and b_s only to $c_r{\downarrow}$. This results in the following set of constraints:

$$D_{a_r\uparrow c_r\uparrow c_a\uparrow z\downarrow a_a\uparrow a_r\downarrow c_r\downarrow} > D_{a_r\uparrow x\downarrow a_s\uparrow} \tag{4.16}$$
$$D_{b_r\uparrow c_r\uparrow c_a\uparrow z\downarrow b_a\uparrow b_r\downarrow c_r\downarrow} > D_{b_r\uparrow y\downarrow b_s\uparrow} \tag{4.17}$$
$$D_{a_r\uparrow c_r\uparrow c_a\uparrow z\downarrow a_a\uparrow a_r\downarrow c_r\downarrow} > D_{a_r\uparrow x\downarrow y\uparrow b_s\downarrow} \tag{4.18}$$
$$D_{b_r\uparrow c_r\uparrow c_a\uparrow z\downarrow b_a\uparrow b_r\downarrow c_r\downarrow} > D_{b_r\uparrow y\downarrow x\uparrow a_s\downarrow} \tag{4.19}$$
$$D_{a_r\uparrow c_r\uparrow c_a\uparrow z\downarrow} > D_{a_r\uparrow x\downarrow y\uparrow} \tag{4.20}$$
$$D_{b_r\uparrow c_r\uparrow c_a\uparrow z\downarrow} > D_{b_r\uparrow y\downarrow x\uparrow} \tag{4.21}$$

With the relaxed constraints (4.16)–(4.21), the circuit also satisfies the specification.

Assuming two inversion delays in the datapath and in $D_{c_r\uparrow c_a\uparrow}$, and one inversion delay in $D_{a_a\uparrow a_r\downarrow}$, the safety margins are 9/4 for (4.16) and (4.17), 9/5 for (4.18) and (4.19), and 5/2 for (4.20) and (4.21). These constraints are thus very weak and should be easy to implement. Even if no environment delays can be reliably assumed and if there are additional datapath inversion delays, the safety margins are 6/4 for (4.16) and (4.17), 6/5 for (4.18) and (4.19), and 3/2 for (4.20) and (4.21), which are still quite good.

None of chain constraints (4.16)–(4.21) are made redundant by the others. If only some of these constraints are used, we obtain violations of the specification. Using only (4.18)–(4.21), $c_r{\downarrow}$ comes before $b_s{\uparrow}$ in

$$b_r\uparrow c_r\uparrow c_a\uparrow y\downarrow x\uparrow z\downarrow b_a\uparrow a_s\downarrow b_r\downarrow c_r\downarrow b_s\uparrow\ .$$

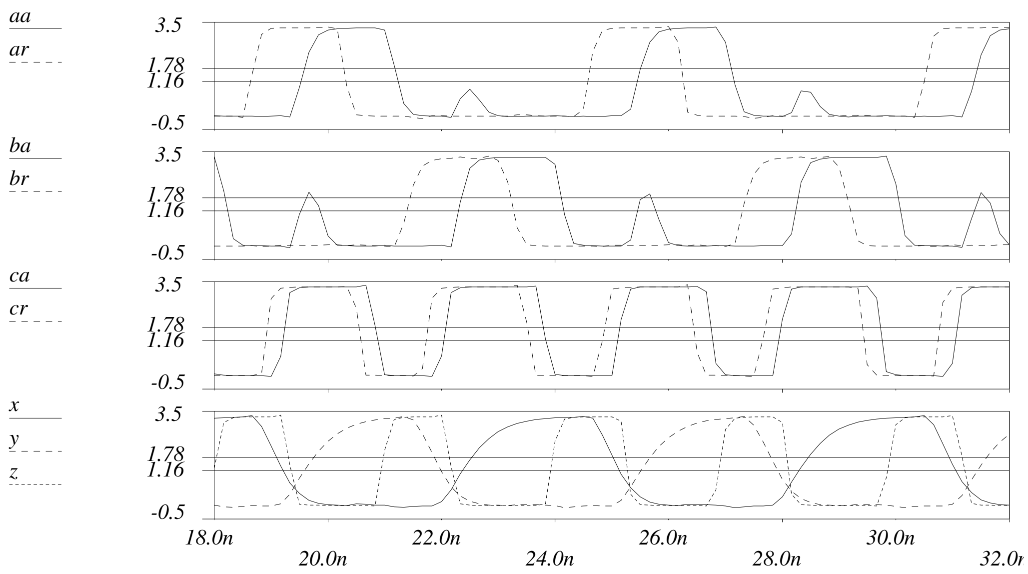


Figure 4.14: Simulation of the failure mode of execution (4.22).

Using only (4.16), (4.17), (4.20), and (4.21), $c_r\!\downarrow$ comes before $a_s\!\downarrow$ in

$$b_r\!\uparrow c_r\!\uparrow c_a\!\uparrow y\!\downarrow x\!\uparrow z\!\downarrow b_a\!\uparrow b_s\!\uparrow b_r\!\downarrow c_r\!\downarrow a_s\!\downarrow .$$

Using only (4.16)–(4.19), $a_a\!\uparrow$ inadvertently responds to $b_r\!\uparrow$ in

$$b_r\!\uparrow c_r\!\uparrow c_a\!\uparrow z\!\downarrow a_a\!\uparrow a_r\!\uparrow b_r\!\downarrow . \tag{4.22}$$

To show the importance of these constraints, we have reproduced here a simulation from [NP98] of the failure mode of execution (4.22). This simulation was performed by adding extra load to the cross-coupled NOR gates, so that the delays of these gates violate constraints (4.20) and (4.21). Constraint (4.20) would impose that $z\!\downarrow$ enter the threshold range after $y\!\uparrow$ leaves the range, which does not happen for the simulation in Figure 4.14: y and z are on the bottom plot, and the first $z\!\downarrow$ actually leaves the range before the first $y\!\uparrow$ even enters the range. Also, notice this is not a case of disrupting the chains, since the other events in (4.20) occur as specified, following the first $a_r\uparrow$ in the top plot. If the constraint were complied with, the erroneous pulses would not appear on b_r.

DO-Component

The DO-component, which is shown in Figure 4.15 (a), is used to implement repetition [Ber93]. When activated along passive handshake port a, it evaluates the guard connected to b, and depending on the outcome, either performs a handshake along c and evaluates again, or completes the handshake along a. A precise specification of the four-phase behavior is in Figure 4.16 as a finalization process.

Further we note that the b_0 guard can only be changed after the handshake on a has been completed; thus, b_0 will be stable at least till $a_a\downarrow$.

The specification is shown in Figure 4.16 as a process automaton for finalization. To reduce clutter, we often omit some illegal transitions from our state machines, in addition to the transitions that produce self-loops at every state; whenever we do so, we indicate the omissions in notes. Also, we assigned labels to two of the states in Figure 4.16, to avoid fully drawing certain incoming edges; see labels '0' and '1' for the states on the left, and 'to 0' and 'to 1' for some edges on the right side of Figure 4.16. This process automaton has two main cycles, one for b_0 low and one for b_0 high, which communicate via edges labeled b_0. Notice that signal b_0 is supposed to be constant from $b_a\downarrow$ till $a_a\downarrow$ or $c_r\uparrow$.

A possible gate realization of the DO component (taken from [Pee96, p. 79]) is shown in Figure 4.15 (b). The component marked 'C' and having a '+' sign for one of the inputs is a generalized C-element [Mar90], given by the following production

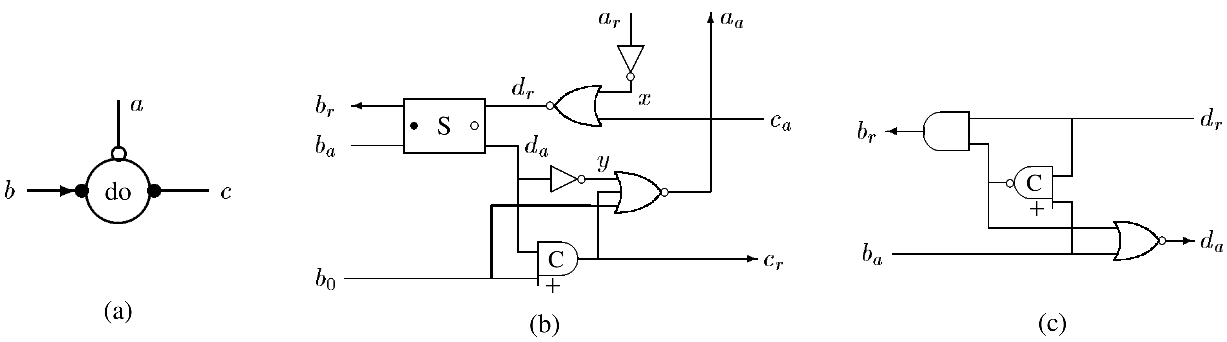


Figure 4.15: DO-component: (a) symbol; (b) circuit [Pee96]; (c) S-element implementation.

rules:

$$\begin{cases} b_0 \wedge d_a & \to & c_r\uparrow \\ \neg d_a & \to & c_r\downarrow \end{cases}$$

The box marked 'S' is an S-element[1], whose implementation is shown in Figure 4.15 (c). The S-element is specified by the following command ([Pee96, p. 75]):

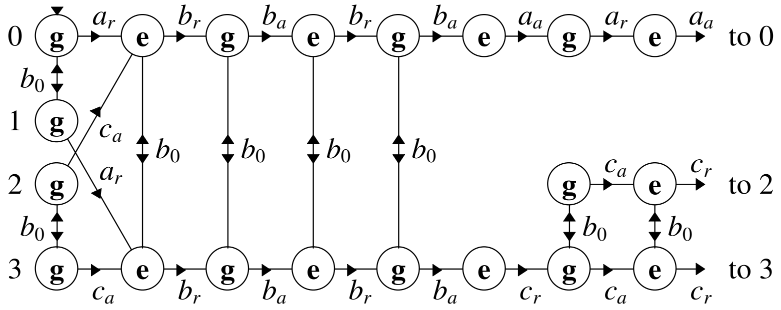
$$*(d_r\uparrow b_r\uparrow b_a\uparrow b_r\downarrow b_a\downarrow d_a\uparrow d_r\downarrow d_a\downarrow)$$

To start, we have verified that the S-element specification is refined by the product of hazard-intolerant models of the gates in Figure 4.15 (c).

In the analysis below, we assume the circuit in 4.15 (b) starts at a state where signals x and y are high and all other signals are low. If all inputs are held low, the circuit does indeed stabilize at this state.

In [Pee96], it was also assumed that the delay from a $d_a\downarrow$ event to a $c_r\downarrow$ event through the C-element (two inversions) is larger than the delay from $d_a\downarrow$ to $y\uparrow$ (one

[1]The S-element is also called 'Q' in [Mar90]



NOTES

- illegal b_r, c_r, a_a events lead to a permanent-escape state
- illegal b_a, b_0, c_a, a_r to a permanent-reject
- all actions that are not shown produce self-loops at every state

Figure 4.16: Finalization process for the DO-component specification.

inversion), which translates into the chain constraint

$$D_{d_a\downarrow c_r\downarrow} > D_{d_a\downarrow y\uparrow}. \tag{4.23}$$

With constraint (4.23) and hazard-intolerant models of all gates, we obtained the following counter-example to refinement, indicating a hazard by $d_a\downarrow$ at the inverter whose output is y:

$$a_r\uparrow x\downarrow d_r\uparrow b_r\uparrow b_0\uparrow b_a\uparrow b_r\downarrow b_a\downarrow d_a\uparrow c_r\uparrow c_a\uparrow d_r\downarrow d_a\downarrow c_r\downarrow$$

Is this hazard a true danger of violating the specification? Using an inertial model for the inverter and hazard-intolerant models for all other gates, the circuit with constraint (4.23) does refine the DO-component specification above.

Better yet, we do not need to rely on an inertial model for the inverter. With constraint (4.23) and the additional constraint

$$D_{d_a\uparrow c_r\uparrow} > D_{d_a\uparrow y\downarrow} \tag{4.24}$$

no hazards occur: under hazard-intolerant models for all gates, the circuit with constraints (4.23) and (4.24) refines the DO-component specification above. The circuit works fine under the (rather weak) chain constraints (4.23) and (4.24), which imply a safety margin of at least 2/1.

4.3 Stocktaking

We have verified three circuits from [Pee96] as case studies for the analysis technique of [Neg97b]. This exercise has pointed to several problems at the border between discrete and analog viewpoints on circuit behavior. We have given an account of these problems and the way we dealt with them. The circuits were found to be correct, and in one case a notable relaxation of the delay constraints could be obtained. The three analysis sessions in Section 4.2.4 took 7, 48, and 8 seconds on a Sun Sparc 5/110 to perform 4, 6, and 4 verifications or diagnoses, respectively, including the delay constraints. Also, we have performed analog simulations for the circuits under study, and the simulations have been consistent with the discrete-state analysis.

As mentioned in Section 1.2, process spaces are closely related to several previous language-oriented treatments of concurrency. Although maybe we could have performed our discrete-state analysis using one of these previous formalisms, note that, in their present forms, [Dil89], [Jos92], and [Ver94b] forbid connecting outputs by parallel composition, whereas in our analysis we need to take the product of chain constraint processes and gate processes that share control over certain actions, and we need the structured verification properties to cover this connectivity case too. CSP, [Ebe91b], and [VK98, VT95] do allow shared outputs in parallel composition, but they have, for instance, restrictions on the processes that can participate in comparisons. The connectivity restrictions in CSP are less limiting than those of other models; on the other hand, CSP handles deadlock by a rather complicated with explicit representations of the events that may be refused at the end of each trace.

Because chain constraints provide for the cases where the involved signal transitions occur out of order or do not occur at all, we did not treat them as simple linear inequalities on delays. On the other hand, chain constraints are easy to handle as processes, and as such they can deal with deadlock, hazards, and non-determinism. The safety margin measure from [KN94] could also be applied for chain constraints.

Chapter 5

Switch-Level Correctness Concerns

Switch-level models [Bry87] offer a viable trade-off between accuracy and efficiency of the analysis of digital MOS circuits. Such models approximate an MOS transistor as an on/off switch, and simplify charge sharing and path strength calculations to the extreme. Nevertheless, such models can capture logical behavior that can be used further for gate-level analysis, and can deal with specific correctness concerns for MOS transistor networks, such as floating states, collisions, and shorts.

In a floating state, a circuit wire is isolated both from ground and from the power supply. A collision occurs if a wire is connected both to ground and to the power supply through transistor channel networks of commensurate strengths. A short is similar to a collision, except that the connections to ground and power supply may be of different strengths. If any of these situations occur in stable states, we call them persistent; otherwise we call them transient. Floating states, collisions, and shorts do not necessarily constitute malfunctions, especially if they are transient;

in fact, floating states and transient non-collision shorts are routinely put to use in certain design styles. Still, floating states and collisions may be undesirable in other designs, because the respective wire voltages can decay to an unknown intermediate value; also, non-collision shorts may be undesirable because of undue power dissipation. To accommodate all these cases, our technique permits us both to detect and to ignore such situations.

Switch-level analysis methods have been proposed in [BCDM86], [Bry87], [BS95], [KS95], [KSL95], and [ZH92], among others. However, except for [BS95], these methods have focused only on steady-state or synchronous behavior.

The methods in [Bry87] and [KSL95] extract the steady-state response of switch-level networks, characterizing signal relationships in stable states. The result can be used further in gate-level analysis, and [ZH92] and [KS95] also apply it to sequential, synchronous circuits that stabilize over a clock period. The method in [BCDM86] extracts a state graph from a transistor netlist directly, assuming all gates in a circuit have the same delay. The framework for MOS circuit analysis in [BS95] can be used for various checks of asynchronous behavior and can even incorporate timing information, but the subsequent timed analysis is only for stable outcome.

The technique we propose here aims to complement the methods above and to provide additional flexibility. We do not extract the behavior of a switch-level network, but we perform computationally-intensive verifications of switch-level correctness concerns, as well as gate-level analysis. Our analysis is event-driven, in the sense that it focuses on signal transitions rather than signal levels and does not assume (nor forbid) synchrony. The event-driven approach permits us to use certain processes that are not easy to cast in a Boolean framework. Notably, these processes can model several assumptions regarding the behavior of charge-retaining circuit nodes, diverse switch-level correctness concerns, and even relative speeds of the components. As in Chapter 4, these speed assumptions do not require knowledge of numerical bounds on component delays, thus, very importantly, the circuit can be analyzed before sizing its transistors and wires. Also see [Neg98].

We tour the features of the technique by means of a case study, a self-timed RAM circuit from [NS95]. Previous verification of this circuit was reported in [NS95], but it did not address the switch-level correctness concerns, and it did not cover parts of the design where the delay constraints intervene.

5.1 Switch-Level Correctness Concerns

We start our analysis by dividing an MOS transistor network into channel-connected subnetworks [Bry87], determined by the absence of gate-drain and gate-source connections. This split depends only on the circuit configuration, not its operation, and stays the same throughout the analysis. The reason for this split is that gate capacitances usually introduce comparatively large delays that need to be modeled separately. We refer to such large-load circuit nodes as charge-retaining nodes.

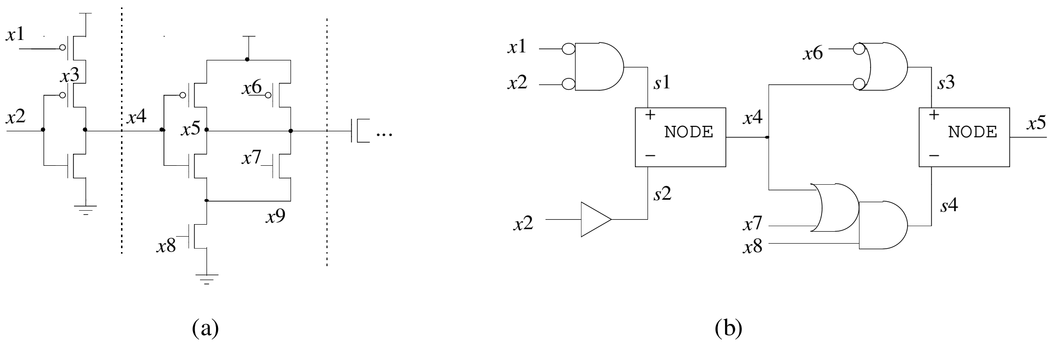


Figure 5.1: Modeling a switch-level network: (a) MOS transistor network split into channel-connected subnetworks; (b) replacement network.

Normally, each channel-connected subnetwork contains an N-transistor network and a P-transistor network, adjacent to ground and power, respectively, called the pull-down and the pull-up networks.

As was done in previous switch-level analysis techniques, we use Boolean variables to indicate the presence of conducting paths. We associate two such variables, call them *path signals*, to each charge-retaining node: one for a path to ground and one for a path to the power supply. For now, we assume the paths have equal strengths; we deal with differing strengths later.

For example, consider the network in Figure 5.1 (a). We introduce the following path signals:

$s1$: there is a conducting path from $x4$ to power supply;
$s2$: there is a conducting path from $x4$ to ground;
$s3$: there is a conducting path from $x5$ to power supply;
$s4$: there is a conducting path from $x5$ to ground.

The pull-down and pull-up networks, controlling the path signals, are modeled by processes which turn out to be similar to Boolean gates. This is not surprising, since such processes emulate the logical relationships between voltage signals and path signals. In Figure 5.1 (b), we actually use Boolean gate symbols for such processes.

In the following, we discuss the particular finalization processes we used to model the components of a replacement network like that in Figure 5.1 (b). These processes are our main mechanism for dealing with various switch-level correctness concerns and peculiarities of behavior. We also discuss certain modeling options that we later discard, but this does not mean that the same *faux pas* have to be taken every time the technique is applied; the purpose of the discarded options is to help explain and justify the option we consider appropriate.

The NODE component in Figure 5.1 (b) represents charge retention in the presence or absence of conducting paths. We have several options in modeling a node by a finalization process, depending on the correctness concerns and behavior assumptions that are commonly used in a given design style. For the design style of the circuit at hand, we use the process in Figure 5.2. A node component controls a voltage signal according to two path signals, representing conducting paths to ground and the power supply. In Figure 5.2, these signals are z, up, and dn respectively. In state 0, the initial state, there are no conducting paths and the voltage is low. (In applications, we change the initial state as needed.) To help interpret the model, notice that: z is always high in states 4, 5, 6, 7, regardless of how the states were reached, and it is always low in 0, 1, 2, 3; up is high in 2, 3, 6, 7 and low elsewhere; dn is high in 1, 3, 5, 7 and low elsewhere. As in Chapter 4, some events and the permanent escape states are omitted from the figures for node processes.

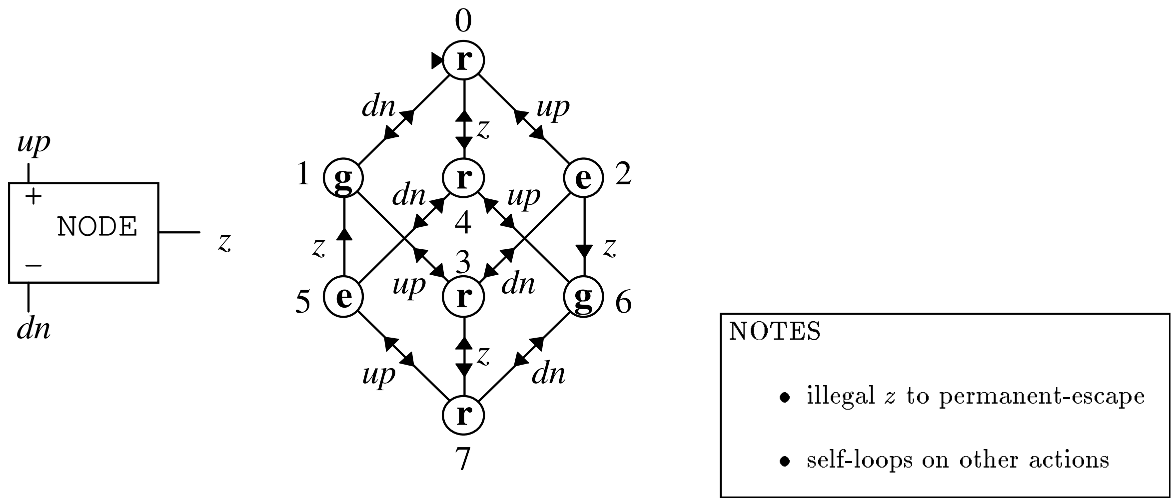
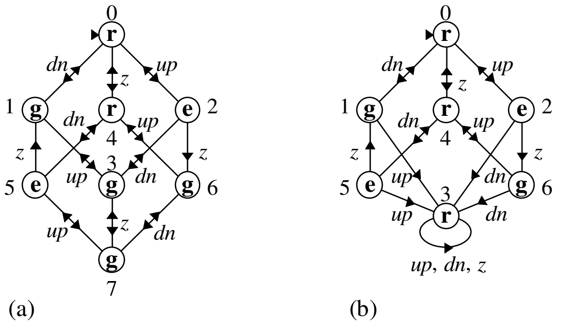


Figure 5.2: Node icon and possible node process.



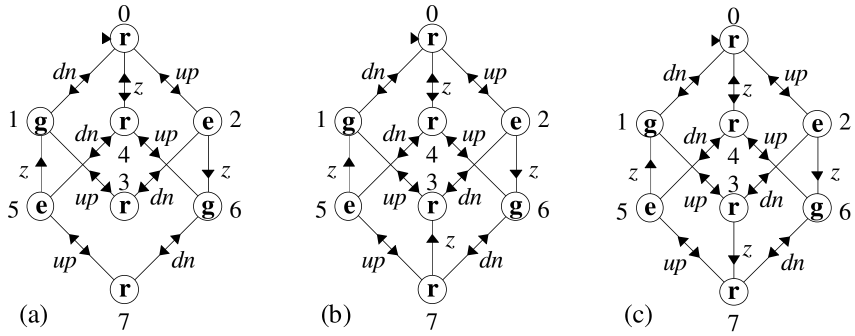
NOTES

- illegal z to permanent-escape
- self-loops on other actions

Figure 5.3: Alternate node processes: do we forbid collisions?

For the model in Figure 5.2, we forbid persistent collisions by taking states 3 and 7 to be rejects. It is the responsibility of the environment to avoid stopping in such a state, where both path signals are high at the same time. Alternately, we could consider collisions to be legal, as in Figure 5.3 (a). In Figure 5.2, we allow transient collisions; for instance, execution *up z dn up z* is a goal, despite passing through state 7. Another alternative would be to make transient collisions illegal by taking all executions that pass through a collision state to be rejects, as in Figure 5.3 (b); there, states 3 and 7 have been merged.

For the model in Figure 5.2, we assumed that, when both paths are conducting, we don't know whether the voltage is driven high, low, or close enough to a threshold level for the neighboring gates to sense small fluctuations as signal transitions. Accordingly, in Figure 5.2 we allow the node voltage to switch arbitrarily during a collision, following the cycle between states 3 and 7. Alternately, we could assume the node voltage constant during collisions, to model, for instance, that a feedback inverter pair introduces a hysteresis (as in many of the cells in Figure 5.7). For that, we could take the z edges that leave states 3 or 7 to lead to a permanent-escape state instead, as in Figure 5.4 (a). Another alternative would be that the voltage might fall but never rise during a collision, for a case where thresholds of



NOTES

- illegal z to permanent-escape
- self-loops on other actions

Figure 5.4: Additional alternatives for node processes: do we forbid switching during collisions?

subsequent gates are higher than the intermediate level where the node voltage stabilizes during a collision. For that, we could allow a z edge from state 7 to state 3, but take the z edge that leaves state 3 to lead to a permanent escape, as in Figure 5.4 (b). Symmetrically, we could consider instead that the voltage may rise but never fall during a collision, as in Figure 5.4 (c). Yet another alternative would be to consider that the voltage either never rises or never falls during a collision, but it may always undergo the opposite transition during a collision. For this, we could take the meet of the models in Figure 5.4 (b) and Figure 5.4 (c).

Symmetric alternatives can be obtained for floating states, and the modifications can be combined to obtain a family of node models for diverse assumptions and correctness concerns. In this case study we settled for the model in Figure 5.2, which forbids persistent collisions and persistent floating states, permits transient collisions and transient floating states, and accounts for switching of the node voltage both while floating and during collisions. Nevertheless, the alternatives discussed above can be used for analyzing circuits designed under different conventions.

So far we have only considered paths of equal strengths. For differing path strengths, approximated by integers as in [Bry87], the function realized by the Boolean model of a pull-down network is as follows:

$[\exists$ a conducting path to ground of strength $n]$
$\vee$ ($[\exists$ conducting path to ground of strength $n-1]$
  $\wedge\ \neg\ [\exists$ conducting path stronger than $n-1$ leading to the power supply])
$\vee$ ($[\exists$ conducting path to ground of strength $n-2]$
  $\wedge\neg\ [\exists$ conducting path stronger than $n-2$ leading to the power supply])
$\vee$ ...

The model for a pull-up network is symmetric.

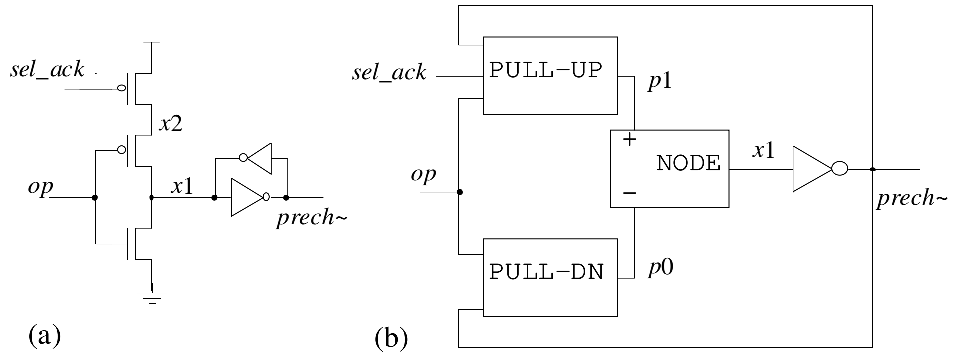


Figure 5.5: Modeling paths in a CMOS cell (the precharge control cell from [NS95]: (a) circuit and signals; (b) replacement network.

For example, the precharge control cell from [NS95] shown in Figure 5.5 (a) (with some new signal labels) is modeled by the network in Figure 5.5 (b), which includes two channel-connected subnetworks: one for the strong inverter and one for the rest of the cell. For the latter subnetwork, the pull-up and pull-down networks are modeled by complex Boolean gates, represented by the boxes in Figure 5.5 (b) that control the path signals $p1$ and $p0$. The function realized by the complex-gate model of the PULL-UP subnetwork in Figure 5.5 (b) is

$$p1 \;=\; (\neg sel\_ack \wedge \neg op) \vee (\neg prech^{\sim} \wedge \neg op) \;,$$

where the first disjunct indicates the presence of a strong conducting path to the power supply, while the second indicates there is a weak conducting path to the power supply and there are no strong conducting paths to ground. The function realized by PULL-DN is, similarly,

$$p0 \;=\; op \vee (prech^{\sim} \wedge \neg(\neg sel\_ack \wedge \neg op)) \;.$$

5.2 Local Isochrony

For the strong inverter in the precharge control cell, we use an inertial model, as justified later. Now, we check refinement between the formal interface for the precharge control cell and the replacement network in Figure 5.5 (b). The formal interface is a hazard-intolerant model (Section 4.1) of the following production rules from [NS95].

Precharge control cell:

$$\begin{cases} op & \rightarrow & prech^{\sim}\uparrow \\ \neg op \wedge \neg sel\_ack & \rightarrow & prech^{\sim}\downarrow \end{cases}$$

The hazard-intolerant model is our default model for cell interfaces, and assumes the cells are operated without hazards. (We verify later that the precharge control cell is indeed operated without hazards.) For the initial state, we assume that the signals sel_ack, op, and $prech^{\sim}$ are low. We obtain the following counter-example execution:

$$op\uparrow\ p0\uparrow\ x1\downarrow\ prech^{\sim}\uparrow\ x1\uparrow\ prech^{\sim}\downarrow\ op\downarrow\ p0\downarrow \tag{5.1}$$

which wrongly indicates that $prech^{\sim}$ may switch! This switching is because a path to ground seems to start conducting at $p0\uparrow$ as the path to the power supply continues to conduct, as in the initial state, despite $op\uparrow$.

We would expect, however, that in reality $prech^{\sim}$ does not have time to charge and discharge before the path to the power supply is terminated, and that the transistors 'open' and 'close' much quicker than the voltage signals rise or fall. If so, the execution in (5.1) would be avoided, because, by the time $prech^{\sim}\uparrow$ occurs, the path to the power supply will have been terminated by a $p1\downarrow$ event. To introduce this assumption, we modify the finalization processes for the 'gates' that control path signals in the replacement network. We start with inertial models, justified by the assumption that paths are formed and interrupted very quickly after an input signal undergoes a transition. Then, we modify the process by taking the slower transitions of the node voltage to lead to a permanent-escape state from any unstable states of the replacement gate. We refer to the gate processes modified this way as *quickened models*.

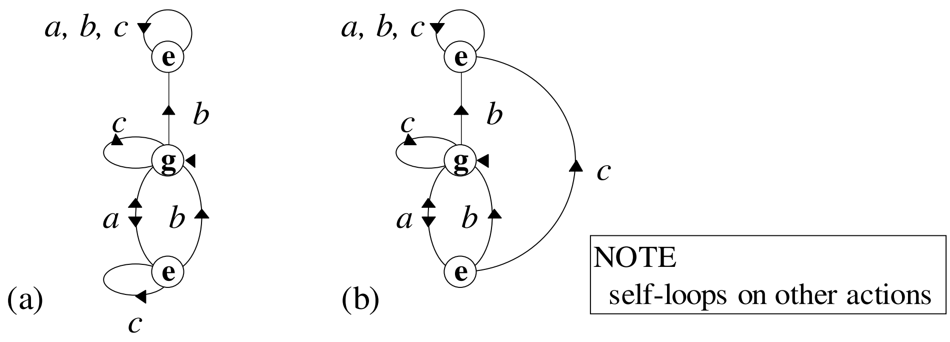


Figure 5.6: Obtaining a quickened model from an inertial model: (a) inertial model; (b) quickened model.

For example, the process automaton in Figure 5.6 (a) represents the finalization process of an inertial buffer with input a and output b. Action c is external to the buffer and thus c events do not affect the state of the buffer. Assuming c transitions are slow, we take in Figure 5.6 (b) the c events from the unstable state to lead to a permanent-escape state, making it the responsibility of the device to avoid such events. How can the buffer physically avoid these external events? By being 'quick' to leave its unstable state.

Incidentally, the transformation in Figure 5.6 can also be obtained by chain constraints.[1] The product of the process in Figure 5.6 (a) with the processes for chain constraints $D_{a\uparrow b\uparrow} < D_{a\uparrow c\uparrow}$, $D_{a\uparrow b\uparrow} < D_{a\uparrow c\downarrow}$, $D_{a\downarrow b\downarrow} < D_{a\downarrow c\uparrow}$, and $D_{a\downarrow b\downarrow} < D_{a\downarrow c\downarrow}$ is precisely the process represented by the process automaton of Figure 5.6 (b). Since the levels of signals are irrelevant here, one can also replace the chain constraints above by a single constraint $D_{ab} < D_{ac}$ having as the base state (where the race between ab and ac starts) the initial state of the process in Figure 5.6 (a).

Note that none of these assumptions is intrinsic to our technique, and other assumptions might be used to fit a particular application. For example, the problem pointed to by the execution in (5.1) would also be avoided by an assumption that the node does not switch voltage while floating or during collisions, instead of using quickened models. This alternate assumption can be justified, say, if the hysteresis of the output inverter pairs keeps the node voltages constant during collisions, and if the weak inverters ensure that the respective nodes are never floating. However, we use quickened models because they provide local compatibility with [Bry87] and other switch-level analysis methods: if we assume that the path signals stabilize much quicker than the voltage signals, it suffices to consider the steady-state behavior of the N and P subnetworks, which can be extracted, say, by [Bry87] from a transistor netlist. Note, nevertheless, that the transient behavior of the nodes is not hindered by this assumption, which refers to the N and P subnetworks but not to entire cells.

Finally, we find that refinement holds between the hazard-intolerant interface of the precharge control cell and the new replacement network. For the reasons discussed above, we settled for quickened inertial models for the gates and a node of the type in Figure 5.2. It follows, under these assumptions, that there are no persistent floating states and no persistent collisions for this cell.

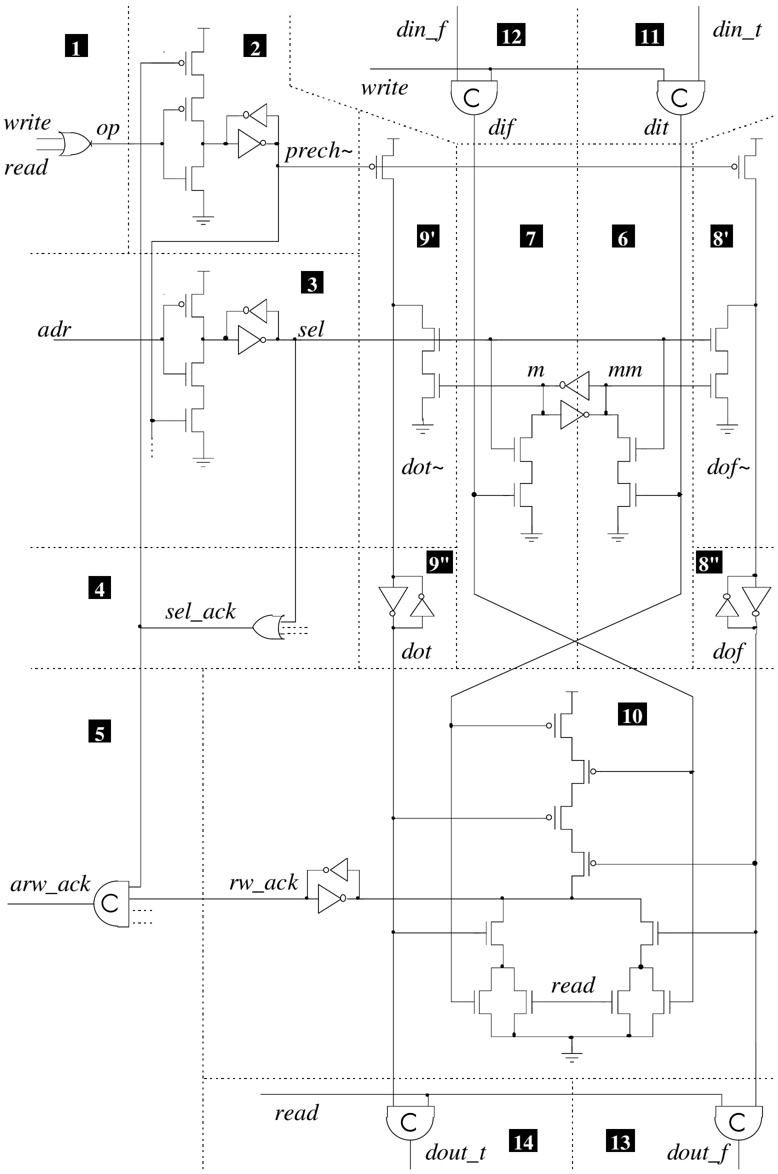
[1] We thank D. L. Dill for this observation.

The inertial model for the strong inverter is justified by checking refinement of this model by a replacement network for the inverter, consisting of quickened inertial models and a node of the type in Figure 5.2.

5.3 Case Study: A Self-Timed RAM

We analyze a self-timed RAM circuit from [NS95], intended to operate as follows. Signals *read* and *write* are control signals; *adr* is used as a select signal; *dit* and *dif* are data inputs, for 'true' and 'false', and they are mutually exclusive; *dot* and *dof* are data outputs, with a similar encoding; *arw_ack* is an acknowledge signal for read and write operations. Initially, all these signals are low. A read cycle starts with the environment raising *read* and *adr*; then, the circuit responds by raising *arw_ack* and either *dot* or *dof*; the environment lowers *read* and *adr*; the circuit lowers *arw_ack* and either *dot* or *dof*, completing the cycle. A write cycle starts with the environment raising *write*, *adr*, and either *dif* or *dit*; the circuit raises *arw_ack*; the environment lowers *write*, *adr*, and either *dif* or *dit*; finally, the circuit lowers *arw_ack*, completing the cycle. These cycles have four phases: the signals involved are initially low, then some input signals rise, some output signals rise in response, all input signals return to low, and all output signals return to low.

The schematic of the circuit we analyze is given in Figure 5.7, after a figure from [NS95]. We have introduced some new signal labels. The ~ signs are part of the signal labels and stand for active low. The specification of the circuit is given by a finalization process in Figure 5.8. As mentioned in Chapter 4, to reduce clutter, we often omit some illegal transitions from our state machines, in addition to the transitions that produce self-loops at every state; whenever we do so, we indicate the omissions in notes. Also, we assigned labels to two of the states in Figure 5.8,



CELLS

1 OR gate
2 Precharge control
3 Select control
4 OR gate
5 Acknowledge control
6, 7 Memory
8', 9' Memory output, after splitting the inversions
8'', 9'' Data-path inversions, split from memory output
10 Data-path acknowledge
11, 12 Data input
13, 14 Data output

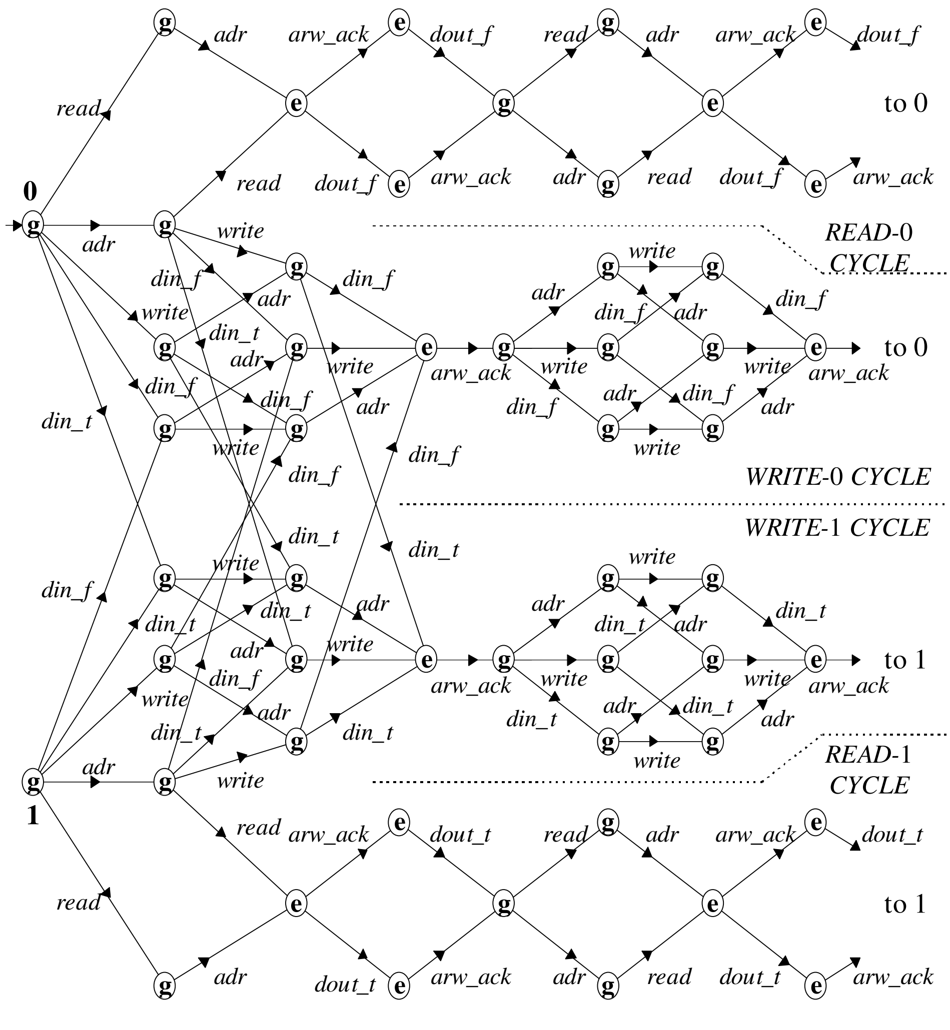
Figure 5.7: Schematic of the self-timed RAM circuit of [NS95], with adjusted partition into cells.

to avoid drawing fully certain incoming edges; see labels '0' and '1' for the states on the left, and 'to 0' and 'to 1' for some edges on the right side of Figure 5.8.

For initial conditions, we assume signals mm, $dof^{\sim}$, and $dot^{\sim}$ in Figure 5.7 are high, and all other labeled signals of Figure 5.7 are low. By taking mm to start high, we consider that a zero is initially stored in the memory cell. Symmetric results would be obtained by assuming a one is stored initially.

Various problems may occur if the circuit is used immediately after power-up, before m and mm stabilize to opposite levels. However, in practice it is often assumed that a circuit is not used before it stabilizes in a desired state after power-up. One can easily check that the circuit under study will stabilize in the initial state or in the symmetric state described above if all its inputs are held low, as in the initial state of the formal specification. Thus, while noting the danger, we limit our analysis to the initial conditions where m is low and mm is high.

Apart from the overall specification and the low-level implementation, we use interface specifications for individual cells to divide the analysis problem into sub-problems. Following [NS95], we represent cell interfaces by production rules [Mar90]. Essentially, these descriptions amount to specifying the instability functions of the cells, as discussed in Section 4.1. As such, these descriptions can be automatically translated by FIREMAPS into finalization processes. By default, we use hazard-intolerant processes for all cell interfaces, to detect hazards as safety violations; later, we use other models for those cells where hazards are tolerated. In Section 5.2, we have given the formal interface for the precharge control cell; the interface specifications for the other cells are as follows.



NOTES
- illegal *dout_f*, *dout_t*, *arw_ack* to permanent escape
- illegal *dint_f*, *dint_t*, *read*, *write*, *adr* to permanent reject
- self-loops on all other actions

Figure 5.8: Circuit specification.

Select control cell (as given in [NS95]; we correct it later):

$$\begin{cases} adr \wedge \neg prech^{\sim} & \rightarrow \quad sel\!\uparrow \\ \neg adr & \rightarrow \quad sel\!\downarrow \end{cases}$$

The multiple-input select OR gate becomes just a wire if there is only one cell:

$$\begin{cases} sel & \rightarrow \quad sel\_ack\!\uparrow \\ \neg sel & \rightarrow \quad sel\_ack\!\downarrow \end{cases}$$

Acknowledge control cell, if there is just one cell:

$$\begin{cases} rw\_ack \wedge sel\_ack & \rightarrow \quad arw\_ack\!\uparrow \\ \neg rw\_ack \wedge \neg sel\_ack & \rightarrow \quad arw\_ack\!\downarrow \end{cases}$$

The original memory cell interface from [NS95] assumes that, in a write operation, m and mm are assigned simultaneously. In the circuit, however, these assignments are causally related and must occur one after the other. To follow the implementation more closely, we split the cell interface into two parts, one that drives m and one that drives mm, as follows.

$$\begin{cases} \neg(mm \vee (dif \wedge sel)) & \rightarrow \quad m\!\uparrow \\ mm \vee (dif \wedge sel) & \rightarrow \quad m\!\downarrow \end{cases} \qquad \begin{cases} \neg(m \vee (dit \wedge sel)) & \rightarrow \quad mm\!\uparrow \\ m \vee (dit \wedge sel) & \rightarrow \quad mm\!\downarrow \end{cases}$$

Memory output cells (we discuss some changes later):

$$\begin{cases} sel \wedge m & \rightarrow \quad dot\!\uparrow \\ \neg prech^{\sim} & \rightarrow \quad dot\!\downarrow \end{cases} \qquad \begin{cases} sel \wedge mm & \rightarrow \quad dof\!\uparrow \\ \neg prech^{\sim} & \rightarrow \quad dof\!\downarrow \end{cases}$$

Data-path acknowledge cell:

$$\begin{cases} ((dit \vee read) \wedge dot) \vee ((dif \vee read) \wedge dof) & \rightarrow \quad rw\_ack\!\uparrow \\ \neg(dit \vee dif \vee dot \vee dof) & \rightarrow \quad rw\_ack\!\downarrow \end{cases}$$

After a refinement check fails for the select control cell under the currently used models established above, we obtain the following counter-example execution:

$$adr\uparrow \tag{5.2}$$

The execution in (5.2) indicates a false deadlock: the original interface from [NS95] expects a $sel\uparrow$ to occur in response to $adr\uparrow$, whereas the actual CMOS gate does not raise its output signal, since $prech^{\sim}$ is still low. This points to a flaw in that interface, which should be instead:

$$\begin{cases} adr \wedge prech^{\sim} & \rightarrow & sel\uparrow \\ \neg adr & \rightarrow & sel\downarrow \end{cases}$$

For this new cell interface, refinement holds.

The $prech^{\sim}$ signal was not shown in the schematic in [NS95]; however, from the precharge control cell interface in [NS95], one deduces that $prech^{\sim}$ is the output of the precharge control cell, active low. Regardless of how $prech^{\sim}$ is interpreted, the original cell interfaces do not correspond to the transistor schematic. More comments about this are given in Section 5.5.

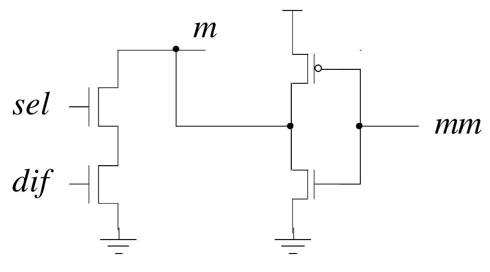


Figure 5.9: Part of memory cell.

The memory cell circuit splits into two channel-connected subnetworks. The part that drives m is shown in Figure 5.9; the weak transistors are drawn smaller. We find that refinement holds between the memory cell interface that drives m and the network in Figure 5.9, and between the memory cell interface that drives mm and the remaining part of the memory cell (for models as above).

The memory output cells are symmetric and we only discuss here the cell that drives *dot*. We obtain the following execution as a counter-example to refinement:

$$sel\!\uparrow\; m\!\uparrow\; p8\!\uparrow \tag{5.3}$$

where $p8$ is the path signal of the pull-down subnetwork, and $p8\!\uparrow$ indicates that a path is formed to ground. Since initially $prech^{\sim}$ is low, the pull-up network is conducting too, and the execution in (5.3) leaves the cell in a persistent collision. The environment and the rest of the circuit should avoid this danger by interrupting the path to the power supply before the pull-down network starts conducting, and we check later whether or not they avoid this danger.

For the data path acknowledge cell, verification was straightforward and refinement does indeed hold between the cell interface and the replacement network.

As the circuit is not completely speed-independent, we use chain constraints to model its relative delay assumptions. Finding the chain constraints typically requires several verifications; to shortcut this procedure, we use gate-level cell interfaces instead of the larger replacement networks of the switch-level cells. We check later that the chain constraints we find this way do suffice for switch-level correctness as well. In the final verification, after finding candidate chain constraints, we use the replacement networks of the memory output cells, and the interface specifications of all other cells, since the replacement networks of all other cells did refine their interface specifications.

Using hazard-intolerant models for all cells and the corrected interface for the select control cell, we obtain the following counter-example execution:

$$\begin{aligned} &write{\uparrow}\ din\_t{\uparrow}\ dit{\uparrow}\ adr{\uparrow}\ op{\uparrow}\ prech^{\sim}{\uparrow}\ sel{\uparrow}\ mm{\downarrow} \\ &\quad m{\uparrow}\ dot{\uparrow}\ sel\_ack{\uparrow}\ rw\_ack{\uparrow}\ arw\_ack{\uparrow} \end{aligned} \tag{5.4}$$

The $mm{\downarrow}$ event causes a hazard for the memory output cell that drives dof, since $dof{\uparrow}$ was enabled after $sel{\uparrow}$, but $dof{\uparrow}$ did not occur and was disabled by $mm{\downarrow}$.

The presence of this hazard is mentioned in [NS95], where it is indicated that this hazard is tolerated for the sake of efficiency. Thus, in the following, we use inertial models for the parts of the memory output cells that drive $dof^{\sim}$ and $dot^{\sim}$, followed by hazard-intolerant models for the inverter cells that drive dof and dot. This way, we model the storage of a logical value on the capacitances of $dot^{\sim}$ and $dof^{\sim}$, since the internal data bus normally has a large capacitance. This amounts to replacing the hazard-intolerant model of the memory output by a 2-bounded model.

The new interfaces are as follows.

Memory output cells:

$$\left\{\begin{array}{lcl} \neg prech^{\sim} & \rightarrow & dot^{\sim}{\uparrow} \\ sel \wedge m & \rightarrow & dot^{\sim}{\downarrow} \end{array}\right. \qquad \left\{\begin{array}{lcl} \neg prech^{\sim} & \rightarrow & dof^{\sim}{\uparrow} \\ sel \wedge mm & \rightarrow & dof^{\sim}{\downarrow} \end{array}\right.$$

Data-path inversions:

$$\left\{\begin{array}{lcl} \neg dot^{\sim} & \rightarrow & dot{\uparrow} \\ dot^{\sim} & \rightarrow & dot{\downarrow} \end{array}\right. \qquad \left\{\begin{array}{lcl} \neg dof^{\sim} & \rightarrow & dof{\uparrow} \\ dof^{\sim} & \rightarrow & dof{\downarrow} \end{array}\right.$$

With the new interfaces, refinement still doesn't hold. We obtained the following counter-example execution:

$$\begin{aligned}&write{\uparrow}\ op{\uparrow}\ prech^{\sim}{\uparrow}\ adr{\uparrow}\ sel{\uparrow}\ din\_t{\uparrow}\ dit{\uparrow}\ dof^{\ \sim}{\downarrow}\\ &\quad mm{\downarrow}\ m{\uparrow}\ dot^{\sim}{\downarrow}\ dot{\uparrow}\ sel\_ack{\uparrow}\ rw\_ack{\uparrow}\\ &\quad arw\_ack{\uparrow}\ adr{\downarrow}\ write{\downarrow}\ op{\downarrow}\ sel{\downarrow}\ sel\_ack{\downarrow}\ prech^{\sim}{\downarrow}\\ &\quad dof^{\ \sim}{\uparrow}\ dot^{\sim}{\uparrow}\ dot{\downarrow}\end{aligned} \tag{5.5}$$

which shows a hazard at the data path inversion cell that drives dof. The hazard is caused by $dof^{\ \sim}{\uparrow}$, which disables a $dof{\downarrow}$ transition that was enabled by $dof^{\ \sim}{\downarrow}$ but never occurred.

The hazard of (5.5) was mentioned in [NS95], which indicated that it will not actually occur because "the inverter is the fastest gate in a CMOS circuit". Accordingly, we can assume that the data path inverters will switch before $arw\_ack{\uparrow}$ is issued.

This assumption is too general; we actually impose some weaker chain constraints that are easier to model and that follow from the assumption above:

$$D(dof^{\sim}{\downarrow}\ rw\_ack{\uparrow}\ arw\_ack{\uparrow}) > D(dof^{\sim}{\downarrow}\ dof{\uparrow})$$

$$D(dot^{\sim}{\downarrow}\ rw\_ack{\uparrow}\ arw\_ack{\uparrow}) > D(dot^{\sim}{\downarrow}\ dot{\uparrow})$$

Under these chain constraints and models for the memory output cell, refinement holds at the gate level.

To obtain speed-independence, [NS95] proposes an alternate design for the memory cell and the memory output cells, reproduced here in Figure 5.10 and represented by the interfaces:

$$\left\{\begin{array}{rcl}\neg m \wedge \neg(dif \wedge write) & \rightarrow & mm{\uparrow}\\ \neg(\neg m \wedge \neg(dif \wedge write)) & \rightarrow & mm{\downarrow}\end{array}\right.$$

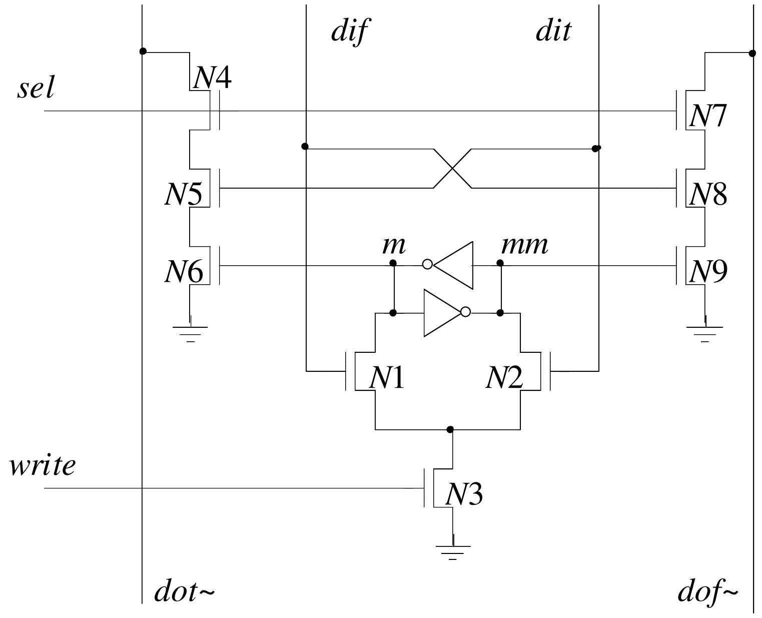


Figure 5.10: Speed-independent RAM design from [NS95], plus transistor labels for reference.

$$\begin{cases} \neg mm \wedge \neg(dit \wedge write) & \rightarrow & m\uparrow \\ \neg(\neg mm \wedge \neg(dit \wedge write)) & \rightarrow & m\downarrow \end{cases}$$

$$\begin{cases} \neg prech^{\sim} & \rightarrow & dof^{\sim}\uparrow \\ sel \wedge dif \wedge mm & \rightarrow & dof^{\sim}\downarrow \end{cases}$$

$$\begin{cases} \neg prech^{\sim} & \rightarrow & dot^{\sim}\uparrow \\ sel \wedge dit \wedge m & \rightarrow & dot^{\sim}\downarrow \end{cases}$$

In Figure 5.10, transistors $N1$, $N2$, $N3$ and the two inverters form a single channel-connected network. However, signals *dit* and *dif* are mutually exclusive; thus, no conducting paths can be formed through both transistors $N1$ and $N2$. Following this observation, we split the memory cell into two cells, to simplify the analysis.

One of the new cells is as shown in Figure 5.9; the other new cell is symmetric to this one.

At this point we check refinement using the alternate cell interfaces and no constraints. We obtain the following counter-example execution:

$$read\uparrow\ adr\uparrow\ op\uparrow\ prech^{\sim}\uparrow\ sel\uparrow\ sel\_ack\uparrow \tag{5.6}$$

which indicates that the circuit may deadlock in the middle of a read cycle, while the specification expects data output and acknowledge events. The problem is that one of the memory output cells should have discharged $dof^{\sim}$ after $prech^{\sim}\uparrow$ and $sel\uparrow$, but its pull-down network was not conducting. The schematic of the alternate, speed-independent circuit from [NS95] is thus wrong. More comments about this are given in Section 5.5.

The text in [NS95], however, does hint that the newly added transistors should be enabled during a read operation, i.e., $N5$ and $N8$ are enabled whenever $read$ is high. On the other hand, in the given schematic, this hint implies that a conducting path will be formed through $N1$ and $N2$ whenever $read$ is high, thus shorting the inverter outputs and endangering the stored data.

A way to fix this problem is to control $N1$, $N2$, $N5$, and $N8$ by dif, dit, $dit \vee read$, and $dif \vee read$, respectively, but this requires disconnecting the gates of $N1$ and $N8$, and of $N2$ and $N5$, which were shown connected in [NS95]. We verified that refinement holds for the schematic modified this way.

Since the memory output cells do not refine their interfaces, we re-do the verification using their replacement networks instead of their formalized interfaces. Refinement does hold between the overall circuit specification and the product of processes for the following items: hazard-intolerant gate-level interface specifications for all cells except the memory output cells, the chain constraints, the quickened models of the pull-down and pull-up networks of the memory output cells, the inertial models of the strong inverters of the memory output cells, and the nodes of the memory output cells (of the type in Figure 5.2). (The reasons for using these particular models were discussed above.) Since we had checked these cell interfaces against the switch-level networks, it follows that the overall circuit specification is refined by the product of the switch-level replacement networks. In particular, it follows that none of the node models stabilizes in a reject state. Thus, there are no persistent collisions and no persistent floating states under the assumptions discussed above.

5.4 Detecting Persistent Shorts

So far we have ignored persistent shorts except for the case where they are persistent collisions. (Recall that shorts occur when there are conducting paths from a circuit node both to ground and to the power supply, and collisions are particular cases of shorts where the connections to ground and power have roughly the same strengths.) In most applications we do not care about shorts that involve paths of differing strengths, thus detection of collisions would suffice. In fact, transient shorts are widely used in feedback loops with weak inverters. However, persistent shorts are sometimes undesirable because they lead to excessive power consumption.

To detect persistent shorts (not transient shorts), we forbid them altogether by means of imaginary cells that duplicate the replacement cells. The genuine replacement cells are also kept in the verified network. The duplicate cells use simplified nodes and modified path strength schemes. The process for a duplicate node is shown in Figure 5.11 (a): it does not control a voltage signal, but it forbids

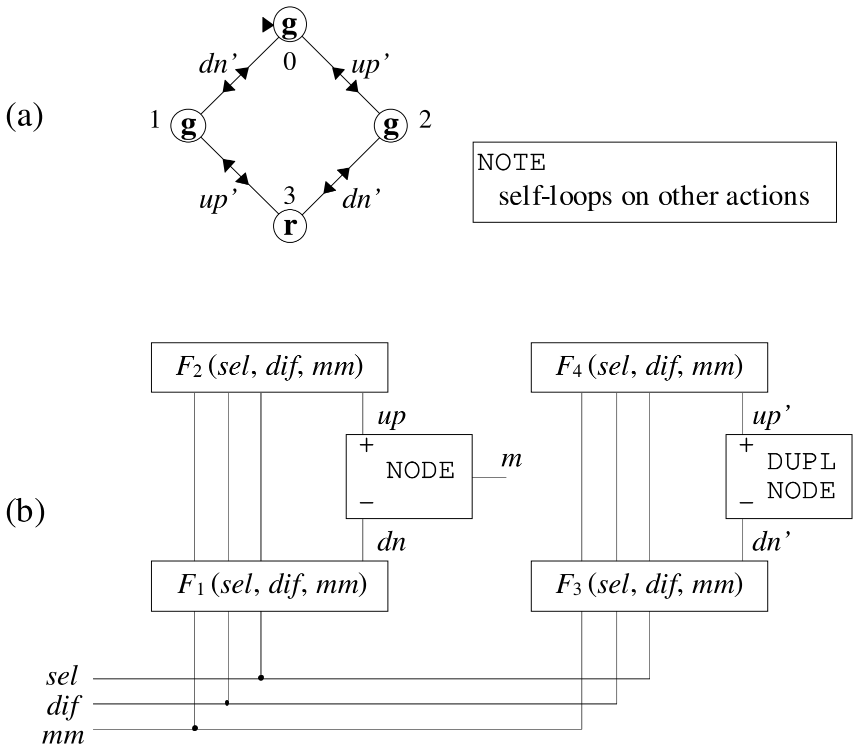


Figure 5.11: Model for detecting persistent shorts in the channel-connected subnetwork in Figure 5.9: (a) duplicate node; (b) short-detecting replacement network.

persistent collisions by declaring collisions to be rejects. A duplicate cell permits us to detect shorts as collisions at the duplicate node by declaring all paths to be of equal strength; for the pull-down network of such a cell, the path signal is thus:

$$
\begin{aligned}
&[\exists \text{ a conducting path to ground of strength } n] \\
&\vee [\exists \text{ a conducting path to ground of strength } n-1] \\
&\vee [\exists \text{ a conducting path to ground of strength } n-2] \\
&\vee \ldots
\end{aligned}
$$

and the pull-up network is determined by a similar formula in terms of paths to the power supply instead of paths to ground.

For the memory cell in Figure 5.9, the short-detecting replacement network in shown in Figure 5.11 (b). The Boolean functions the path signals are, respectively,

$$
\begin{aligned}
F_1(sel, m, mm) &= (sel \wedge dif\,) \vee mm \\
F_2(sel, m, mm) &= \neg((sel \wedge dif\,) \vee mm) \wedge \neg mm \;=\; \neg((sel \wedge dif\,) \vee mm) \\
F_3(sel, m, mm) &= (sel \wedge dif\,) \vee mm \\
F_4(sel, m, mm) &= \neg mm
\end{aligned}
$$

The short-detecting replacement network above does not refine its cell interface: if for instance *sel* and *dif* were kept high and *mm* kept low, a persistent short would occur. But can such situations happen in the context of the circuit and specified environment? We re-run the verification with respect to the overall circuit specification, after including the duplicates of the memory output cells, as well as all other cells without duplicates, and we find that refinement holds. It is not necessary to include duplicates for the other cells because their weak paths are only through the weak feedback inverters, and, due to the strong inverters, they will agree with the strong paths in any stable states. Thus, persistent shorts at these other cells can only be through strong paths, and would have been detected as persistent collisions, which, we have already verified, do not occur in the given circuit and environment. It follows there are no persistent shorts either, under the assumptions discussed.

5.5 Stocktaking

Switch-level analysis proved to be an interesting exercise for event-driven discrete-state modeling. Our main mechanism for handling diverse switch-level correctness concerns and behavior assumptions is by the family of models for charge-retaining nodes in Section 5.1; these assumptions and conditions can be strengthened or weakened in several ways. To justify our choices, we also discussed certain modeling options which we have later discarded. A promising direction for further work may be to provide a guide to which models apply for other MOS design styles.

One notable assumption was that replacement gates are much quicker than the transitions of an external signal. The quickened models in Section 5.2 incorporate this assumption, making it the responsibility of the replacement gates to avoid, through timing, these external transitions while these gates are unstable.

It was not obvious beforehand that these assumptions and conditions can be incorporated in an analysis based on concurrent processes. Notice for instance that the quickened models introduce control over actions that are external to their devices, and chain constraint processes bring additional control over actions that are controlled by cell processes. In the analysis, we often compute product and decide refinement using such models, and we use the algebraic properties for hierarchical and modular verification, which, in process spaces, do apply for these unusual connectivity cases too. For the reasons mentioned in Sections 1.2 and 4.3, we believe analysis of this kind would be more difficult to do in other models of concurrency. Due to abstract executions and absence of connectivity restrictions, we expect process spaces will provide sufficient flexibility for many such niche applications.

In our case study the cells were rather small, and we found it easier to feed in the Boolean functions of channel-connected N and P subnetworks than to describe each transistor individually and to extract such functions from a full transistor netlist. If needed, one can use the specialized method of [Bry87] to extract the behavior of the N and P subnetworks from transistor netlists, then input the results into our technique. Alternately, our technique can be extended to work directly from the transistor netlists, by using a separate process for each transistor. However, we expect our technique to find sufficient applications to circuits that have small but numerous cells, as in the case study, and we leave for future work the connections to [Bry87] and the other extensions mentioned above.

The circuit in [NS95] was verified previously, but only for a stability protocol at the gate level, and excluding the speed-dependent part; [NS95] reports almost 1/2 hour on a standard workstation for these verifications. For comparison, a FIREMAPS session for gate-level verifications covering hazard-freedom, deadlock-freedom, etc., took only 32 seconds on a Sun Sparc 5/110 workstation. A full session spanning the verifications reported in this paper both at gate-level and switch-level took just over 3 minutes (182 s) in FIREMAPS on the Sun Sparc 5/110, reaching almost 10^{11} states during the largest task (the test for persistent shorts in Section 5.4).

We found some minor flaws in the original cell interfaces (the execution in (5.2)) and in the schematic of an alternate circuit from [NS95] (the execution in (5.6)). According to [Sta97], the cell interface flaw from [NS95] was not present in the files that were originally verified, while the alternate memory design should be used with control signals different from those in Figure 5.7; for more details, see Section 5.3 above. Also, a corrected version of the cell interface for (5.2) appears in [Nie97, p. 44]. Anyway, the case study shows that our technique could detect these problems. As for the main circuit from [NS95] (not the cell interfaces or the alternate circuit), under the initial stable state assumption, the switch-level behavior assumptions, and the delay constraints above, it passed our verifications.

Chapter 6

Handling Liveness and Progress

In Chapter 3, we described finalization properties that are determined by finite total traces, which represent the entire operation of a system until it comes to a stop. Some systems, however, might never stop operating, in the sense that it is a legal behavior for them to issue new events forever. Such infinite operation is exposed to pathologies that are more subtle than illegal events or global deadlock. Accordingly, if infinite operation is of interest, one needs to impose correctness concerns that are more elaborate than safety or finalization. One of these correctness concerns, called *liveness*, is described informally in [LL90] as 'good things eventually do happen'. Let us modify this quote to describe informally another correctness concern, called *progress*, as 'good things do happen within a bounded time'.

In an event-oriented viewpoint, the 'good things' in the informal descriptions above refer to desired events. Liveness avoids that a desired event is postponed forever, and progress avoids that a desired event is postponed for an unbounded time. For instance, we might impose a fairness condition on the behavior of the etiquette machine in Example 2.1, to ensure that the machine chooses fairly between the reply greetings "How are you" and "How do you do"; this condition would be violated if the machine always responds by "How are you". Also, we might impose that the machine replies within a bounded time, which may not happen if the communication between Listener and Speaker in Example 2.5 were implemented, say, by a lossy channel over which a message is retransmitted until it is received. There may not exist any bounds on how many times a message has to be retransmitted until it gets through the channel, so such communications may have unbounded delays. Note that progress does not refer to the numerical values of bounds on delays between events, but just to the existence of such bounds; as a result, progress is a metric-free correctness concern, just like safety, finalization, and liveness. More details and examples will be provided in the following sections.

In modeling real-life systems by formal objects, a subjective step always intervenes, and, in asynchronous circuits, the problem is compounded by the absence of a standard for the behavior of common gates under hazards. For these reasons, we do not offer firm universal rules of how to attach safety, finalization, liveness, or progress specifications to real-life systems; we only provide guidelines, examples, and some applications. This means that we do not define the liveness and progress processes of a discrete-state system, because of the absence of a generally accepted model of a discrete-state system and the absence of generally accepted model of the behavior of a circuit under hazards. Instead, we introduce liveness and progress processes by examples in Sections 6.1 and 6.2, we propose a classification of liveness and progress faults in Section 6.3. Moreover, in Section 6.4 we attach default liveness and progress processes to simpler finitary specifications by guessing what their liveness and progress properties might be, as these simpler specifications do not explicitly determine such properties. However, although our default processes appear to correspond to our intuition of liveness and progress properties for many asynchronous circuits and other discrete-state systems, this is not always the case, as we discuss in Subsection 6.4.4. Where these default processes are not appropriate, one may employ the more general liveness and progress processes we discuss first; hence the label 'default' for the processes we later extract from the simpler specifications.

In Chapter 3, we studied processes that use only finite traces as executions. Here, we also include infinite traces in execution sets along with finite traces, and this permits us to study both liveness and progress.

For language $L \subseteq \mathcal{U}^*$, we denote by L^ω the set of sequences obtained by concatenating infinitely many words from $L \setminus \{\varepsilon\}$, and by L^∞ the set of sequences obtained by concatenating finitely or infinitely many words from L. Thus, $L^\infty = L^* \cup L^\omega$. This notation for L^∞ and L^ω is the usual; see for instance [Tho90].

For sequence $u \in \mathcal{U}^\infty$, language $L \subseteq \mathcal{U}^\infty$, and alphabet $\Gamma \subseteq \mathcal{U}$, let the *projection* of u on Γ be a sequence $u{\downarrow}\Gamma \in \mathcal{U}^\infty$ obtained by deleting from u all occurrences of symbols from outside Γ, and let the *projection* and *extension* of L on Γ be languages $L{\downarrow}\Gamma \subseteq \mathcal{U}^\infty$ and $L{\uparrow}\Gamma \subseteq \mathcal{U}^\infty$, as follows:

$$\begin{aligned} &L{\downarrow}\Gamma = \{u{\downarrow}\Gamma \mid u \in L\}, \text{ and} \\ &L{\uparrow}\Gamma = \{u \in \mathcal{U}^\infty \mid u{\downarrow}\Gamma \in L\} \ . \end{aligned}$$

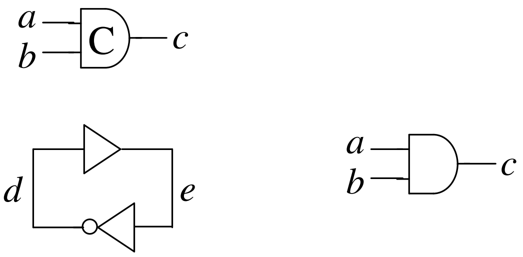
For example, if $\mathcal{U} = \{a, b, c\}$ then $abba^\omega{\downarrow}\{a, c\} = a^\omega$, $abab{\downarrow}\{c\} = \varepsilon$, $abba{\uparrow}\{a, b\} = c^*ac^*bc^*bc^*ac^\infty$, and $abba{\uparrow}\{a, c\} = \emptyset$. (A word u is sometimes used to denote the language $\{u\}$. Unary operators, like Kleene star and ω-closure, bind more strongly than binary operators, such as concatenation.)

For sequences t and e, we write $t \leq e$ if t is a finite prefix of e (i.e., any prefix of e except, if e is infinite, e itself). We refer to finite prefixes as just prefixes. The *prefix-closure* of language L is the set **pref** L of all prefixes of words from L. A *limit* of a language L is a sequence e such that for every prefix of e, there is a longer prefix of e that is in L. The set of limits of a language L is denoted by **lim** L, which is $\{e \in \mathcal{U}^\infty \mid \forall\, u \leq e : \exists\, v \in L : u \leq v \leq e\}$; that is, finite limits are words from L, and infinite limits have infinitely many prefixes from L. Also see sets of infinite limits in [Tho90].

6.1 Liveness

In Chapter 3, we used finite total traces to represent the operation of a system from the time when the system starts to operate until it comes to a legal or illegal stop. In general, we say a (finite or infinite) trace is *complete* if it represents an observation of the entire operation of a system since start, regardless of whether the system stops or not. For studying liveness, we consider that a discrete-state system s is represented by its *liveness process* over $\mathcal{U}^\infty$, denoted by convention as λs, that classifies each trace as accessible or acceptable according to whether it may occur as a complete trace of a system. Note that all operators and algebraic properties of process spaces are inherited for liveness processes, by taking the execution set to be $\mathcal{U}^\infty$.

Example 6.1 The refinement relationship on liveness processes can be used to detect faults such as deadlock and unfairness in a digital circuit. In this example, we check whether the circuit in Figure 6.1 (a) is a correct implementation of the AND gate in Figure 6.1 (b). Clearly, the C-element does not implement an AND gate, but there is a catch. Notice that the circuit also includes an oscillating loop of a buffer and an inverter, which means that the operation of the circuit never



(a) (b)

Figure 6.1: Example circuit and gate.

stops, and recall that finalization processes handle only cases where the systems do come to a stop. Thus, we need to use processes that make finer distinctions than the finalization processes; in this example, we use liveness processes.

Let C (a C-element), BUF (a buffer), and INV (an inverter) denote the three components in Figure 6.1 (a) (from top to bottom). In determining the liveness processes we will consider the liveness properties of the components, as explained below.

Let $\mathcal{U} = \{a, b, c, d, e\}$. We take

$$\textbf{as}\,\lambda\text{BUF} = ((de)^\infty \cup (de)^* dd\{d, e\}^\infty) \uparrow \{d, e\}\ ,$$

$$\textbf{at}\,\lambda\text{BUF} = ((de)^\infty \cup (de)^* d \cup (de)^* e\{d, e\}^\infty) \uparrow \{d, e\}\ .$$

Notice that λ BUF is quite similar to a finalization process. The finite words in $(de)^*$ are goals of λBUF because, at any time, the environment of BUF may stop producing d events. The infinite word $(de)^\omega$ is a goal of λBUF because the environment and BUF need not stop at all. The words in $(de)^* dd\{d, e\}^\infty$ are rejects and are accessible to BUF. After two consecutive d events, which are not expected, BUF may stop at any time or may not stop at all. The fact that subsequent d or e events may come in any order means that this particular BUF contract does not care about what might happen after a hazard; in this particular case, a hazard is qualified as the fault of the environment no matter what happens afterwards. The words in $(de)^*d$ are escapes, since BUF should eventually produce an e after receiving a d. The words in $(de)^*e\{d,e\}^\infty$ are also escapes, since the environment does not expect two consecutive e events. Notice that the escapes in $(de)^*e\{d,e\}^\infty$ are not only safety faults, but also liveness faults, because, after two consecutive e events, the environment may demand, for instance, events that BUF cannot guarantee to produce (e.g., events whose actions are not in the output alphabet of BUF).

By similar considerations, we take

$$\begin{aligned}\mathbf{as}\,\lambda\text{INV} &= ((de)^*d \cup (de)^\omega \cup (de)^*e\{d,e\}^\infty) \uparrow \{d,e\}\ ,\\ \mathbf{at}\,\lambda\text{INV} &= (\mathbf{pref}\,(de)^\infty \cup (de)^*dd\{d,e\}^\infty) \uparrow \{d,e\}\ .\end{aligned}$$

For the C-element, we use the following liveness process:

$$\begin{aligned}\mathbf{as}\,\lambda\text{C} = {} & ((aa \cup bb \cup abc \cup bac)^\omega \cup (aa \cup bb \cup abc \cup bac)^* \cdot (\varepsilon \cup a \cup b)\\ & \cup (aa \cup bb \cup abc \cup bac)^* \cdot (ab \cup ba) \cdot (a \cup b)(\{a,b,c\}^\infty) \uparrow \{a,b,c\}\ ,\\ \mathbf{at}\,\lambda\text{C} = {} & (\mathbf{pref}\,(aa \cup bb \cup abc \cup bac)^\infty\\ & \cup (aa \cup bb \cup abc \cup bac)^* \cdot (c \cup ac \cup bc)(\{a,b,c\}^\infty)) \uparrow \{a,b,c\}\ .\end{aligned}$$

This liveness process is quite similar to that of the buffer: the words in $((ab \cup ba)c)^*$ are goals for λC, because the environment may stop producing a or b events; the words in $((ab \cup ba)c)^\omega$ also are goals, since the communication between the environment and the C-element needs not stop at all; etc.

Let $u = abca(de)^\omega$. We have $u \in \mathbf{as}\,\lambda\text{C} \cap \mathbf{as}\,\lambda\text{BUF} \cap \mathbf{as}\,\lambda\text{INV}$. On the other hand, $u \notin \mathbf{as}\,\lambda\text{AND}$ because AND should eventually produce a second c after the second a (the a signal becomes low, thus the c signal should eventually become low). Thus, we have $\mathbf{as}\,(\lambda\textsc{c} \times \lambda\textsc{buf} \times \lambda\textsc{inv}) \not\subseteq \mathbf{as}\,\lambda\textsc{and}$, and, therefore, $\lambda\textsc{and} \not\sqsubseteq \lambda\textsc{c} \times \lambda\textsc{buf} \times \lambda\textsc{inv}$.

The violation can be interpreted as follows. Word u appears to cause a deadlock, since a 'wait-for cycle' occurs, involving C and the environment of AND. After $abca$, C waits for the environment of AND to send another input event, while the environment of AND requires C to produce an output event.

Note that deadlock occurs in parts of the system despite the fact that some other parts never stop producing events d and e. Although the liveness processes in this example were quite similar to the hazard-intolerant finalization processes described in Section 4.1, notice that this fault would not be detected by checking robustness or refinement on the finalization processes of the entire system: every finite word over $\{a, b, c, d, e\}$ is either an escape of $\varphi\textsc{buf}$ or an escape of $\varphi\textsc{inv}$, and thus an escape for the product of the finalization processes in the system. In this sense, finalization processes would not be helpful for examining the system altogether, because they only deal with situations where the system comes to a stop, whereas a part of this system never comes to a stop. For this reason, we resort to infinite traces and liveness processes to detect such local deadlock in a non-stopping system.

To detect the deadlock above, one may alternately apply a finalization analysis for part of the system only (note that $\varphi\textsc{c} \not\sqsubseteq \varphi\textsc{and}$), but that is difficult to generalize for arbitrary systems where the deadlocked part may not be so clearly isolated from the rest of the system. By contrast, the liveness approach taken above does not rely on the structural peculiarities of this system, and can be applied in general. □

```
task body P0 is
begin
   loop
      select
         Critical_Section_1;
      or
         Critical_Section_2;
      end select;
   end loop;
end P0;
```

Figure 6.2: Specification for starvation example.

Example 6.2 Let us consider a classical example of starvation adapted from [BA90, p. 35]. The concurrent program in Figure 6.3 attempts to ensure mutual exclusion between the critical sections of tasks `P1` and `P2` by using variables `C1` and `C2`. `C1` and `C2` are initially set at 1. We state the specification as the task `P0` in Figure 6.2. Let the actions be

r_{xyz} = `P`x reads from `C`y value z ,

w_{xyz} = `P`x writes in `C`y value z ,

ecs_x = enter `Critical_Section_`x ,

lcs_x = leave `Critical_Section_`x ,

$encs_x$ = enter `Non_Critical_Section_`x ,

$lncs_x$ = leave `Non_Critical_Section_`x .

```
C1, C2:  Integer range 0..1 := 1;
```

```
task body P1 is
begin
  loop
    Non_Critical_Section_1;
    C1 := 0;
    loop
    exit if C2 = 1;
      C1 := 1;
      C1 := 0;
    end loop;
    Critical_Section_1;
    C1 := 1;
  end loop;
end P1;
```

```
task body P2 is
begin
  loop
    Non_Critical_Section_2;
    C2 := 0;
    loop
    exit if C1 = 1;
      C2 := 1;
      C2 := 0;
    end loop;
    Critical_Section_2;
    C2 := 1;
  end loop;
end P2;
```

Figure 6.3: Implementation for starvation example.

Consider infinite trace

$$u = encs_2 lncs_2 (encs_1 lncs_1 w_{110} r_{121} w_{220} r_{210} ecs_1 lcs_1 w_{111} w_{221})^\omega \ .$$

Word u is an accessible complete trace for `P1`, because `P1` executes its main loop, in which it can stay forever. Word u is also an accessible complete trace for `P2`, because `P2` executes its inner loop and reads `C1 = 1` at every iteration, and thus can stay in its inner loop forever. On the other hand, we take u not to be an accessible complete trace for `P0`, because `P0` executes a non-deterministic choice in its loop (between ecs_1 and ecs_2), and, for any non-zero probability of ecs_2, `P0` should eventually choose ecs_2. Thus, $u \in \textbf{as}\,(\lambda\texttt{P1} \times \lambda\texttt{P2})$ and $u \notin \textbf{as}\,\lambda\texttt{P0}$. Consequently, we have $\textbf{as}\,\lambda\texttt{P0} \not\supseteq \lambda\textbf{as}\,(\lambda\texttt{P1} \times \lambda\texttt{P2})$ and thus $\lambda\texttt{P0} \not\sqsubseteq \lambda\texttt{P1} \times \lambda\texttt{P2}$.

The violation can be interpreted as follows. Word u causes starvation because it never allows `P2` to enter its critical section. Starvation is also widely known as unfairness. □

6.2 Progress

In Section 6.1, we interpret a finite or infinite complete trace as the entire operation of a system since start. For studying progress, we use a finite or infinite trace to represent partial observations of a system from start until arbitrarily late moments of time, but not necessarily the entire operation of the system. In the context of this interpretation, we refer to traces as *unbounded*. Loosely speaking, an unbounded execution will be considered 'legal' if, for any time bound, that executions has a prefix that is a legal partial execution and that can take longer than the bound.

Example 6.3 To illustrate unbounded traces by contrast to complete traces, consider the etiquette machine of Example 2.1 with an additional fairness stipulation that the machine should choose fairly between the two polite reply greetings $m1$ and $m2$, using the shorthands in Figure 2.4 (a). Execution $(e1m1)^\infty$ should be an escape for the liveness process, since the machine should reply with $m2$ from time to time. On the other hand, this execution should be a goal for the progress process, because partial observations of legal behavior until arbitrarily late moments in time are represented by finite prefixes $(e1m1)^*$ or $(e1m1)^*e1$ of this execution. □

For studying progress, we consider that a discrete-state system s is represented by its *progress process*, over $\mathcal{U}^\infty$, denoted by convention as πs, that qualifies each trace as accessible or acceptable according to whether it may occur as an unbounded trace of a system. The main usage of such specifications is to ensure that certain events will occur within a bounded time, whatever the bound may be. This way, we can reason about the existence of upper bounds on certain delays, but we do not need to know what the bounds are. Consequently, progress processes do not incorporate numerical information, and progress is a metric-free, behavior concern. Also note that all operators and algebraic properties of process spaces are inherited for progress processes, by taking the execution set to be $\mathcal{U}^\infty$, as we did for liveness.

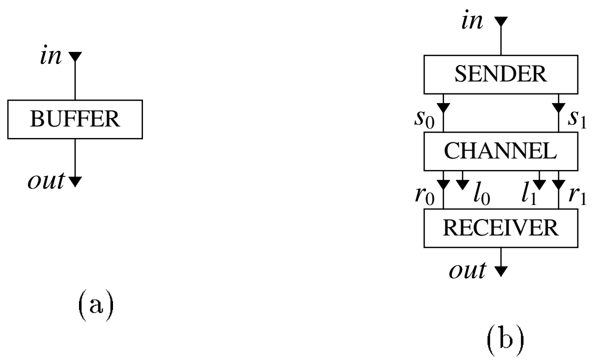


Figure 6.4: Communication protocol for livelock example.

Example 6.4 Consider the following communication protocol. The specification is a buffer, as in Figure 6.4 (a). After an input message *in*, an output message *out* follows within a bounded delay, and then the operation may be repeated indefinitely. The implementation uses a lossy channel with two interfaces, as in Figure 6.4 (b). After *in*, SENDER starts sending messages s_0. Some of these messages may be lost, causing l_0 events which reset CHANNEL and are not seen by RECEIVER. (The l_0 events are not used by the system; they only model the loss of a message.) Some messages may get through, causing r_0 events to occur. After an r_0, RECEIVER issues *out*. According to the specification, another *in* is now allowed (there exists some feedback from *out* to *in* in the environment). The operation is then repeated, with s_1, l_1, and r_1 instead of s_0, l_0, and r_0, and so on.

Consider $u = in(s_0 l_0)^\omega$. Word u is an accessible unbounded trace of SENDER, because the environment is under no obligation to provide another *in*. It is also an accessible unbounded trace of CHANNEL: although, presumably, CHANNEL cannot choose l_0 over r_0 infinitely many times, CHANNEL can choose l_0 any finite number of times. Thus, CHANNEL can follow u for an unbounded time. Word u is also an accessible unbounded trace of RECEIVER, since RECEIVER remains in its initial state, where it is not expected to issue an output event. Since $u \in \mathbf{as}\,\pi\mathrm{SENDER} \cap \mathbf{as}\,\pi\mathrm{CHANNEL} \cap \mathbf{as}\,\pi\mathrm{RECEIVER}$, we have $u \in \mathbf{as}\,(\pi\mathrm{SENDER} \times \pi\mathrm{CHANNEL} \times \pi\mathrm{RECEIVER})$. On the other hand, u is not an accessible unbounded trace of BUFFER, because an *out* event must follow *in* within a bounded time. Hence, $u \notin \mathbf{as}\,\pi\mathrm{BUFFER}$. Therefore, $\pi\mathrm{BUFFER} \not\sqsubseteq \mathbf{as}\,(\pi\mathrm{SENDER} \times \pi\mathrm{CHANNEL} \times \pi\mathrm{RECEIVER})$.

The violation can be interpreted as follows. Word u causes a livelock. After *in*, SENDER starts producing s_0 events. For fairness, CHANNEL will eventually choose r_0; however, this choice can be delayed for any amount of time. On the other hand, the specification demands an *out* within a bounded delay, which cannot be granted, regardless of the value of the bound. □

Example 6.5 The deadlock fault in Example 6.1 also constitutes a violation of progress. The progress processes of the components in Example 6.1 are exactly the same as their liveness processes.[1] Hence, the fault in Example 6.1 can be detected with progress processes in the same manner as with liveness processes. □

6.3 A Taxonomy of Lock Faults

The fact that the progress execution set is the same as the liveness execution set (namely, $\mathcal{U}^\infty$) invites a comparison between liveness and progress. By this comparison, we obtain a classification of liveness and progress faults (call them *lock faults*), in absolute and relative versions, which appears to generalize intuitive notions of deadlock, starvation, and livelock. Although these notions are not disputed in the literature at an informal level, the formalizations proposed previously are often restricted to particular cases, and we are not aware of a formal classification related to ours. First, we describe informally these notions.

A widespread view of deadlock is the following definition from [Tan92, p. 242]:

> A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

This definition refers to a particular notion of processes, as software entities that run in an operating system. In more general settings where processes are formal

[1] This does not always hold for other components. For instance, execution u from Example 6.4 is in $\mathbf{as}\,\pi$CHANNEL but not in $\mathbf{as}\,\lambda$CHANNEL.

objects, deadlock is considered to be a state from which no further action is possible (see for instance [Hoa85], [Mil89], or [BS95]) or a state where all processes can stop but some processes demand further actions [Ver94b]. On the other hand, deadlock detection in a system of processes involves not only detecting situations where all processes come to a stop, but also situations where only a subset of processes are deadlocked. Moreover, the deadlocked processes might perform some other actions while waiting: they may have internal clocks, or they may conduct external communications with operating system processes. Therefore, formalizing deadlock as a single state can detect global deadlocks where an entire system comes to a grinding stop, but is not convenient for detecting deadlocks that involve only part of a system while other actions can happen indefinitely; see Example 6.1. Although such localized deadlock is sometimes given other names (such as "livedeadlock" in [Sme90]), it does make the object of well-known deadlock detection algorithms (see for instance [Tan92, p. 247]) and for us it is just a particular case of deadlock.

A typical starvation situation (also called "infinite overtaking" in [Hoa85, p. 80]) is where a process chooses among several options infinitely many times, but at least one of the available options is never taken. Starvation is usually formalized by fairness properties; see [Dil89, p. 138] for an example. As in [Dil89], we can detect both deadlock and starvation by means of liveness processes; see Example 6.2. In addition, we propose to distinguish between starvation and deadlock by means of progress processes.

A typical livelock situation is the possibility of an action occurring arbitrarily late, whereas that action is expected to occur 'reliably' within some bounded delay. Livelock is often caused by unbounded internal communication (see Example 6.4) but not every case of unbounded internal communication fits our notion of livelock. In fact, we do not consider unbounded internal communication to be a fault by itself. For example, in many digital circuits an internal clock may tick an unbounded number of times while waiting for an input; this is not a faulty behavior, because the circuit is not expected to ensure that input events occur at all. Both livelock and deadlock can be detected by progress processes. However, we do not agree with [BA90, p. 34] that livelock is a form of deadlock, and we distinguish between livelock and deadlock by means of liveness processes.

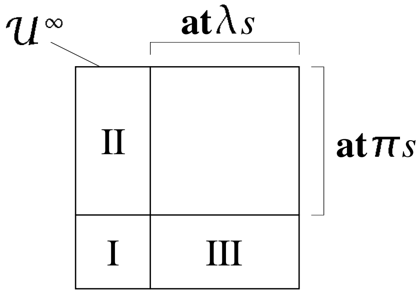


Figure 6.5: Absolute lock faults.

Let s be a discrete-state system. For robustness, we require that, for all $u \in \mathcal{U}^\infty$, we have $u \in \mathbf{at}\,\lambda s$ and $u \in \mathbf{at}\,\pi s$. This condition can be violated in three ways: (i) $u \notin \mathbf{at}\,\lambda s$ and $u \notin \mathbf{at}\,\pi s$; (ii) $u \notin \mathbf{at}\,\lambda s$ but $u \in \mathbf{at}\,\pi s$; and (iii) $u \in \mathbf{at}\,\lambda s$ but $u \notin \mathbf{at}\,\pi s$. We call these situations *deadlock*, *starvation*, and *livelock*, respectively. In the Venn diagram in Figure 6.5 (a), they correspond to the regions marked 'I', 'II', and 'III', respectively. In this formalization, there are no other liveness or progress faults.

Lock faults can also be considered in a relative sense, whereby two discrete-state systems s_1 and s_2 are compared. System s_1 can be thought to be a specification, and s_2 an implementation. For s_2 to be better or as good as s_1 with respect to liveness and progress, we require $\lambda s_1 \sqsubseteq \lambda s_2$ and $\pi s_1 \sqsubseteq \pi s_2$, i.e., $\mathbf{as}\,\lambda s_1 \supseteq \mathbf{as}\,\lambda s_2$, $\mathbf{as}\,\pi s_1 \supseteq \mathbf{as}\,\pi s_2$, $\mathbf{at}\,\lambda s_1 \subseteq \mathbf{at}\,\lambda s_2$, and $\mathbf{at}\,\pi s_1 \subseteq \mathbf{at}\,\pi s_2$. A word can be in sixteen

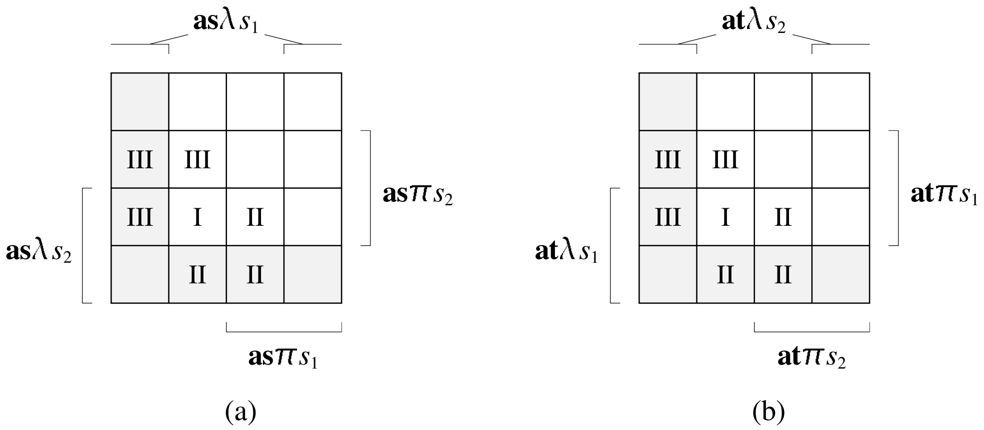


Figure 6.6: Relative lock faults.

positions with respect to **as** λs_1, **as** λs_2, **as** πs_1, and **as** πs_2, as shown in the diagram in Figure 6.6 (a). (Figure 6.6 (b) is similar to Figure 6.6 (a), except that s_1 and s_2 have been swapped.) We define *relative deadlock*, *relative starvation*, and *relative livelock* as the existence of a word in a region marked 'I', 'II', and 'III', respectively, in either Figure 6.6 (a) or (b).

For instance, the fault in Example 6.1 and Example 6.5 is a relative deadlock; the fault in Example 6.2 is relative starvation; and, the fault in Example 6.4 is a relative livelock. Denoting by s_1 the respective specifications and by s_2 the respective implementations, we have: in Example 6.1 and Example 6.5, $u \in$ **as** λs_2, $u \in$ **as** πs_2, $u \notin$ **as** λs_1, and $u \notin$ **as** πs_1; in Example 6.2, $u \in$ **as** λs_2, $u \in$ **as** πs_2, $u \notin$ **as** λs_1, and $u \in$ **as** πs_1 (we consider that the execution point of the specification can follow u for an unbounded time because the option ecs_1 can be chosen by task `P0` any finite number of times in a row); and, in Example 6.4, $u \notin$ **as** λs_2, $u \in$ **as** πs_2, $u \notin$ **as** λs_1, and $u \notin$ **as** πs_1.

Relationships between relative and absolute lock faults follow from the relationship between refinement and robustness in Theorem 2.12. For example, a relative

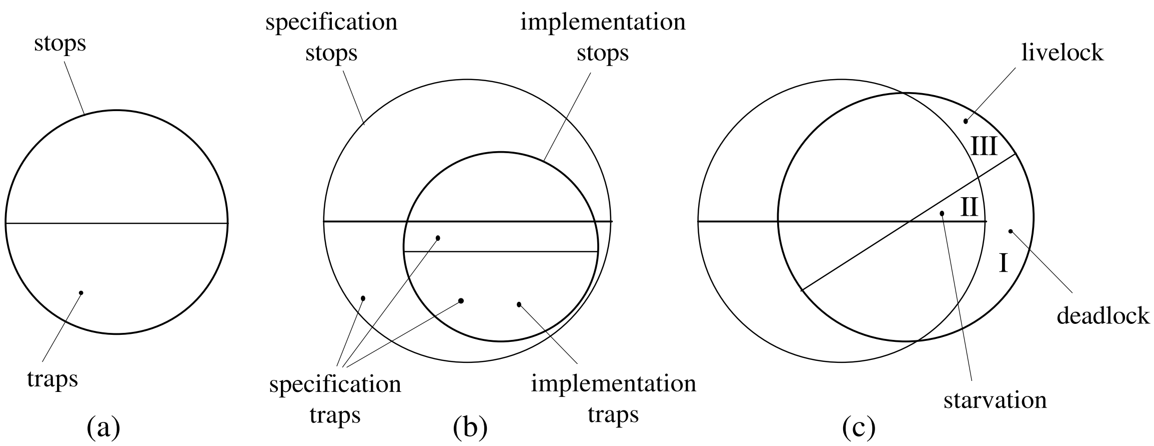


Figure 6.7: Aspirins for lock headaches: relative lock faults revisited.

deadlock can be viewed as a deadlock between an implementation and the environment of a specification.

In many discrete-state systems, every complete trace is also an unbounded trace. Therefore, we often have, for discrete-state system s, $\mathbf{as}\,\lambda s \subseteq \mathbf{as}\,\pi s$ and $\mathbf{at}\,\lambda s \subseteq \mathbf{at}\,\pi s$. In such situations, the shaded areas in Figure 6.6 (a) and (b) are empty, and the figure can be redrawn in a more suggestive manner.

Let us call *traps* the accessible complete traces of a system, and *stops* the accessible unbounded traces. The aspirin-shaped diagram in Figure 6.7 (a) illustrates a subset relationship that usually holds between traps and stops. Figure 6.7 (b) illustrates a situation without such lock faults, where all implementation traps are specification traps as well, and all implementation stops are specification stops too. This is a re-drawn version of the diagrams in Figure 6.6, omitting the shaded areas and the intersections marked 'I', 'II', and 'III'. Figure 6.7 (c) illustrates the general case, where there may be lock faults; this corresponds to the diagrams in Figure 6.6, without the shaded areas but including areas 'I', 'II', and 'III'.

6.4 Implicit Liveness and Progress Properties

A major obstacle to verifying liveness or progress is the amount of information needed by the verification procedure. The difficulty is that common descriptions of discrete-state systems, such as finite automata, Petri nets, or digital circuit schematics together with, say, logic relationships between the inputs and outputs of each component, specify only the finite traces of the systems to verify, while two systems with different liveness properties can have the same finite traces.

In this situation it may seem natural to extend the model with some representation of infinite traces, and to ask the users to specify the liveness or progress properties of their systems explicitly, as done for the liveness and progress processes above. Such *user-directed* approaches have a high degree of generality, because they allow many types of properties to be specified. However, such an approach can be tedious and error-prone, and provides no indications of *appropriateness* and *completeness* of a specification. In other words, such approaches do not address the problems whether the liveness or progress properties specified by the users, are necessary, and whether they are sufficient to forbid, say, the danger of starvation in a particular implementation. (This stumbling point is also mentioned in [CMP92].)

To illustrate how liveness and progress properties are easy to overlook and hard to specify explicitly, let us consider a gate specified by a Boolean function. Even if we remember to specify that the output of the gate should eventually respond to an input excitation, we might forget to specify that the response should occur within a bounded time. Also, consider a mutual exclusion element which ensures that two processors do not access the same resource at the same time. Trivial implementations are an arbiter that serves none of the processes [CMP92] and an arbiter that always favors one of the processes.

In this section, we consider a different approach to resolve the ambiguity of finite-trace models. Often, liveness and progress properties seem to be *implicitly assumed*, like bounded-time response and fairness in the examples above: we don't normally use Boolean gates that can have unbounded delays or arbiters that can be unfair. Accordingly, one can attach unique liveness and progress properties to a finite-trace description, to match these implicitly assumed properties. If these attached properties do indeed correspond to the intended liveness and progress properties of a class of systems, then the users need only to provide finite-trace descriptions for systems in that class, and we can extract implicitly assumed liveness and progress properties to check freedom from deadlock, starvation, and livelock solely on the basis of finite-trace descriptions.

Also see [NB95] and [NB98].

6.4.1 Finite-Trace Structures

We extract the implicit liveness and progress properties from a particular form of finite-trace descriptions, called *trace structures*. We use trace structures instead of safety and finalization processes because of some particular features (input and output alphabets) that are instrumental in defining the implicit properties in question. Future research, however, should eliminate this dependence on particular features and extend the definition of implicit properties to other types of systems, such as ones that have bi-directional ports (i.e., ports that are sometimes inputs and other times outputs).

A *trace structure* is a triple $P = \langle \mathbf{i}P, \mathbf{o}P, \mathbf{lg}P \rangle$ of two disjoint alphabets $\mathbf{i}P$ and

$\mathbf{o}P$ and a prefix-closed, non-empty language $\mathbf{lg}P$ over $\mathbf{i}P \cup \mathbf{o}P$. The words of $\mathbf{lg}P$ are called *traces* of P. The *alphabet* of P, denoted by $\mathbf{a}P$, is $\mathbf{i}P \cup \mathbf{o}P$. The symbols in $\mathbf{a}P$ are called *actions* of P. For sequence $e \in \mathcal{U}^\infty$ and trace structure P, e_P is a shorthand for $e\!\downarrow\!\mathbf{a}\,P$.

A trace structure P can represent a process in the following manner. Symbols in $\mathbf{a}P$ stand for ports. Symbols in $\mathbf{o}P$, called *outputs*, are ports controlled by the process; they include the internal ports and the genuine output ports. Symbols in $\mathbf{i}P$, called *inputs*, represent ports controlled by the environment. Traces in $\mathbf{lg}P$ stand for finite sequences of events that may have occurred in the modeled process up to a certain time, thus they correspond to goals of safety processes. This interpretation led to the restriction that $\mathbf{lg}P$ be prefix-closed, which we take for trace structures in order to facilitate the definitions of default liveness and progress processes. However, for general safety processes we do not impose this restriction, since we do not need it in the algebraic treatment, and it is always possible that new systems or viewpoints may be discovered where the safety goal sets should not be prefix-closed.

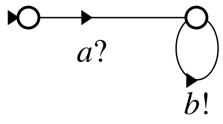
Example 6.6 The trace structure for the C-element in Example 6.1 is

$$\langle\{a,b\},\{c\},((ab \cup ba)c)^*(\varepsilon \cup a \cup b)\rangle$$

$\square$

A *limit* of trace structure P is a limit of $\mathbf{lg}\,P$. For trace structure P, let $\mathbf{lim}\,P$ be the set of limits of $\mathbf{lg}\,P$. We have $\mathbf{lg}\,P \subseteq \mathbf{lim}\,P$.

Although trace structures can model non-deterministic processes as illustrated in Subsection 6.4.2, trace structures that have regular languages can be represented by deterministic automata as follows. A *behavior automaton* is a tuple



unused: $c!$

Figure 6.8: A behavior automaton.

$A = \langle \mathbf{i}A, \mathbf{o}A, \mathbf{st}A, \mathbf{ed}A, \mathbf{init}A \rangle$ such that $\mathbf{i}A$ and $\mathbf{o}A$ are finite and disjoint subsets of $\mathcal{U}$, $\mathbf{st}A$ is a finite and non-empty set of *states*, $\mathbf{init}A \in \mathbf{st}A$ is the *initial state*, and $\mathbf{ed}A \subseteq \mathbf{st}A \times (\mathbf{i}A \cup \mathbf{o}A) \times \mathbf{st}A$ is a (finite) set of labelled *edges* such that no two edges leaving the same state have the same label. If (q, b, q') is an edge, then b is its *label*. Note that some symbols of $\mathbf{i}A$ or $\mathbf{o}A$ might not actually appear on edges.

In figures, behavior automata have an initial state marked with an incoming arrow, and edges marked with actions. Inputs have a '?' sign and outputs have a '!' sign. Inputs or outputs that do not appear on any edges are listed below the automaton, with the annotation 'unused'.

The *language* of a behavior automaton is the set of all traces spelled by finite paths that start at the initial state. Note that the language of a behavior automaton is always prefix-closed and contains ε. For example, the language of the behavior automaton Figure 6.8 is and $\mathbf{pref}(ab^*)$.

6.4.2 Implicit Liveness Properties

In formalizing the implicitly assumed complete traces of a discrete-state system, we have obtained a property that unifies a strong fairness property of infinite traces (e.g., see [Fra86]) with a quiescence property of finite traces (e.g., see [Jon87]). The property is formally the same for infinite and finite sequences, but, for clarity, the

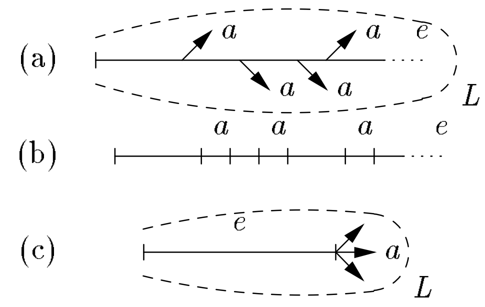


Figure 6.9: Recurrently enabled and fired symbols.

intuitive explanations are given separately for the two cases.

Symbol a is *recurrently enabled* by sequence e with respect to language L if $\forall\, t \leq e,\ \exists\, u : tu \leq e \wedge tua \in L$. The *set of recurrently enabled symbols* of e with respect to L is denoted by $\mathbf{ren}_L e$. Finite sequence t *immediately enables* a symbol a in language L if $ta \in L$. Note that, if e is infinite, the recurrently enabled symbols of e with respect to L are those symbols that are immediately enabled in L by infinitely many prefixes of e. See Figure 6.9 (a). If e is finite, the recurrently enabled symbols of e with respect to L are the symbols immediately enabled by e in L. See Figure 6.9 (c). For example, $\mathbf{ren}_{\mathbf{pref}\,((ab)^*c)}(ab)^\omega = \{a, b, c\}$ and $\mathbf{ren}_{\mathbf{pref}\,((ab)^*c)}ab = \{a, c\}$.

Symbol a is *fired* by sequence e if a appears in e at least once. Symbol a is *recurrently fired* by e if $\forall\, t \leq e,\ \exists\, u : tua \leq e$. The *set of recurrently fired symbols* of e is denoted by $\mathbf{rfi}\, e$. Note that the recurrently fired symbols are exactly the symbols fired infinitely often. See Figure 6.9 (b). For every finite sequence e, $\mathbf{rfi}\, e = \emptyset$, i.e., e has no recurrently fired symbols. For example, $\mathbf{rfi}\, ac(ab)^\omega = \{a, b\}$ and $\mathbf{rfi}\, aba = \emptyset$.

For alphabet Σ and language L, limit e of L is *strongly live* with respect to Σ and L if e recurrently fires all symbols from Σ that e recurrently enables in L, i.e., if $\mathbf{ren}_L e \cap \Sigma \subseteq \mathbf{rfi}\, e$. For infinite e, strong liveness is the same as strong fairness. For finite e, strong liveness means 'no symbols from Σ are enabled at the end of e'.

Limit e of trace structure P is an *output trap* of P if e is strongly live with respect to $\mathbf{o}\, P$ and $\mathbf{lg}\, P$. The *set of output traps* of P is denoted by $\mathbf{otp}\, P$. (Note that $\mathbf{otp}\, P \subseteq \mathbf{lim}\, P$, and that the set of output traps of a trace structure is uniquely determined by its language and alphabets.) Output traps formalize our idea of implicitly assumed accessible complete traces of a system. For an intuitive picture, consider that the execution point of a system follows limit e. The recurrently enabled output symbols can be viewed as exerting a pressure to be fired by the process; that pressure is relieved for recurrently fired symbols only. If an output symbol a is recurrently enabled but is not recurrently fired by e, the pressure builds up and e is not complete because an a event is due to be fired by the process.

Example 6.7 Consider an elementary arbitration element called SELECTOR in [Ebe91b]: $\langle \{a\},\ \{b, c\},\ \mathbf{pref}\,(a(b \cup c))^* \rangle$. The set of output traps is $(a(b \cup c))^*$ $\cup$ $\{e \in \{ab, ac\}^\omega \mid e$ fires b infinitely many times and e fires c infinitely many times$\}$. The finite limits from $(a(b \cup c))^*$ are output traps because they do not immediately enable any output symbols. The infinite limits that fire both b and c infinitely many times are output traps because b and c are the only outputs and are recurrently fired. The remaining finite limits, those in $(a(b \cup c))^* a$, are not output traps because they immediately enable b and c. The remaining infinite limits are not output traps because they cease to fire one of the outputs after some finite prefix, but they recurrently enable both outputs. Intuitively, the remaining finite limits owe an output event and the remaining infinite limits are unfair to either b or c. $\square$

The following definition associates our implicitly assumed liveness properties to a trace structure.

Definition 6.1 *The* default liveness process *of trace structure P is a process $\lambda_d P$ over $\mathcal{U}^\infty$ such that*

$$\begin{aligned} \mathbf{as}\,\lambda_d P &= (\mathbf{otp}\,P \cup ((\mathbf{lg}\,P \cdot \mathbf{i}\,P)\backslash\mathbf{lg}\,P) \cdot (\mathbf{a}\,P)^\infty){\uparrow}\mathbf{a}\,P \\ \mathbf{at}\,\lambda_d P &= (\mathbf{lim}\,P \cup ((\mathbf{lg}\,P \cdot \mathbf{o}\,P)\backslash\mathbf{lg}\,P) \cdot (\mathbf{a}\,P)^\infty){\uparrow}\mathbf{a}\,P \ . \end{aligned}$$

The goal set of $\lambda_d P$ is thus $(\mathbf{otp}\,P){\uparrow}\mathbf{a}\,P$; output traps are both accessible and acceptable as complete traces. Limits of $\mathbf{lg}\,P$ that are not output traps are escapes for $\lambda_d P$, meaning that the device is supposed to ensure that every recurrently enabled output of P is recurrently fired; in other words, the device should not stop in a state where an output is due, and, if there are infinitely many opportunities to fire an output, the device should not ignore that output forever. (These implicit liveness properties appear to fit well many practical systems, but there is no guarantee that they would always hold; where the implicit liveness properties are inappropriate, the general liveness process from Section 6.1 have to be used instead of $\lambda_d P$.) Since $\mathbf{lg}\,P$ is prefix-closed and non-empty, it contains ε. Every finite or infinite word e from outside $\mathbf{lim}\,P$ has a maximal prefix t in $\mathbf{lg}\,P$, and thus can be written as $e = tau$, where a is an action and u is a finite or infinite word, such that $ta \notin \mathbf{lg}\,P$. If a is an input, then e should have been avoided by the environment. This is a safety violation, but it is also a liveness violation, since, after the illegal input a, the device may require the environment to ensure occurrence of actions that are not in e. Thus, such e is a reject for the default liveness process. Symmetrically, if a is an output, then e is an escape for $\lambda_d P$. An example will be provided shortly.

By applying refinement to default liveness processes, we obtain a *relative liveness* condition in which a trace-structure specification is compared to a trace-structure implementation for liveness faults. Although this condition refers to implicitly assumed liveness properties, the verdict of this condition depends exclusively on the finite-trace descriptions given in the form of trace-structures. This does not contradict the point made in [Bla86] that finite traces do not have sufficient modeling power to specify arbitrary liveness properties. In our case, trace structures specify just these implicitly assumed liveness properties. Nevertheless, our approach appears to work well in many situations, including the following example adapted from [Bla86].

Example 6.8 The circuit in Figure 6.10 (a) represents an unfair implementation for an arbiter. The implementation uses two mutual exclusion elements, or mutex for short, whose trace structures are shown by the automaton in Figure 6.10 (b). A typical application of a mutex is to arbitrate the exclusive access of two independent clients to a shared resource. Each of the two clients sees an input and an output of the mutex. A client requests the resource by raising the corresponding input, and the mutex grants the resource by raising the corresponding output in response to a raised input. The resource is released by the client by lowering the corresponding input, after which the mutex lowers the corresponding output. Mutual exclusion is realized by ensuring that the two outputs are not both high at any particular time. The specification is also a mutex, whose terminals are labelled as in Figure 6.10 (c). The implementation also contains several delay elements, a flip-flop, which we assume to start in the state where Q is high, and two forks. The behavior automata in Figure 6.10 (d), (f), and (e) represent the trace structure of a delay element with input I and output O, the trace structure of the flip-flop (starting with Q high) and the trace structure of a fork with input I and outputs O0 and O1. The Q output of the flip-flop may change when S or R are high: it is reset when R is high and it is set when S is high and R is low. In the circuit in Figure 6.10 (a), we assume that

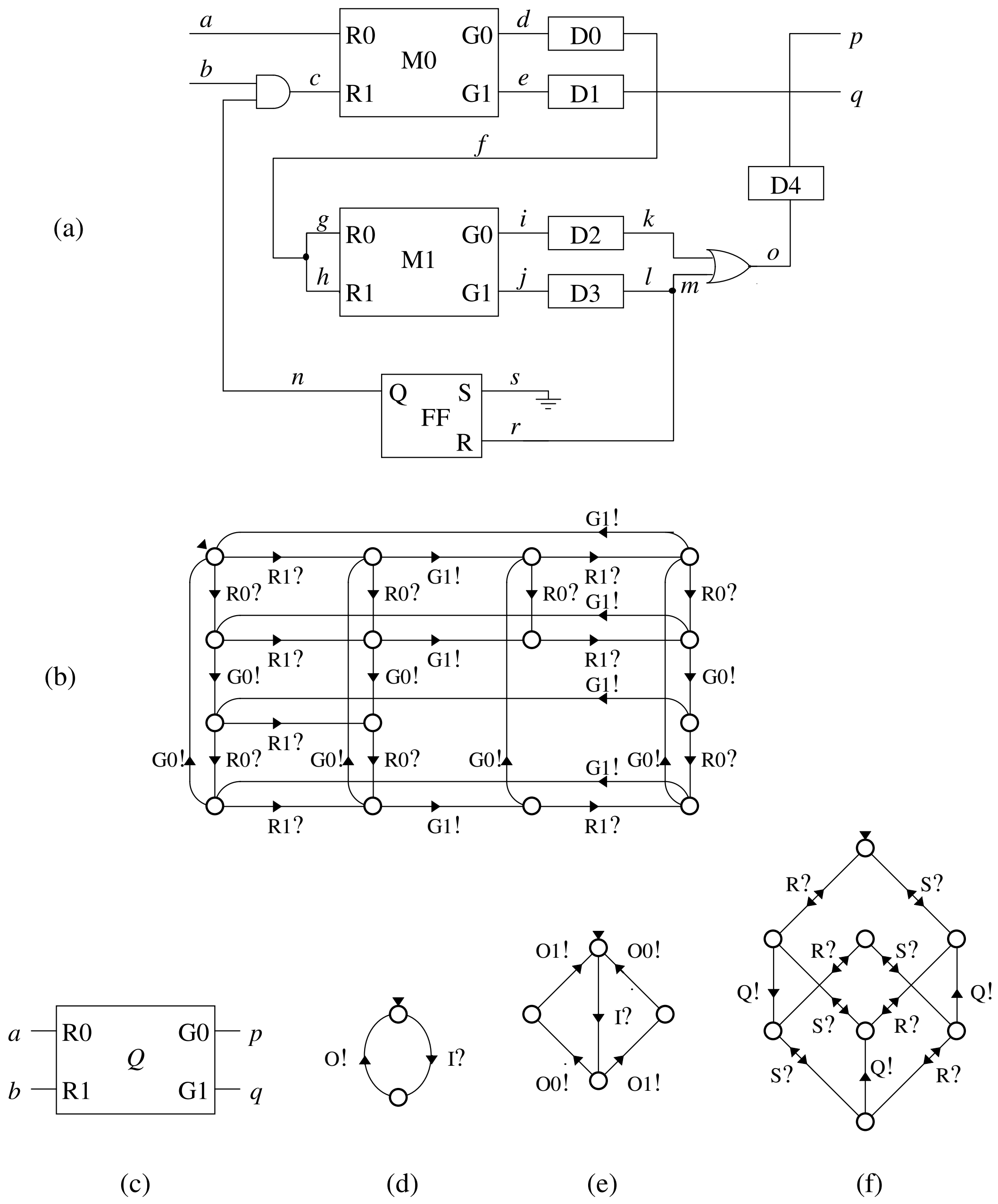


Figure 6.10: Unfair implementation of mutual exclusion.

n is initially high and all other signals are initially low.

The second MUTEX has the inputs wired together so that it enters a metastable state when a rising edge is applied. If the G1 branch wins, then the flip-flop is reset. After that, any c request is retracted and no further c requests are taken by the first MUTEX, because of the AND gate. Because of this danger, the implementation is unfair. Moreover, this implementation has several hazards, but they can be avoided by sizing the relative delays of the components and are not relevant for our discussion.

This unfairness is detected as a counter-example to the refinement condition on default liveness processes. Consider the infinite trace $e = \alpha\beta b(\gamma\gamma\delta\delta)^\omega$, where $\alpha = adfghjlrmnop$, $\beta = adfghjlrmop$, $\gamma = adfghikop$, and $\delta = adfghjlmrop$. The word α raises the outputs d of M0 and j of M1, resetting the flip-flop; β brings M0 and M1 back to their initial states. The word $\gamma\gamma$ represents a cycle of operation where outputs d of M0 and i of M1 are turned on and off; there are identical sequences of rising and falling transitions. The word $\delta\delta$ represents a cycle of operation where outputs d of M0 and j of M1 are turned on and off. Thus, e formalizes the danger described above: it resets the flip-flop and then never grants a q in response to the b event, even though M0 and M1 behave 'fairly' (since c does not rise, M0 does not have to raise e either). Let $I = \{$M0, M1, D0, D1, D2, D3, D4, FF, AND, OR$\}$. One verifies that, for each element P of I, we have $e{\downarrow}\mathbf{a}\,P \in \mathbf{otp}P$. In particular, note that $e_{\mathrm{M0}} = adad(adadadad)^\omega = (ad)^\omega$, $\mathbf{ren}_{\mathbf{lg}\mathrm{M0}} e_{\mathrm{M0}} = \{a, c, d\}$, and $\mathbf{rfi} e_{\mathrm{M0}} = \{a, d\} \supseteq \{a, c, d\} \cap \{d, e\} = \mathbf{ren}_{\mathbf{lg}\mathrm{M0}} e_{\mathrm{M0}} \cap \mathbf{o}\mathrm{M0}$. Thus, one verifies that $\forall\, P \in I : e \in \mathbf{as}\,\lambda_d P$. Thus, $e \in \mathbf{as} \times_{P \in I} \lambda_d P$. However, let Q be the specification of the circuit, which is the mutex shown in Figure 6.10 (c). We have that $e_Q = abpap(apapapap)^\omega = abp(ap)^\omega$, and $\mathbf{rfi} e_Q = \{a, p\}$, while $\mathbf{ren}_{\mathbf{lg}_Q} e_Q = \{a, b, p, q\}$ and thus $\mathbf{rfi} e_Q \not\supseteq \mathbf{ren}_{\mathbf{lg}_Q} e_Q \cap \mathbf{o}Q = \{p, q\}$. Thus, $e_Q \notin \mathbf{otp}Q$. Also, one verifies that $e_Q \in \mathbf{lim}\,\mathbf{lg}Q$ thus $e \notin \mathbf{as}\,\lambda_d Q$. By the counter-example of e, we have $\lambda_d Q \not\sqsubseteq \times_{P \in I} \lambda_d P$.

In words, although M0 and M1 behave fairly towards their choices, the overall implementation ignores the request b.

Assuming the default progress process for the specification (that is, assuming that $\pi Q = \pi_d\ Q$), the unfairness fault above fits into our classification of relative lock faults (Section 6.3) as a case of starvation, because e is an accessible execution for the $\pi_d Q$: since $\mathbf{rfi} e_Q = \{a, p\}$, $\mathbf{cen}_{\mathbf{lg}Q} e_Q = \{b\}$, thus $\mathbf{cen}_{\mathbf{lg}Q} e_Q \cap \mathbf{o}Q = \emptyset \subseteq \mathbf{rfi} e_Q = \{a, p\}$, and thus $e_Q \in \mathbf{osp}Q$. □

6.4.3 Implicit Progress Properties

Implicitly assumed unbounded traces can be associated to finitary representations by a property similar to that for complete traces. To formalize that property, we introduce the notion of *weak liveness* which unifies weak fairness and quiescence.

The approach in [CMP92] classifies liveness properties in a finer manner than [AS85]; apparently, it does not relate to our notion of progress. The liveness condition in [LT87] is close in form to our weak liveness formalization; however, the condition in [LT87] is not a condition for progress because [LT87] take infinite sequences to represent complete sequences, not unbounded sequences. On the other hand, our notion of progress, proposed in [Neg93] and [Neg95], does relate to the "finitary fairness" assumption recommended in [AH94] and [VK98].

In formalizing a notion of implicitly assumed unbounded traces of a discrete-state system, we have obtained a property that unifies a weak fairness property of infinite traces (e.g., see [Fra86]) with a quiescence property of finite traces (e.g.,

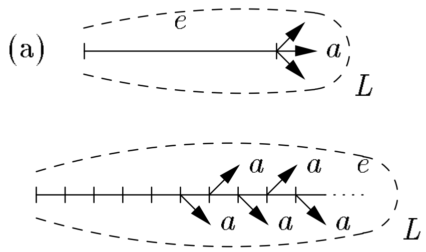


Figure 6.11: Continuously enabled symbols.

see [Jon87]). The property is formally the same for infinite and finite sequences, but, for clarity, the intuitive explanations are given separately for the two cases.

Symbol a is *continuously enabled* by sequence e with respect to language L if $\exists\, t \leq e$ such that $\forall\, u$ such that $tu \leq e$, we have $tua \in L$. The *set of continuously enabled symbols* of e with respect to L is denoted by $\mathbf{cen}_L e$. Note that, if e is infinite, the continuously enabled symbols of e with respect to L are those symbols that are immediately enabled in L by all prefixes of e larger than some prefix t. See Figure 6.9 (b). If e is finite, the continuously enabled symbols of e with respect to L are the symbols immediately enabled by e in L. See Figure 6.9 (a). For example, $\mathbf{cen}_{\mathbf{pref}\,(ab^*c)} a(b^\omega) = \{b, c\}$ and $\mathbf{cen}_{\mathbf{pref}\,((ab)^*c)} ab = \{a, c\}$.

For alphabet Σ and language L, limit e of L is *weakly live* with respect to Σ and L if e recurrently fires all symbols from Σ that e continuously enables in L, i.e., if $\mathbf{cen}_L e \cap \Sigma \subseteq \mathbf{rfi}\, e$. For infinite e, weak liveness is the same as weak fairness. For finite e, weak liveness means that no symbols from Σ are enabled at the end of e—the same as quiescence and the same as strong liveness.

Limit e of trace structure P is an *output stop* of P if e is weakly live with respect to $\mathbf{o}\,P$ and $\mathbf{lg}\,P$. The *set of output stops* of P is denoted by $\mathbf{osp}\,P$. (Note that $\mathbf{osp}\,P \subseteq \mathbf{lim}\,P$, and that the set of output stops of a trace structure is uniquely determined by its language and alphabets.) Output stops formalize our idea of implicitly assumed accessible unbounded traces of a system. For an intuitive picture, consider that the execution point of a system follows limit e. The continuously enabled output symbols can be viewed as exerting a pressure to be fired by the device; if the symbol cannot be fired, it will have to be disabled. Otherwise, there usually are upper bounds on delays.

Example 6.9 Consider a SELECTOR $\langle\{a\},\ \{b,c\},\ \mathbf{pref}\,(a(b\cup c))^*\rangle$. The set of output stops is $(a(b\cup c))^* \cup \{ab,ac\}^\omega$. The finite limits from $(a(b\cup c))^*$ are output stops because they do not immediately enable any output symbols. All infinite limits in this example are output stops because no outputs are continuously enabled: the outputs in this case are disabled periodically after words of the form $(a(b\cup c))^*$. The remaining finite limits, those in $(a(b\cup c))^*a$, are not output stops because they immediately enable b and c. □

The following definition associates our implicitly assumed progress properties to a trace structure.

Definition 6.2 *The* default progress process *of trace structure P is a process $\pi_d P$ over $\mathcal{U}^\infty$ such that*

$$\begin{aligned}\mathbf{as}\,\pi_d P &= (\mathbf{osp}\,P \cup ((\mathbf{lg}\,P\cdot\mathbf{i}\,P)\backslash\mathbf{lg}\,P)\cdot(\mathbf{a}\,P)^\infty){\uparrow}\mathbf{a}\,P\\ \mathbf{at}\,\pi_d P &= (\mathbf{lim}\,P \cup ((\mathbf{lg}\,P\cdot\mathbf{o}\,P)\backslash\mathbf{lg}\,P)\cdot(\mathbf{a}\,P)^\infty){\uparrow}\mathbf{a}\,P\ .\end{aligned}$$

The goal set of $\pi_d P$ is thus $(\mathbf{osp}\,P){\uparrow}\mathbf{a}\,P$; output stops are both accessible and acceptable as unbounded traces. Limits of $\mathbf{lg}\,P$ that are not output stops are escapes for $\pi_d P$, meaning that the device is supposed to ensure that every continuously enabled output of P is recurrently fired; in other words, the device should not stop in a state or a group of states where an output is due. This means that outputs are expected to be fired within some bounded time, unless they are disabled by some other events. (These implicit progress properties appear to fit well many practical systems, but there is no guarantee that they would always hold; where the implicit progress properties are inappropriate, the general progress process from Section 6.2 have to be used instead of $\pi_d P$.) The other escapes and rejects of $\pi_d P$ are as explained for $\lambda_d P$ in Subsection 6.4.2. An example will be provided shortly.

By applying refinement to default progress processes, we obtain a *relative progress* condition in which a trace-structure specification is compared to a trace-structure implementation for progress faults. Although this condition refers to implicitly assumed progress properties, the verdict of this condition depends exclusively on the finite-trace descriptions given in the form of trace-structures. Trace structures specify just the implicitly assumed progress properties, not arbitrary progress properties; nevertheless, our approach appears to work well in many situations.

Example 6.10 Informally speaking, we consider that progress violations are caused by limits that are 'bounded' for the specification, but are 'unbounded' for the implementation. In Example 6.4, execution u is accessible for π_dSENDER, π_dCHANNEL, and π_dRECEIVER, but not for π_dBUFFER because u does not contain an illegal input event for the BUFFER, and because u is not an output stop for the BUFFER since it continuously enables *out*. □

6.4.4 Limitations of the Default Processes

As part of the representation discipline, the users should ensure that default liveness and progress processes capture the properties of their systems; otherwise, general liveness and progress processes should be employed. For instance, default liveness and progress processes do not seem to be appropriate for the switch-level node models in Chapter 5.

Example 6.11 Some of the node models of Section 5.1 (a) forbid persistent collisions by requiring their environments to eventually leave a collision state. For instance, state 3 of the process automaton in Figure 5.2 is a reject state, making *up dn* a finalization reject. Also, *up dn* should be a liveness reject, since it is the environment's fault if this is a complete operation of the device-environment system.

On the other hand, a trace structure for the node in Figure 5.2 should normally have *up dn* in its language and as a limit, because this trace is legal as a partial observation of the behavior of this device-environment system. By the construction of λ_d, this trace is acceptable for the default liveness process, while, as pointed above, it should be a reject of the liveness process of the node. In this case, the implicit liveness properties did not correspond to the actual ones. □

Example 6.12 One may want to require that the collision state of the node in Example 6.11 be left within a bounded time, not just eventually. If so, *up dn* should be a reject of the progress process of that node, but one can check that it is not a reject of the default progress process of the node. In this case, the implicit progress properties did not correspond to the actual ones. □

Note, however, that Examples 6.11 and 6.12 are not a limitation of the general liveness and progress processes from Sections 6.1, 6.2, and 6.3, but only of the default liveness processes introduced in this section. In these examples, the general liveness and progress processes are used to model our intended properties of the nodes, and to determine the limitations of the default processes by comparison.

On the other hand, we believe that the advantages offered by the possibility to extract default processes from finitary descriptions, and a good match to the intended liveness and progress properties for most systems, can outweight the fact that there are few cases in which the default processes do not properly capture the intuition.

Chapter 7

Beyond Trace-Based Applications

In previous chapters, we have applied the general process space formalism by taking executions to be finite or infinite sequences of events. While these have been our main applications so far, process spaces can address a much larger range of problems. In this chapter we briefly discuss some other ways to apply the general process space theory by using different types of executions. Although these additional discussions cannot be very involved, their purpose is two-fold: to take steps towards some applications of promising practical value, and, at the same time, to illustrate the power and usage of the general theory by indicating some unusual applications.

7.1 Steady-State Networks

In this section we consider an application that does not refer to the behavior of a system in time, but rather to the stable states of a system. Also, the systems considered in this section are not discrete-state systems; their state spaces are parameterized by state variables that take values from the set of reals. Moreover, as is often the case in practice, there may be some uncertainty in values of the parameters (coefficients) for which only ranges are given. An example of such systems are electrical networks. If there are n real state variables of interest in the network, one may take the execution set to be $\mathbb{R}^n$. The processes corresponding to parts or sub-networks are determined according to the contract paradigm described in Chapter 2.

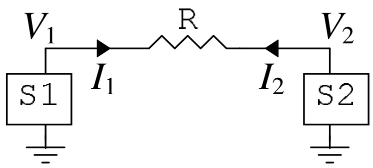


Figure 7.1: Electrical network for steady-state parameter example.

Example 7.1 Figure 7.1 represents a steady-state network (a DC circuit). The outputs of two voltage sources `S1` and `S2` are connected by a resistor `R`. There are four variables of interest: V_1, I_1, V_2, and I_2, the output voltages and currents of the two sources; correspondingly, we take the executions to be vectors (V_1, I_1, V_2, I_2) from $\mathbb{R}^4$ (the units are in the international system). We assume that source $\mathtt{S}x$ delivers an electromotive force between $V_{x\,\min}$ and $V_{x\,\max}$ as long as I_x is between $I_{x\,\min}$ and $I_{x\,\max}$. Accordingly, we represent source `S1` by a process $\eta\mathtt{S1}$ over $\mathbb{R}^4$ such that

$$\begin{aligned}
\mathbf{g}\,\eta\mathtt{S1} &= \{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid V_{1\,\min} \le V_1 \le V_{1\,\max} \wedge I_{1\,\min} \le I_1 \le I_{1\,\max}\}\ , \\
\mathbf{at}\,\eta\mathtt{S1} &= \{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid I_{1\,\min} \le I_1 \le I_{1\,\max}\}\ .
\end{aligned}$$

(Recall that $\mathbf{g}$ denotes the goal set of a process.) Notice for instance that an execution with $I_1 > I_{1\,\max}$ and $V_1 < V_{1\,\min}$ is made a reject for $\eta\mathtt{S1}$. This execution is clearly not an intended behavior; thus it should not be a goal. Because the source does not guarantee a voltage within the range if the current is outside the range, this execution should not be an escape. Source S2 is represented by a similar process ηS2.

We also assume that the resistance of R is r and that R can operate under all voltages. Accordingly,

$$\eta\mathrm{R} \;=\; (\ \{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid I_1 + I_2 = 0 \ \wedge\ rI_1 = V_1 - V_2\}\ ,\ \mathbb{R}^4\ )\ .$$

Now, we assume that $V_{1\,\min} = V_{2\,\min} = 4.9$V, $V_{1\,\max} = V_{2\,\max} = 5.2$V, $I_{1\,\max} = I_{2\,\max} =$ 25mA, $I_{1\,\min} = I_{2\,\min} =$ -25mA, and $r = 10\Omega$. Robustness of product is not satisfied for parameters in these ranges. For execution $z = (5.2, 0.03, 4.9, -0.03)$, we have $z \in \mathbf{as}\,\eta\mathrm{S1} \cap \mathbf{as}\,\eta\mathrm{S2} \cap \mathbf{as}\,\eta\mathrm{R}$, but $z \notin \mathbf{at}\,\eta\mathrm{S1}$. It follows that $\eta\mathrm{S1} \times \eta\mathrm{S2} \times \eta\mathrm{R} \notin \mathcal{R}_{\mathbb{R}^4}$.

The violation can be interpreted as follows. Due to slack in the values of the electromotive forces, a current larger than 25mA may occur, which may damage the sources. □

7.2 Dynamical Systems

Process spaces may be useful in the study of dynamical systems. If there are n real state variables of interest (counting derivatives as well), one may take the execution set to be the set of functions from $\mathbb{R}$ (the time domain[1]) to $\mathbb{R}^n$. Some simplifications are possible for particular types of dynamical systems. For instance, the execution set can be taken to contain only continuous functions or can be taken to contain distributions whose Laplace transforms are ratios of polynomials.

[1]Other time domains, such as the set of integers, can be also be used instead of $\mathbb{R}$.

The processes corresponding to dynamical systems are determined according to the contract paradigm described in Chapter 2.

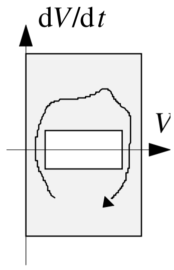


Figure 7.2: Requirement of Brockett ring type.

One possible use for process spaces in dynamical systems may be to set proof obligations sufficient to allow for a simpler paradigm, such as the discrete-state modeling. [GC94] demonstrates several difficulties with a toggle element at very high and very low switch frequencies. However, [GC94] uses a sine wave for input. Some of these difficulties may turn out to be avoidable if all voltages in the circuit are restricted to be either low, high, or changing quickly. This requirement (a Brockett ring requirement [Bro89]) amounts to restricting the phase trajectories for each variable (i.e., the possible sets of pairs $(V, dV/dt)$) to be within a region of the phase plane such as the shaded area in Figure 7.2 (a region topologically isomorphic to an annulus, meaning that there exists a continuous bijective mapping of the phase plane to itself that transforms that region into a ring-shaped region), under appropriate differentiability assumptions. Accordingly, one can restrict the goal sets to comprise only functions that satisfy such a requirement. If an execution violates this requirement on an input variable of a dynamical system, that execution will be considered a reject for the process representing that system. If an execution violates the requirement on an output variable, that execution will be considered an escape. The refinement condition may be used to prove that, for each cell in a circuit, if the input signals of that cell satisfy the Brockett ring requirement, then that cell operates according to a discrete-state model, and moreover, its output signals also satisfy the Brockett ring requirement. The robustness condition will tell whether the Brockett ring requirement is satisfied by the inputs of all cells, to ensure that the discrete-state approximation can be used without bad effects.

7.3 Input Control

By *input control* we mean absence of dangling (unconnected) inputs in a digital circuit; this is a connectivity correctness concern. Dangling inputs are sometimes regarded as dangerous because their voltages may fluctuate excessively in the presence of electromagnetic interference.

In [Dil89] a "circuit algebra" was proposed, to justify certain algebraic properties of processes by their relevance for connectivity concerns. Also, in [Ver94a], studies of various connectivity concerns are proposed, on the basis of the testing paradigm. These treatments of connectivity concerns have many similarities to studies of behavior; however, it is not clear whether these treatments as originally presented can be viewed as instances of a theory based on abstract executions.

For studying input control, we consider that each discrete-state system s is represented by its uncontrolled action process, a process vs over $\mathcal{U}$. For a circuit component, we say an action from $\mathcal{U}$ is *uncontrolled* if that action does not represent an output of that component. The uncontrolled action processes are determined by the contract paradigm described in Chapter 2, applied to uncontrolled actions. We do not offer a general recipe of how to attach uncontrolled action processes to an arbitrary circuit component, since this would have to refer to a definition of what it means for a component port to be an input. However, in the vast majority of digital circuits, one can distinguish inputs from outputs easily by their impedances. Then, regarding the actions as executions, an uncontrolled action process classifies actions as rejects, goals, or escapes according to whether they may 'occur' as uncontrolled actions of a given system; usually, this means that outputs are escapes (the device should make sure they may not 'occur' as uncontrolled actions), inputs are rejects (the environment should make sure they are not uncontrolled), and all other actions in $\mathcal{U}$ are goals (as far as the device and environment are concerned, those actions can be uncontrolled). The following examples illustrate uncontrolled action processes and meanings of robustness and product in this interpretation.

Example 7.2 Let the components of the circuit in Figure 6.1 (a) be C, BUF, and INV, from top to bottom. We check whether the circuit is a 'good substitute' for the AND gate with respect to input control.

Letting $\mathcal{U} = \{a, b, c, d, e\}$, we have (after some straightforward manipulations):

$$\begin{array}{ll} & \upsilon\mathrm{C} \times \upsilon\mathrm{BUF} \times \upsilon\mathrm{INV} \\ = & (\{a,b,d,e\},\{c,d,e\}) \times (\{a,b,c,d\},\{a,b,c,e\}) \times (\{a,b,c,e\},\{a,b,c,d\}) \\ = & (\{a,b\},\{c\} \cup \overline{\{a,b\}}) \\ = & (\{a,b\},\{c,d,e\}) \\ \notin & \mathcal{R}_{\{a,b,c,d,e\}} \ . \end{array}$$

Note that $\mathbf{r}\,(\upsilon\mathrm{C} \times \upsilon\mathrm{BUF} \times \upsilon\mathrm{INV}) = \{a, b\}$ indicates precisely the two dangling inputs in the circuit. □

Example 7.3 Input control can also be considered in a relative sense. Let us check whether the circuit in Figure 6.1 (a) is a correct implementation of the AND gate in Figure 6.1 (b), with respect to input control. For that, we demand $\upsilon\mathrm{AND} \sqsubseteq$

$\upsilon\mathrm{C} \times \upsilon\mathrm{BUF} \times \upsilon\mathrm{INV}$. The uncontrolled action processes of the components are as in Example 7.2. We have

$$\begin{array}{ll} & \mathbf{as}\,(\upsilon\mathrm{C} \times \upsilon\mathrm{BUF} \times \upsilon\mathrm{INV}) \\ = & \{a, b\} \qquad \text{(as in Example 7.2)} \\ \subseteq & \{a, b, d, e\} \\ = & \mathbf{as}\,\upsilon\mathrm{AND} \end{array}$$

and

$$\begin{array}{ll} & \mathbf{at}\,(\upsilon\mathrm{C} \times \upsilon\mathrm{BUF} \times \upsilon\mathrm{INV}) \\ = & \{c, d, e\} \qquad \text{(as in Example 7.2)} \\ = & \mathbf{at}\,\upsilon\mathrm{AND} \end{array}$$

The refinement condition is thus satisfied. Informally speaking, the two dangling inputs of the circuit are controlled by the environment of the AND gate. $\square$

7.4 Timing

Timing properties can be determined on the basis of time-stamped partial traces of a discrete-state system, i.e., the finite or infinite sequences of pairs of actions and time-stamps representing events that occur up to a certain time. Here, we take the time domain to be $[0, \infty)$, the set of non-negative real numbers.[2]

For pair $(a, t) \in \mathcal{U} \times [0, \infty)$, let $\mathbf{a}\,(a, t) = a$ and $\mathbf{t}\,(a, t) = t$. For finite sequence x, let $\mathbf{d}\,x$ be the domain of indices of x, which is the set $\{i \in \mathbb{Z} \mid 0 \leq i < l\}$, where $\mathbb{Z}$ is the set of integer numbers and l is the length of x. Let the elements of x be

[2]Other time domains can also be used.

$x_0, \ldots, x_{l-1}$. For example, the domain of indices of aba is $\{0, 1, 2\}$, $(aba)_1$ is b, and the domain of indices of ε is $\emptyset$.

With this notation, we take the execution set to be

$$\mathcal{F} = \{x \in (\mathcal{U} \times [0, \infty))^* \mid \forall i, j \in \mathbf{d}\, x : (i \leq j \Rightarrow \mathbf{t}\, x_i \leq \mathbf{t}\, x_j)\} \ .$$

In words, the executions for timing are the finite sequences of pairs of an action and a time-stamp such that time-stamps are in increasing order. For instance, $(a, 0.4)(b, 0.4)(c, 2) \in \mathcal{F}$, but $(a, 1)(b, 0.7) \notin \mathcal{F}$. For studying timing, one can for instance represent a discrete-state system by a process over $\mathcal{F}$, as prescribed by the contract paradigm described in Chapter 2, applied to time-stamped partial traces. Under this representation, timing faults such as excessive worst-case delays can be detected as violations of refinement or robustness.

7.5 True Concurrency

In Chapters 3, 4, and 6, we have assumed that events are instantaneous. Simultaneous occurrence of two events was modeled as two possible interleavings of those events. In the "true concurrency" paradigm, simultaneous occurrence of two atomic events is different from the possibility of both interleavings, which is perhaps a more accurate but a more complicated point of view.

For a "true concurrency" model, one can consider actions to be elements of $\wp(\mathcal{U})$ (subsets of $\mathcal{U}$), as opposed to atomic actions, which are elements of $\mathcal{U}$. Then, events are occurrences of actions, and atomic events are occurrences of atomic actions. The execution sets are determined by the contract paradigm in Chapter 2, applied to sequences over $\wp(\mathcal{U})$. Determining the goal, reject, and escape sets for true concurrency processes is roughly similar to determining such sets for the safety, finalization, liveness, or progress processes in previous chapters, although goals are now sequences over $\wp(\mathcal{U})$ instead of sequences over $\mathcal{U}$. Determining reject and escape sets has a subtlety, however, if an illegal event contains both an illegal input event and an illegal output event. In such situations, the resulting execution is typically an escape rather than a reject.

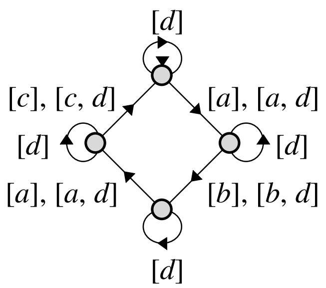


Figure 7.3: Goal set for multiple-winner example: all sequences of sets of actions that are spelled by a path starting at the initial state.

Example 7.4 Consider a component TOGGLE with input a and outputs b and c, which repeatedly inputs an a event and outputs either a b event or a c event, alternating b and c. For illustrative purposes, assume there also exists an atomic action d which is not seen by the TOGGLE. The true concurrency safety process of TOGGLE is a process σ'TOGGLE over $\wp(\mathcal{U})^*$. The goal set of σ'TOGGLE is as shown in Figure 7.3 by an informal state graph (not a process automaton or a behavior automaton) where true concurrency events, that is sets of atomic events from $\mathcal{U}$, are represented as lists of atomic events within square brackets. Note that d atomic events may occur arbitrarily. Execution $[a][a, b]$ is a reject of σ'TOGGLE, because TOGGLE may issue a b after the first a, but a second a is not expected until an output atomic event of TOGGLE. On the other hand, execution $[a][a, c]$ is an escape of σ'TOGGLE, because $[a, c]$ contains both an illegal input atomic event (no a events are expected until an output atomic event) and an illegal output atomic event (it is not the turn of c). $\square$

Chapter 8

Process Abstractions

In Chapter 6, we have used a projection operation that deletes certain events from traces, and a related extension operation that inserts events arbitrarily in traces. Also, in Section 4.2 we have noted that, in applications, we often change the initial states of process automata, to correspond to the actual initial values of signals in a circuit; such operations of re-initializing a process are known as derivatives or as "after"-operations.

At first sight, it may seem that, since process spaces do not refer to the structure of executions, they would not be able to capture projections or derivatives. It turns out that significant properties of these operations can be obtained from binary relations between sets of abstract executions, without any references to alphabets or other structural details of executions. Moreover, other operations, such as relabeling, reflection, rotation, and process compositions, can be built from binary relations between executions as well, thus inheriting these properties.

The algebraic properties mentioned above include monotonicity with respect to process comparisons, laws for composition, Galois connection properties, and, very importantly, criteria that authorize us to perform verifications on images of processes through such maps instead of directly on the original processes. Certain classes of systems are shown to allow for equivalent verifications on images through a given map, and certain classes of maps are shown to always produce pessimistic or optimistic approximations of all processes.

A benefit of studying these properties in a unified framework is that we capture the algebraic structure of several process maps for the price of one. We start with mathematical preliminaries regarding maps between sets, which may be of independent theoretical interest because they generalize certain properties of language homomorphisms. We then present our mathematical treatment of maps between processes, followed by a more intuitive section on criteria for verifications on images of processes through such maps. After presenting this theory, we demonstrate the concepts on applications to the operations mentioned above and to some other operations as well.

The research in this chapter was tentatively started together with J.A. Brzozowski and J. C. Ebergen in [BEN96]. Although the joint work was interrupted shortly thereafter, we would like to acknowledge their feedback, as well as Ebergen's formalization of projection operations for processes with alphabets.

8.1 Preliminaries

As a first step towards building a theory of process transforms, we first consider maps between power sets of two arbitrary sets $\mathcal{E}_1$ and $\mathcal{E}_2$.

In the mathematical notation and terminology, we follow [Bir67], [DP90], or introduce our own.

The *converse* of a relation $\rho \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ is a relation $\rho^{\smile} \subseteq \mathcal{E}_2 \times \mathcal{E}_1$ such that, for every $u \in \mathcal{E}_1$ and $v \in \mathcal{E}_2$, $u\rho v \Leftrightarrow v\rho^{\smile}u$. (The sign $\times$ is used both for Cartesian product and the process product, but it will be clear from the context which of these two operations is used.) The *composite* of relations $\rho_1 \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ and $\rho_2 \subseteq \mathcal{E}_2 \times \mathcal{E}_3$ is a relation $\rho_1 \circ \rho_2 \subseteq \mathcal{E}_1 \times \mathcal{E}_3$ such that, for every $u \in \mathcal{E}_1$ and $w \in \mathcal{E}_3$, $u(\rho_1 \circ \rho_2)w \Leftrightarrow \exists\, v \in \mathcal{E}_2 : u\rho_1 v \wedge v\rho_2 w$. The *image* of set $X \subseteq \mathcal{E}_1$ through relation $\rho \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ is the set $\mathrm{Im}_\rho(X) = \{v \in \mathcal{E}_2 \mid \exists\, u \in X : u\rho v\}$. For functions $h_1 : B \to C$ and $h_2 : A \to B$, $(h_1 \circ h_2)(x) = h_1(h_2(x))$.[1]

A *poset* is a pair $\langle P, \preceq\rangle$ consisting of a set P and a partial order relation $\preceq$ on P. A *Galois connection* between posets $\langle A, \preceq_A\rangle$ and $\langle B, \preceq_B\rangle$ is a pair (α, β) of maps $\alpha : A \to B$ and $\beta : B \to A$ such that, for every $u \in A$ and $v \in B$, $u \preceq_A \beta(v) \Leftrightarrow v \preceq_B \alpha(u)$. Map $\gamma : A \to B$ is *order-reversing* if $\forall\, u_1, u_2 \in A : u_1 \preceq u_2 \Rightarrow \gamma(u_1) \succeq \gamma(u_2)$.

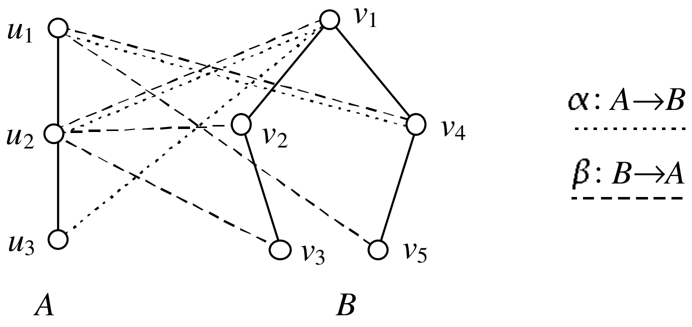


Figure 8.1: Example of a Galois connection.

Example 8.1 The continuous lines in Figure 8.1 represent the Hasse diagrams of

[1]As usual, the notation for composition of relations is in reverse order from composition of functions, as follows. The *graph* of a function $h : C \to D$ is a relation $\eta \subseteq C \times D$ such that $u\eta v \Leftrightarrow h(u) = v$. If η_1 and η_2 are the graphs of functions $h1$ and $h2$, then the graph of $h_2 \circ h_1$ is $\eta_1 \circ \eta_2$.

two posets A and B. The dotted lines represent a map $\alpha : A \to B$, and the dashed lines represent a map $\beta : B \to A$. These two maps form a Galois connection. $\square$

A Galois connection correlates the orders in two posets in a particular manner, as described by some algebraic properties mentioned in [Bir67] and [DP90], given below.

Proposition 8.1 *For posets $\langle A, \preceq_A \rangle$ and $\langle B, \preceq_B \rangle$ and maps $\alpha : A \to B$ and $\beta : B \to A$, we have:*

(a) *(α, β) is a Galois connection if and only if*

- **(i)** $\forall\, u \in A, v \in B : u \preceq_A \beta(\alpha(u))$ *and* $v \preceq_B \alpha(\beta(v))$, *and*
- **(ii)** *α and β are order-reversing.*

(b) *If (α, β) is a Galois connection then*

- **(iii)** $\forall\, u \in A, v \in B : \alpha(\beta(\alpha(u))) = \alpha(u)$ *and* $\beta(\alpha(\beta(v))) = \beta(v)$.

Example 8.2 One verifies that the maps in Figure 8.1 satisfy properties (i), (ii), and (iii) of Proposition 8.1, in addition to the defining property of Galois connections above. $\square$

As we will show, the process transforms that are of interest to us can be obtained from maps between sets of executions that satisfy the following laws.

Lemma 8.2 *For map $f : \wp(\mathcal{E}_1) \to \wp(\mathcal{E}_2)$, the following properties are equivalent.*

(a) *There exists a map $f^{\smile} : \wp(\mathcal{E}_2) \to \wp(\mathcal{E}_1)$ such that*

$$\forall\, X \subseteq \mathcal{E}_1, Y \subseteq \mathcal{E}_2 : \; X \cap f^{\smile}(Y) = \emptyset \;\Leftrightarrow\; Y \cap f(X) = \emptyset\ ;$$

(b) *For every set $\mathcal{M}$ of subsets of $\mathcal{E}_1$,*

$$f(\bigcup_{X \in \mathcal{M}} X) \;=\; \bigcup_{X \in \mathcal{M}} f(X) \;;$$

(c) *There exists a relation* $\rho \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ *such that*

$$f = \mathrm{Im}_\rho \;.$$

For map $h : \wp(C) \to \wp(D)$, let $\overline{h}$ be a map from $\wp(C)$ to $\wp(D)$ that satisfies $\forall\, X \subseteq C : \overline{h}(X) = \overline{h(X)}$. Noting that two sets A and B are disjoint if and only if $A \subseteq \overline{B}$, Property (a) above is the same as saying that the maps $\overline{f}$ and $\overline{f^{\smile}}$ form a Galois connection between $(\wp(\mathcal{E}_1), \subseteq)$ and $(\wp(\mathcal{E}_2), \subseteq)$. This shows how the maps in question keep consistency of the subset orders in the two power sets. Property (b) characterizes such maps as those maps that preserve union. Property (c) provides a representation for such maps, showing they can be constructed from binary relations.

Example 8.3 The maps that satisfy the properties in Lemma 8.2 generalize language homomorphisms.

Our definitions of language homomorphisms are adapted from [HU79]. For finite sets Σ_1 and Σ_2, a *language homomorphism* is a map $h : \Sigma_1 \to \Sigma_2^*$ extended to Σ_1^* and $\wp(\Sigma_1^*)$ by the following laws:

$$h(\varepsilon) = \varepsilon \;,$$
$$\forall\, u_1, u_2 \in \Sigma_1^* :\; h(u_1 u_2) = h(u_1) h(u_2) \;,$$
$$\forall\, L \subseteq \Sigma_1^* :\; h(L) = \bigcup_{u \in L} h(u) \;.$$

Consider, for instance, the *deletion* map from [Dil89]. For finite sets Γ and Σ such that $\Gamma \subseteq \Sigma$, $\mathbf{del}(\Gamma)(\cdot) : \Sigma \to \Sigma^*$ satisfies

$$\forall\, a \in \Sigma \setminus \Gamma : \mathbf{del}(\Gamma)(a) = a \text{ , and}$$
$$\forall\, a \in \Gamma : \mathbf{del}(\Gamma)(a) = \varepsilon \text{ .}$$

The resulting language homomorphism $\mathbf{del}(\Gamma)(\cdot) : \wp(\Sigma^*) \to \wp(\Sigma^*)$ deletes from the argument words all occurrences of symbols from Γ.

Note that the third law in the definition of language homomorphims above is equivalent to Property (c) of Lemma 8.2, if we take $\rho \subseteq \Sigma^* \times \Sigma^*$ so that $u \rho v \Leftrightarrow v = h(u)$, (i.e., so that ρ is the graph of h). Also, it is noted in [HU79] that language homomorphims commute with finite union; this property can easily be extended to infinite unions as in Lemma 8.2, Property (b).

For language homomorphism h, the *inverse homomorphic image* of a language $L' \subseteq \Sigma_2^*$ is

$$h^{-1}(L') = \{u \mid h(u) \in L'\} \text{ .}$$

Despite this notation and terminology, which are standard in language theory, h^{-1} is *not* the inverse of function h. In fact, language homomorphisms need not be bijective, thus they might not admit inverses; deletion, for instance, is not injective since two symbols from Γ have the same image, ε.

On the other hand, inverse homomorphic images do correspond to converses of maps as in Property (a) of Lemma 8.2. For all $L \subseteq \Sigma_1^*$ and $L' \subseteq \Sigma_2^*$,

$$L \cap h^{-1}(L') = \emptyset \;\Leftrightarrow\; \neg\exists\, u \in L :\; h(u) \in L' \;\Leftrightarrow\; h(L) \cap L' = \emptyset \text{ .}$$

□

Proof of Lemma 8.2

((b)⇒(a)) For an arbitrary subset Y of $\mathcal{E}_2$, let

$$f^{\smile}(Y) = \{u \in \mathcal{E}_1 \mid f(\{u\}) \cap Y \neq \emptyset\}.$$

Then, for arbitrary subset X of $\mathcal{E}_1$,

$$\begin{array}{ll} & X \cap f^{\smile}(Y) \neq \emptyset \\ \Leftrightarrow & X \cap \{u \in \mathcal{E}_1 \mid f(\{u\}) \cap Y \neq \emptyset\} \neq \emptyset \\ \Leftrightarrow & \exists\, u \in X : f(\{u\}) \cap Y \neq \emptyset \\ \Leftrightarrow & Y \cap \bigcup_{u \in X} f(\{u\}) \neq \emptyset \\ \Leftrightarrow & Y \cap f(\bigcup_{u \in X}\{u\}) \neq \emptyset \ . \qquad \text{(cf. Property (b))} \end{array}$$

((c)⇒(b)) For arbitrary $\mathcal{M} \subseteq \wp(\mathcal{E}_1)$,

$$\begin{array}{ll} & f(\bigcup_{X \in \mathcal{M}} X) \\ = & \{v \in \mathcal{E}_2 \mid \exists\, u \in \bigcup_{X \in \mathcal{M}} X : u\rho v\} \\ = & \{v \in \mathcal{E}_2 \mid \exists\, X \in \mathcal{M} : \exists\, u \in X : u\rho v\} \\ = & \{v \in \mathcal{E}_2 \mid \exists\, X \in \mathcal{M} : v \in f(X)\} \\ = & \bigcup_{X \in \mathcal{M}} f(X) \ . \end{array}$$

((a)⇒(c)) Let $\rho = \{(u, v) \in \mathcal{E}_1 \times \mathcal{E}_2 \mid v \in f(\{u\})\}$, and consider arbitrary subset X of $\mathcal{E}_1$. We prove that $f(X) = \{v \mid \exists\, u \in X : u\,\rho\, v\}$. For arbitrary $v \in \mathcal{E}_2$, we have

$$\begin{array}{ll} & v \in f(X) \\ \Leftrightarrow & f(X) \cap \{v\} \neq \emptyset \\ \Leftrightarrow & X \cap f^{\smile}(\{v\}) \neq \emptyset \qquad \text{(by (a))} \\ \Leftrightarrow & \exists\, u \in X : \{u\} \cap f^{\smile}(\{v\}) \neq \emptyset \\ \Leftrightarrow & \exists\, u \in X : f(\{u\}) \cap \{v\} \neq \emptyset \qquad \text{(by (a) again)} \\ \Leftrightarrow & \exists\, u \in X : v \in f(\{u\}) \\ \Leftrightarrow & \exists\, u \in X : u\,\rho\, v \ . \end{array}$$

□

A representation property similar to (a) $\Leftrightarrow$ (c) is a previous result mentioned in [Bir67].

Definition 8.1 *Function f between the power sets of sets $\mathcal{E}_1$ and $\mathcal{E}_2$ is a* union-preserving map *(*UPM *for short) if it satisfies the properties from Lemma 8.2.*

By definition, UPMs preserve unions. Are there similar properties with respect to other set operators?

Lemma 8.3 *For UPM f between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, for subsets X and Y of $\mathcal{E}_1$, and for family $\mathcal{M} \subseteq \wp(\mathcal{E}_1)$ of subsets of $\mathcal{E}_1$, we have*

(a) $X \subseteq Y \;\Rightarrow\; f(X) \subseteq f(Y)$; *(monotonicity)*

(b) $f(\bigcap_{X \in \mathcal{M}} X) \subseteq \bigcap_{X \in \mathcal{M}} f(X)$;

(c) $f(\emptyset) = \emptyset$. *(strictness)*

Proof For Part (a), note

$$X \subseteq Y \;\Rightarrow\; X \cup Y = Y \;\Rightarrow\; f(X) \cup f(Y) = f(Y) \;\Rightarrow\; f(X) \subseteq f(Y) \; .$$

(Alternately, Part (a) can be proven by using the continuity of f, which results from Property (b) in Lemma 8.2 by taking $\mathcal{M}$ to be a directed set; see e.g. [DP90] for definitions of directed sets and continuity of maps between posets.)

Part (b) follows from Part (a) by noting $\forall\, X \in \mathcal{M} : \bigcap_{Z \in \mathcal{M}} Z \subseteq X$, thus, $\forall\, X \in \mathcal{M} : f(\bigcap_{Z \in \mathcal{M}} Z) \subseteq f(X)$.

Part (c) results from Property (b) in Lemma 8.2 by taking $\mathcal{M} = \emptyset$. $\square$

Note that, in general, we do not have $f(\bigcap_{X \in \mathcal{M}} X) = \bigcap_{X \in \mathcal{M}} f(X)$. Take, for instance, UPM $f : \wp(\{1,2\}) \to \wp(\{3\})$ such that $f(\{1\}) = f(\{2\}) = \{3\}$; we have $f(\{1\} \cap \{2\}) = f(\emptyset) = \emptyset$, but $f(\{1\}) \cap f(\{2\}) = \{3\}$.

In Lemma 8.2, Parts (a) and (c) refer to the existence of other objects (a Galois connection and a binary relation) that have certain properties in combination with the UPM being defined. The relation in Part (c) is unique, by the way it is defined through its image. The following lemma shows the Galois connection is unique too, and indicates how to construct it.

Lemma 8.4 *For UPM f, Galois connection $(\overline{f}, \overline{f^{\smile}})$, and relation ρ,*

$$f = \mathrm{Im}_\rho \;\Leftrightarrow\; f^{\smile} = \mathrm{Im}_{\rho^{\smile}} \; .$$

Proof For all u and Y, we have

$$\begin{array}{lll} & u \in f^{\smile}(Y) & \\ \Leftrightarrow & \{u\} \cap f^{\smile}(Y) \neq \emptyset & \\ \Leftrightarrow & Y \cap f(\{u\}) \neq \emptyset & \text{(by Lemma 8.2 since } f \text{ is a UPM)} \\ \Leftrightarrow & \exists\, v \in Y : v \in f(\{u\}) & \\ \Leftrightarrow & \exists\, v \in Y : u \,\rho\, v & \\ \Leftrightarrow & \exists\, v \in Y : v \,\rho^{\smile}\, u & \\ \Leftrightarrow & u \in Im_{\rho^{\smile}}(Y) \; . & \end{array}$$

□

The construction in Lemma 8.4 also serves as a definition.

Definition 8.2 *For UPM f between $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, the* underlying relation *of f is the (unique) relation $\rho_f \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ that satisfies $f = \mathrm{Im}_{\rho_f}$, and the* converse *of f is the (unique) map $f^{\smile} : \wp(\mathcal{E}_2) \to \wp(\mathcal{E}_1)$ for which $(\overline{f}, \overline{f^{\smile}})$ is a Galois connection.*

Example 8.4 An interesting UPM that is neither a language homomorphism nor the converse of a language homomorphism is prefix closure **pref**; this map was used in Chapter 6. The underlying relation is $\geq$, the converse of the prefix relation $\leq$ on strings. Thus, not only **pref** is not a language homomorphism, but also its unique underlying relation is not a function.

Both the prefix closure of languages of finite words and the prefix closure for languages of finite and infinite words are UPMs. It follows that these maps between languages inherit all the properties we prove for UPMs. □

Converses and compositions of UPMs correspond to converses and compositions of underlying relations.

Lemma 8.5 *For UPMs $f : \wp(\mathcal{E}_1) \to \wp(\mathcal{E}_2)$ and $g : \wp(\mathcal{E}_2) \to \wp(\mathcal{E}_3)$,*

(a) $f^{\smile}$ *is a UPM and* $\rho_{(f^{\smile})} = (\rho_f)^{\smile}$;

(b) $g \circ f$ *is a UPM and* $\rho_{g\circ f} = \rho_f \circ \rho_g$;

(c) $(g \circ f)^{\smile} = f^{\smile} \circ g^{\smile}$.

Proof

(a) This is a re-statement of Lemma 8.4.

(b) Let X be an arbitrary subset of $\mathcal{E}_1$. Then,

$$
\begin{aligned}
& g \circ f(X) \\
= \quad & \{w \in \mathcal{E}_3 \mid \exists\, v \in f(X) : v\, \rho_g\, w\} \\
= \quad & \{w \in \mathcal{E}_3 \mid \exists\, u \in X, v \in \mathcal{E}_2 : u\, \rho_f\, v \ \wedge\ v\, \rho_g\, w\} \\
= \quad & \{w \in \mathcal{E}_3 \mid \exists\, u \in X : u\, (\rho_f \circ \rho_g)\, w\} \ .
\end{aligned}
$$

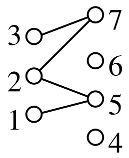


Figure 8.2: Example relation.

(c) Trivial. $\square$

In general, the converses of UPMs, derived from relations, are not the same as inverses of these functions. That is, we may have $f^{\smile}(f(X)) \neq X$. Take for instance $X = \{1\}$ in Figure 8.2; we have $f^{\smile}(f(\{1\})) = \{1, 2\}$. Also see Example 8.3.

Example 8.5 The converse of the deletion UPM $\mathbf{del}(\{beep\}) : \wp(\mathcal{E}_1) \to \wp(\mathcal{E}_2)$ from Example 8.3 is a map $\mathbf{del}^{-1}(\{beep\}) : \wp(\mathcal{E}_2) \to \wp(\mathcal{E}_1)$, called *inverse deletion* in [Dil89], which inserts *beep* actions arbitrarily in the words of a sublanguage of $\mathcal{E}_2$. For instance, $\mathbf{del}^{-1}(\{beep\})(\{e1\ m2,\ \varepsilon\}) = beep^*\ e1\ beep^*\ m2\ beep^* \cup beep^*$.

The term "inverse deletion" in [Dil89] refers to the inverse homomorphic images, and not to the inverse of a function of languages; see Example 8.3. $\square$

The following lemma shows that taking the converse of a UPM is an involution and preserves a subset relation extended to UPMs.

Lemma 8.6 *For UPMs f and g between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, we have*

(a) $(f^{\smile})^{\smile} = f$;

(b) $\forall\, X \subseteq \mathcal{E}_1 : f(X) \subseteq g(X) \;\Leftrightarrow\; \forall\, Y \subseteq \mathcal{E}_2 : f^{\smile}(Y) \subseteq g^{\smile}(Y)$.

Proof

(a) By Lemma 8.5, we have

$$\rho_{((f^\smile)^\smile)} = (\rho_{(f^\smile)})^\smile = ((\rho_f)^\smile)^\smile = \rho_f \; .$$

Thus,

$$(f^\smile)^\smile = \mathrm{Im}_{\rho_{((f^\smile)^\smile)}} = \mathrm{Im}_{\rho_f} = f \; .$$

(b) Due to Part (a), we only need to show ($\Rightarrow$). Let $Y \subseteq \mathcal{E}_2$ arbitrary. We have

$$\begin{array}{lll} & f^\smile(Y) & \\ = & \{u \in \mathcal{E}_1 \mid \exists\, v \in Y : v\rho_{(f^\smile)}u\} & \\ = & \{u \in \mathcal{E}_1 \mid \exists\, v \in Y : v(\rho_f)^\smile u\} & \text{(Lemma 8.5 (b))} \\ = & \{u \in \mathcal{E}_1 \mid \exists\, v \in Y : u\rho_f v\} & \\ = & \{u \in \mathcal{E}_1 \mid \exists\, v \in Y \cap f(\{u\})\} & \\ \subseteq & \{u \in \mathcal{E}_1 \mid \exists\, v \in Y \cap g(\{u\})\} & \\ & & \text{(by the hypothesis that } \forall\, X \subseteq \mathcal{E}_1 : f(X) \subseteq g(X)) \\ = & g^\smile(Y) \; . & \text{(as shown above for } f) \end{array}$$

□

8.2 Process Abstractions

In this section we discuss maps between two process spaces, $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$ (the *domain* and the *co-domain*). We define the maps that are of interest to us by a commutative diagram involving two of the process composition operators. These operators can be any of $\sqcup$, $\oplus$, $+$, $\sqcap$, $\otimes$, and $\times$; call them $*$ and $\odot$.

Definition 8.3 *Map F between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$ is a* $*\odot$-PA *(for* 'process abstraction'*) if*

$$\forall \mathcal{B} \subseteq \mathcal{S}_{\mathcal{E}_1} : \; F(*_{p\in\mathcal{B}}\, p) \; = \; \odot_{p\in\mathcal{B}} F(p) \; .$$

Definition 8.3 is for arbitrary $*$ and $\odot$, but most applications use only $\sqcap\sqcap$-PAs, which are defined by taking $* = \odot = \sqcap$. In the following, we focus on maps of this type. Other process abstractions, when needed, can be obtained by applying the duality and ternary symmetry principles from Chapter 2.

We have mentioned process abstractions are constructed from UPMs, which, in turn, are represented by relations on execution sets. Our construction is as follows.

Theorem 8.7 (representation) *For $\sqcap\sqcap$-PA F between $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, there exist UPMs f, g, h, and i from $\wp(\mathcal{E}_1)$ to $\wp(\mathcal{E}_2)$ such that, for all p from $\mathcal{S}_{\mathcal{E}_1}$,*

$$F(p) = (f(\mathbf{as}\,p) \cup h(\mathbf{r}\,p),\ \overline{g(\mathbf{as}\,p) \cup i(\mathbf{r}\,p)})\ . \tag{8.1}$$

Proof We construct the UPMs in question from the images through F of two special processes that depend on a subset $X \subseteq \mathcal{E}_1$: $(X, \mathcal{E}_1)$ and $(X, \overline{X})$.

Let f, g, h, and i be maps between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$ such that

$$f(X) = \mathbf{as}\,F((X, \mathcal{E}_1))\ , \tag{8.2}$$

$$g(X) = \mathbf{r}\,F((X, \mathcal{E}_1))\ , \tag{8.3}$$

$$h(X) = \mathbf{as}\,F((X, \overline{X}))\ , \tag{8.4}$$

$$i(X) = \mathbf{r}\,F((X, \overline{X}))\ . \tag{8.5}$$

To show f, g, h, and i are UPMs, we show they preserve union.

$$\begin{array}{cll}
& f(\bigcup_{X\in\mathcal{M}} X) & \\
= & \mathbf{as}\,F((\bigcup_{X\in\mathcal{M}} X, \mathcal{E}_1)) & \text{(by choice of } f\text{)} \\
= & \mathbf{as}\,F(\sqcap_{X\in\mathcal{M}}(X, \mathcal{E}_1)) & \\
= & \mathbf{as}\,(\sqcap_{X\in\mathcal{M}} F((X, \mathcal{E}_1))) & \text{(Definition 8.3 for } * = \odot = \sqcap\text{)} \\
= & \bigcup_{X\in\mathcal{M}} \mathbf{as}\,F((X, \mathcal{E}_1)) & \\
= & \bigcup_{X\in\mathcal{M}} f(X)\ . & \text{(by choice of } f\text{)}
\end{array}$$

Thus, f is a UPM. The proofs for g, h, i are similar.

Now, we compute $F(p)$ for an arbitrary process p.

$$\begin{aligned}
& F((\mathbf{as}\,p, \overline{\mathbf{r}\,p})) \\
= \quad & F((\mathbf{as}\,p, \mathcal{E}_1) \sqcap (\mathbf{r}\,p, \overline{\mathbf{r}\,p})) \\
= \quad & F((\mathbf{as}\,p, \mathcal{E}_1)) \sqcap F((\mathbf{r}\,p, \overline{\mathbf{r}\,p})) \\
= \quad & (f(\mathbf{as}\,p), \overline{g(\mathbf{as}\,p)}) \sqcap (h(\mathbf{r}\,p), \overline{i(\mathbf{r}\,p)}) \\
= \quad & (f(\mathbf{as}\,p) \cup h(\mathbf{r}\,p), \overline{g(\mathbf{as}\,p) \cup i(\mathbf{r}\,p)}) \ .
\end{aligned}$$

□

From Definition 8.3, one can immediately deduce two structure-preserving properties of a $\sqcap\sqcap$-PA. Monotonicity with respect to refinement is particularly important in applications.

Theorem 8.8 *For $\sqcap\sqcap$-PA F between the process spaces over sets $\mathcal{E}_1$ and $\mathcal{E}_2$, and for processes p and q over $\mathcal{E}_1$, we have*

(a) $p \sqsubseteq q \ \Rightarrow \ F(p) \sqsubseteq F(q)$. *(monotonicity)*

(b) $F(\top) = \top$.

Proof

(a) For any processes p and q,

$$\begin{aligned}
& p \sqsubseteq q \Leftrightarrow p \sqcap q = p \Rightarrow F(p \sqcap q) = F(p) \\
& \Leftrightarrow F(p) \sqcap F(q) = F(p) && \text{(since } F \text{ is a } \sqcap\sqcap\text{-PA)} \\
& \Leftrightarrow F(p) \sqsubseteq F(q) \ .
\end{aligned}$$

(b) Take $\mathcal{B}$ in Definition 8.3 to be empty. □

Although Theorem 8.7 permits us to choose a $\sqcap\sqcap$-PA and its four UPMs in many ways, not all the choices are needed in applications. In the following, we focus on a simpler type of $\sqcap\sqcap$-PAs.

Definition 8.4 *For $\sqcap\sqcap$-PA F and UPMs f, g, h, and i that satisfy equation (8.1) from Theorem 8.7, we write $F = (f, g, h, i)^{\sqcap\sqcap}$. We say such F is* simple *(F is a* $\sqcap\sqcap$-SPA *for short), and we write $f^{\sqcap\sqcap} = F$ and $f = F_{\sqcap\sqcap}$, if $f = i$ and for all X we have $g(X) = h(X) = \emptyset$.*

The notation $f = F_{\sqcap\sqcap}$ in Definition 8.4 is well-defined because f is unique for a given $\sqcap\sqcap$-SPA F; observe that, for any subset X from the domain of f, we have $f(X) = \mathbf{as}\, F(X)$.

Simple $\sqcap\sqcap$-PAs preserve accessible and reject sets.

Lemma 8.9 *For UPM f between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$ and for process p over $\mathcal{E}_1$,*

$$\mathbf{as}\, f^{\sqcap\sqcap}(p) = f(\mathbf{as}\, p) \ ,$$
$$\mathbf{r}\, f^{\sqcap\sqcap}(p) = f(\mathbf{r}\, p) \ .$$

Proof Replace g, h, and i in Theorem 8.7 by $\emptyset$, $\emptyset$, and f, respectively. □

Informally speaking, the idea for PAs is to ensure consistency between the execution sets of a process and their images through the map. However, a map may create ambiguity when the images of disjoint sets, such as the accessible and the escape sets of a process, are not disjoint. This ambiguity can be resolved in several ways by assigning the co-domain executions in question to one or another execution set of the image process. Maps of the $\sqcap\sqcap$-SPA type are particularly important, because they resolve such ambiguity pessimistically, in favor of the accessible and reject sets of the image process, as in Lemma 8.9.

Example 8.6 The product of Listener and Speaker in Example 2.5 refines the etiquette machine of Chapter 2 if the latter is considered over execution set $\{e1,$

$e2$, ex, $m1$, $m2$, mx, $beep\}$ as in Example 2.6, but the etiquette machine is not refined by this product. The reason is that the etiquette machine does not 'control' action *beep*. For instance, execution $u = beep\ e1\ beep\ m1\ beep$ is accessible for the etiquette machine but not for Listener $\times$ Speaker, since u is an escape for Listener.

What if we delete action *beep* from Listener$\times$Speaker, using the deletion UPM of Example 8.3? Letting $F = (\mathbf{del}(\{beep\}))^{\sqcap\sqcap}$, we have that F(Listener$\times$Speaker) is refined by the etiquette machine, if the latter is considered over execution set $\{e1, e2, ex, m1, m2, mx\}$ as in Example 2.1. In particular, execution $v = e1m1$, obtained by deleting all *beep*s from u, is accessible for F(Listener$\times$Speaker), even though u is an escape for Listener$\times$Speaker. This is because we can also obtain v by deleting all *beep*s from $u' = e1\ beep\ m1$, while u' is accessible for Listener$\times$Speaker.

Notice that, in deciding that v will be accessible for the image process, we take the position that we do not care about 'happy ending' cases where the original process actually avoids u, as long as there exist 'sad ending' cases where the original process does not avoid u'. This is the effect of using a $\sqcap\sqcap$-SPA. $\square$

A key question regarding process abstractions is how they correlate the notions of correctness in two process spaces.

Theorem 8.10 (reciprocity) *For $\sqcap\sqcap$-SPA F between the process spaces over sets $\mathcal{E}_1$ and $\mathcal{E}_2$, let $F^{\smile}$ be the $\sqcap\sqcap$-SPA $((F_{\sqcap\sqcap})^{\smile})^{\sqcap\sqcap}$ (call it the* converse *of F). Then, for all processes p over $\mathcal{S}_{\mathcal{E}_1}$ and q over $\mathcal{S}_{\mathcal{E}_2}$,*

$$p \times F^{\smile}(q) \in \mathcal{R}_{\mathcal{E}_1} \ \Leftrightarrow\ F(p) \times q \in \mathcal{R}_{\mathcal{E}_2}\ .$$

Proof Let $f = F_{\sqcap\sqcap}$. We have

$$p \times F^{\smile}(q) \in \mathcal{R}_{\mathcal{E}_1}$$

$$
\begin{array}{ll}
\Leftrightarrow & \mathbf{as}\, p \cap \mathbf{r}\, F^{\smile}(q) = \mathbf{r}\, p \cap \mathbf{as}\, F^{\smile}(q) = \emptyset \\
& \hfill (\text{calculus, from definitions of } \times \text{ and } \mathcal{R}) \\
\Leftrightarrow & \mathbf{as}\, p \cap f^{\smile}(\mathbf{r}\, q) = \mathbf{r}\, p \cap f^{\smile}(\mathbf{as}\, q) = \emptyset \qquad (\text{Lemma 8.9 and choice of } F^{\smile}) \\
\Leftrightarrow & f(\mathbf{as}\, p) \cap \mathbf{r}\, q = f(\mathbf{r}\, p) \cap \mathbf{as}\, q = \emptyset \qquad (\text{Lemma 8.2, Property (a)}) \\
\Leftrightarrow & \mathbf{as}\, F(p) \cap \mathbf{r}\, q = \mathbf{r}\, F(p) \cap \mathbf{as}\, q = \emptyset \\
\Leftrightarrow & F(p) \times q \in \mathcal{R}_{\mathcal{E}_2} \ . \qquad (\text{calculus, from definitions of } \times \text{ and } \mathcal{R}) \qquad \square
\end{array}
$$

Together with Theorem 2.12 from Chapter 2, Theorem 8.10 shows that the pair of maps $(-F, -F^{\smile})$ is a Galois connection with respect to the converse of reflection:

$$p \sqsupseteq -F^{\smile}(q) \ \Leftrightarrow\ q \sqsupseteq -F(p) \ .$$

This shows how a $\sqcap\sqcap$-SPA keeps consistency of the refinement relations of $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$.

We now show that compositions of $\sqcap\sqcap$-SPAs correspond to compositions of UPMs. This property is important for determining transitivity properties for process maps, as will be illustrated for instance in Subsection 8.4.4.

Lemma 8.11 *For $\sqcap\sqcap$-SPAs F and G between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, we have*

$$G \circ F = (G_{\sqcap\sqcap} \circ F_{\sqcap\sqcap})^{\sqcap\sqcap} \ .$$

Proof For any process p over $\mathcal{E}_1$,

$$
\begin{array}{ll}
& (G_{\sqcap\sqcap} \circ F_{\sqcap\sqcap})^{\sqcap\sqcap}(p) \\
= & ((G_{\sqcap\sqcap} \circ F_{\sqcap\sqcap})(\mathbf{as}\, p), \overline{(G_{\sqcap\sqcap} \circ F_{\sqcap\sqcap})(\mathbf{r}\, p)}) \\
= & (G_{\sqcap\sqcap}(\mathbf{as}\, F(p)), \overline{G_{\sqcap\sqcap}(\mathbf{r}\, F(p))}) \qquad (\text{Lemma 8.9}) \\
= & (\mathbf{as}\, G(F(p)), \overline{\mathbf{r}\, G(F(p))}) \qquad (\text{Lemma 8.9 again}) \\
= & G(F(p)) \ . \qquad \square
\end{array}
$$

The process abstractions described so far, namely $\sqcap\sqcap$-PAs, can be reflected and rotated in both domain and co-domain, to obtain thirty-five other types of process abstractions. In most applications, only $\sqcap\sqcap$-PAs are used; however, in some applications we have also encountered process abstractions of other types. As an example, we indicate here how to handle $\sqcup\times$-PAs, which are defined by taking $* = \sqcup$ and $\odot = \times$ in Definition 8.3.

Proposition 8.12 *For $\sqcup\times$-PA F between $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, there exist UPMs f, g, h, and i from $\wp(\mathcal{E}_1)$ to $\wp(\mathcal{E}_2)$, such that, for every process p over $\mathcal{E}_1$,*

$$\mathbf{v}\, F(p) = f(\mathbf{at}\, p) \cup g(\mathbf{e}\, p) \ , \tag{8.6}$$

$$\mathbf{e}\, F(p) = h(\mathbf{at}\, p) \cup i(\mathbf{e}\, p) \ . \tag{8.7}$$

For F, f, g, h, and i that satisfy (8.6) and (8.7), we write $F = (f, g, h, i)^{\sqcup\times}$. Then, for every process p over $\mathcal{E}_1$,

$$(f, g, h, i)^{\sqcup\times}(p) = \backslash(f, g, h, i)^{\sqcap\sqcap}(-p) \ .$$

Thus, one can deal with a $\sqcup\times$-PA just by rotations and reflections applied to the result and argument of a $\sqcap\sqcap$-PA. It follows by duality and ternary symmetry that the properties shown so far for $\sqcap\sqcap$-PAs apply to $\sqcup\times$-PAs and to the other types of process abstractions as well.

8.3 Verifications on Images

The basic verification problem is to determine robustness of a product of processes; the verification of refinement can be reduced to this basic problem by using Theorem 2.12. We are aiming to study relationships between robustness of $\times_{p\in\mathcal{B}}\, p$ and robustness of $\times_{p\in\mathcal{B}}F(p)$: under which properties of maps or processes does one of these robustness conditions imply the other?

Two properties which are relevant for verification on images happen to hold for all processes and $\sqcap\sqcap$-SPAs.

Lemma 8.13 *For $\sqcap\sqcap$-SPA F between the process spaces over sets $\mathcal{E}_1$ and $\mathcal{E}_2$, for processes p and q over $\mathcal{E}_1$, and for set $\mathcal{B} \subseteq \mathcal{S}_{\mathcal{E}_1}$ of processes, we have*

(a) $\times_{p\in\mathcal{B}}F(p) \sqsubseteq F(\times_{p\in\mathcal{B}}\, p)$;

(b) $p \in \mathcal{R}_{\mathcal{E}_1} \;\Rightarrow\; F(p) \in \mathcal{R}_{\mathcal{E}_2}$.

Proof Let UPM f be such that $F = f^{\sqcap\sqcap}$.

(a) We have

$$
\begin{aligned}
& F(\times_{p\in\mathcal{B}}\, p) \\
= \quad & F(\ (\ \textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p\ ,\ \bigcap_{p\in\mathcal{B}} \mathbf{at}\, p \cup \overline{\bigcap_{q\in\mathcal{B}} \mathbf{as}\, q}\ )\ ) && \text{(Definition 2.9)} \\
= \quad & F(\ (\ \textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p\ ,\ \overline{(\bigcup_{p\in\mathcal{B}} \overline{\mathbf{at}\, p}) \cap \bigcap_{q\in\mathcal{B}} \mathbf{as}\, q}\ )\ ) \\
& && \text{(De Morgan's laws for } \cup \text{ and } \cap) \\
= \quad & F(\ (\ \textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p\ ,\ \overline{(\bigcup_{p\in\mathcal{B}} \mathbf{r}\, p) \cap \bigcap_{q\in\mathcal{B}} \mathbf{as}\, q}\ )\ ) \\
= \quad & F(\ (\ \textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p\ ,\ \overline{\bigcup_{p\in\mathcal{B}}((\bigcap_{q\in\mathcal{B}} \mathbf{as}\, q) \cap \mathbf{r}\, p)}\ )\ ) && \text{(distributivity of } \cap \text{ through } \cup) \\
= \quad & (\ f(\textstyle\bigcap_{p\in\mathcal{B}} \mathbf{as}\, p),\ \overline{f(\bigcup_{p\in\mathcal{B}}((\bigcap_{q\in\mathcal{B}} \mathbf{as}\, q) \cap \mathbf{r}\, p))}\ ) && \text{(Lemma 8.9)} \\
\sqsupseteq \quad & (\ \textstyle\bigcap_{p\in\mathcal{B}} f(\mathbf{as}\, p),\ \overline{\bigcup_{p\in\mathcal{B}}((\bigcap_{q\in\mathcal{B}} f(\mathbf{as}\, q)) \cap f(\mathbf{r}\, p))}\ )
\end{aligned}
$$

(Lemma 8.2 (Property (b)) and Lemma 8.3 (b), since now the accessible and the reject sets of the process in question are both larger than they were in the previous step)

$$
\begin{aligned}
= \quad & \times_{p\in\mathcal{B}}(f(\mathbf{as}\, p), \overline{f(\mathbf{r}\, p)}) \\
& && \text{(by rephrasing Definition 2.9 as done above for } \times_{p\in\mathcal{B}}\, p) \\
= \quad & \times_{p\in\mathcal{B}}F(p)\ . && \text{(Lemma 8.9)}
\end{aligned}
$$

(b) Since $f(\emptyset) = \emptyset$ (Lemma 8.3), we have

$$p \in \mathcal{R}_{\mathcal{E}_1} \;\Leftrightarrow\; \mathbf{r}\,p = \emptyset \;\Rightarrow\; f(\mathbf{r}\,p) = \emptyset \;\Leftrightarrow\; \mathbf{r}\,F(p) = \emptyset \;\Leftrightarrow\; F(p) \in \mathcal{R}_{\mathcal{E}_2}\,. \qquad \Box$$

Certain simple properties of an underlying relation entail several additional properties for the corresponding UPMs and $\sqcap\sqcap$-SPAs.

Definition 8.5 *Let $\rho \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ be a relation. We say that ρ is* surjective *if $\mathrm{Im}_\rho(\mathcal{E}_1) = \mathcal{E}_2$; that ρ is* injective *if, for all $u_1, u_2 \in \mathcal{E}_1$ and $v \in \mathcal{E}_2$ such that $u_1 \rho v$ and $u_2 \rho v$, we have $u_1 = u_2$; that ρ is* co-surjective *if $\rho^{\smile}$ is surjective; and that ρ is* co-injective *if $\rho^{\smile}$ is injective.*

Observe that a relation $\rho \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ is a *partial function* from $\mathcal{E}_1$ to $\mathcal{E}_2$ if and only if ρ is co-injective; relation $\rho \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ is a *function* from $\mathcal{E}_1$ to $\mathcal{E}_2$ iff ρ is co-injective and co-surjective. A function is *bijective* iff it is injective and surjective.

8.3.1 Optimistic Approximations

Injective relations generate optimistic approximations of processes, in the sense that verifications of robustness of product on image processes may yield false positives but never false negatives.

Example 8.7 Suppose we try to check robustness of the product of Listener and Speaker from Example 2.5 by examining all executions that have exactly four actions. (The number four was chosen arbitrarily.) For this, we can use as underlying relation the identity relation of set B, where $B = AAAA$ and $A = \{e1, e2, ex, m1, m2, mx, beep\}$. The identity relation of set B is $\mathbf{id}_B = \{(u, u) \mid u \in B\}$, and can be used as a relation on A^* as well, since $B \subset A^*$. Let F be the corresponding $\sqcap\sqcap$-SPA, that is, $F = (\mathrm{Im}_{\mathbf{id}_B})^{\sqcap\sqcap}$.

We observe that execution $v = e1\ e1\ beep\ m1$ is a reject for $F(\text{Listener}) \times F(\text{Speaker})$. Since $\mathbf{id}_B$ is injective, we know this execution corresponds to a reject of the product of the original processes, since there is only one domain execution, call it u, that is in relation to the reject we have found, and u must be a reject for Listener and a goal for Speaker, since v is a reject for $F(\text{Listener})$ and a goal for $F(\text{Speaker})$. Incidentally, $u = v$ in this example, but this needs not be the case for other injective relations. □

We are interested not just in injective relations, but also in processes for which a given $\sqcap\sqcap$-SPA behaves as if it were based on an injective relation in the sense described above.

Definition 8.6 *Let f be a UPM between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, and let F be a $\sqcap\sqcap$-PA between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$. Subset X of $\mathcal{E}_1$ is* f-subfixed *if $f^{\smile}(f(X)) \subseteq X$. Process p over $\mathcal{E}_1$ is* optimistically approximated *by F (p is* F-optimistic *for short) if $F^{\smile}(F(p)) \sqsupseteq p$.*

Proposition 8.14 *For $\sqcap\sqcap$-SPA $F = f^{\sqcap\sqcap}$, process p is F-optimistic if and only if* $\mathbf{as}\,p$ *and* $\mathbf{r}\,p$ *are f-subfixed.*

Proof This follows immediately from definitions. □

Lemma 8.15 *For UPM f between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, and for set $\mathcal{M}$ of f-subfixed sets from $\mathcal{E}_1$, we have*

$$f(\textstyle\bigcap_{X\in\mathcal{M}} X) = \bigcap_{X\in\mathcal{M}} f(X)\ .$$

Proof The ($\subseteq$) part is Lemma 8.3 (b). For showing ($\supseteq$), assume for the sake of contradiction that

$$\exists\, v \in (\bigcap_{X\in\mathcal{M}} f(X)) \setminus f(\bigcap_{X\in\mathcal{M}} X)\ .$$

If $\mathcal{M} = \emptyset$, the assumption is trivially contradicted. If $\mathcal{M} \neq \emptyset$, let $X_1 \in \mathcal{M}$ and $u_1 \in X_1$ such that $u_1 \rho_f v$. For arbitrary X in $\mathcal{M}$, by our assumption there exists $u \in X : u \rho_f v$. Since $u_1 \rho_f v$ and $u \in X$, $u_1 \in f^{\smile}(f(X))$. Since X is f-subfixed, $u_1 \in X$. Thus, $u_1 \in \bigcap_{X\in\mathcal{M}} X$. Since $u_1 \rho_f v$, it follows that $v \in f(\bigcap_{X\in\mathcal{M}} X)$, contradicting the assumption. $\square$

Theorem 8.16 (optimistic approximation) *For $\sqcap\sqcap$-SPA F between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, and for set $\mathcal{B}$ of F-optimistic processes from $\mathcal{S}_{\mathcal{E}_1}$,*

$$\times_{p\in\mathcal{B}}\, p \in \mathcal{R}_{\mathcal{E}_1} \;\Rightarrow\; \times_{p\in\mathcal{B}} F(p) \in \mathcal{R}_{\mathcal{E}_2}\ .$$

Proof Let $f = F_{\sqcap\sqcap}$. We have

$$\begin{array}{lll}
& \times_{p\in\mathcal{B}} p \in \mathcal{R}_{\mathcal{E}_1} & \\
\Rightarrow & F(\times_{p\in\mathcal{B}} p) \in \mathcal{R}_{\mathcal{E}_2} & \text{(Lemma 8.13 (b))} \\
\Leftrightarrow & f(\bigcup_{p\in\mathcal{B}}((\bigcap_{q\in\mathcal{B}} \mathbf{as}\, q) \cap \mathbf{r}\, p)) = \emptyset & \text{(Definitions 2.9 and 2.2)} \\
\Leftrightarrow & \bigcup_{p\in\mathcal{B}} f((\bigcap_{q\in\mathcal{B}} \mathbf{as}\, q) \cap \mathbf{r}\, p) = \emptyset & \text{(Property (b) in Lemma 8.2)} \\
\Leftrightarrow & \bigcup_{p\in\mathcal{B}}((\bigcap_{q\in\mathcal{B}} f(\mathbf{as}\, q)) \cap f(\mathbf{r}\, p)) = \emptyset & \text{(Lemma 8.15)} \\
\Leftrightarrow & \bigcup_{p\in\mathcal{B}}((\bigcap_{q\in\mathcal{B}} \mathbf{as}\, F(q)) \cap \mathbf{r}\, F(p)) = \emptyset & \text{(Lemma 8.9)} \\
\Leftrightarrow & \times_{p\in\mathcal{B}} F(p) \in \mathcal{R}_{\mathcal{E}_2} \qquad \text{(Definitions 2.9 and 2.2)} & \square
\end{array}$$

Theorem 8.16 tells us that verifications of robustness of product (our basic verification problem) on images produce no false negatives (no false flaws) for optimistically approximated processes; hence the term 'optimistic'.

Now we characterize relations whose $\sqcap\sqcap$-SPAs approximate all processes optimistically. The benefit of knowing that a relation has this property is that verifications on images with no false negatives can be performed for all processes rather than just for a restricted class.

Theorem 8.17 (injective reduction) *For $\sqcap\sqcap$-SPA $F = f^{\sqcap\sqcap}$ between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, the following properties are equivalent:*

(a) *relation ρ_f is injective;*

(b) *every set $X \subseteq \mathcal{E}_1$ is f-subfixed;*

(c) *every process p over $\mathcal{E}_1$ is F-optimistic.*

Proof

((a) $\Rightarrow$ (b)) For arbitrary w from $f^{\smile}(f(X))$, we have

$$\exists\, v \in f(X), u \in X : u\rho_f v \wedge w\rho_f v \ .$$

Since ρ_f is injective, $u = w$ and thus $w \in X$. Since $w \in f^{\smile}(f(X))$ was arbitrary, we have $f^{\smile}(f(X)) \subseteq X$, that is X is subfixed.

((b) $\Rightarrow$ (a)) For $u, w \in \mathcal{E}_1$ and $v \in \mathcal{E}_2$, if $u\rho_f v$ and $w\rho_f v$ then $v \in f(\{u\})$ and $w \in f^{\smile}(\{v\})$, thus $w \in f^{\smile}(f(\{u\}))$. Since all subsets of $\mathcal{E}_1$ are f-subfixed, $f^{\smile}(f(\{u\})) \subseteq \{u\}$ and thus $w \in \{u\}$ and $w = u$. Thus, ρ_f is injective.

((b) $\Rightarrow$ (c)) Let $p \in \mathcal{S}_{\mathcal{E}_1}$ arbitrary.

$$\begin{array}{cll} & F^{\smile}(F(p)) & \\ = & (f^{\smile}(f(\mathbf{as}\,p)), \overline{f^{\smile}(f(\mathbf{r}\,p))}) & \text{(Definition of } F^{\smile} \text{ and Lemma 8.9)} \\ \sqsupseteq & (\mathbf{as}\,p, \overline{\mathbf{r}\,p}) & \text{(since } \mathbf{as}\,p \text{ and } \mathbf{r}\,p \text{ are } f\text{-subfixed)} \\ = & p \ . & \end{array}$$

((b) $\Rightarrow$ (c)) Let $X \subseteq \mathcal{E}_1$ arbitrary.

$$\begin{array}{ll} & f^{\smile}(f(X)) \\ = & \mathbf{as}\, F^{\smile}(F((X,\mathcal{E}_1))) \\ \subseteq & \mathbf{as}\,(X,\mathcal{E}_1) \qquad \text{(since } (X,\mathcal{E}_1) \text{ is } F\text{-optimistic)} \\ = & X\ . \end{array}$$

$\square$

8.3.2 Pessimistic Approximations

Co-surjective relations generate pessimistic approximations of processes, in the sense that verifications of robustness of product on image processes may yield false negatives but never false positives.

Example 8.8 Applying the deletion operation for the product of Listener and Speaker in Example 8.6 will make execution $e1\ e1\ m1\ m1$ a reject for the image process, since it is in relation with the reject execution $e1\ e1\ beep\ m1\ beep\ m1$ of the original product process.

Notice that executions $e1\ beep\ e1\ m1\ beep\ m1$ and $e1\ e1\ m1\ m1\ beep$ are also in the deletion relation with $e1\ e1\ m1\ m1$, although the first is a goal and the second an escape for the original product process. In words, by what we can tell without seeing *beep*, the original product process might tolerate $e1\ e1\ m1\ m1$ from time to time; however, we don't care about that possibility if we know that it is also possible to have a reject.

In this sense, although some information may be lost by a relation that is co-surjective but not injective, the verifications of robustness of product on the image processes are sufficient to guarantee correctness of the original processes. Some information is lost by allowing common images for 'good' executions like $e1\ beep$

$e1\ m1\ beep\ m1$ and 'bad' executions like $e1\ e1\ beep\ m1\ beep\ m1$, but the image execution will be qualified as 'bad' as long as it has at least one 'bad' original. □

We are interested not just in co-surjective relations, but also in processes for which a given ⊓⊓-SPA behaves as if it were based on an co-surjective relation in the sense described above.

Definition 8.7 *Let f be a UPM between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, and let F be a ⊓⊓-PA between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$. Subset X of $\mathcal{E}_1$ is f-*superfixed *if $f^{\smile}(f(X)) \supseteq X$. Process p over $\mathcal{E}_1$ is* pessimistically approximated *by F (p is F-*pessimistic* for short) if $F^{\smile}(F(p)) \sqsubseteq p$.*

Proposition 8.18 *For ⊓⊓-SPA $F = f^{\sqcap\sqcap}$, process p is F-pessimistic if and only if* $\mathbf{as}\, p$ *and* $\mathbf{r}\, p$ *are f-superfixed.*

Proof This follows immediately from definitions. □

Lemma 8.19 *For UPM f between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, and for f-superfixed subset X of $\mathcal{E}_1$, we have*

$$f(X) = \emptyset \Rightarrow X = \emptyset\ .$$

Proof We have

$$\begin{array}{lll} & f(X) = \emptyset & \\ \Rightarrow & f^{\smile}(f(X)) = \emptyset & \text{(since } f^{\smile} \text{ is also a UPM)} \\ \Rightarrow & X = \emptyset\ . & \text{(since } X \subseteq f^{\smile}(f(X))\text{)} \end{array}$$

□

Theorem 8.20 (pessimistic approximation) *For ⊓⊓-SPA F between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, and for set $\mathcal{B}$ of F-pessimistic processes from $\mathcal{S}_{\mathcal{E}_1}$,*

$$\times_{p\in\mathcal{B}}\, p \in \mathcal{R}_{\mathcal{E}_1} \;\Leftarrow\; \times_{p\in\mathcal{B}} F(p) \in \mathcal{R}_{\mathcal{E}_2}\ .$$

Proof Let $f = F_{\sqcap\sqcap}$. We have

$$\begin{array}{lll}
 & \times_{p\in\mathcal{B}} F(p) \in \mathcal{R}_{\mathcal{E}_2} & \\
\Rightarrow & F(\times_{p\in\mathcal{B}}\, p) \in \mathcal{R}_{\mathcal{E}_2} & \text{(Lemma 8.13 (a))} \\
\Leftrightarrow & f(\mathbf{r}\ \times_{p\in\mathcal{B}}\ p) = \emptyset & \\
\Rightarrow & \mathbf{r}\ \times_{p\in\mathcal{B}}\, p = \emptyset & \text{(Lemma 8.9)} \\
\Rightarrow & \times_{p\in\mathcal{B}} p \in \mathcal{R}_{\mathcal{E}_1}\ . \quad \text{(Lemma 8.19)} & \square
\end{array}$$

Theorem 8.20 tells us that verifications of robustness of product (our basic verification problem) on images produce no false positives (no false passes) for pessimistically approximated processes; hence the term 'pessimistic'.

Now we characterize relations whose $\sqcap\sqcap$-SPAs approximate all processes pessimistically. The benefit of knowing that a relation has this property is that verifications on images with no false positives can be performed for all processes rather than just for a restricted class.

Theorem 8.21 (co-surjective reduction) *For $\sqcap\sqcap$-SPA $F = f^{\sqcap\sqcap}$ between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$, the following properties are equivalent:*

(a) *relation ρ_f is co-surjective;*

(b) *every set $X \subseteq \mathcal{E}_1$ is f-superfixed;*

(c) *every process p over $\mathcal{E}_1$ is F-pessimistic.*

Proof

((a) $\Rightarrow$ (b)) For arbitrary u from X, we have

$$
\begin{array}{lll}
& \exists\, v : u\rho_f v & (\rho_f \text{ is co-surjective}) \\
\Rightarrow & \exists\, v : u\rho_f v \wedge v\rho_f{}^{\smile} u & (\text{Lemma 8.5 (a)}) \\
\Rightarrow & \exists\, v : u(\rho_f \circ \rho_f{}^{\smile})u & \\
\Rightarrow & u \in f^{\smile}(f(X)) \ . &
\end{array}
$$

Since $u \in X$ was arbitrary, we have $X \subseteq f^{\smile}(f(X))$.

((b) $\Rightarrow$ (a)) Let $u \in \mathcal{E}_1$ arbitrary. Since $\{u\} \subseteq f^{\smile}(f(\{u\}))$, there exists $v \in f(\{u\})$ such that $u\rho_f v$. Since u was arbitrary, ρ_f is co-surjective.

((b) $\Rightarrow$ (c)) Let $p \in \mathcal{S}_{\mathcal{E}_1}$ arbitrary.

$$
\begin{array}{ll}
& F^{\smile}(F(p)) \\
= & (f^{\smile}(f(\mathbf{as}\, p)), \overline{f^{\smile}(f(\mathbf{r}\, p))}) \\
\sqsubseteq & (\mathbf{as}\, p, \overline{\mathbf{r}\, p}) \\
= & p \ .
\end{array}
$$

((b) $\Rightarrow$ (c)) Let $X \subseteq \mathcal{E}_1$ arbitrary.

$$
\begin{array}{ll}
& f^{\smile}(f(X)) \\
= & \mathbf{as}\, F^{\smile}(F((X, \mathcal{E}_1))) \\
\supseteq & \mathbf{as}\, (X, \mathcal{E}_1) \\
= & X \ .
\end{array}
$$

□

8.3.3 Independence

One can combine optimistic and pessimistic approximations to achieve verifications on images that are equivalent to verifications of robustness of products of original processes. Our notion of independence of a process from a ⊓⊓-PA is simply the conjunction of optimistic and pessimistic approximations.

Definition 8.8 *Let f be a UPM between power sets $\wp(\mathcal{E}_1)$ and $\wp(\mathcal{E}_2)$, and let F be a $\sqcap\sqcap$-SPA between process spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$. Subset X of $\mathcal{E}_1$ is f-*fixed* if $f^{\smile}(f(X)) = X$. Process p over $\mathcal{E}_1$ is F-*independent* if $F^{\smile}(F(p)) = p$.*

It follows from Theorems 8.16 and 8.20 that verifications of robustness of product on images of F-independent processes through F are equivalent to verifications of robustness of product directly on the original processes. Furthermore, the condition for independence of all processes from a given $\sqcap\sqcap$-SPA F is injectivity and co-surjectivity of the underlying relation. This can be proved by combining Theorems 8.17 and 8.21, or directly as follows. Underlying relations that are both injective and co-surjective are sometimes encountered in applications, such as relabeling of a process automata; these relations are characterized by the property that they admit a right-inverse. Using the right-inverse, one may reconstruct an original process from an image process, so no information is lost. Recall that, when composing relations, the relation on the right applies last.

Theorem 8.22 *For relation ρ on sets $\mathcal{E}_1$ and $\mathcal{E}_2$, the following properties are equivalent.*

(a) *ρ is injective and co-surjective;*

(b) $\rho \circ (\rho^{\smile}) = \mathbf{id}_{\mathcal{E}_1}$ *;*

(c) *all processes over $\mathcal{E}_1$ are $(\mathrm{Im}_\rho)^{\sqcap\sqcap}$-independent.*

Proof Let $\eta = \rho^{\smile}$ and $f = \mathrm{Im}_\rho$.

(**(a)**$\Rightarrow$**(b)**) Assuming ρ is injective and co-surjective, we show $u\ (\rho \circ \eta)\ w$ iff $u = w$.

Since ρ is co-surjective, there exists a v in $\mathcal{E}_2$ such that $u\ \rho\ v$. By the definition of converse relations, we have $v\ \eta\ u$ as well. Thus, $u\ (\rho \circ \eta)\ u$.

By the definitions of converse and composite relations, if $u\ (\rho \circ \eta)\ w$ then there exists v such that $u\ \rho\ v$ and $w\ \rho\ v$. Since ρ is injective, $u = w$.

((b)⇒(a)) Assuming $u\ (\rho \circ \eta)\ w$ iff $u = w$, we show ρ is co-surjective and injective.

Let $u \in \mathcal{E}_1$. Since $u\ (\rho \circ \eta)\ u$, there exists, by the definition of composite relations, a v in $\mathcal{E}_2$ such that $u\ \rho\ v$ (and $v\ \eta\ u$). Since u was arbitrary, ρ is co-surjective.

Let $u_1, u_2 \in \mathcal{E}_1$ and $v \in \mathcal{E}_2$ such that $u_1\ \rho\ v$ and $u_2\ \rho\ v$. By the definitions of converse and composite relations, we have $u_1\ (\rho \circ \eta)\ u_2$. By the assumption that $u\ (\rho \circ \eta)\ w$ iff $u = w$, it follows that $u_1 = u_2$.

((b)⇒(c)) For every subset X of $\mathcal{E}_1$, we have $f^{\smile}(f(X)) = X$. Applying this to the execution sets of an arbitrary process p, we obtain that p is $f^{\sqcap\sqcap}$-independent.

((c)⇒(b)) For every subset X of $\mathcal{E}_1$, let p_X be the process $(X, \mathcal{E}_1)$. Since by hypothesis $(f^{\smile})^{\sqcap\sqcap}(f^{\sqcap\sqcap}(p_X)) = p_X$ and by Lemmas 8.9 and 8.11, $f^{\smile}(f(\mathbf{as}\, p_X)) = \mathbf{as}\, f^{\smile}(f(p_X))$, we have $f^{\smile}(f(X)) = f^{\smile}(f(\mathbf{as}\, p_X)) = \mathbf{as}\, p_X = X$. □

8.4 Examples of Applications

Process abstractions can model a surprisingly large variety of maps between processes. Here we list several such maps, give illustrative examples, and discuss the meaning of some of the general algebraic properties of process abstractions in the particular applications.

Where relevant, we indicate the conditions under which one can perform verifications on images of processes through process abstractions instead of verifications of the original processes. In particular, where relevant we indicate what are the converse maps and the independent processes of a certain map, and whether the underlying relation is injective or co-surjective.

8.4.1 Relabeling of Actions

Relabeling operators simply replace the names of some of the actions or ports in a system by the names of other actions or ports. To model relabelings, we use a *substitution function* $h : \mathcal{U}_1 \to \mathcal{U}_2$, which is an injective function between two sets $\mathcal{U}_1$ and $\mathcal{U}_2$ of actions. The replacements occur simultaneously for all actions in $\mathcal{U}_1$, and the injectivity condition ensures that there are no labeling conflicts.

The execution sets are $\mathcal{E}_1 = (\mathcal{U}_1)^*$ and $\mathcal{E}_2 = (\mathcal{U}_2)^*$. The underlying relationship is a function, which is also denoted by h because it naturally extends the substitution function as follows:

$$h(\varepsilon) = \varepsilon, \text{ and}$$
$$\forall\, u \in \mathcal{E}_1, a \in \mathcal{U}_1 : h(ua) = h(u)h(a)\ .$$

Since the underlying relation is an injective function, it is an injective and co-surjective relation. Thus, by Theorems 8.17 and 8.21, verifications on the relabeled processes are equivalent to verifications on the original processes; by Theorem 8.22, all processes are independent for the corresponding $\sqcap\sqcap$-SPA $(\mathrm{Im}_h)^{\sqcap\sqcap}$. It follows from Theorem 8.8 that relabeling is monotonic with respect to refinement.

Example 8.9 For the etiquette machine of Example 2.1, if the substitution function replaces $m1$ by mx and mx by $m1$, then the resulting machine does not refine the original: $e1\ mx$ is an escape for the original process and a goal for the image process. However, if we apply the same substitution to the environment process as well, the product of the environment's image and the machine's image (through the corresponding $\sqcap\sqcap$-SPA) is robust, just like the product of the original environment and machine processes. $\square$

The concepts and properties above extend naturally to infinite traces and other types of executions that rely on action sets.

8.4.2 Restricted Execution Sets

Sometimes it is convenient to analyze a system by examining only some of its executions. For studying restrictions of execution sets, we use identity relations on subsets.

For set $\mathcal{E}$, let the identity relation on $\mathcal{E}$ be $\mathbf{id}_{\mathcal{E}} = \{(u, u) \mid u \in \mathcal{E}\}$. For sets $\mathcal{E}_1$, $\mathcal{E}_2$, and A such that $A \subseteq \mathcal{E}_1$ and $A \subseteq \mathcal{E}_2$, $\mathbf{id}_A$ is also a binary relation between $\mathcal{E}_1$ and $\mathcal{E}_2$.

For instance, when simulating a process, only one execution is considered; let that execution be denoted by u. The behavior under simulation can be represented by another process, which is the image of the original process through the $\sqcap\sqcap$-SPA corresponding to relation $\mathbf{id}_{\{u\}}$. Due to the injectivity of $\mathbf{id}_{\{u\}}$, verifications performed on simulated processes will not produce false negatives (false flaws). Since there is at most one non-escape execution in the co-domain, verifications in the co-domain consist of checking that that execution is not a reject for the respective processes.

8.4.3 Process Derivatives

Derivatives of formal expressions were introduced by J.A. Brzozowski in [Brz64]. Similar operators have been applied to various types of processes in [Hoa85] (the "after" operator) and other theories of concurrency. For language $L \subseteq \mathcal{U}^*$ and word $w \in \mathcal{U}^*$,

$$D_w L = \{v \in \mathcal{U}^* \mid wv \in L\} \ .$$

Process derivatives result from a binary relation d_w on $\mathcal{U}^*$ such that $u \ d_w \ v$ iff $v = wu$. Then, $D_w = \mathrm{Im}_{d_w}$ is the corresponding UPM.

Definition 8.9 *For process p over $\mathcal{U}^*$ and word $w \in \mathcal{U}^*$, the* derivative *of p by w is the process*

$$(D_w)^{\sqcap\sqcap}(p) \ .$$

Monotonicity of process derivatives with respect to refinement, as well as other commutative diagram properties, follows from Theorem 8.8. Using Lemma 8.11, composition properties such as $D_{w_2}^{\sqcap\sqcap}(D_{w_1}^{\sqcap\sqcap}(p)) = D_{w_1 w_2}^{\sqcap\sqcap}(p)$ follow from the corresponding properties of d_w on single executions.

Derivatives can be used to re-initialize a process.

Example 8.10 Suppose we run a plant producing etiquette machines like that in Example 2.1, but some of the buyers would only use them in their reject states. Then we might as well deliver pre-annoyed etiquette machines: we can set up a derivative shop that applies $D_{ex}^{\sqcap\sqcap}$ to an etiquette machine, thereby changing its initial state to a reject state. □

Since relation d_w is injective, verifications of robustness of product on derivative processes yield no false negatives. This amounts to checking the behavior after w_1, w_2, etc.; if flaws are discovered, they are also flaws of the original process p.

8.4.4 Deletion, Hiding, and Projection

Projections, hidings, and deletions have underlying relations of the form described in Example 8.3. These relations are co-surjective, so verifications of robustness of product on image processes through $\sqcap\sqcap$-SPAs yield no false positives.

By Propositions 8.14 and 8.18, the independent processes of deletion, hiding, or projection are those processes whose accessible and reject execution sets are invariant to deleting and then inserting arbitrarily actions from a given alphabet.

Monotonicity of these operations (with respect to refinement) follows from Theorem 8.8 (a). Also, Lemma 8.11 enables us to study properties for compositions of projections by considering just the underlying relationships instead of mappings between processes. For example, let $\mathcal{U}_1$ and $\mathcal{U}_2$ be subsets of $\mathcal{U}$. One obtains the same results if one deletes from an execution u all actions of $\overline{\mathcal{U}_1}$ and then all actions of $\overline{\mathcal{U}_2}$, or if one deletes from u directly all actions of $\overline{\mathcal{U}_1 \cap \mathcal{U}_2}$. From Lemma 8.11 it follows that, for process p, we have $p{\downarrow}\mathcal{U}_1{\downarrow}\mathcal{U}_2 = p{\downarrow}(\mathcal{U}_1 \cap \mathcal{U}_2)$, $p{\downarrow}\mathcal{U}_1{\downarrow}\mathcal{U}_2 = p{\downarrow}\mathcal{U}_2{\downarrow}\mathcal{U}_1$, $p{\downarrow}\mathcal{U}_1{\downarrow}\mathcal{U}_1 = p{\downarrow}\mathcal{U}_1$, etc.

The concepts and properties above extend naturally to infinite traces and other types of executions that involve action sets.

8.4.5 Reflection and Rotations

Unary process space operators, such as reflection, forward rotation, and backward rotation, can also be studied as process abstractions.

Proposition 8.23 *In process space $\mathcal{S}_{\mathcal{E}}$, reflection is $(\mathrm{Im}_{\mathbf{id}_{\mathcal{E}}})^{\sqcup\sqcap}$, forward rotation is $(\mathrm{Im}_{\mathbf{id}_{\mathcal{E}}})^{\times\sqcap}(p)$, and backward rotation is $(\mathrm{Im}_{\mathbf{id}_{\mathcal{E}}})^{\otimes\sqcap}(p)$.*

Since the underlying relationships of these PAs are injective and co-surjective, all processes are independent for these PAs.

8.4.6 Process Compositions

A Galois connection between two functions that derive from a parallel composition operator was noticed in [Ver94b, p. 49], although that observation was not carried any further.

We show here that binary process space operators, such as product and meet, can be regarded as process abstractions that derive from relations. Product corresponds to a $++$-PA and meet corresponds to a $\sqcup\sqcup$-PA as follows.

Proposition 8.24 *For processes p and q over $\mathcal{E}$,*

$$p \times q = (\mathbf{id}_{\mathbf{as}\,q}, \mathbf{empty}, \mathbf{empty}, \mathbf{id}_{\mathbf{g}\,q})^{++}\ (p)\ ,\ \textit{and}$$
$$p \sqcap q = (\mathbf{id}_{\mathbf{at}\,q}, \mathbf{empty}, \mathbf{empty}, \mathbf{id}_{\mathbf{e}\,q})^{\sqcup\sqcup}\ (p)\ ,$$

where **empty** *denotes a UPM that always returns the empty set.*

Since we have restricted our treatment to $\sqcap\sqcap$-SPAs, while the $\sqcap\sqcap$-PAs in Proposition 8.24 are not $\sqcap\sqcap$-SPAs, we do not generalize here the observation of [Ver94b]. That observation may be an indication that some of the properties we have shown for $\sqcap\sqcap$-SPAs can be extended to general $\sqcap\sqcap$-PAs; however, we leave such extensions for further work.

8.4.7 Divergence

One of the influential trends in concurrency theory is to model the behavior after an undesirable input by allowing any actions to occur; see for instance [Hoa85] and [Jos92]. We can capture this viewpoint in process spaces by declaring that a reject manifests itself just by being accessible; for that, we can transform the rejects of a process into goals.

Definition 8.10 *Let* **r2g** *(for* rejects-to-goals*) be a map between processes over the same execution set, such that, for any process* p*,*

$$\mathbf{as}\ \mathbf{r2g}(p) = \mathbf{as}\, p\ ;$$

$$\mathbf{r}\ \mathbf{r2g}(p) = \emptyset\ .$$

Example 8.11 After hearing an "XXX" or an environment greeting out of turn, the etiquette machine of Example 2.1 may lose its composure and start babbling anything, although in this case it does not become unhappy. The **r2g** map applied to the original etiquette machine process yields a process for which executions like $e1\ m1\ ex\ m1\ m1\ m1$ are not rejects, but are accessible. □

The rejects-to-goals map is also a $\sqcap\sqcap$-PA, although not a $\sqcap\sqcap$-SPA.

Proposition 8.25 *In process space* $\mathcal{S}_{\mathcal{E}}$*,* $\mathbf{r2g} = (\mathbf{id}_{\mathcal{E}}, \emptyset, \emptyset, \emptyset)^{\sqcap\sqcap}$.

8.5 Further Applications

We have demonstrated the applicability of process abstractions to several types of maps between processes. One immediate benefit is that the operations of relabeling, projection, derivatives, restriction, etc., are shown to satisfy several algebraic properties, such as properties of composition, criteria for verifications of images, etc.

More importantly, since our process maps are derived from a mathematical construct as basic as binary relations, we expect that many other maps of practical interest can be studied this way. It may be feasible, for instance, to use the prefix closure (see Example 8.4) or similar operations for studying links among default processes for liveness and progress and perhaps new default processes for safety and finalization. Relations between analog and digital models along the lines in Chapter 4 and Chapter 7 are also within reach: the problem of modeling these analog-digital relations is reduced to that of choosing appropriate relations between analog trajectories and sequences of events.

Chapter 9

Concluding Remarks

For a large part, process spaces constitute a reworking, simplification, or generalization of previous theories of concurrency. In doing this reworking, however, we depart from the usual approach of extending previous models by additional features such as timing, infinite traces, etc. We take the opposite approach, which is to take things out of previous formalisms, take them apart, and show that what is left allows for further applications. In particular, we consider connectivity restrictions undesirable, and we eliminate them entirely.

At the same time, there is a fair number of new applications and new algebraic properties that do not seem to overlap with previously known results. While we generalize our theory as much as possible, we ensure that sufficient new applications exist to the formal verification of asynchronous circuits to motivate by themselves the existence of the new theory.

In addition, we illustrate by examples that process spaces hold promise of future applications to other types of systems and correctness concerns, and we also mention some clues that invite further development of the theory.

As a separate development, an extension of process spaces has been used to obtain representation theorems for ternary algebras [BLN97, Esi97].

9.1 Contributions of the Thesis

We formalize the basics of a general theory of processes that relies on abstract executions. Like several previous treatments of concurrency, process spaces have operators for parallel composition, non-deterministic choice, mirroring, hiding of actions, and process derivatives, and conditions for satisfaction of a specification and for autonomous operation. The algebraic structure obeys a duality principle, and captures generic properties for verification.

In addition, process spaces have several new operations and properties, among which we highlight a ternary symmetry principle based on a rotation operation, criteria for pessimistic approximation and optimistic approximation in terms of underlying binary relations on executions, and a notion of independence from an arbitrary process abstraction. Our properties work for finite and infinite families of processes as well, and have no connectivity restrictions. The results for infinite families of processes answer a challenge from [Ver94b, p. 111] regarding infinite substitutions; see Section 2.5.

Diverse types of executions have been used in previous models (see Section 1.2), and there are substantial similarities among treatments of various correctness concerns in CSP and trace theory. On the other hand, abstract executions are a novel approach, and this generalization required elimination of notions such as actions, ports, variables, states, or events, which, in one form or another, stand at the basis of previous theories of concurrency. In particular, the process space operations, correctness conditions, and their algebraic properties are studied without restrictions regarding inputs, outputs, or alphabets of actions. Although parallel composition and refinement have been defined in CSP and trace theory essentially by set intersections and subset relationships on trace sets, those operations had restrictions on the actions of the processes involved, and it was not obvious beforehand that the algebraic properties for structured verification could be extended, with minor modifications, beyond these restrictions.

Even less predictable was that projections, hidings, and relabelings could be formalized and handled by means abstract executions as well. Not only does this formalization capture many of their algebraic properties, but it also points out that these operations have a lot in common with process derivatives, reflection, and even parallel composition. Although related to previous results studied by [CGL92], [LGS+95], and [Sif83] in an operational setting, our criteria for reduced verification and our reciprocity theorem in terms of abstract executions are new and more general. The methods in [CDYC97], [DWT95], and [SYP+97] are directed towards particular applications, and are less general than the above.

At the same time, we carry out two niche applications for the verification of asynchronous circuits, which, we believe, demonstrate some advantages of process spaces over previous treatments of discrete-state systems, by exploiting the absence of connectivity restrictions in process spaces. We use a BDD-based tool to verify published asynchronous circuits. The performance for flat verification (see Chapter 3) is not top-of-the-line among BDD-based tools, but it does bring many practical designs within our striking distance. In addition, the tool provides support for modular and hierarchical verification without any connectivity restrictions. This flexibility is useful for verifying larger designs.

The chain constraint approach permits us to use relative delay constraints, while keeping the advantages of metric-free analysis. Relative delay constraints can replace certain circuit components to reduce area, power consumption, and latency. Metric-free analysis enables pre-layout verification, because numerical bounds on component and wire delays are not known with sufficient precision until after the transistor sizes and wire lengths are set. Our approach differs from previous methods for handling delay constraints for asynchronous circuits by not requiring knowledge of numerical bounds on component delays.

The generality of the theory is illustrated by guidelines and examples of modeling several types of systems, ranging from concurrent programs to steady-state linear electrical networks, and of handling various correctness concerns for discrete-state systems.

Previous attempts to extract liveness properties from finitary specifications have concluded that finitary descriptions are insufficiently powerful. However, many designers have not rushed to embrace infinitary formalisms that have been proposed to address this shortcoming; for instance, Petri nets (which essentially specify finitary languages) are massively used to this day to specify systems that involve arbitration and deadlock concerns. While we agree that general liveness properties are not captured by finitary specifications, it turns out that default liveness and progress properties can be attached to finitary specifications in a way that suits many applications.

At the same time, we provide the possibility for specifying general liveness properties with infinitary specifications, to accommodate systems that might not fit our default liveness properties, and to present our algebraic treatment of liveness and progress as instances of process spaces by using infinite sequences of actions as executions.

Our formal classification of lock faults appears to be new, although the proposed formalizations for deadlock, livelock, and starvation match informal examples in [Ver94b, p. 66] and [Dil89, p. 138], for instance.

9.2 Current Limitations and Further Work

The performance, interface, and application areas of FIREMAPS can be greatly enhanced. On the immediate wish-list, there is an update of the BDD engines to use partitioned transition relations, zero-suppressed BDDs, dynamical variable ordering, etc. Also, the criteria for optimistic and pessimistic approximation and independence should be implemented, to provide other instruments for taming state explosion. A graphical user interface needs to be designed, and the present script language of the tool should be extended to allow input in the form of signal transition graphs or Petri nets. Infinitary language specifications and timed specifications should eventually be incorporated.

The applications to concurrent programs, protocols, and dynamical systems should be carried further to actual verification techniques. Also, we expect applications to other types of systems, such as verification of data integrity and data-base operations, to be within reach.

For the chain constraint application, we should do more to actually find the chain constraints that would suffice to guarantee a system is correct, rather than just providing hints in counter-example executions. Moreover, although we are doing well without using numerical timing properties, in the future they may be needed. Also, the analysis of switch-level correctness concerns should be extended to MOS transistor networks that are not necessarily partitioned into P and N networks adjacent to the power and ground supplies, respectively. Good targets for additional applications to asynchronous circuit verification are any circuits that use unusual design tricks. For example, circuits with bi-directional ports, such as the single-track handshake circuits [BB96], could exploit the absence of formal distinctions between inputs and outputs in process spaces.

The theoretical story is not over either. A lot work needs to be done to extend our approximation, independence, and reciprocity results to process abstractions that are not SPAs; however, the relationship between such process maps and the product operator hints that interesting results can be obtained. The relationships between delay-insensitivity and our notion of independence should also be investigated.

Finally, we would like to recommend that process maps be used for building formal bridges between existing models of discrete-state systems, in the hope that the various viewpoints in circulation will converge towards a unified theory of concurrency.

Bibliography

[ACD$^+$92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwacks, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the Real-Time Systems Symposium*, pages 157–166. IEEE Computer Society Press, 1992.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:186–235, 1994.

[AH94] R. Alur and T. A. Henzinger. Finitary fairness. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 52–61, 1994.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[BA90] M. Ben-Ari. *Principles of Distributed and Concurrent Programming.* Prentice-Hall, 1990.

[BB96] K. van Berkel and A. Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[BC88] Christian Berthet and Eduard Cerny. An algebraic model for asynchronous circuits verification. *IEEE Transactions on Computers*, 37(7):835–847, July 1988.

[BCDM86] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic circuit verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[BEN96] J. A. Brzozowski, J. C. Ebergen, and R. Negulescu. On abstractions and process spaces. Working manuscript, 1996.

[Ber92] K. van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[Ber93] K. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[BHP95] K. van Berkel, F. Huberts, and A. Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[Bir67] G. Birkhoff. *Lattice Theory*. American Mathematical Society, third edition, 1967.

[BK85] J. A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[Bla86] D. L. Black. On the existence of delay-insensitive fair arbiters: Trace theory and its limitations. *Distributed Computing*, 1:205–225, 1986.

[BLN96] J. A. Brzozowski, J. J. Lou, and R. Negulescu. A characterization of finite ternary algebras. Technical Report CS–96–26, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1996.

[BLN97] J. A. Brzozowski, J. J. Lou, and R. Negulescu. A characterization of finite ternary algebras. *International Journal of Algebra and Computation*, 7(6):713–721, 1997.

[BN97] J. A. Brzozowski and R. Negulescu. Automata of asynchronous behaviors. In *Proceedings of the 1997 Workshop on Implementing Automata (WIA97)*, London, Ontario, Canada, September 1997. Also in *Lecture Notes in Computer Science*, LNCS 1436, Springer, Berlin, 1998.

[BN98] J. A. Brzozowski and R. Negulescu. Automata of asynchronous behaviors. *Theoretical Computer Science*, 1998. To appear in special issue based on WIA97.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[Bro89] R. W. Brockett. Smooth dynamical systems which realize arithmetical and logical operations. In H. Nijmeijer and J. M. Schumacher, editors, *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, volume 135 of *Lecture Notes in Control and Information Sciences*, pages 19–30. Springer-Verlag, 1989.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry87] R. E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 634–649, 1987.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[BS95] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[BY75] J. A. Brzozowski and M. Yoeli. Models for analysis of races in sequential circuits. Lecture Notes in Computer Science, pages 26–31, June 1975.

[BY76] J. A. Brzozowski and M. Yoeli. *Digital Networks*. Prentice-Hall, 1976.

[Car82] L. Cardelli. *An Algebraic Approach to Hardware Description and Verification*. PhD thesis, University of Edinburgh, U.K., 1982.

[CDYC97] S. Chakraborty, D. L. Dill, Kenneth Y. Yun, and Kun-Yung Chang. Timing analysis for extended burst-mode circuits. In *Proc. Interna-*

tional Symposium on Advanced Research in Asynchronous Circuits and Systems. IEEE Computer Society Press, April 1997.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[CMP92] E. Chang, Z. Manna, and A. Pnueli. The safety-progress classification. Technical Report STAN–CS–92–1408, Stanford University, Dept. of Computer Science, 1992.

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

[DNH83] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Compuer Science*, 34:83–133, 1983.

[DNS92] D. L. Dill, S. M. Nowick, and R. F. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in System Design*, 1(1):29–60, July 1992.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[DWT95] D. L. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximations. In *Proc. International Workshop on Computer Aided Verification*, pages 409–422, 1995.

[EB95] J. Ebergen and R. T. Berks. VERDECT: A verifier for asynchronous circuits. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995.

[Ebe89] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[Ebe91a] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[Ebe91b] J. C. Ebergen. Parallel computations and delay-insensitive circuits. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 85–104. Springer-Verlag, 1991.

[Esi97] Z. Esik. A Cayley theorem for ternary algebras. *International Journal of Algebra and Computation*, 1997.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.

[GC94] M. R. Greenstreet and P. Cahoon. How fast will the flip flop? In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 77–86, November 1994.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Hen88] M. Hennessy. *Algebraic theory of processes*. Foundations of Computing. MIT Press, 1988.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[Jon87] B. Jonsson. Modular verification of asynchronous networks. In *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

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

[JU90] M. B. Josephs and J. T. Udding. An algebra for delay-insensitive circuits. In R. P. Kurshan and E. M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, 1990.

[KM91] R. Kurshan and K. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design*, 10(11):1356–1371, 1991.

[KN94] M. Kuwako and T. Nanya. Timing-reliability evaluation of asynchronous circuits based on different delay models. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 22–31, November 1994.

[KS95] T. Kam and P. A. Subrahmanyam. Comparing layouts with HDL models: A formal verification technique. *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, pages 503–509, April 1995.

[KSL95] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin. Verity—a formal verification program for custom CMOS circuits. *IBM Journal of Research and Development*, (1/2):149–165, March 1995.

[Kur94] R. P. Kurshan. The complexity of verification. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 365–371, 1994.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, (6):1–35, 1995.

[LL90] L. Lamport and N. Lynch. Distributed computing: Models and methods. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Methods and Semantics*, pages 1159–1196. The MIT Press-Elsevier, 1990.

[LMBSV92] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[Mar90] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[Maz86] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets, Part II: Applications and Relationships to Other Models of Concurrency*, volume 255 of Lecture Notes in Computer Science, pages 279–324. Springer-Verlag, 1986.

[McM92] K. L. McMillan. *Symbolic Model Checking—An Approach to the State Explosion Problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1992.

[MD92] K. McMillan and D. L. Dill. Algorithms for interface timing verification. In *Proceedings of the 1992 ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mil90] R. Milner. Semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Methods and Semantics*, pages 1201–1242. The MIT Press–Elsevier, 1990.

[Mil94] G. J. Milne. *Formal Specification and Verification of Digital Systems.* McGraw-Hill, 1994.

[MJCL97] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium* *on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[Mye95] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[NB95] R. Negulescu and J. A. Brzozowski. Relative liveness: From intuition to automated verification. In *Asynchronous Design Methodologies*, pages 108–117. IEEE Computer Society Press, May 1995.

[NB98] R. Negulescu and J. A. Brzozowski. Relative liveness: From intuition to automated verification. *Formal Methods in System Design*, 12(1):73–115, January 1998.

[Neg93] R. Negulescu. Lock phenomena in asynchronous circuits. Graduate course project, 1993.

[Neg95] R. Negulescu. Process spaces. Technical Report CS–95–48, Dept. of Computer Science, Univ. of Waterloo, December 1995.

[Neg97a] R. Negulescu. A technique for finding and verifying speed-dependences in gate circuits. Technical Report CS–97–28, Dept. of Computer Science, Univ. of Waterloo, Canada, August 1997.

[Neg97b] R. Negulescu. A technique for finding and verifying speed-dependences in gate circuits. In *Proceedings of the 1997 ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 189–198, Austin, Texas, USA, December 1997.

[Neg98] R. Negulescu. Event-driven verification of switch-level correctness concerns. In *Int. Conf. on Application of Concurrency to System Design*, pages 213–223, March 1998.

[Nie97] L. S. Nielsen. *Low-Power Asynchronous VLSI Design*. PhD thesis, Deptartment of Information Technology, Technical University of Denmark, Lyngby, Denmark, October 1997.

[NP98] R. Negulescu and A. Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 159–170, 1998.

[NS95] L. S. Nielsen and J. Staunstrup. Design and verification of a self-timed RAM. In *Proceedings of the IFIP International Conference on VLSI*, pages 751–758, 1995.

[PDF+98] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[Pee96] A. M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pra91] I. S. W. B. Prasetya. Solving the design equation in the failures model. Master's thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1991.

[RM94] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In *Proc. International Workshop on Computer Aided Verification*, pages 468–480, 1994.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, January 1987.

[SHSJ73] N.E. Steenrod, P.R. Halmos, M.M. Schiffer, and J.A.Dieudonné. *How to Write Mathematics*. American Mathematical Society, 1973.

[Sid95] P. R. Sidorowicz. Verification of the control circuitry of the CFPP architecture. Graduate course project, 1995.

[Sif83] J. Sifakis. Property preserving homomorphisms of transition systems. In E. Clarke and D. Kozen, editors, *Proceedings of the 4th Workshop on Logics of Programs*, Pittsburgh, U.S.A., June 1983.

[Sme89] R. Smedinga. *Control of Discrete Events*. PhD thesis, Univ. of Groningen, The Netherlands, 1989.

[Sme90] R. Smedinga. Discrete event systems: Deadlock, livelock, and livedeadlock. In *Proceedings of the 11th IFAC World Congress*, volume III, Tallinn, Estonia, August 1990.

[SS91] R.F. Sproull and I.E. Sutherland. Logical effort: Designing for speed on the back of an envelope. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*. MIT Press, March 1991. Also see `http://www.mkp.com/Logical_Effort` .

[Sta97] J. Staunstrup. Personal communication, 1997.

[Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[SYP+97] A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella, and L. Lavagno. Partial order based approach to synthesis of speed-independent circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 254–265. IEEE Computer Society Press, April 1997.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Methods and Semantics*, pages 135–191. The MIT Press–Elsevier, 1990.

[Ver] T. Verhoeff. Encyclopedia of delay-insensitive systems (EDIS). `http://edis.win.tue.nl/` .

[Ver94a] T. Verhoeff. The testing paradigm applied to network structure. Computing Science Notes 94/10, Dept. of Math. and C. S., Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.

[Ver94b] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[VK98] A. Valmari and I. Kokkarinen. Unbounded verification results by finite-state compositional techniques: 10^{any} states and beyond. In *Int. Conf. on Application of Concurrency to System Design*, pages 75–85, March 1998.

[VT95] A. Valmari and M. Tiernari. Compositional failure-based semantic models for basic LOTOS. *Formal Aspects of Computing*, 7:440–468, 1995.

[WR87] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM J. Control and Optimization*, 25(3):637–659, May 1987.

[ZH92] C. Zhou and C. A. R. Hoare. A model for synchronous switching circuits and its theory of correctness. *Formal Methods in System Design*, 1:7–28, 1992.